UCSD p-System*

- Assembler
- Linker

Part One: UCSD p-System Assembler

*UCSD p-System is a trademark of the Regents of the University of California.

1041569-1
[Part A]

UCSD p-System*

- Assembler
- Linker

Part One: UCSD p-System Assembler

*UCSD p-System is a trademark of the Regents of the University of California.

Copyright © 1981, Texas Instruments Incorporated
See important warranty information at back of book.

This manual was developed by staff members of the Texas Instruments Education and Communications Center.

This software is copyrighted 1979, 1981 by the Regents of the University of California, SofTech Microsystems, Inc., Texas Instruments Incorporated, and other copyright holders as identified in the program code. No license to copy this software is conveyed with this product. Additional copies for use on additional machines are available through Texas Instruments Incorporated. No copies of the software other than those provided for in Title 17 of the United States Code are authorized by Texas Instruments Incorporated.

UCSD Pascal and UCSD p-System are trademarks of the Regents of the University of California. Item involved met its quality assurance standards applicable to Version IV.0.

TABLE OF CONTENTS

GENERAL INFORMATION 9
- 1.1 Using this Manual 11
- 1.2 Set-Up Instructions 12
- 1.3 Special Keys 17

OPERATION OF THE ASSEMBLER 19
- 2.1 Establishing Input and Output Files 20
- 2.2 Responses to Listing Prompt 21
- 2.3 Output Modes 22
- 2.4 Error Conditions 23

FORMAT INFORMATION 24
- 3.1 Registers 24
  - 3.1.1 Program Counter Register (PC) 24
  - 3.1.2 Workspace Pointer Register (WP) 24
  - 3.1.3 Status Register (ST) 24
- 3.2 Object Code Format 30
  - 3.2.1 Byte Organization 30
  - 3.2.2 Word Organization 30
  - 3.2.3 Memory Organization 30
- 3.3 Source Code Format 31
  - 3.3.1 Character Set 31
  - 3.3.2 Identifiers 31
  - 3.3.3 Character Strings 32
  - 3.3.4 Constants 32
  - 3.3.5 Expressions 32
- 3.4 Source Statement Format 38
  - 3.4.1 Label Field 38
  - 3.4.2 Op-code Field 39
  - 3.4.3 Operand Field 40
  - 3.4.4 Comment Field 40
- 3.5 Source File Format 41
  - 3.5.1 Assembly Routines 41
  - 3.5.2 Global Declarations 41
  - 3.5.3 Absolute Sections 42

TABLE OF CONTENTS

ADDRESSING MODES 43
4.1 General Addressing Modes 43
4.1.1 Workspace Register Addressing 44
4.1.2 Workspace Register Indirect Addressing 44
4.1.3 Workspace Register Indirect Auto-Increment Addressing . . . 45
4.1.4 Symbolic Memory Addressing 45
4.1.5 Indexed Memory Addressing 46
4.2 Program Counter Relative Addressing 47
4.3 CRU Bit Addressing 48
4.4 Immediate Addressing 49
4.5 Addressing Summary 50

INSTRUCTION FORMATS 52
5.1 Format I -- Two General Address Instructions 53
5.2 Format II -- Jump Instructions 54
5.2.1 Format II -- Bit I/O Instructions 55
5.3 Format III -- Logical Instructions 56
5.4 Format IV -- CRU Multi-bit Instructions 57
5.5 Format V -- Register Shift Instructions 58
5.6 Format VI -- Single Address Instructions 59
5.7 Format VII -- Control Instructions 60
5.8 Format VIII -- Immediate Instructions 61
5.9 Format IX -- Extended Operation Instruction 63
5.9.1 Format IX -- Multiply and Divide Instructions 64

ARITHMETIC INSTRUCTIONS 65
6.1 Add Words--A 67
6.2 Add Bytes--AB 69
6.3 Absolute Value--ABS 71
6.4 Add Immediate--AI 72
6.5 Decrement--DEC 73
6.6 Decrement by Two--DECT 74
6.7 Divide--DIV 75
6.8 Increment--INC 77
6.9 Increment by Two--INCT 78
6.10 Multiply--MPY 79
6.11 Negate--NEG 81
6.12 Subtract Words--S 82
6.13 Subtract Bytes--SB 83

6.14 Instruction Examples 85
6.14.1 Incrementing and Decrementing Examples 85
6.14.2 General Example 89

JUMP AND BRANCH INSTRUCTIONS 90
7.1 Branch--B 93
7.2 Branch and Link--BL 94
7.3 Branch and Load Workspace Pointer--BLWP 95
7.4 Jump If Equal--JEQ 96
7.5 Jump If Greater Than--JGT 97
7.6 Jump If High or Equal--JHE 98
7.7 Jump If Logical High--JH 99
7.8 Jump If Logical Low--JL 100
7.9 Jump If Low or Equal--JLE 101
7.10 Jump If Less Than--JLT 102
7.11 Unconditional Jump--JMP 103
7.12 Jump If No Carry--JNC 104
7.13 Jump If Not Equal--JNE 105
7.14 Jump If No Overflow--JNO 106
7.15 Jump If Odd Parity--JOP 107
7.16 Jump On Carry--JOC 108
7.17 Return with Workspace Pointer--RTWP 109
7.18 Execute--X 110
7.19 Extended Operation--XOP 111
7.20 Instruction Examples 113
7.20.1 Common Workspace Subroutine Example 113
7.20.2 Context Switch Subroutine Example 115
7.20.3 Passing Data to Subroutines 119
7.20.4 Extended Operations 121
7.20.5 Execute Example 122

COMPARE INSTRUCTIONS 123
8.1 Compare Words--C 125
8.2 Compare Bytes--CB 127
8.3 Compare Immediate--CI 128
8.4 Compare Ones Corresponding--COC 129
8.5 Compare Zeros Corresponding--CZC 131

TABLE OF CONTENTS

CONTROL AND CRU INSTRUCTIONS 133
9.1 Load CRU--LDCR 136
9.2 Set CRU Bit to One--SBO 137
9.3 Set CRU Bit to Zero--SBZ 138
9.4 Store CRU--STCR 139
9.5 Test Bit--TB 141
9.6 Other Instructions 142
9.7 CRU Input/Output 143
9.7.1 CRU I/O Instructions 143
9.7.2 Accessing Specific Bits 143
9.7.3 SBO Example 144
9.7.4 SBZ Example 144
9.7.5 TB Example 144

LOAD AND MOVE INSTRUCTIONS 145
10.1 Load Immediate--LI 147
10.2 Load Interrupt Mask Immediate--LIMI 148
10.3 Load Workspace Pointer Immediate--LWPI 150
10.4 Move Word--MOV 151
10.5 Move Byte--MOVB 153
10.6 Store Status--STST 154
10.7 Store Workspace Pointer--STWP 155
10.8 Swap Bytes--SWPB 156
10.9 Instruction Example 157

LOGICAL INSTRUCTIONS 159
11.1 AND Immediate--ANDI 161
11.2 OR Immediate--ORI 163
11.3 Exclusive OR--XOR 165
11.4 Invert--INV 167
11.5 Clear--CLR 169
11.6 Set to One--SETO 170
11.7 Set Ones Corresponding--SOC 171
11.8 Set Ones Corresponding, Byte--SOCB 173
11.9 Set Zeros Corresponding--SZC 175
11.10 Set Zeros Corresponding, Byte--SZCB 177

WORKSPACE REGISTER SHIFT INSTRUCTIONS 179
12.1 Shift Right Arithmetic--SRA 181
12.2 Shift Right Logical--SRL 183
12.3 Shift Left Arithmetic--SLA 185
12.4 Shift Right Circular--SRC 187
12.5 Instruction Example 189

ASSEMBLER DIRECTIVES 191
13.1 Procedure Delimiting Directives 194
13.2 Data and Constant Definition Directives 196
13.3 Location Counter Modification Directives 198
13.4 Listing Control Directives 199
13.5 Program Linkage Directives 202
13.6 Conditional Assembly Directives 204
13.7 Macro Definition Directives 205
13.8 Miscellaneous Directives 206

CONDITIONAL ASSEMBLY 208
14.1 Conditional Expressions 209

MACRO LANGUAGE 210
15.1 Macro Definitions and Calls 211
15.2 Parameter Passing 213
15.3 Scope of Labels in Macros 215
15.3.1 Local Labels as Macro Parameters 215

ASSEMBLER OUTPUT 217
16.1 Source Listing 218
16.2 Error Messages 219
16.3 Code Listing 220
16.3.1 Forward References 220
16.3.2 External References 220
16.3.3 Multiple Code Lines 220
16.4 Symbol Table 221
16.5 Example 222

TABLE OF CONTENTS

APPENDICES 224
17.1 Memory Organization 225
17.1.1 Directly Addressable Memory 225
17.1.2 Memory-Mapped Devices 228
17.2 Memory, CRU, and Interrupt Structure 230
17.2.1 CRU Allocation 230
17.2.2 Interrupt Handling 231
17.3 Character Set 233
17.4 Assembler Directive Table 236
17.5 Hexadecimal Instruction Table 239
17.6 Alphabetical Instruction Table 242
17.7 Program Development with Multi-Drive Systems 245
17.7.1 Two-Drive System 245
17.7.2 Three-Drive system 245

IN CASE OF DIFFICULTY 246

WARRANTY 247

SECTION 1: GENERAL INFORMATION

The UCSD p-System* Assembler allows you to write programs in the powerful assembly language of the TMS9900 microprocessor built into the TI-99/4 and TI-99/4A Home Computers. The TMS9900 has all of the features expected from an advanced microprocessor, including both byte- and word-oriented commands, a variety of addressing modes, and fast context switching.

The use of assembly language instead of a higher-level language, such as BASIC or Pascal, has several advantages. The execution of assembly language programs is much faster. In addition, assembly language gives you access to all machine resources, including functions not available from higher-level languages.

An assembly language consists of symbolic names which represent machine instructions, memory addresses, or program data. The instructions are mnemonic codes, which are easier to use and remember than the symbols of object code. In addition, you use expressions as operands and can use decimal numbers in expressions and as operands. Further, the use of assembly language relieves you of the tedious task of writing machine language instructions and keeping track of binary machine addresses within the program.

An assembly language program (called source code) is converted by an assembler into a sequence of machine instructions (called object code). Assemblers create either relocatable or absolute object code. Relocatable code includes information that allows a loader to place it in any available area of memory, while absolute code must be loaded into a specific area of memory. Symbolic addresses in programs that are assembled to relocatable object code are called relocatable addresses.

With the UCSD p-System, you can develop assembly language programs that run under the control of a host program in Pascal or another high-level language.

*trademark of the Regents of the University of California.

GENERAL INFORMATION

The UCSD p-System Assembler, in conjunction with the Linker and some support programs, meets this need. It is a single-pass assembler modeled after The Last Assembler (TLA), developed at the University of Waterloo. The basic concept behind both the TLA and the UCSD p-System Assembler is the use of a central machine-independent core that is common to all versions of the UCSD p-System Assembler. This central core is augmented with machine-specific modules to handle the architecture of each specific machine.

The simplest configuration for running the Assembler requires the TI Home Computer, the TI Color Monitor (or a video modulator and a television set), the Memory Expansion unit, the p-Code peripheral, and a Disk Memory System with at least one Disk Memory Drive. With this equipment, plus the diskette containing the Editor and Filer and the diskette containing the Assembler, you can develop and assemble programs. To enhance your system, you can add Disk Memory Drives, the RS232 Interface, or other peripherals available from Texas Instruments.

If you are using the Assembler to develop your own programs, first create the files with the UCSD p-System Editor (sold separately). Next, assemble the program you created with the Assembler. Then, to link several files, use the Linker. (For more detailed information, refer the the UCSD p-System Editor manual and the UCSD p-System Linker manual.)

The Assembler is a one-pass assembler, with the ability to patch forward references after the fact.

The Assembler predefines registers and optionally produces a list of the source and object code and the symbol table.

After a file has been assembled and linked, you can load and run it as described in the UCSD p-System p-Code owner's manual.

1.1 USING THIS MANUAL

This manual assumes that you already know a programming language, preferably an assembly language. If you do not, there are many fine books available which teach the basics of assembly language use. After you know these basics, this manual gives the details of TMS9900 assembly language.

This manual provides details on assembling programs on the TI Home Computer and includes explanations of the following.

- Using the Assembler.
- All TMS9900 assembly language instructions and pseudo-instructions.
- Assembler output.

When terms that may be new to you are first used, they are defined. Section 2 explains the basics of using the Assembler. Sections 3 through 13 are a detailed description of the TMS9900 assembly language. Sections 14 through 16 discuss conditional assembly, the use of macro language, and assembler output. Section 17 is the Appendix. The last section provides service and warranty information.

1.2 SET-UP INSTRUCTIONS

The steps involved in creating and linking an assembly language file and Pascal file are included in this section. Please read this material completely before proceeding.

Use your Disk Manager or the Filer to make a backup copy of the diskette which contains the Assembler. You may use this copy for your own use. The original should be kept in a safe place.

Note: For the recommended placement of files on a multi-drive system, see the Appendix.

1. Be sure that the Memory Expansion unit, the p-Code peripheral, and the Disk Memory System are attached to the computer and turned on. Refer to the appropriate owner's manuals for product details.

2. To create an assembly language program, use the p-System Editor. Insert the Editor diskette into a disk drive.

3. Turn on the monitor and computer console. The System promptline appears. **Note:** If you turn on the computer before inserting a diskette in a disk drive, you must insert a diskette and then press **I** to initialize the System before you can proceed.

4. Press **E**, for E(dit, to load the Editor.

5. Refer to the the UCSD p-System Editor owner's manual for detailed directions on entering a program. When you have completed your program, press **Q** for Q(uit. Then press **W** for W(rite.

6. Remove the Editor diskette and insert the diskette on which you wish to save the assembly language program. If you have one disk drive, the assembly language program must be saved on the diskette that contains the Assembler and Linker. If the assembly language program is too long to fit on this diskette, then two disk drives are required.

7. Enter the filename for the assembly language program and press <return>.

8. Place the diskette that contains the Assembler and Linker and the assembly language program to be assembled in a disk drive.

9. Press **A**, for A(ssemble, to load the Assembler.

10. The screen displays the message

```
Assembling...
```

while the Assembler is loaded. If the workfile, SYSTEM.WRK.TEXT, exists, that file is assembled, the assembly language code produced is saved as SYSTEM.WRK.CODE, and you may proceed to step 11.

If SYSTEM.WRK.TEXT does not exist, the following prompt appears.

```
Assemble?
```

Enter the location and name of the assembly language program which you wish to have assembled. For example, to assemble the program TESTA.TEXT, which is contained on the diskette in disk drive 2 (#5), enter

```
#5:TESTA
```

Next the prompt

```
To what codefile?
```

appears. Enter the location and name of the file to which you wish the code to be saved. For example, if you wish the code to be saved as TESTA.CODE on the diskette in disk drive 2 (#5), enter

```
#5:TESTA
```

If you wish the code to be saved as SYSTEM.WRK.CODE on the diskette in disk drive 1 (#4), just press <return>.

Next the prompt

```
Output file for assembled listing: (<cr> for none)
```

appears. If you wish to have the output file saved, enter the location and file name. Otherwise press <return>.

11. While the file is being assembled, an account of the progress and any error messages are displayed. The following is the display when a small assembly language program named TESTA is assembled.

```
9900    Assembler   IV.0 [a.3]
<0      >
TESTA
<5      >...
Assembly complete:          3 lines
     0    errors flagged on this assembly
```

A description of the meaning of this display is given in Section 17.

12. When the assembling process is finished, the System promptline reappears. If you have only one disk drive, you must transfer the completed code to the diskette which contains the Compiler. Use the Filer to make this transfer, as described in the UCSD p-System Filer owner's manual.

13. Next you must create and compile a Pascal program to call the assembly language program. To create a Pascal program, use the System Editor. Insert the Editor diskette into a disk drive.

14. Press **E**, for E(dit, to load the Editor.

15. Refer to the the UCSD p-System Editor owner's manual for detailed directions on entering a program. When you have completed your program, press **Q** for Q(uit. Then press **W** for W(rite.

16. Remove the Editor diskette and insert the diskette on which you wish to save the Pascal program. If you have one disk drive, the Pascal program must be saved on the diskette that contains the Compiler. If the Pascal program plus the assembly language program are too long to fit on this diskette, then two disk drives are required.

17. Enter the filename for the Pascal program and press <return>.

18. If you wish, the assembly language code may be put in a library, such as SYSTEM.LIBRARY, by using the utility LIBRARY. (See the UCSD p-System Utility owner's manual.) Then the code may be R(un. The compilation process is as described in step 21, and the linking is done as part of the R(un process, and proceeds as described in step 22. The file is then run.

    If you do not wish to put the assembly language code into a library, then you must compile the Pascal code, link the assembly language code to the Pascal code, and X(ecute the program. These steps are described below.

19. Place the diskette that contains the Compiler, the Pascal program to be compiled, and the code from the assembly language program in a disk drive.

20. Press **C**, for C(ompile, to load the Compiler.

21. Refer to the the UCSD p-System Compiler owner's manual for detailed directions on creating and compiling a program.

22. If you are R(unning the program, the linking process continues as described in the UCSD p-System Linker manual. Your program is then run. Otherwise, when the compiling process is finished, the System promptline reappears.

23. The assembly language code must be linked with the Pascal code in order to run the files. If you have one disk drive, transfer the code files for the Pascal program and assembly language program to the diskette containing the Assembler and Linker. Use the Filer to make this transfer, as described in the UCSD p-System Filer owner's manual. If the Pascal program plus the assembly language program are too long to fit on this diskette, then two disk drives are required. If you have two disk drives, the Pascal code and assembly language code should be saved on the same diskette. If you have three disk drives, the Pascal program may be saved on any diskette.

24. Refer to the UCSD p-System Linker manual for detailed instructions on linking Pascal and assembly language programs.

25. If you are R(unning the file, it is now run. Otherwise, when the linking process is finished, the System promptline reappears. You may run the linked file with the X(ecute command, as described in the UCSD p-System Editor owner's manual.

If you have only one disk drive, the size of the programs which you may assemble, compile, and link is limited to the memory available on the diskette which contains the Compiler and the diskette which contains the Assembler and Linker. If you have two disk drives, then the programs and their code may occupy that memory available plus the space on the diskette which contains the program. You can compile the largest programs if you have three disk drives. See the Appendix for information on using multi-drive systems.

1.3 SPECIAL KEYS

In this manual, the keys that you press are indicated by surrounding them with <angle brackets>. The name <return> is used when the Pascal prompts on the screen refer to <return> or <cr> (carriage return). You should press the <ENTER> key. Pressing any key for more than approximately half a second causes that key to be repeated.

To obtain lower-case letters, press the key with the letter on it. To obtain all upper-case letters on the TI-99/4, use the alpha lock toggle to change to upper-case. On the TI-99/4A you may use the alpha lock toggle or press the <ALPHA LOCK> key. To obtain a single upper-case letter on the TI-99/4 when the computer is in lower-case mode, simultaneously press the key and the small space key on the left side of the keyboard or the space bar. On the TI-99/4A, press the key and <SHIFT>.

Name	TI-99/4	TI-99/4A	Action
<del>	SHIFT F	FCTN 1	Deletes a character.
<ins>	SHIFT G	FCTN 2	Inserts a character.
<flush>	SPACE 3	FCTN 3	Stops writing output to the screen.
<break>	SPACE 4	FCTN 4	Stops the program and initializes the System.
<stop>	SPACE 5	FCTN 5	Suspends the program until this key is pressed again.
<alpha lock>	SPACE 6	FCTN 6 or ALPHA LOCK	Acts as a toggle to convert uppper-case letters to lower-case and back again.
<screen left>	SPACE 7	FCTN 7	Moves the text displayed on the screen to the left 20 columns at a time.
<screen right>	SPACE 8	FCTN 8	Moves the text displayed on the screen to the right 20 columns at a time.
<line del>	SHIFT Z	FCTN 9	Deletes the current line of information.
{	SPACE 1	FCTN F	Types the left brace.
}	SPACE 2	FCTN G	Types the right brace.
[	SPACE 9	FCTN R	Types the left bracket.
]	SPACE 0	FCTN T	Types the right bracket.
<etx/eof>	SHIFT C	CTRL C	Indicates the end of a file.
<esc>	SPACE .	CTRL .	Tells the program to ignore previous text.
<tab>	SHIFT A	CTRL I	Moves the cursor to the next tab.
<up-arrow>	SHIFT E	FCTN E	Moves the cursor up one line.
<left arrow> or <backspace>	SHIFT S	FCTN S	Moves the cursor to the left one character.

Name	TI-99/4	TI-99/4A	Action
<right-arrow>	SHIFT D	FCTN D	Moves the cursor to the right one character.
<down-arrow>	SHIFT X	FCTN X	Moves the cursor down one line.
<return>	ENTER	ENTER	Tells the computer to accept the information you type.

SECTION 2: OPERATION OF THE ASSEMBLER

To access the Assembler, press **A** when the System promptline is displayed. This command executes the file named SYSTEM.ASSMBLER. (Note the missing E in the filename. This is required for conformance with the Filer's restrictions on file name lengths.) If this is not the name of the desired assembler version, be sure to save the existing file SYSTEM.ASSMBLER under a different name before changing the desired assembler's name to SYSTEM.ASSMBLER.

The Assembler has two associated support files: an op-codes file and an error file. These should always be stored along with the Assembler code file.

For the Assembler to run correctly, the proper op-codes file must be present on some on-line diskette, with the name 9900.OPCODES. This file contains all predefined symbols (instruction and register names) and their corresponding values for the associated assembly language. If this file is not on-line, the Assembler displays

```
9900 not on any vol
```

and stops the assembly.

The Assembler also has its own error file, named 9900.ERRORS, which contains a list of error messages. This file need not be present for running the Assembler, but it can greatly aid in removing the syntax errors from a newly written program.

2.1 ESTABLISHING INPUT AND OUTPUT FILES

When the Assembler is first accessed from the promptline, it attempts to open the work file as its input file. If a work file exists, the first prompt is the listing prompt described in Section 2.2 and the generated code file is named SYSTEM.WRK.CODE. If no work file exists, the following prompt appears.

```
Assemble?
```

Type the file name of the file you wish to have assembled and press <return>. To stop the assembly, simply press <return>. Otherwise, the next prompt is as follows.

```
To what codefile?
```

Type the desired name of the output code file and press <return>. Pressing only <return> causes the Assembler to name the output *SYSTEM.WRK.CODE. Entering $ causes the code file to be created with the same filename prefix as the source file. The Assembler then displays its standard listing prompt.

2.2 RESPONSES TO LISTING PROMPT

Before assembling begins, the following prompt appears on the screen.

```
9900 Assembler IV.0 [A.3]
Output file for assembled listing: (<CR> for none)
```

You can respond with one of the following.

- The <esc> key and <return>, which stops the assembly and returns to the System promptline. If you type a file name and then decide to stop Assembler execution, press <line del>, followed by <esc> and <return>.

- CONSOLE: or #1:, either of which sends an assembled listing of the source program to the screen during assembly.

- PRINTER: or #6:, either of which which sends an assembled listing to a printer.

- REMOUT: or #8:, either of which which sends an assembled listing to a printer.

- A <return>, which causes the Assembler to suppress generation of an assembled listing and ignore all listing directives.

- Any other response, which causes the Assembler to write the assembled listing to a text file of that name. Any existing text file of that name is removed in the process.

For instance, the following response causes a list file named LISTING.TEXT to be created on disk drive 2.

```
#5:listing
```

It is your responsibility to ensure that the specified unit is on-line. The Assembler prints an error message and stops the assembly process if it attempts to open an off-line I/O unit.

2.3 OUTPUT MODES

If the listing generated by the Assembler is sent to some unit other than the screen or if no listing is generated, the Assembler writes a running account of the assembly process on the screen for your benefit.

One dot appears on the screen for every line assembled. On every 50th line the number of lines currently assembled is shown on the left side of the screen surrounded by angle brackets.

When an include file directive is processed by the Assembler, the screen displays the current source statement in the form:

```
.INCLUDE <file name>
```

This allows you to keep track of which include file is currently being assembled.

At the end of the assembly, the screen shows the total number of lines assembled in the source program and the total number of errors found in the source program.

2.4 ERROR CONDITIONS

When the Assembler finds an error, it prints the current source statement, if it is applicable to the error, and the error number. (This does not apply to undefined labels and System errors.) It then attempts to retrieve and print an error message from the errors file. If the errors file cannot be opened because the file is not on an on-line device or there is not enough memory, no additional message appears. This is followed by the prompt

```
<sp> (continue), <esc> (terminate), E(dit
```

A space continues the assembly. An escape stops the assembly. Pressing E invokes the Editor. The Assembler considers certain System errors to be fatal. These errors stop the assembly regardless of the response given to the above prompt.

If you press E, the System accesses the Editor, which opens the file containing the error and positions the cursor at the location where the error occurred. This works correctly when the source program is wholly contained in one file. When include files are used, you should set up the input and output files manually (see Section 2.1) for the Editor to position the cursor in the file that contains the error.

In most cases, pressing <spacebar> restarts the assembly process with no problems. Since assembly language source statements are independent with respect to syntax, it is not difficult for the Assembler to continue generating a code file. Thus, a code file exists at the end of an assembly if you press <spacebar> for every (nonfatal) error prompt that appears. Of course, the code produced may not reflect what you intended with your source program.

At the end of an assembly, an error message is printed for each undefined label. You can ignore occurrences of undefined labels if they are irrelevant to the desired execution of the code file.

In addition to generating a code file, the Assembler makes use of a file which is removed from the diskette upon normal termination of the assembly. Occasionally a System error may occur that prevents the Assembler from removing this file. If this happens, a new file named LINKER.INFO may appear. It can be removed because it is useless outside of the assembly process.

SECTION 3: FORMAT INFORMATION

This section discusses how the TI Home Computer and the TMS9900 microprocessor allow you to use Registers, transfer vectors, Workspaces, source statement formats, expressions, constants, symbols, terms, and character strings. It also describes the format of object code, source code, source statements, and source files.

3.1 REGISTERS

A register is a memory word that serves a specific purpose. Registers in Random Access Memory (RAM) are called "software" registers. A set of 16 consecutive registers is called a "workspace."

Three "hardware" registers are located in the CPU itself. They are the Program Counter Register, the Workspace Pointer Register, and the Status Register.

3.1.1 Program Counter Register (PC)

The Program Counter Register (PC) keeps track of the location of the next instruction in memory. The PC manages the program and maintains a sequential and orderly flow of instructions.

3.1.2 Workspace Pointer Register (WP)

The Workspace Pointer Register (WP) contains the address of the current software workspace.

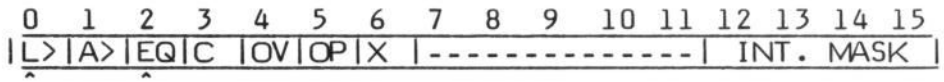
3.1.3 Status Register (ST)

The Status Register (ST) contains indications of the present status of the computer. Each bit of the status register is initialized to zero when the computer is turned on. Then, as each instruction is performed, the computer indicates the status by changing the appropriate "switches" as a result of that instruction. By this method the bits are set (changed to 1) and reset (changed to 0) by machine instructions. Status bits have the following meanings.

Name	Bit Number	Meaning
L>	0	Logical greater than
A>	1	Arithmetic greater than
EQ	2	Equal
C	3	Carry
OV	4	Overflow
OP	5	Odd parity
X	6	Extended operation
-	7-11	Reserved
INT. MASK	12-15	Interrupt mask

In the diagrams in this manual, bits that are checked or set have a caret (^) printed under them. The following is a representation of the Status Register with the L> and EQ bits set.

Status Register



The following table indicates the bits in the Status Register that may be affected by the various assembly language instructions.

Status Bits Affected by Instructions[1]

Mnemonic	L>	A>	EQ	C	OV	OP	X
A	X	X	X	X	X	-	-
AB	X	X	X	X	X	X	-
ABS	X	X	X	X	X	-	-
AI	X	X	X	X	X	-	-
ANDI	X	X	X	-	-	-	-
B	-	-	-	-	-	-	-
BL	-	-	-	-	-	-	-
BLWP	-	-	-	-	-	-	-
C	X	X	X	-	-	-	-
CB	X	X	X	-	-	X	-
CI	X	X	X	-	-	-	-
CLR	-	-	-	-	-	-	-
COC	-	-	X	-	-	-	-
CZC	-	-	X	-	-	-	-
DEC	X	X	X	X	X	-	-
DECT	X	X	X	X	X	-	-
DIV	-	-	-	-	X	-	-
INC	X	X	X	X	X	-	-
INCT	X	X	X	X	X	-	-
INV	X	X	X	-	-	-	-
JEQ	-	-	-	-	-	-	-
JGT	-	-	-	-	-	-	-
JH	-	-	-	-	-	-	-
JHE	-	-	-	-	-	-	-
JL	-	-	-	-	-	-	-
JLE	-	-	-	-	-	-	-
JLT	-	-	-	-	-	-	-
JMP	-	-	-	-	-	-	-
JNC	-	-	-	-	-	-	-
JNE	-	-	-	-	-	-	-
JNO	-	-	-	-	-	-	-
JOC	-	-	-	-	-	-	-

Mnemonic	L>	A>	EQ	C	OV	OP	X
JOP	-	-	-	-	-	-	-
LDCR	X	X	X	X	-	2	-
LI	X	X	X	-	-	-	-
LIMI	-	-	-	-	-	-	-
LWPI	-	-	-	-	-	-	-
MOV	X	X	X	-	-	-	-
MOVB	X	X	X	-	-	X	-
MPY	-	-	-	-	-	-	-
NEG	X	X	X	X	X	-	-
ORI	X	X	X	-	-	-	-
RTWP	X	X	X	X	X	X	X
S	X	X	X	X	X	-	-
SB	X	X	X	X	X	X	-
SBO	-	-	-	-	-	-	-
SBZ	-	-	-	-	-	-	-
SETO	-	-	-	-	-	-	-
SLA	X	X	X	X	X	-	-
SOC	X	X	X	-	-	-	-
SOCB	X	X	X	-	-	X	-
SRA	X	X	X	X	-	-	-
SRC	X	X	X	X	-	-	-
SRL	X	X	X	X	-	-	-
STCR	X	X	X	-	-	2	-
STST	-	-	-	-	-	-	-
STWP	-	-	-	-	-	-	-
SWPB	-	-	-	-	-	-	-
SZC	X	X	X	-	-	-	-
SZCB	X	X	X	-	-	X	-
TB	-	-	X	-	-	-	-
X	3	3	3	3	3	3	3
XOP	3	3	3	3	3	3	3
XOR	X	X	X	-	-	-	-

Notes:

[1]In addition to these instructions, the instructions CKOF, CKON, IDLE, LREX, and RSET are included for completeness. None affect any status bits or have any other useful effect on the Home Computer.

[2]When an LDCR or STCR instruction transfers eight or fewer bits, the OP bit is set or reset as in byte instructions. Otherwise, these instructions do not affect the OP bit.

[3]The X instruction does not affect any status bit. The instruction executed by the X instruction sets status bits normally. When an XOP instruction is implemented by software, the XOP bit is set, and the subroutine sets status bits normally.

3.1.3.1 Logical Greater Than--(L>)

The logical greater than bit is set when an unsigned number is compared with a smaller unsigned number. Note that the most significant bit represents 2^{15} in a word and 2^{7} in a byte rather than the sign of the number.

3.1.3.2 Arithmetic Greater Than--(A>)

The arithmetic greater than bit is set when a signed number is compared with a smaller signed number. The most significant bits of the words or bytes being compared represent the sign of the number, zero for positive or one for negative. For positive numbers, the remaining bits represent the binary value. For negative numbers, the remaining bits represent the two's complement of the binary value.

3.1.3.3 Equal--(EQ)

The equal bit is set when the two words or bytes being compared are equal. The significance of equality is the same whether the comparison is between unsigned binary numbers or two's complement numbers.

3.1.3.4 Carry--(C)

The carry bit is set by a carry of 1 from the most significant bit (sign bit) of a word or byte during arithmetic and shift operations. Thus the carry bit is used by shift operations to store the last bit shifted out of the Workspace Register being shifted.

3.1.3.5 Overflow--(OV)

The overflow bit is set when the result of an arithmetic operation is too large or too small to be represented correctly in two's complement representation.

In addition operations, the overflow bit is set when the most significant bits of the operands are equal and the most significant bit of the result is not equal to the most significant bit of the destination operand.

In subtraction operations, the overflow bit is set when the most significant bits of the operands are not equal and the most significant bit of the result is not equal to the most significant bit of the destination operand.

For a divide operation, the overflow bit is set when the most significant 16 bits of the dividend are greater than or equal to the divisor.

For an arithmetic left shift, the overflow bit is set if the most significant bit of the workspace register being shifted changes value.

For the absolute value and negate instructions, the overflow bit is set when the source operand is the maximum negative value (8000H).

3.1.3.6 Odd Parity--(OP)

In byte operations the odd parity bit is set when the parity of the result is odd and is reset when the parity is even. The parity of a byte is odd when the number of bits having values of one is odd. When the number of bits having values of one is even, the parity of the byte is even. The odd parity bit is equal to the least significant bit of the sum of the bits in the byte.

3.1.3.7 Extended Operation--(X)

The extended operation instruction (XOP) is available in some TI-99/4A computers. The only way to determine if your computer supports this instruction is to try it. (See Section 7.19.) Extended operation instructions permit a limited extension of the existing instruction set to include additional instructions. In the computer, these additional instructions are implemented by software routines.

When the program contains an XOP instruction that is software implemented, the computer locates the XOP Workspace Pointer (WP) and Program Counter (PC) words in the XOP reserved memory locations and loads the WP and PC. Then the computer transfers control to the XOP instruction set through a context switch. When the context switch is complete, the XOP workspace contains the calling routine's return data in Workspace Registers 13, 14, and 15.

The extended operation bit is set when the software implemented extended operation is initiated.

3.1.3.8 Interrupt Mask

The interrupt mask is status bits 12 through 15. Any device with a level number less than or equal to the value in the interrupt mask is permitted by the TMS9900 microprocessor to interrupt a running program. Thus if the interrupt mask has a value of 2 (binary 0010), any device with a level of 0, 1, or 2 may interrupt a running program. On the TI Home Computer, all interrupts are on level 2. Thus only values of 0 and 2 are useful.

Note: The p-System is not designed to allow interrupts, and assembly language programs that enable interrupts probably cannot return to the calling program or to the System.

3.2 OBJECT CODE FORMAT

The internal representation of the object code and data is dependent on bytes, words, and memory organization. The following sections describe the organization of these basics.

3.2.1 Byte Organization

A byte consists of eight bits. The bits can represent eight binary values or a single character of data. The bits can also represent a one-byte machine instruction or a number which is interpreted either as a signed two's-complement number in the range of -128 through 127 or an unsigned number in the range of 0 through 255.

3.2.2 Word Organization

A word consists of sixteen bits, or two adjacent bytes, in memory. A word can contain a one-word machine instruction, any combination of byte quantities, a number which can be interpreted either as a signed two's-complement number in the range of -32,768 through 32,767, or an unsigned number in the range of 0 through 65,535.

3.2.3 Memory Organization

The TI Home Computer is based on the TMS9900 microprocessor, which is word oriented and byte addressable. The instructions and data words are constrained to word boundaries. A word boundary is defined as an even byte address.

In the Assembler, data directives are defined such that they produce integral numbers of words. You are responsible for maintaining word alignment of instructions and data words. Failure to do so is flagged with an error message. Nonalignment occurs when a directive creates an odd number of data bytes.

The two bytes that make up a 16-bit word are termed the most-significant and least-significant byte, or MSB and LSB respectively. The computer treats the first byte as the MSB and the second byte as the LSB.

3.3 SOURCE CODE FORMAT

Source code must be in a specific format in order to be translated into object code. This format requires using acceptable characters, identifiers, character strings, constants, and expressions.

3.3.1 Character Set

The following characters are used to construct source code.

- Upper- and lower-case letters: A through Z and a through z
- Numerals: 0 through 9
- Special symbols: ! @ # $ % ^ & * ( ) < > ~ [ ] . , / ; : " ' + - = ? _
- Space character and tab character

3.3.2 Identifiers

Identifiers consist of an alphabetical character followed by a series of alphanumeric characters and/or underscore characters. Upper- and lower-case and the underscore character are not significant. This definition of identifiers is equivalent to the Standard Pascal definition. For example, all of the following identifiers are equivalent.

```
FormArray
FORM_ARRAY
formarray
```

Identifiers can be used in label and constant definitions, machine instructions, assembler directives, macro identifiers, and label and constant references.

Predefined identifiers are reserved by the Assembler as symbolic names for machine instructions and registers and cannot be used as names for labels, constants, or procedures. The dollar sign ($) is the location counter character. This is a character which, when used in an expression, represents the current value of the location counter in the program during the assembly process.

3.3.3 Character Strings

A character string is written as a series of ASCII characters surrounded by double quotes. A string can contain up to 80 characters, but cannot cross source lines. A double quote can be embedded in a character string by entering it twice; for example, "This contains ""embedded"" double quotes." The Assembler directive .ASCII requires a character string for its operand. Strings also have limited uses in expressions.

3.3.4 Constants

Numeric constants may be binary, decimal, hexadecimal, or octal. Character constants of up to two characters may be used. The radix (base) of an integer constant lacking a trailing radix character is set to the Assembler's current default radix. The initial default radix for the TI Home Computer is decimal (base 10).

3.3.4.1 Binary Constants

A binary integer constant is a series of bits or binary digits (0, 1) followed by the letter T. The range of values is 0 through 1111111111111111T for a word constant and 0 through 11111111T for a byte constant.

The following are examples of valid binary constants.

```
0T
01000100T
11101T
```

3.3.4.2 Decimal Constants

A decimal integer word constant is written as a series of numerals (0 through 9) followed by a period. Its range of values is -32,768 through 32,767 as a signed two's-complement number. As a byte constant, its range of values is -128 through 127 as a signed two's-complement number or 0 through 255 as an unsigned number.

The following examples show valid decimal constants.

```
001
256
-4096
```

3.3.4.3 Hexadecimal Constants

A hexadecimal integer word constant is written as a series of up to four significant hexadecimal numerals (0 through 9, A through F) followed by the letter H. The leading numeral of a hexadecimal constant must be a numeric character, so a dummy 0 (zero) must precede a value that starts with A through F. The range of values is 0 through FFFF.

The following examples show valid hexadecimal constants.

```
0AH
100H
0FFFEH        The leading 0 is required here.
```

Byte constants have the same syntax but can have at most two significant hexadecimal numerals with a range of 0 through FF.

3.3.4.4 Octal Constants

An octal integer word constant is written as a series of up to six significant octal numerals (0 through 7) followed by the letter Q. The range of values is 0 through 177777. Byte constants can have at most three significant octal numerals, with a range of 0 through 377.

The following are examples of valid octal constants.

```
17Q
457Q
177776Q
```

3.3.4.5 Character Constants

Character constants are special cases of character strings and can be used in expressions. The maximum length is two characters for a word constant and one character for a byte constant.

The following are examples of valid character constants.

"A"
"BC"
"YA"

An assembly-time constant is written as an identifier that has been assigned a constant value by the .EQU directive (see Section 13.2). The constant's value is completely determined at assembly time from the expression following the directive. Assembly-time constants must be defined before you refer to them.

3.3.5 Expressions

Expressions can be used as symbolic operands for machine instructions and Assembler directives. An expression can be any one of the following.

- A label, which might refer to a defined address or an address farther down in the source code (implying that the label is presently undefined), an externally referenced address, or an absolute address.
- A constant.
- A series of labels or constants separated by arithmetic or logical operators.
- The null expression, which evaluates to 0 (zero).

An expression containing more than one label is a valid expression under certain circumstances. In the following examples, R1, R2, and R3 are relocatable labels, and A1, A2, and A3 are absolute values.

Example	Description
R1-R2	Subtracting a relocatable value from another relocatable value yields an absolute value.
R1-R2+A1	Any number of absolute values may be added to an absolute value to obtain an absolute value.
R1+R2-R3	If the number of relocatable values added together is exactly one more than the number of relocatable values subtracted, the result is a relocatable value.
R1+A1+A2	Any number of absolute values may be added to a relocatable value to obtain a relocatable value.
A1/A2	An absolute value divided by another absolute value gives an absolute value.
A1*A2	An absolute value multiplied by another absolute value gives an absolute value.

It is illegal to add together two relocatable values, to multiply relocatable values, to multiply a relocatable value by an absolute value, or to negate a relocatable value.

In relocatable programs, absolute expressions cannot be used as the operands of instructions which require location-counter-relative address modes.

An expression can contain no more than one externally defined label, and the label's value must be added to the expression. An expression containing an external reference cannot contain a forward-referenced label, and the rest of the expression must be absolute.

An expression can contain no more than one forward-referenced identifier. A forward-referenced identifier is assumed to be a relocatable label defined farther down in the source code. Any other identifiers must be defined before they are used in an expression. An expression containing a forward-referenced label cannot also contain an externally defined label.

The following operators are available for use in expressions.

Unary operations:

+	Plus
-	Minus (two's-complement negation)
~	Logical not (one's-complement negation)

Binary operations:

+	Plus
-	Minus
^	Exclusive OR
*	Multiplication
/	Signed integer division (DIV)
//	Unsigned integer division (DIV)
%	Unsigned remainder division (MOD)
!	Bitwise OR
&	Bitwise AND

The following operators are available for use only with conditional assembly directives.

=	Equal
<>	Not equal

The symbols below can be used as alternatives to the single-character definitions presented above. Occurrences of these alternative definitions require at least one blank character both before and after them.

.OR = !
.AND = &
.NOT = ~
.XOR = ^
.MOD = %

The Assembler evaluates expressions from left to right; there is no operator precedence. All operations are performed on word quantities. Unary operators are available only with constants and absolute addresses. Angle brackets (< and >) must enclose subexpressions which contain embedded unary operators.

Angle brackets can also be used in expressions to override the left-to-right evaluation of operands. Subexpressions enclosed in angle brackets are completely evaluated before the rest of the expression is evaluated.

The following are examples of valid expressions. The default radix is decimal.

CNST+4	The sum of the value of identifier CNST and 4.
BELOW-2	The result of subtracting 2 from the value of identifier BELOW.
2-TIMER	The result of subtracting the value of identifier TIMER from 2. TIMER must be absolute.
3*2+MACRO	The product of 3 times 2 added to the value of the identifier MACRO.
BLBD+3*2	The sum of the identifier BLBD and 3, which is multiplied by 2. BLBD must be absolute.
650/2-PAST	The result of dividing 650 by 2 and subtracting the value of identifier PAST from the quotient. PAST must be absolute.
	Null expression: result is constant 0.
-4*12+<6/2>	The result of negative 4 times 12 added to 6 divided by 2. This evaluates to -45 (decimal).
85+2+<-5>	The sum of 85, 2, and negative 5. This evaluates to 82 (decimal).
0!1&<~0>	Zero or 1 and not zero. This evaluates to 1.
0 .OR 1 .AND <.NOT 0>	This is the same expression as above. It evaluates to 1.

3.4 SOURCE STATEMENT FORMAT

An assembly language source program consists of source statements which can contain machine instructions, Assembler directives, comments, or nothing (a blank line). Each source statement is defined as one line of a text file. Assembly language identifiers can be either upper-case or lower-case alphabetic characters. Source statements are divided into a label field, an op-code field, an operand field, and a comment field.

3.4.1 Label Field

The label field begins in the left-most character position of each source line. Macro identifiers and machine instructions must not appear in the start of the label field, but Assembler directives and comments can appear there.

The Assembler supports the use of both standard labels and local (reusable) labels. A standard label is an identifier that appears in the label field of a source statement. It can optionally be terminated by a colon (which is not used when referring to the label). As in Pascal, only the first eight characters of the label are important; the rest are ignored by the Assembler. Also, as in Pascal, the underscore character is not significant.

The following are examples of valid labels.

```
BIOS
L3456:      Referred to as "L3456"
THE_KIND
LONG_LABEL  The ninth character is ignored
```

A standard label is a symbolic name for a unique address or constant and can be declared only once in a source program. A label is optional for machine instructions and for many of the Assembler directives. A source statement consisting of only a label is a valid statement which has the effect of assigning the current value of the location counter to the label. This is equivalent to placing the label in the label field of the next source statement that generates object code. Labels defined in the label field of the .EQU directive (see Section 13.2) are assigned the value of the expression in the operand field.

Local labels are non-mnemonic labels which allow source statements to be labeled without taking up storage space in the symbol table. They can contribute to the cleanliness of program design by reserving the use of mnemonic label names for conceptually more important sections of code.

Local labels have a dollar sign ($) in the first character position, with the remaining characters being digits. As in regular labels, only the first eight digits are significant. The scope of a local label is limited to the lines of source statements between the declarations of consecutive standard labels. Thus, the jump to label $4 in the following example is illegal.

```
LABEL1
                    LI          R1,SOURCE
                    MOV         @LEN,R2
$3          MOVB    *R1+,R0
                    JEQ         $4      ; Illegal use of label.
                    DEC         R2
                    JNE         $3      ; Legal use of label.
LABEL2
                    SETO        R2
$4          ...
```

Up to 21 local labels can be defined between two standard labels. Upon encountering a standard label, the Assembler removes all existing local label definitions. Thus, all local label names must be redefined after that point. Local labels cannot be used in the label field of the .EQU directive (see Section 13.2).

3.4.2 Op-code Field

The op-code field begins with either the first nonblank character following the label field or the first nonblank character following the left-most character position when the label is omitted. The op-code field is terminated by one or more blanks. The op-code field contains an identifier which can be one of the following types.

- Machine instruction
- Assembler directive
- Macro call

3.4.3 Operand Field

The operand field begins with the first nonblank character following the op-code field and is terminated by an optional number of blanks. The operand field can contain as many expressions as are required by the preceding op-code.

3.4.4 Comment Field

The comment field, which can be preceded by an optional number of blanks, begins with a semicolon (;) and extends to the end of the source line. The comment field can contain any printable ASCII characters. The comment field is listed on assembled listings but has no other effect on the assembly process.

3.5 SOURCE FILE FORMAT

Assembly source files are generated with the UCSD P-System Editor, which is described in the Editor manual, and saved as files of type TEXT. A source file is constructed from assembly routines (procedures and functions) and global declarations.

3.5.1 Assembly Routines

A source file can contain more than one assembly routine. Each assembly routine ends when the following routine begins. Each routine in a source file is a separate entity and contains its own relocation information. Each assembled routine can be referred to individually by a Pascal host program during linking.

Assembly routines must begin with a .PROC, .FUNC, .RELPROC, or .RELFUNC directive. The last routine in the source file must be terminated by the .END directive. See Section 13.1 and the UCSD p-System Linker manual for a description of these directives.

At the end of each routine, the Assembler's symbol table is cleared of all but predefined and globally declared symbols, and the Location Counter (LC) is reset to zero.

3.5.2 Global Declarations

An assembly routine cannot directly access objects declared in another assembly routine, even if the routines are assembled in the same source file. However, it is occasionally desirable for a set of routines to share a common group of declarations. Therefore the Assembler allows global data declarations.

Any objects declared before the first occurrence of a .PROC or .FUNC directive in a source file can be referred to by all subsequent assembly routines. No code can be generated before the first procedure delimiting directive. Thus, the "global" objects are limited to the directives that do not generate code (.EQU, .REF, .DEF, .MACRO, .LIST, etc.).

3.5.3 Absolute Sections

You may find it necessary to access absolute addresses in memory, regardless of where an assembly routine is located in memory. For instance, a program may need to access Read Only Memory (ROM) routines. Absolute sections allow you to define labels and data space using the standard syntax and directives with the extra ability to specify absolute (non-relocatable) label addresses starting at any location in memory.

Absolute sections are initiated by the directive .ASECT (for absolute section) and terminated by the directive .PSECT (for program section, which is the default setting during assembly). When the .ASECT directive is encountered, the Absolute section Location Counter (ALC) becomes the current location counter. The .ORG directive can be used to set the ALC to any desired value. Label definitions are non-relocatable and are assigned the current value of the ALC. The data directives .WORD, .BLOCK, and .BYTE increment the ALC (instead of the regular LC).

Data directives in an absolute section cannot place initial values in the locations specified as they can when used in the program section. Thus, you can use absolute sections for constructing a template of label and memory address assignments.

The equate directive (.EQU) can be used in an absolute section, but the labels can only be equated to absolute expressions. The only other directives allowed within an absolute section are .LIST, .NOLIST, .END, and the conditional assembly directives.

Absolute sections can appear as global objects.

The following is a simple example of an absolute section.

```
            .ASECT              ; Start absolute section.
            .ORG    8374H       ; Set ALC to 8374 hexadecimal.
                                ; Note that no data values are
                                assigned.
                                ; Label assignments below.
KBD         .BYTE               ; Keyboard select.
KEYVAL      .BYTE               ; Key selected.
JOYY        .BYTE               ; Joystick y-position.
JOYX        .BYTE               ; Joystick x-position.
            .BLOCK 4
STATUS      .BYTE               ; Keyscan status return.
            .PSECT              ; End absolute section.
```

SECTION 4: ADDRESSING MODES

This section describes the addressing modes used in assembly language. Examples of programming in each addressing mode are included.

4.1 GENERAL ADDRESSING MODES

A source operand is the number, address, string, etc., which is to be manipulated or operated upon. A destination operand is the address where the result of the performed manipulation is stored. Instructions that specify a general address for a source or destination operand may be in one of five addressing modes. These addressing modes and their uses are discussed in this section.

The following lists the T-field value, which indicates the type of addressing mode (see Section 5), and gives an example for each of the addressing modes.

Addressing Modes

Addressing Mode	T-field value	Example
Workspace Register	00	5
Workspace Register Indirect	01	*7
Symbolic Memory[1, 2]	10	@LABEL
Indexed Memory[1, 3]	10	@LABEL(5)
Workspace Register Indirect Auto-increment	11	*7+

Notes:

[1]The instruction requires an additional word for each T-field value of 10. The additional word contains a memory address.

[2]The four-bit field immediately following the T-field value of 10_2, called the S (for a source operand) or D (for a destination operand) field, is set to zero by the Assembler.

[3]The T-field value of 10_2 indicates both symbolic and indexed memory addressing modes. If the four-bit field which follows it contains a zero value, it is a symbolic addressing mode. If it is non-zero, it is an indexed addressing mode, and the non-zero value is the number of the index register. Therefore, Workspace Register 0 cannot be used for indexing.

4.1.1 Workspace Register Addressing

Workspace Register addressing specifies the Workspace Register that contains the operand. A Workspace Register address is specified by a value of 0 through 15, optionally preceded with an "R". For example, Workspace Register 8 may be referred to as "8" or "R8".

Examples:

```
MOV     R4,R8       ;Copies the contents of Workspace Register 4 into
                    Workspace Register 8.

MOV     4,8         ;Same as the preceding example.

COC     R15,R10     ;Compares the bits of Workspace Register 10 that
                    correspond to the one bits in Workspace Register
                    15 to one.
```

4.1.2 Workspace Register Indirect Addressing

Workspace Register indirect addressing specifies a Workspace Register that contains the address of the operand. An indirect Workspace Register address is preceded by an asterisk (*).

Examples:

```
A       *R7,*R2     ;Adds the contents of the word at the address in
                    Workspace Register 7 to the contents of the word
                    at the address in Workspace Register 2 and places
                    the sum in the word at the address in Workspace
                    Register 2.

MOV     *R7,R0      ;Copies the contents of the word at the address
                    given in Workspace Register 7 into Workspace
                    Register 0.
```

4.1.3 Workspace Register Indirect Auto-Increment Addressing

Workspace Register indirect auto-increment addressing specifies a Workspace Register that contains the address of the operand. After the address is obtained from the Workspace Register, the Workspace Register is incremented by 1 for a byte instruction or by 2 for a word instruction. A Workspace Register auto-increment address is preceded by an asterisk and followed by a plus sign (+).

Examples:

```
S       *R3+,R2         ;Subtracts the contents of the word at the address
                        in Workspace Register 3 from the contents of
                        Workspace Register 2, places the result in
                        Workspace Register 2, and increments the address
                        in Workspace Register 3 by two.

CB      R5,*R6+         ;Compares the first byte of the contents of
                        Workspace Register 5 with the contents of the byte
                        at the address in Workspace Register 6 and
                        increments the address in Workspace Register 6 by
                        one.
```

4.1.4 Symbolic Memory Addressing

Symbolic memory addressing specifies the memory address that contains the operand. A symbolic memory address is preceded by an "at" sign (@).

Examples:

```
S       @FIX1,@LIST4    ;Subtracts the contents of the word at location
                        FIX1 from the contents of the word at location
                        LIST4 and places the difference in the word at
                        location LIST4.

C       R0,@STORE       ;Compares the contents of Workspace Register 0
                        with the contents of the word at location STORE.

MOV     @12,@7CH        ;Copies the word at address 000CH into location
                        007CH.
```

4.1.5 Indexed Memory Addressing

Indexed memory addressing specifies the memory address that contains the operand. The address is the sum of the contents of a Workspace Register and a symbolic address. An indexed memory address is preceded by an "at" sign (@) and followed by a term enclosed in parentheses. The Workspace Register specified by the term within the parentheses is the index register. Workspace Register 0 may not be specified as an index register.

Examples:

```
A       @2(R7),R6       ;Adds the contents of the word found at the
                        address computed by adding 2 to the contents of
                        Workspace Register 7 to the contents of Workspace
                        Register 6 and places the sum in Workspace
                        Register 6.

MOV     R7,@LIST4-6(R5) ;Copies the contents of Workspace Register 7 into
                        a word of memory. The address of the word of
                        memory is the sum of the contents of Workspace
                        Register 5 and the value of symbol LIST4 minus 6.
```

4.2 PROGRAM COUNTER RELATIVE ADDRESSING

Program Counter relative addressing is used only by jump instructions. A Program Counter relative address is written as an expression that corresponds to an address at a word boundary. The Assembler evaluates the expression and subtracts the sum of the current location plus two. One-half of the difference is the value placed in the object code. This value must be in the range of -128 through +127. When the instruction is in relocatable code (that is, when the Location Counter is relocatable), the relocation type of the evaluated expression must match the relocation type of the current Location Counter. When the instruction is in absolute code, the expression must be absolute.

Example:

```
JMP     THERE           ;Jumps unconditionally to location THERE.
```

4.3 CRU BIT ADDRESSING

The CRU, or Communications Register Unit, is a command-driven bit-addressable I/O interface. An instruction can set, reset, or test any bit in the CRU array or move data between the memory and CRU data fields. The CRU software base address is contained in the 16 bits of Workspace Register 12. From the CRU software base address, the processor is able to determine the CRU hardware base address and the resulting CRU bit address.

The CRU bit instructions use a well-defined expression that represents a displacement from the CRU base address (bits 3 through 14). The displacement, in the range of -128 through +127, is added to the base address in Workspace Register 12. See Section 9 for more information.

Example:

```
SBO     8           ;Sets CRU bit to one at the CRU address 8 greater
                    than the CRU base address.
```

4.4 IMMEDIATE ADDRESSING

Immediate instructions use the contents of the word following the instruction word as the operand of the instruction. The immediate value is an expression, and the Assembler places its value in the word following the instruction. Immediate instructions that require two operands have a Workspace Register address preceding the immediate value.

Example:

```
LI      R5,1000H        ;Places 1000H into Workspace Register 5.
```

4.5 ADDRESSING SUMMARY

The following table shows the addressing mode required for each instruction of the Assembler instruction set. The first column lists the instruction mnemonic. The second and third columns specify the required address, listed below.

G - General address:
- Workspace Register address
- Indirect Workspace Register address
- Symbolic memory address
- Indexed memory address (R0 not allowed)
- Indirect Workspace Register auto-increment address

WR - Workspace Register address
PC - Program counter relative address
CRU - CRU bit address
I - Immediate value
* - The address into which the result is placed when two operands are required

Instruction Addressing

Mnemonic	First Operand	Second Operand	Mnemonic	First Operand	Second Operand
A	G	G*	LDCR	G	Note 1
AB	G	G*	LI	WR*	I
ABS	G	-	LIMI	I	-
AI	WR*	I	LREX	-	-
ANDI	WR*	I	LWPI	I	-
B	G	-	MOV	G	G*
BL	G	-	MOVB	G	G*
BLWP	G	-	MPY	G	WR*
C	G	G	NEG	G	-
CB	G	G	ORI	WR*	I
CI	WR	I	RSET	-	-
CKOF	-	-	RTWP	-	-
CKON	-	-	S	G	G*
CLR	G	-	SB	G	G*
COC	G	WR	SBO	CRU	-
CZC	G	WR	SBZ	CRU	-
DEC	G	-	SETO	G	-
DECT	G	-	SLA	WR*	Note 2
DIV	G	WR*	SOC	G	G*
IDLE	-	-	SOCB	G	G*
INC	G	-	SRA	WR*	Note 2
INCT	G	-	SRC	WR*	Note 2
INV	G	-	SRL	WR*	Note 2

Mnemonic	First Operand	Second Operand
JEQ	PC	-
JGT	PC	-
JH	PC	-
JHE	PC	-
JL	PC	-
JLE	PC	-
JLT	PC	-
JMP	PC	-
JNC	PC	-
JNE	PC	-
JNO	PC	-
JOC	PC	-
JOP	PC	-

Mnemonic	First Operand	Second Operand
STCR	G*	Note 1
STST	WR	-
STWP	WR	-
SWPB	G	-
SZC	G	G*
SZCB	G	G*
TB	CRU	-
X	G	-
XOP	G	Note 3
XOR	G	WR*

Notes:

[1]The second operand is the number of bits to be transferred, from 0 through 15, with 0 meaning 16 bits.

[2]The second operand is the shift count, from 0 through 15. 0 indicates that the count is in bits 12 through 15 of Workspace Register 0. When the count is 0 and bits 12 through 15 of Workspace Register 0 equal 0, the count is 16.

[3]The second operand specifies the extended operation, from 0 through 15. The disposition of the result may or may not be in the first operand address, as determined by you.

SECTION 5: INSTRUCTION FORMATS

An assembler instruction occupies one word (16 bits) of memory. Each word is divided into appropriately sized bit fields which are arranged in one of nine formats. These formats are discussed below and are referred to in the discussions of the instructions in the following sections. You must clearly understand addressing modes, as described in Section 4, before reading this section.

Each format contains one or more of the following bit fields.

Op-Code - Machine operation code.

B - Byte indicator: 1 for byte instructions, 0 for word instructions.

Td - Type of addressing mode of the destination operand.

D - Destination operand.

Ts - Type of addressing mode of the source operand.

S - Source operand.

DISP - Displacement value (signed).

C - Count (bit count).

W - Workspace register.

5.1 FORMAT I -- TWO GENERAL ADDRESS INSTRUCTIONS

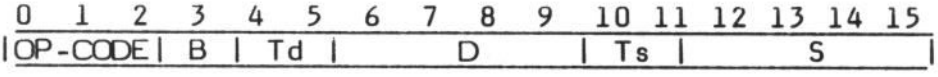
The operand field of Format I instructions contains two general addresses separated by a comma. The first address is the source address and the second is the destination address. The Format I mnemonic operation codes are listed below and discussed in subsequent sections.

A	Add words
AB	Add Bytes
C	Compare words
CB	Compare Bytes
MOV	MOVe word
MOVB	MOVe Byte
S	Subtract words
SB	Subtract Bytes
SOC	Set Ones Corresponding
SOCB	Set Ones Corresponding, Byte
SZC	Set Zeros Corresponding
SZCB	Set Zeros Corresponding, Byte

Example:

```
SUM     A       @LABEL1,*R7     ;Adds the contents of the word at
                                location LABEL1 to the contents of
                                the word at the address in
                                Workspace Register 7 and places the
                                sum in the word at the address in
                                Workspace Register 7.  SUM is the
                                location of the instruction.
```

Format I instructions are assembled as follows.



When either Ts or Td (but not both) equal binary 10, the instruction occupies two words of memory. The second word contains a memory address used with S or D to develop the effective address. When both Ts and Td equal binary 10, the instruction occupies three words of memory. The second word contains the memory address of the source operand, and the third word contains the memory address of the destination operand.

5.2 FORMAT II -- JUMP INSTRUCTIONS

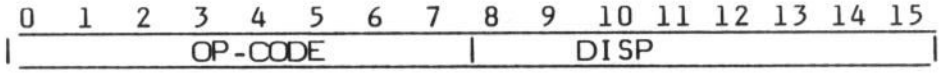
Format II instructions use Program Counter (PC) relative addresses coded as expressions corresponding to instruction locations on word boundaries. The Format II jump mnemonic operation codes are listed below and discussed in subsequent sections See Section 5.2.1 for a discussion of the Format II CRU bit I/O instructions.

JEQ	Jump if EQual
JGT	Jump if Greater Than
JH	Jump if logical High
JHE	Jump if High or Equal
JL	Jump if logical Low
JLE	Jump if Low or Equal
JLT	Jump if Less Than
JMP	unconditional JuMP
JNC	Jump if No Carry
JNE	Jump if Not Equal
JNO	Jump if No Overflow
JOC	Jump On Carry
JOP	Jump if Odd Parity

Example:

```
NOW     JMP     BEGIN     ;Jumps unconditionally to the instruction at
                          location BEGIN. The address of location BEGIN
                          must not be greater than the address of location
                          NOW by more than 128 words, nor less than the
                          address of location NOW by more than 127
                          words.
```

Format II instructions are assembled as follows.



The signed displacement value is shifted one bit position to the left and added to the contents of the Program Counter after the Program Counter has been incremented to the address of the following instruction. In other words, it is a displacement in words from the instruction address plus two.

5.2.1 Format II -- Bit I/O Instructions

In addition to jump instructions, the CRU bit I/O instructions also follow Format II. The operand field of Format II CRU bit I/O instructions contains a well-defined expression which evaluates to a CRU bit address, relative to the contents of Workspace Register 12. The Format II CRU bit I/O instructions are listed below and discussed in subsequent sections. See Section 5.2 for a discussion of the Format II jump instructions.

SBO	Set Bit to logic One
SBZ	Set Bit to logic Zero
TB	Test Bit

Example:

```
SBO     5       ;Sets a CRU bit to one.
```

5.3 FORMAT III -- LOGICAL INSTRUCTIONS

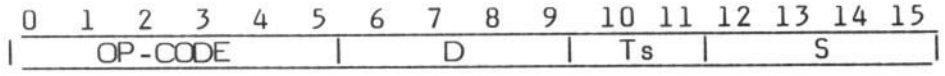
The operand field of Format III instructions contains a general address followed by a comma and a Workspace Register address. The general address is the source address. The Workspace Register address is the destination address. The Format III mnemonic operation codes are listed below and discussed in subsequent sections.

COC	Compare Ones Corresponding
CZC	Compare Zeros Corresponding
XOR	eXclusive OR

Example:

```
COMP    XOR     @LABEL8(R3),R5      ;Performs an exclusive OR operation
                                    on the contents of a memory word
                                    and the contents of Workspace
                                    Register 5 and places the result in
                                    Workspace Register 5. The address
                                    of the memory word is the sum of
                                    the contents of Workspace Register
                                    3 and the value of the symbol
                                    LABEL8.
```

Format III instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address of the source operand.

5.4 FORMAT IV -- CRU MULTI-BIT INSTRUCTIONS

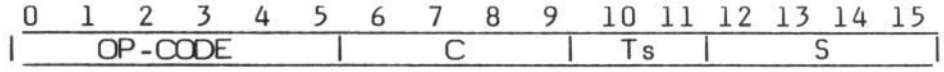
The operand field of Format IV instructions contains a general address followed by a comma and a well-defined expression. The general address is the memory address from which or into which bits are transferred. The CRU address for the transfer is the contents of bits 3 through 14 of Workspace Register 12. The well-defined expression is the number of bits to be transferred and must have a value of 0 through 15. A 0 value specifies a 16 bit transfer. For eight or fewer bits the general address is a byte address. For nine or more bits the general address is a word address. The Format IV mnemonic operation codes are listed below and discussed in subsequent sections.

LDCR	LoaD CRU
STCR	STore CRU

Example:

```
LDCR    *R6+,8  ;Places eight bits from the byte of memory at the address in
                 Workspace Register 6 into eight consecutive CRU lines and
                 increments Workspace Register 6 by 1.
```

Format IV instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

5.5 FORMAT V -- REGISTER SHIFT INSTRUCTIONS

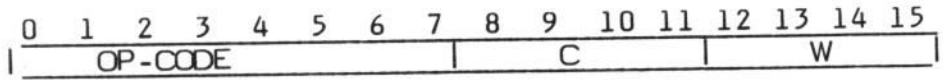
The operand field of Format V instructions contains a Workspace Register address followed by a comma and a well-defined expression. The contents of the Workspace Register are shifted a number of bit positions specified by the well-defined expression. When the term equals zero, the shift count must be placed in bits 12-15 of Workspace Register 0. The Format V mnemonic operation codes are listed below and discussed in subsequent sections.

SLA	Shift Left Arithmetic
SRA	Shift Right Arithmetic
SRC	Shift Right Circular
SRL	Shift Right Logical

Example:

```
SLA     R6,4    ;Shifts the contents of Workspace Register 6 to the left 4 bit
                 positions and replaces the vacated bits with zeros.
```

Format V instructions are assembled as follows.



5.6 FORMAT VI -- SINGLE ADDRESS INSTRUCTIONS

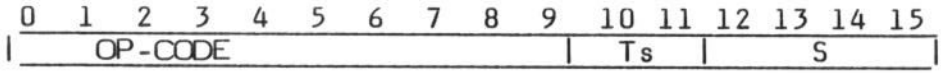
The operand field of Format VI instructions contains a general address. The Format VI mnemonic operation codes are listed below and discussed in subsequent sections.

ABS	ABSolute value
B	Branch
BL	Branch and Link
BLWP	Branch and Load Workspace Pointer
CLR	CLeaR
DEC	DECrement
DECT	DECrement by Two
INC	INCrement
INCT	INCrement by Two
INV	INVert
NEG	NEGate
SETO	SEt To One
SWPB	SWaP Bytes
X	eXecute

Example:

```
CNT     INC     R7      ;Adds one to the contents of Workspace Register
                        7 and places the sum in Workspace Register 7.
                        CNT is the location into which the instruction is
                        placed.
```

Format VI instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address of the source operand.

5.7 FORMAT VII -- CONTROL INSTRUCTIONS

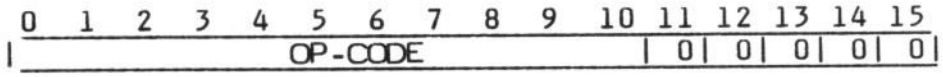
Format VII instructions require no operand field. The Format VII mnemonic operation codes are listed below and discussed in subsequent sections. All but the last instruction have no effect on the TI Home Computer.

CKOF	ClocK OFf
CKON	ClocK ON
IDLE	IDLE
LREX	Load or REstart eXecution
RSET	ReSET
RTWP	ReTurn with Workspace Pointer

Example:

```
RTWP    ;Returns control to the calling program and restores the context of the
        calling program by placing the contents of Workspace Registers 13, 14,
        and 15 into the Workspace Pointer Register, the Program Counter, and
        the Status Register.
```

Format VII instructions are assembled as follows.



The op-code field contains 11 bits that define the machine operation. The five least significant bits are zeros.

5.8 FORMAT VIII -- IMMEDIATE INSTRUCTIONS

The operand field of Format VIII instructions contains a Workspace Register address followed by a comma and an expression. The Workspace Register is the destination address, and the expression is the immediate operand. The Format VIII mnemonic operation codes are listed below and discussed in subsequent sections.

AI	Add Immediate
ANDI	AND Immediate
CI	Compare Immediate
LI	Load Immediate
ORI	OR Immediate

There are two additional Format VIII instructions that require only an expression in the operand field. The expression is the immediate operand. The destination is implied in the name of the instruction. These instructions are listed here.

LIMI	Load Interupt Mask Immediate
LWPI	Load Workspace Pointer Immediate

Another modification of Format VIII requires only a Workspace Register address in the operand field. The Workspace Register address is the destination. The source is implied in the name of the instruction. The following mnemonic operation codes use this modified Format VIII.

STST	STore STatus
STWP	STore Workspace Pointer

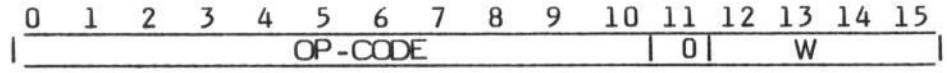
Examples:

```
ANDI    R4,000FH  ;Performs an AND operation on the contents of Workspace
                  Register 4 and immediate operand 000FH.

LWPI    WRK1      ;Places the address defined for the symbol WRK1 into the
                  Workspace Pointer Register.

STWP    R4        ;Places the contents of the Workspace Pointer Register into
                  Workspace Register 4.
```

Format VIII instructions are assembled as follows.



A zero bit separates the two fields. The instructions that have no Workspace Register operand place zeros in the W field. The instructions that have immediate operands place the operands in the word following the word that contains the op-code, i.e., these instructions occupy two words each.

5.9 FORMAT IX -- EXTENDED OPERATION INSTRUCTION

The extended operation instruction can be used on some TI Home Computers. See Section 7.19 for more information.

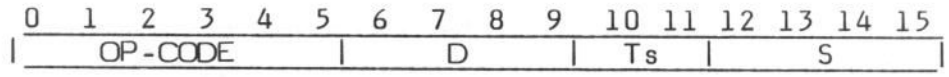
The operand field of the Format IX extended operation instruction contains a general address and a well-defined expression. The general address is the address of the operand for the extended operation. The term specifies the extended operation to be performed and must be in the range of 0 through 15. The Format IX mnemonic operation code is listed below and discussed in subsequent sections. See Section 5.9.1 for a discussion of the Format IX multiply and divide instructions.

XOP	eXtended OPeration

Example:

```
XOP     @LABEL(R4),12     ;Performs extended operation 12 using the
                          address computed by adding the value of symbol
                          LABEL to the contents of Workspace Register 4.
```

Format IX instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

5.9.1 Format IX -- Multiply and Divide Instructions

The operand field of Format IX multiply and divide instructions contains a general address followed by a comma and a Workspace Register address. The general address is the address of the multiplier or divisor, and the Workspace Register address is the address of the Workspace Register that contains the multiplicand or dividend. The Workspace Register address is also the address of the first of two Workspace Registers to contain the result. The Format IX multiply and divide instructions are listed below and discussed in subsequent sections. See Section 5.9 for a discussion of the Format IX extended operation instruction.

MPY	MultiPlY
DIV	DIVide

Example:

```
MPY     @ACC,R9         ;Multiplies the contents of Workspace Register 9
                        by the contents of the word at location ACC
                        and places the product in Workspace Registers 9
                        and 10, with the 16 least significant bits of the
                        product in Workspace Register 10.
```

Multiply and divide instructions are assembled in the same format as shown in Section 5.9, except that the D field contains the Workspace Register operand.

SECTION 6: ARITHMETIC INSTRUCTIONS

The following arithmetic instructions are described in this section.

Instruction	Mnemonic	Section
Add words	A	6.1
Add Bytes	AB	6.2
ABSolute value	ABS	6.3
Add Immediate	AI	6.4
DECrement	DEC	6.5
DECrement by Two	DECT	6.6
DIVide	DIV	6.7
INCrement	INC	6.8
INCrement by Two	INCT	6.9
MultiPlY	MPY	6.10
NEGate	NEG	6.11
Subtract words	S	6.12
Subtract Bytes	SB	6.13

Examples are given in Section 6.14.

Each instruction consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

ARITHMETIC INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field. Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

6.1 ADD WORDS--A

Op-code: A000 (Format I)

Syntax definition:

[<label>] b A b <gas>,<gad> b [<comment>]

Example:

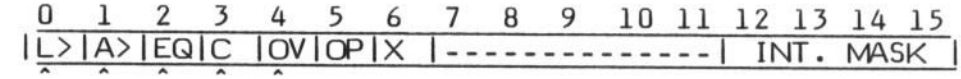
```
LABEL    A       @ADR1(R2),@ADR2(R3)  ;Adds the word at the address found
                                      by adding ADR1 to the contents of
                                      Workspace Register 2 to the word at
                                      the address found by adding ADR2
                                      to the contents of Workspace
                                      Register 3 and puts the result in the
                                      word at the second address.
```

Definition:

Adds a copy of the source operand (word) to the destination operand (word) and replaces the destination operand with the sum. The computer compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gas) + (gad) => (gad)

Application notes:

The A instruction adds both signed and unsigned integer words. For example, if the address labeled TABLE contains 3124H and Workspace Register 5 contains 8H, the instruction

```
A       R5,@TABLE
```

results in the contents of TABLE changing to 312CH and the contents of Workspace Register 5 not changing. The logical and arithmetic greater than status bits are set and the equal, carry, and overflow status bits are reset.

6.2 ADD BYTES--AB

Op-code: B000 (Format I)

Syntax definition:

[<label>] b AB b <gas>,<gad> b [<comment>]

Example:

```
LABEL   AB      R3,R2       ;Adds the left byte of Workspace Register 3 to
                            the left byte in Workspace Register 2 and places
                            the result in the left byte of Workspace Register
                            2.
```

Definition:

Adds a copy of the source operand (byte) to the destination operand (byte) and replaces the destination operand with the sum. When the source or destination operand is addressed in the Workspace Register mode, only the leftmost byte (bits 0 through 7) of the addressed Workspace Register is used. The computer compares the sum to zero and sets/resets the status bits to indicate the results of the comparison. When there is a carry of the most significant bit of the byte, the carry status bit is set. When there is an overflow, the overflow status bit is set. The odd parity bit is set when the bits in the sum (destination operand) establish odd parity and is reset when the bits in the sum establish even parity.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, overflow, and odd parity.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^  ^  ^
```

Execution results:

(gas) + (gad) => (gad)

Application notes:

The AB instruction is used to add signed or unsigned integer bytes. For example, if Workspace Register 3 contains 7400H, memory word 2122H contains 0F318H and Workspace Register 2 contains 2123H, the instruction

```
AB      R3,*R2+
```

changes the contents of memory word 2122H to 0F38CH because 74H (the value in Workspace Register 3) plus 23H (the value in memory byte 2123H) is 8CH. The left byte of memory word 2122H is unchanged. The contents of Workspace Register 2 are changed to 2124H, while the contents of Workspace Register 3 remain unchanged. The logical greater than, overflow, and odd parity status bits are set, while the arithmetic greater than, equal, and carry status bits are reset.

6.3 ABSOLUTE VALUE--ABS

Op-code: 0740 (Format IV)

Syntax definition:

[<label>] b ABS b <gas> b [<comment>]

Example:

```
LABEL    ABS     *2          ;Replaces the contents of the word starting at
                             the address in Workspace Register 2 with its
                             absolute value.
```

Definition:

Computes the absolute value of the source operand and replaces the source operand with the result. The absolute value is the two's complement of the source operand when the sign bit (bit zero) is equal to one. When the sign bit is equal to zero, the source operand is unchanged. The computer compares the original source operand to zero and sets/resets the status bits to indicate the results of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^     ^
```

Execution results:

(gas) => (gas)

Application notes:

The ABS instruction is useful for taking the absolute value of an operand. For example, if the third word in array LIST contains the value 0FF3CH and Workspace Register 7 contains the value 4H, the instruction

```
        ABS     @LIST(R7)
```

changes the contents of the third word (bytes 4 and 5) in array LIST to 00C4H. The logical greater than status bit is set, while the arithmetic greater than and equal status bits are reset.

6.4 ADD IMMEDIATE--AI

Op-code: 0220 (Format III)

Syntax definition:

[<label>] b AI b <wa>,<iop> b [<comment>]

Example:

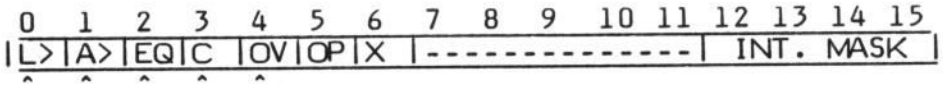
```
LABEL    AI      R2,7        ;Adds 7 to the contents of Workspace Register
                             2.
```

Definition:

Adds a copy of the immediate operand (the contents of the word following the instruction word in memory) to the contents of the Workspace Register specified in the wa field and replaces the contents of the Workspace Register with the results. The computer compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(wa) + iop => (wa)

Application notes:

The AI instruction adds an immediate value to the contents of a Workspace Register. For example, if Workspace Register 6 contains a zero, the instruction

```
AI      R6,12
```

changes the contents of Workspace Register 6 to 12. The logical greater than and arithmetic greater than status bits are set, while the equal, carry, and overflow status bits are reset.

6.5 DECREMENT--DEC

Op-code: 0600 (Format IV)

Syntax definition:

[<label>] b DEC b <gas> b [<comments>]

Example:

```
LABEL   DEC     R2          ;Decrements the contents of Workspace Register
                            2 by 1.
```

Definition:

Subtracts a value of one from the source operand and replaces the source operand with the result. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^  ^
```

Execution results:

(gas) - 1 => (gas)

Application notes:

The DEC instruction subtracts a value of one from any addressable operand. The DEC instruction is also useful in counting and indexing byte arrays. For example, if COUNT contains a value of 1, the instruction

```
DEC     @COUNT
```

results in a value of zero in location COUNT and sets the equal and carry status bits while resetting the logical greater than, arithmetic greater than, and overflow status bits. The carry bit is always set except on transition from zero to minus one.

6.6 DECREMENT BY TWO--DECT

Op-code: 0640 (Format IV)

Syntax definitions:

[<label>] b DECT b <gas> b [<comment>]

Example:

```
LABEL    DECT    @ADDR       ;Decrements the contents of ADDR by 2.
```

Definition:

Subtracts two from the source operand and replaces the source operand with the result. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.

```
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
|L>|A>|EQ|C  |OV|OP|X  |--------------|  INT. MASK  |
 ^   ^   ^   ^   ^
```

Execution results:

(gas) - 2 => (gas)

Application notes:

The DECT instruction is useful in counting and indexing word arrays. Also, the DECT instruction enables you to subtract a value of two from any addressable operand. For example, if Workspace Register PRT, which has been equated to 3, contains a value of 2C10H, the instruction

```
        DECT    PRT
```

changes the contents of Workspace Register 3 to 2C0EH. The logical greater than, arithmetic greater than and carry status bits are set, while the equal and overflow status bits are reset.

6.7 DIVIDE--DIV

Op-code: 3C00 (Format IX)

Syntax definition:

[<label>] b DIV b <gas>,<wad> b [<comment>]

Example:

```
LABEL       DIV   @ADR(R2),R3     ;Divides the contents of the words in
                                  Workspace Register 3 and Workspace
                                  Register 4 by the contents of the
                                  memory location pointed to by ADR
                                  plus Workspace Register 2 and stores
                                  the integer result in Workspace
                                  Register 3 with the remainder in
                                  Workspace Register 4.
```

Definition:

Divides the destination operand (a consecutive two-word area of workspace) by a copy of the source operand (one word), using unsigned integer rules. Places the integer quotient in the first of the two-word destination operand area and places the remainder in the second word of that same area. This division is graphically represented as follows.

Destination Operand Workspace Registers:

```
 Workspace Register(n)   |Workspace Register(n+1)
0                    15  |0                    15
<-------------------->   |<-------------------->
  Resulting Quotient       Resulting Remainder
<-------------------------------------------->
                Dividend
```

Source operand:

```
Addressable Memory
0                 15
<---------------->
     Divisor
```

The first of the destination operand Workspace Registers, shown above, is

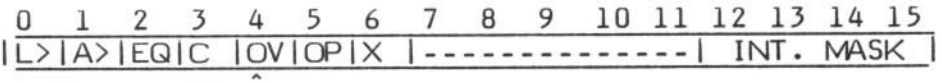
addressed by the contents of the D field. The dividend is right justified in this 2-word area. When the division is complete, the quotient (result) is placed in the first Workspace Register of the destination operand (represented by n) and the remainder is placed in the second word of the destination operand (represented by n+1).

When the source operand is greater than the first word of the destination operand, normal division occurs. If the source operand is less than or equal to the first word of the destination operand, normal division results in a quotient that cannot be represented in a 16-bit word. In this case, the computer sets the overflow status bit, leaves the destination operand unchanged, and cancels the division operation.

If the destination operand is specified as Workspace Register 15, the first word of the destination operand is Workspace Register 15 and the second word of the destination operand is the word in memory immediately following the workspace area.

Status bits affected:

Overflow



Execution results:

(wad and wad +1) divided by (gas) => (wad) and (wad) + 1
The quotient is placed in wad and the remainder is placed in wad + 1.

Application notes:

The DIV instruction performs a division. For example, if Workspace Register 2 contains a zero and Workspace Register 3 contains 12, and the contents of LOC is 5, the instruction

```
DIV     @LOC,2
```

results in 2 in Workspace Register 2 and 2 (the remainder) in Workspace Register 3. The overflow status bit is reset. If Workspace Register 2 contained the value 5, the value contained in the two-word destination operand equals 327,692 and division by the value 5 results in a quotient of 65,538, which cannot be represented in a 16-bit word. This attempted division sets the overflow status bit and the computer cancels the operation.

6.8 INCREMENT--INC

Op-code: 0580 (Format VI)

Syntax definition:

[<label>] b INC b <gas> b [<comment>]

Example:

```
LABEL    INC     COUNT     ;Increments the contents of the address pointed
                           to by COUNT by 1.
```

Definition:

Adds one to the source operand and replaces the source operand with the result. The computer compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^  ^
```

Execution results:

(gas) + 1 => (gas)

Application notes:

The INC instruction may be used to count and index byte arrays, add a value of one to an addressable memory location, or set flags. For example, if COUNT contains a zero, the instruction

```
INC     @COUNT
```

places a 1 in COUNT and sets the logical greater than and arithmetic greater than status bits, while the equal, carry, and overflow status bits are reset.

6.9 INCREMENT BY TWO--INCT

Op-code: 05C0 (Format VI)

Syntax definition:

[<label>] b INCT b <gas> b [<comment>]

Example:

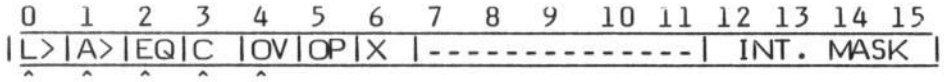
```
LABEL    INCT    R3          ;Increments the contents of Workspace Register
                             3 by 2.
```

Definition:

Adds a value of two to the source operand and replaces the source operand with the sum. The computer compares the sum to zero and sets/resets the status bit to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gas) + 2 => (gas)

Application notes:

The INCT instruction may be used to count and index word arrays and add the value of two to an addressable memory location. For example, if Workspace Register 5 contains the address (2100H) of the fifteenth word of an array, the instruction

```
INCT    5
```

changes Workspace Register 5 to 2102H, which points to the sixteenth word of the array. The logical greater than and arithmetic greater than status bits are set, while the equal, carry, and overflow status bits are reset.

6.10 MULTIPLY--MPY

Op-code: 3800 (Format IX)

Syntax definition:

[<label>] b MPY b <gas>,<wad> b [<comment>]

Example:

```
LABEL   MPY    @ADDR,R3  ;Multiplies the contents of Workspace Register 3
                         by the value of ADDR.  The result is right
                         justified in the 32 bits of Workspace Register 3
                         and Workspace Register 4.
```

Definition:

Multiplies the first word in the destination operand (a consecutive 2-word area in workspace) by a copy of the source operand and replaces the 2-word destination operand with the result. The multiplication operation may be graphically represented as follows.

Destination operand Workspace Registers:

```
 Workspace Register(n)   |Workspace Register(n+1)
0                    15  |0                    15
<-------------------->
                  Multiplicand
<------------------------------------>
                    Product
```

Source operand:

```
                    Addressable Memory
                    0                15
                    <---------------->
                        Multiplier
```

The first word of the destination operand, shown on the previous page, is addressed by the contents of the D field. This word contains the multiplicand (unsigned value of 16 bits) right-justified in the Workspace Register (represented by workspace n above). The 16-bit, unsigned multiplier is located in the source operand. When the multiply operation is complete, the product appears right-justified in the entire 2-word area addressed by the destination field as a 32-bit unsigned value. The maximum value of either input operand is 65,535 and the maximum value of the unsigned product is $65{,}535^2$.

If the destination operand is specified as Workspace Register 15, the first word of the destination operand is Workspace Register 15 and the second word of the destination operand is the memory word immediately following the workspace memory area.

Status bits affected:

None

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(gas) * (wad) = (wad) and (wad)+1

The product (32-bit magnitude) is placed in wad and wad + 1, with the most significant half in wad.

Application notes:

The MPY instruction performs a multiplication. For example, if Workspace Register 5 contains 18, Workspace Register 6 contains 1B31H, and memory location NEW contains 5, the instruction

```
MPY     @NEW,5
```

changes the contents of Workspace Register 5 to 0 and Workspace Register 6 to 90. The source operand is unchanged. The Status Register is not affected by this instruction.

6.11 NEGATE--NEG

Op-code: 0500 (Format VI)

Syntax definition:

[<label>] b NEG b <gas> b [<comment>]

Example:

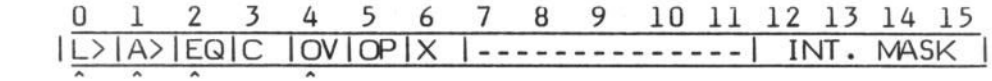
```
LABEL   NEG     R2          ;Replaces the contents of Workspace Register 2
                            with its additive inverse.
```

Definition:

Replaces the source operand with the two's-complement of the source operand. The computer determines the two's-complement value by inverting all bits of the source operand and adding one to the resulting word. The computer then compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and overflow.



Execution results:

-(gas) => (gas)

Application notes:

The NEG instruction changes the contents of an addressable memory location its additive inverse. For example, if Workspace Register 5 contains the value 0A342H, the instruction

```
        NEG     R5
```

changes the contents of Workspace Register 5 to 5CBEH. The logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

6.12 SUBTRACT WORDS--S

Op-code: 6000 (Format I)

Syntax definition:

[<label>] b S b <gas>,<gad> b [<comment>]

Example:

```
LABEL    S        R2,R3        ;Subtracts the contents of Workspace Register 2
                                 from the contents of Workspace Register 3.
```

Definition:

Subtracts a copy of the source operand from the destination operand and places the difference in the destination operand. The computer compares the difference to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set. The source operand remains unchanged.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.

```
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
|L>|A>|EQ|C  |OV|OP|X  |--------------|  INT. MASK  |
 ^   ^   ^   ^   ^
```

Execution results:

(gad) - (gas) => (gad)

Application notes:

The S instruction subtracts signed integer values. For example, if memory location OLDVAL contains a value of 1225H and memory location NEWVAL contains a value of 8223H, the instruction

```
        S        @OLDVAL,@NEWVAL
```

changes the contents of NEWVAL to 6FFEH. The logical greater than, arithmetic greater than, carry, and overflow status bits are set, while the equal status bit is reset.

6.13 SUBTRACT BYTES--SB

Op-code: 7000 (Format I)

Syntax definition:

[<label>] b SB b <gas>,<gad> b [<comment>]

Example:

```
LABEL   SB      R2,R3       ;Subtracts the leftmost byte of Workspace
                            Register 2 from the leftmost byte of Workspace
                            Register 3.
```

Definition:

Subtracts a copy of the source operand (byte) from the destination operand (byte) and replaces the destination operand byte with the difference. When the destination operand byte is addressed in the Workspace Register mode, only the leftmost byte (bits 0-7) in the Workspace Register is used. The computer compares the resulting byte to zero and sets/resets the status bits accordingly. When there is a carry of the most significant bit of the byte, the carry status bit is set. When there is an overflow, the overflow status bit is set. If the result byte establishes odd parity (an odd number of logic one bits in the byte), the odd parity status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, overflow, and odd parity.

```
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
|L>|A>|EQ|C  |OV|OP|X  |--------------|  INT. MASK  |
 ^   ^   ^   ^   ^   ^
```

Execution results:

<gad> - <gas> => <gad>

Application notes:

The SB instruction subtracts signed integer bytes. For example, if Workspace Register 6 contains the value 121CH, memory location 121CH contains the value 2331H, and Workspace Register 1 contains the value 1344H, the instruction

```
SB      *6+,1
```

changes the contents of Workspace Register 6 to 121DH and the contents of Workspace Register 1 to 0F044H. The logical greater than status bit is set, while the other status bits affected by this instruction are reset.

6.14 INSTRUCTION EXAMPLES

This section includes several arithmetic instruction examples for further clarification. The application of these instructions is not necessarily limited to that given.

6.14.1 Incrementing and Decrementing Examples

There are two decrement and two increment instructions that may be used for various types of control when passing through a loop, indexing through an array, or operating within a group of instructions.

The incrementing and decrementing instructions available for use with the Assembler are:

INCrement (INC)
INCrement by Two (INCT)
DECrement (DEC)
DECrement by Two (DECT)

The single increment and decrement instructions are useful for indexing byte arrays and for counting byte operations. The increment by two and decrement by two instructions are useful for indexing word arrays and for counting word operations. The following sections provide some examples of these operations.

6.14.1.1 Increment Instruction Example

The example program shows how the INC instruction is useful in byte operations. The program searches a character array for a character with odd parity. To terminate the search, the last character contains zero. The search begins at the lowest address of the array and maintains an index in a Workspace Register. The character array for this example is called A1 and is also the relocatable address of the array. The code is shown on the next page.

```
        SETO    R1          ;Set counter index to -1 (0FFFFH).
SEARCH  INC     R1          ;Increment index.
        MOVB    @A1(R1),R2  ;Get character.
        JOP     ODDP        ;Jump if found.
        JNE     SEARCH      ;Continue search if not zero.
        .
        .
        .
ODDP    ...
```

6.14.1.2 Decrement Instruction Example

To illustrate the use of a DEC instruction in a byte array, this example inverts a 26-character byte array and places the results in another array of the same size called A2. The contents of A1 are defined with a data .ASCII statement as follows.

```
A1      .ASCII  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Array A2 is defined with the .BLOCK statement as follows.

```
A2      .BLOCK  26
```

The sample code for the solution is:

```
        LI      R5,26               ;Counter and index for A1.
        LI      R4,A2               ;Address of A2.
INVRT   MOVB    @A1-1(R5),*R4+      ;Invert array (Note 1).
        DEC     R5                  ;Reduce counter.
        JGT     INVRT               ;Continue if not complete.
        .
        .
        .
```

Note:

[1] @A1(R5) addresses the elements of array A1 in descending order as Workspace Register 5 is decremented. *R4+ addresses array A2 in ascending order as Workspace Register four is incremented.

Array A2 contains the following as a result of executing this sequence of code:

```
A2      ZYXWVUTSRQPONMLKJIHGFEDCBA
```

The JGT instruction used to terminate the loop allows Workspace Register 5 to serve both as a counter and as an index register.

A special quality of the DEC instruction allows you to simulate a jump greater than or equal to zero instruction. Since DEC always sets the carry status bit except when changing from zero to minus one, it can be used in conjunction with a JOC instruction to form a JGE loop. The example below performs the same function as the preceding example.

```
A1      .ASCII  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
A2      .BLOCK  26
        LI      R5,25           ;Counter and index for A1.
        LI      R4,A2           ;Address of A2.
INVRT   MOVB    @A1(R5),*R4+    ;Invert array.
        DEC     R5              ;Reduce counter.
        JOC     INVRT           ;Continue if not complete.
        .
        .
        .
```

Note: Since the use of JOC makes the loop execute when the counter is zero, the counter is initialized to 25 rather than 26 as in the preceding example.

6.14.1.3 Decrement by Two Instruction Example

To illustrate the use of a DECT instruction in processing word arrays, this example adds the elements of a word array to the elements of another word array and places the results in the second array. The contents of the two arrays are initialized as follows.

```
A1      .WORD   500,300,800,1000,1200,498,650,3,27,0
A2      .WORD   36,192,517,29,315,807,290,40,130,1320
```

The sample code that adds the two arrays is as follows.

```
        LI      R4,20                   ;Initialize counter (Note 1).
SUMS    A       @A1-2(R4),@A2-2(R4)     ;Add arrays (Note 2).
        DECT    R4                      ;Decrement counter by two.
        JGT     SUMS                    ;Repeat addition.
```

Notes:

[1]The counter is preset to 20 which is the number of bytes in the array.
[2]The addressing of the two arrays through the use of the at sign (@) is indexed by the counter, which is decremented after each addition.

The contents of the A2 array after the addition process are as follows.

```
A2      536,492,1317,1029,1515,1305,940,43,157,1320
```

There is another method by which this addition process may be accomplished. This method is shown in the following code.

```
        LI      R4,10           ;Initialize counter (Note 1).
        LI      R5,A1           ;Load address of A1 (Note 2).
        LI      R6,A2           ;Load address of A2 (Note 2).
SUMS    A       *R5+,*R6+       ;Add arrays (Note 3).
        DEC     R4              ;Decrement counter.
        JGT     SUMS            ;Repeat addition (Note 4).
```

Notes:

[1]The counter is preset to 10 (the number of elements in the array).
[2]This address is incremented each time an addition takes place. The increment is via the auto-increment function (+).
[3]The * indicates that the contents of the register are to be used as an address, and the + indicates that it is to be automatically incremented by two each time the instruction is executed.
[4]Workspace Register 4 is only greater than zero for ten executions of the DEC instruction, so control is transferred to SUMS nine times after the initial execution.

After execution, the contents of array A2 are the same for this method as for the first.

6.14.2 General Example

The following program illustrates several of the arithmetic instructions. The program consists of a calling program and a subroutine. The subroutine produces the result of the function X-(|3*Y|+5) where X and Y are variable data, treated as signed integers, and passed to the subroutine from the calling program.

To simplify the example, no error checking is included in the subroutine, and it is assumed that the product of 3*Y is in the range of a signed 16-bit word (-32,768 through 32,767).

```
;CALLING PROGRAM
            .
            .
            .
            BL      @CALC       ;Call subroutine.
VAR         .WORD   37          ;X value.
            .WORD   1804        ;Y value.
            MOV     0,RESULT    ;Save result.
            .
            .
            .
RESULT      .BLOCK  2
            .
            .
            .
CALC        MOV     *R11+,R0    ;Put X value in Register 0.
            MOV     *R11+,R1    ;Put Y value in Register 1.
            ABS     R1          ;Take absolute value of Y.
            MPY     @THREE,1    ;Take 3 times absolute value of Y.
            AI      R2,5        ;Add 5 to previous result.
            S       R2,R0       ;Subtract previous result from X.
            B       *R11        ;Return.
THREE       .WORD   3           ;Constant.
```

SECTION 7: JUMP AND BRANCH INSTRUCTIONS

The following jump and branch instructions are described in this section.

Instruction	Mnemonic	Section
Branch	B	7.1
Branch and Link	BL	7.2
Branch and Load Workspace Pointer	BLWP	7.3
Jump if EQual	JEQ	7.4
Jump if Greater Than	JGT	7.5
Jump if High or Equal	JHE	7.6
Jump if logical High	JH	7.7
Jump if logical Low	JL	7.8
Jump if Low or Equal	JLE	7.9
Jump if Less Than	JLT	7.10
Unconditional JuMP	JMP	7.11
Jump if No Carry	JNC	7.12
Jump if Not Equal	JNE	7.13
Jump if No Overflow	JNO	7.14
Jump if Odd Parity	JOP	7.15
Jump On Carry	JOC	7.16
ReTurn Workspace Pointer	RTWP	7.17
EXecute	X	7.18
EXtended OPeration	XOP	7.19

Examples are given in Section 7.20.

Branch instructions transfer control either unconditionally or conditionally according to the state of one or more bits of the Status Register. The conditional branch (jump) instructions and the status bit or bits tested are shown on the next page.

Status Bits Tested by Jump Instructions

Mnemonic	L>	A>	EQ	C	OV	OP	Jump if:
JH	X	-	X	-	-	-	L>=1 and EQ=0
JL	X	-	X	-	-	-	L>=0 and EQ=0
JHE	X	-	X	-	-	-	L>=1 or EQ=1
JLE	X	-	X	-	-	-	L>=0 or EQ=1
JGT+	-	X	-	-	-	-	A>=1
JLT+	-	X	X	-	-	-	A>=0 and EQ=0
JEQ	-	-	X	-	-	-	EQ=1
JNE	-	-	X	-	-	-	EQ=0
JOC	-	-	-	X	-	-	C=1
JNC	-	-	-	X	-	-	C=0
JNO	-	-	-	-	X	-	OV=0
JOP	-	-	-	-	-	X	OP=1

+Only JGT and JLT use signed arithmetic comparisons. The others are unsigned (logical) comparisons.

For all jump instructions, a displacement of zero results in execution of the next instruction in sequence. A displacement of -1 results in execution of the same instruction (a single-instruction loop).

Each instruction's description consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

JUMP AND BRANCH INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

7.1 BRANCH--B

Op-code: 0440 (Format VI)

Syntax definition:

[<label>] b B b <gas> b [<comment>]

Example:

```
LABEL   B       @THERE      ;Transfers control to location THERE.
```

Definition:

Replaces the Program Counter contents with the source address and transfers control to the instruction at that location.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(gas) => (PC)

Application notes:

The B instruction transfers control to another section of code to change the linear flow of the program. For example, if the contents of Workspace Register 3 is 21CCH, the instruction

```
        B       *R3
```

causes the word at location 21CCH to be used as the next instruction, because this value replaces the contents of the Program Counter when this instruction is executed.

7.2 BRANCH AND LINK--BL

Op-code: 0680 (Format VI)

Syntax definition:

[<label>] b BL b <gas> b [<comment>]

Example:

```
LABEL    BL       @SUBR       ;Calls SUBR as a common Workspace subroutine.
```

Definition:

Places the source address in the Program Counter, places the address of the instruction following the BL instruction (in memory) in Workspace Register 11, and transfers control to the new Program Counter contents.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(old PC) => (Workspace Register 11)
(gas) => (PC)

Application notes:

The BL instruction returns linkage. For example, if the instruction

```
BL       @TRAN
```

occurs at memory location 04BCH, this instruction has the effect of placing memory location TRAN in the Program Counter. Since the instruction BL @TRAN requires two words of machine code (which are placed at addresses 04BCH and 04BEH), the word address immediately following the second word is 04C0H so that value is the address placed in Workspace Register 11.

7.3 BRANCH AND LOAD WORKSPACE POINTER--BLWP

Op-code: 0400 (Format VI)

Syntax definition:

[<label>] b BLWP b <gas> b [<comment>]

Example:

```
LABEL    BLWP    @VECT      ;Branches to subroutine at address (@VECT+2)
                            and executes context switch.
```

Definition:

Places the source operand in the Workspace Pointer and the word immediately following the source operand in the Program Counter. Places the previous contents of the Workspace Pointer in the new Workspace Register 13, places the previous contents of the Program Counter (address of the instruction following BLWP) in the new Workspace Register 14, and places the contents of the Status Register in the new Workspace Register 15. When all store operations are complete, the computer transfers control to the new Program Counter.

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

(gas) => (WP)
(gas + 2) => (PC)
(old WP) => (Workspace Register 13)
(old PC) => (Workspace Register 14)
(ST) => (Workspace Register 15)

Application notes:

The BLWP instruction links to subroutines, program modules, or other programs that do not necessarily share the calling program's workspace. See Section 7.20.3 for an example of using the BLWP instruction.

7.4 JUMP IF EQUAL--JEQ

Op-code: 1300 (Format II)

Syntax definition:

[<label>] b JEQ b <exp> b [<comment>]

Example:

```
LABEL    JEQ     LOC          ;Jumps to LOC if EQ = 1.
```

Definition:

When the equal status bit is set, transfers control by adding the signed displacement in the instruction word to the Program Counter and then placing the sum in the Program Counter to transfer control.

Status bits tested:

Equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Jump if: EQ = 1

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the equal bit is equal to 1: (PC) + Displacement => (PC).
If the equal bit is equal to 0: (PC) => (PC).

Application notes:

The JEQ instruction transfers control when the equal status bit is set.

7.5 JUMP IF GREATER THAN--JGT

Op-code: 1500 (Format II)

Syntax definition:

[<label>] b JGT b <exp> b [<comment>]

Example:

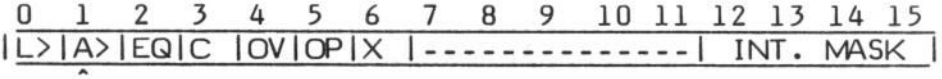
```
LABEL    JGT     THERE       ;Jumps to THERE if A> = 1.
```

Definition:

When the arithmetic greater than status bit is set, adds the signed displacement in the instruction word to the Program Counter and places the sum in the Program Counter. Transfers control to the new Program Counter location.

Status bits tested:

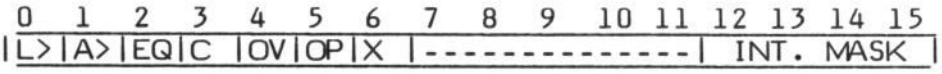
Arithmetic greater than.



Jump if: A> = 1

Status bit affected:

None.



Execution results:

If the arithmetic greater than bit is equal to 1: (PC) + Displacement => (PC).
If the arithmetic greater than bit is equal to 0: (PC) => (PC).

Application notes:

The JGT instruction transfers control if the arithmetic greater than status bit is set.

7.6 JUMP IF HIGH OR EQUAL--JHE

Op-code: 1400 (Format II)

Syntax definition:

[<label>] b JHE b <exp> b [<comment>]

Example:

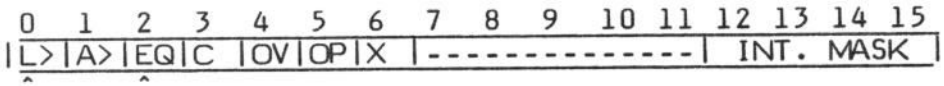
```
LABEL   JHE     BLBD        ;Jumps to location BLBD if either EQ or L> is
                            set.
```

Definition:

When the equal status bit or the logical greater than status bit is set, adds the signed displacement in the instruction word to the Program Counter and replaces the contents of the Program Counter with the sum.

Status bits tested:

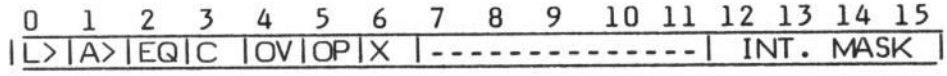
Logical greater than, equal.



Jump if: L> = 1 or EQ = 1

Status bits affected:

None.



Execution results:

If the logical greater than bit is equal to 1 or the equal bit is equal to 1:
(PC) + Displacement => (PC).
If the logical greater than bit and the equal bit are equal to 0: (PC) => (PC).

Application notes:

The JHE instruction transfers control when either the logical greater than or equal status bit is set.

7.7 JUMP IF LOGICAL HIGH--JH

Op-code: 1B00 (Format II)

Syntax definition:

[<label>] b JH b <exp> b [<comment>]

Example:

```
LABEL    JH       CONT        ;If L> equals 1 and EQ equals 0, skips to CONT.
```

Definition:

When the equal status bit is reset and the logical greater than status bit is set, adds the signed displacement in the instruction word to the contents of the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

Logical greater than, equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^     ^
```

Jump if: L> = 1 and EQ = 0

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the logical greater than bit is equal to 1 and the equal bit is equal to 0:
(PC) + Displacement => (PC).
If the logical greater than bit is equal to 0 or the equal bit is equal to 1:
(PC) => (PC).

Application notes:

The JH instruction transfers control when the equal status bit is reset and the logical greater than status bit is set.

7.8 JUMP IF LOGICAL LOW--JL

Op-code: 1A00 (Format II)

Syntax definition:

[<label>] b JL b <exp> b [<comment>]

Example:

```
LABEL    JL       PREVLB      ;If L> and EQ are reset, jumps to PREVLB.
```

Definition:

When the equal and logical greater than status bits are reset, adds the signed displacement in the instruction word to the Program Counter contents and replaces the Program Counter with the sum.

Status bits tested:

Logical greater than, equal.

```
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
|L>|A>|EQ|C |OV|OP|X |----------|  INT. MASK  |
  ^       ^
```

Jump if: L> = 0 and EQ = 0

Status bits affected:

None.

```
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
|L>|A>|EQ|C |OV|OP|X |----------|  INT. MASK  |
```

Execution results:

If the logical greater than bit and the equal bit are equal to 0:
(PC) + Displacement => (PC).
If the logical greater than bit is equal to 1 or the equal bit is equal to 1:
(PC) => (PC).

Application notes:

The JL instruction transfers control when the equal and logical greater than status bits are reset.

7.9 JUMP IF LOW OR EQUAL--JLE

Op-code: 1200 (Format II)

Syntax definition:

[<label>] b JLE b <exp> b [<comment>]

Example:

```
LABEL    JLE      THERE      ;Jumps to THERE when EQ = 1 or L> = 0.
```

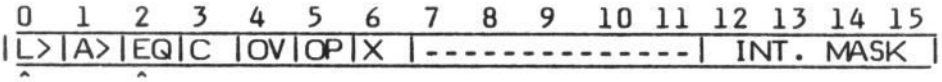
Definition:

When the equal status bit is set or the logical greater than status bit is reset, adds the signed displacement in the instruction word to the contents of the Program Counter and replaces the Program Counter with the sum.

Note: JLE is not "jump if less than or equal."

Status bits tested:

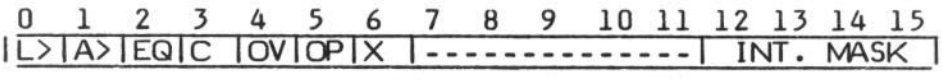
Logical greater than, equal.



Jump if: L> = 0 or EQ = 1

Status bits affected:

None.



Execution results:

If the logical greater than bit is equal to 0 or the equal bit is equal to 1:
(PC) + Displacement => (PC).
If the logical greater than bit is equal to 1 and the equal bit is equal to 0:
(PC) => (PC).

Application notes:

The JLE instruction transfers control when the equal status bit is set or the logical greater than status bit is reset.

7.10 JUMP IF LESS THAN--JLT

Op-code: 1100 (Format II)

Syntax definition:

[<label>] b JLT b <exp> b [<comment>]

Example:

```
LABEL    JLT      THERE      ;Jumps to THERE if A> = 0 and EQ = 0.
```

Definition:

When the equal and arithmetic greater than status bits are reset, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter contents with the sum.

Status bits tested:

Arithmetic greater than, equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
      ^  ^
```

Jump if: A> = 0 and EQ = 0

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the arithmetic greater than bit and the equal bit are equal to 0:
(PC) + Displacement => (PC)
If the arithmetic greater than bit is equal to 1 or the equal bit is equal to 1:
(PC) => (PC).

Application notes:

The JLT instruction transfers control when the equal and arithmetic greater than status bits are reset.

7.11 UNCONDITIONAL JUMP--JMP

Op-code: 1000 (Format II)

Syntax definition:

[<label>] b JMP b <exp> b [<comment>]

Example:

```
LEAVE   JMP     LANA        ;Jumps to address LANA.
```

Definition:

Adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum if the sum is within 100H bytes of the current Program Counter.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(PC) + Displacement => (PC)

The Program Counter is always incremented to the address of the next instruction prior to execution of an instruction. The execution results of jump instructions refer to the Program Counter contents after the contents have been incremented to address the next instruction in sequence. The displacement (in words) is shifted to the left one bit position to orient the word displacement to the word address, and added to the Program Counter contents.

Application notes:

The JMP instruction transfers control to another section of the program.

The pseudo-instruction NOP is equivalent to

```
JMP $+2
```

and moves to the next instruction. It has no effect except to take up time and memory.

7.12 JUMP IF NO CARRY--JNC

Op-code: 1700 (Format II)

Syntax definition:

[<label>] b JNC b <exp> b [<comment>]

Example:

```
LABEL    JNC     NONE        ;Jumps to NONE if C = 0.
```

Definition:

When the carry status bit is reset, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

Carry.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Jump if: C = 0

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the carry bit is equal to 0: (PC) + Displacement => (PC).
If the carry bit is equal to 1: (PC) => (PC).

Application notes:

The JNC instruction transfers control when the carry status bit is reset.

7.13 JUMP IF NOT EQUAL--JNE

Op-code: 1600 (Format II)

Syntax definition:

[<label>] b JNE b <exp> b [<comment>]

Example:

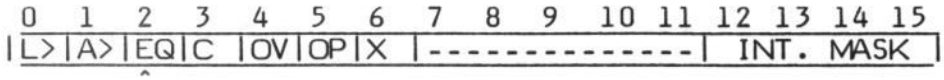
```
LABEL    JNE     LOC2        ;Jumps to LOC2 if EQ = 0.
```

Definition:

When the equal status bit is reset, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

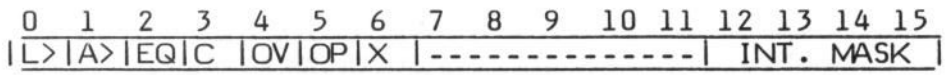
Equal.



Jump if: EQ = 0

Status bits affected:

None.



Execution results:

If the equal bit is equal to 0: (PC) + Displacement => (PC).
If the equal bit is equal to 1: (PC) => (PC).

Application notes:

The JNE instruction transfers control when the equal status bit is reset. For instance, JNE is often useful when testing CRU bits.

7.14 JUMP IF NO OVERFLOW--JNO

Op-code: 1900 (Format II)

Syntax definition:

[<label>] b JNO b <exp> b [<comment>]

Example:

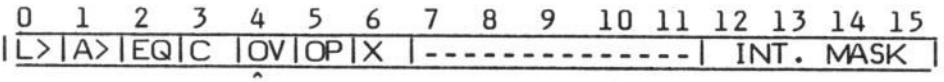
```
LABEL    JNO     NORML      ;Jumps to NORML if OV = 0.
```

Definition:

When the overflow status bit is reset, adds the displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

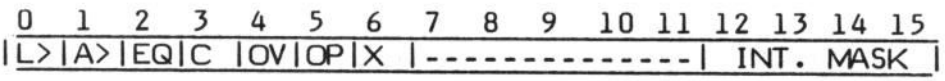
Overflow.



Jump if: OV = 0

Status bits affected:

None.



Execution results:

If the overflow bit is equal to 0: (PC) + Displacement => (PC).
If the overflow bit is equal to 1: (PC) => (PC).

Application notes:

The JNO instruction transfers control when the overflow status bit is reset. JNO normally transfers control during arithmetic sequences where addition, subtraction, incrementing, and decrementing may cause an overflow condition. JNO may also be used following an SLA (Shift Left Arithmetic) operation. If, during SLA execution, the sign of the Workspace Register being shifted changes, the overflow status bit is set. This feature permits transfer, after a sign change, to error correction routines or to another functional code sequence.

7.15 JUMP IF ODD PARITY--JOP

Op-code: 1C00 (Format II)

Syntax definition:

[<label>] b JOP b <exp> b [<comment>]

Example:

```
LABEL    JOP      THERE      ;Jumps to THERE if OP = 1.
```

Definition:

When the odd parity status bit is set, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

Odd parity.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
                 ^
```

Jump if: OP = 1

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the odd parity bit is equal to 1: (PC) + Displacement => (PC).
If the odd parity bit is equal to 0: (PC) => (PC).

Application notes:

The JOP instruction transfers control when there is odd parity. Odd parity indicates that there is an odd number of logic one bits in the byte tested. JOP transfers control if the byte tested contains an odd number of logic one bits. This instruction may be used in data transmissions where the parity of the transmitted byte is used to ensure the validity of the received character at the point of reception.

7.16 JUMP ON CARRY--JOC

Op-code: 1800 (Format II)

Syntax definition:

[<label>] b JOC b <exp> b [<comment>]

Example:

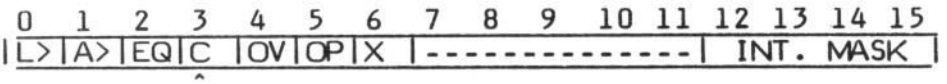
```
LABEL    JOC     PROCED     ;If C = 1, jumps to PROCED.
```

Definition:

When the carry status bit is set, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

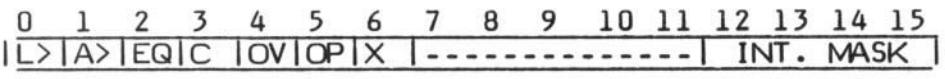
Carry.



Jump if: C = 1

Status bits affected:

None.



Execution results:

If the carry bit is equal to 1: (PC) + Displacement => (PC).
If the carry bit is equal to 0: (PC) => (PC).

Application notes:

The JOC instruction transfers control when the carry status bit is set.

7.17 RETURN WITH WORKSPACE POINTER--RTWP

Op-code: 0380 (Format VII)

Syntax definition:

[<label>] b RTWP b [<comment>]

Example:

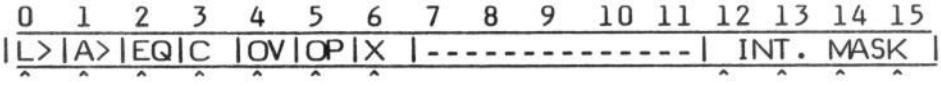
```
LABEL    RTWP    ;Returns from subroutine called by BLWP.
```

Definition:

Replaces the contents of the Workspace Pointer Register with the contents of the current Workspace Register 13. Replaces the contents of the Program Counter with the contents of the current Workspace Register 14. Replaces the contents of the Status Register with the contents of the current Workspace Register 15. The effect of this instruction is to restore the execution environment that existed prior to an interrupt, a BLWP instruction, or an XOP instruction.

Status bits affected:

Restores all status bits to the value contained in Workspace Register 15.



Execution results:

(Workspace Register 13) => (WP)
(Workspace Register 14) => (PC)
(Workspace Register 15) => (ST)

Application notes:

The RTWP instruction restores the execution environment after the completion of an interrupt, a BLWP instruction, or an XOP instruction.

7.18 EXECUTE--X

Op-code: 0480 (Format VI)

Syntax definition:

[<label>] b X b <gas> b [<comment>]

Example:

```
LABEL    X       R2          ;Executes the contents of Workspace Register 2.
```

Definition:

Executes the source operand as an instruction. When the source operand is not a single word instruction, the word or words following the execute instruction are used with the source operand as a 2-word or 3-word instruction. The source operand, when executed as an instruction, may affect the contents of the Status Register. The Program Counter increments by either one, two, or three words depending upon the source operand. If the executed instruction is a branch, the branch is taken. If the executed instruction is a jump and if the conditions for a jump (i.e. the status test indicates a jump) are satisfied, then the jump is taken relative to the location of the X instruction.

Status bits affected:

None, but substituted instruction affects status bits normally.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

An instruction at gas is executed instead of the X instruction.

Application notes:

The X instruction executes the source operand as an instruction. This is primarily useful when the instruction to be executed is dependent upon a previous operation. Refer to Section 7.20 for additional application notes.

7.19 EXTENDED OPERATION--XOP

Op-code: 2C00 (Format IX)

Syntax definition:

[<label>] b XOP b <gas>,<xop> b [<comment>]

Example:

```
LABEL   XOP     @BUFF(R4),1 ;Performs XOP 1 on the word of the address
                            BUFF plus the displacement specified by
                            Workspace Register 4.
```

Definition:

This instruction is on all TI-99/4A Home Computers. However, some only support XOP 2 while others support both XOP 1 and XOP 2. To find out if your TI-99/4A computer supports the XOP 1 instruction, read one word at address 44H. If the word is 0FFD8H, then XOP 1 is available. If it contains other data (most likely 0FFE8H), then XOP 1 is not available.

The op field specifies the extended operation transfer vector in memory. The two memory words at that location contain the Workspace Pointer and Program Counter contents for the software implemented XOP instruction subroutine. Note that the two memory words at this location must contain the necessary Workspace Pointer and Program Counter values prior to the XOP instruction execution for software implemented instructions.

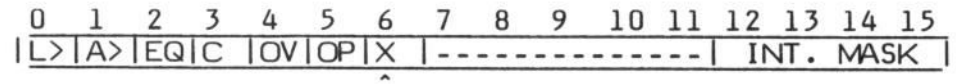
XOP 1 is at address 44H, with vectors 0FFD8H and 0FFF8H. XOP 2 is at address 48H with vectors 83A0H and 8300H. The first entry in the vector is the new workspace address. The second entry is the new Program Counter address.

When the computer is turned on, XOP 1 is set up to be used with development software used by Texas Instruments. However, if you have XOP 1 you may modify the data for your own use.

The effective address of the source operand is placed in Workspace Register 11 of the XOP workspace. The Workspace Pointer contents are placed in Workspace Register 13 of the XOP workspace. The Program Counter contents are placed in Workspace Register 14 of the XOP workspace. The Status contents are placed in Workspace Register 15 of the XOP workspace. Control is transferred to the new Program Counter address and the software implemented XOP is executed. (XOP execution of software implemented XOP instruction is similar to an interrupt trap execution.)

Status bits affected:

Extended operation.



Execution results:

gas => (Workspace Register 11)
(0040H + (op)*4) => (WP)
(0042H + (op)*4) => (PC)
(WP) => (Workspace Register 13)
(PC) => (Workspace Register 14)
(ST) => (Workspace Register 15)
1 => X (XOP status bit)

7.20 INSTRUCTION EXAMPLES

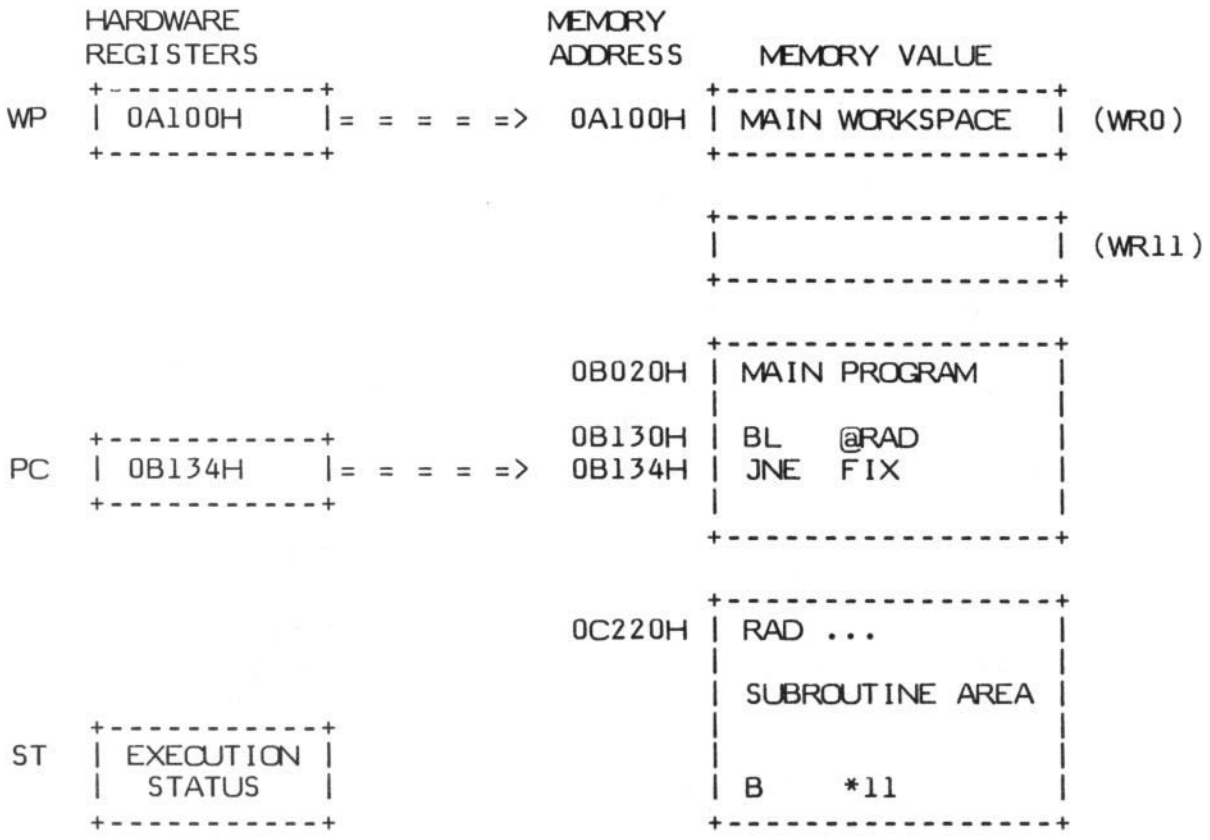
There are two types of subroutine linkage available with the Assembler. One type, called a common workspace subroutine, uses the same set of Workspace Registers that the calling routine uses. The BL instruction stores the contents of the Program Counter in Workspace Register 11 and transfers control to the subroutine.

The other type is a context switch subroutine. The BLWP instruction stores the contents of the Workspace Pointer Register, the Program Counter, and the Status Register in Workspace Registers 13, 14, and 15. The instruction makes the subroutine workspace active and transfers control to the subroutine.

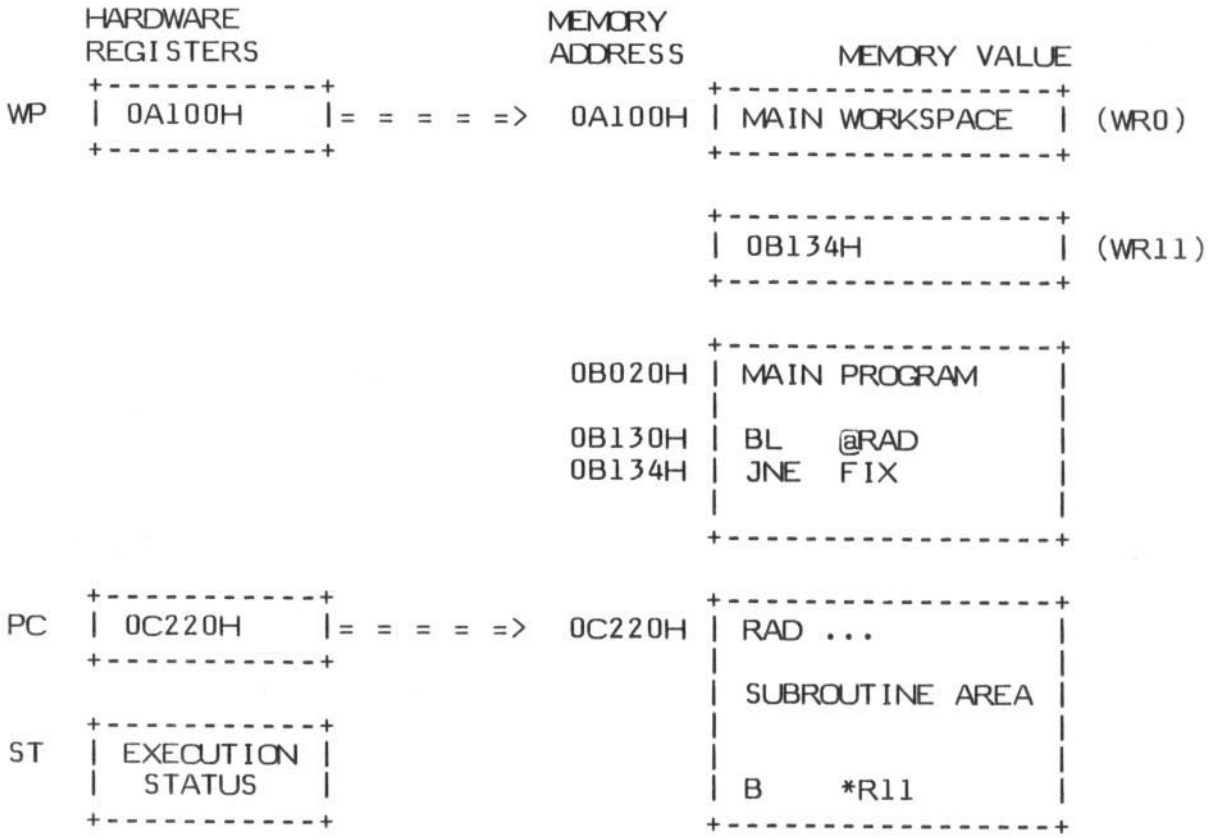
7.20.1 Common Workspace Subroutine Example

The following is an example of memory contents prior to a BL call to a subroutine. The contents of Workspace Register 11 are not important to the main routine. When the BL instruction is executed, the CPU stores the contents of the Program Counter in Workspace Register 11 of the main routine and transfers control to the instruction located at the address indicated by the operand of the BL instruction. This type of subroutine uses the main program workspace. The second example shows the memory contents after the call to the subroutine with the BL instruction.

When the instruction at location 1130H is executed (BL @RAD), the present contents of the Program Counter, which point to the next instruction, are saved in Workspace Register 11. Workspace Register 11 would then contain an address of 1134H. The Program Counter is then loaded with the address of label RAD, which is address 0C220H. This subroutine returns to the main program with a branch to the address in Workspace Register 11 using the B *R11 instruction.



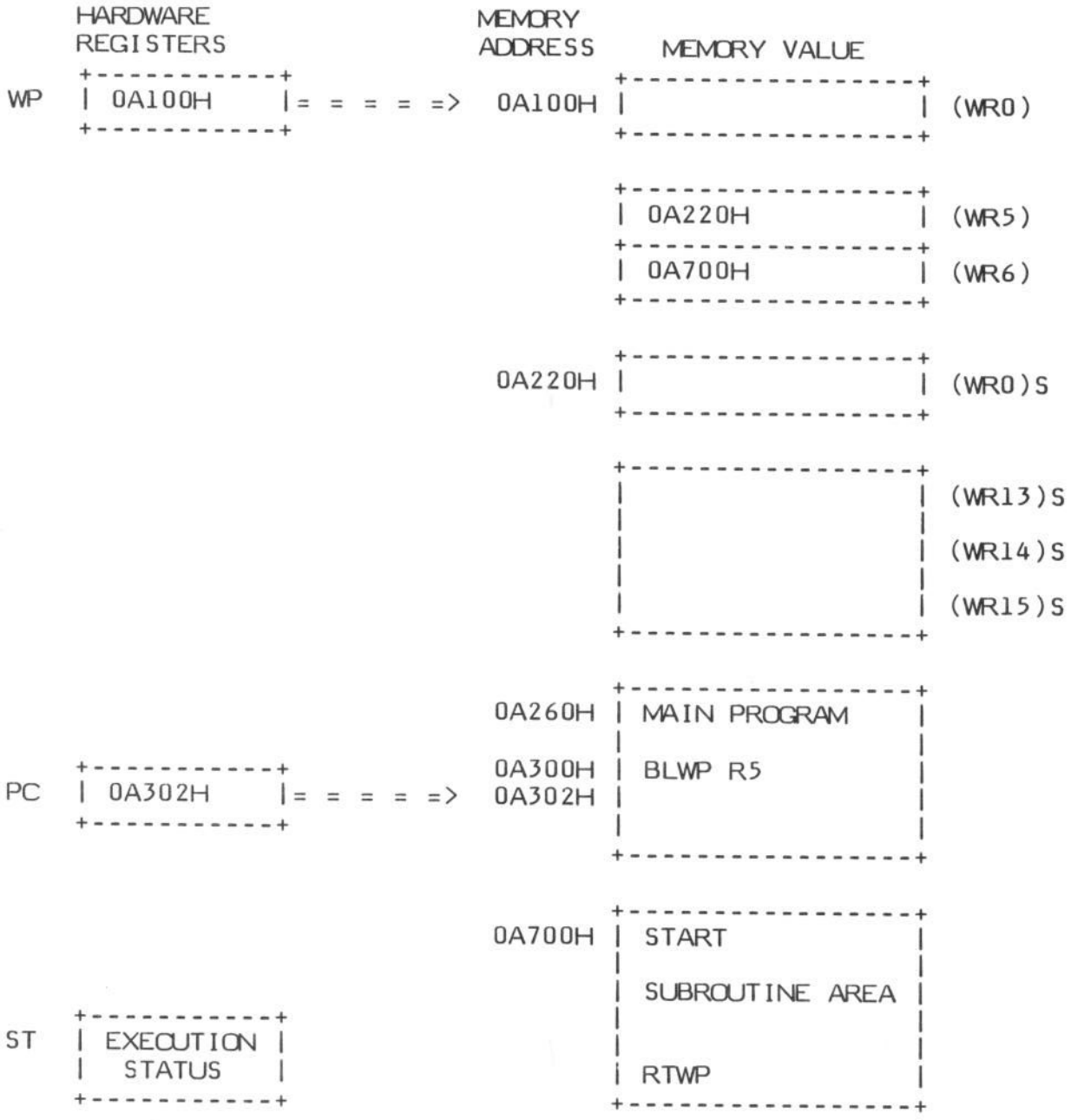
Common Workspace Subroutine Example



PC Contents after BL Instruction Execution

7.20.2 Context Switch Subroutine Example

This example shows the memory contents prior to the call to the subroutine. The contents of the subroutine's Workspace Registers 13, 14, and 15 are not significant. When the BLWP instruction is executed at location 0300H, there is a context switch from the main program to the subroutine. The context switch then places the main program Program Counter, Workspace Pointer, and Status Register contents in Workspace Registers 13, 14, and 15 of the subroutine. This saves the environment of the main program for use on return. The operand of the BLWP instruction specifies that the address vector for the context switch is in Workspace Registers 5 and 6. The address in Workspace Register 5 is placed in the Workspace Pointer Register, and the address in Workspace Register 6 is placed in the Program Counter.

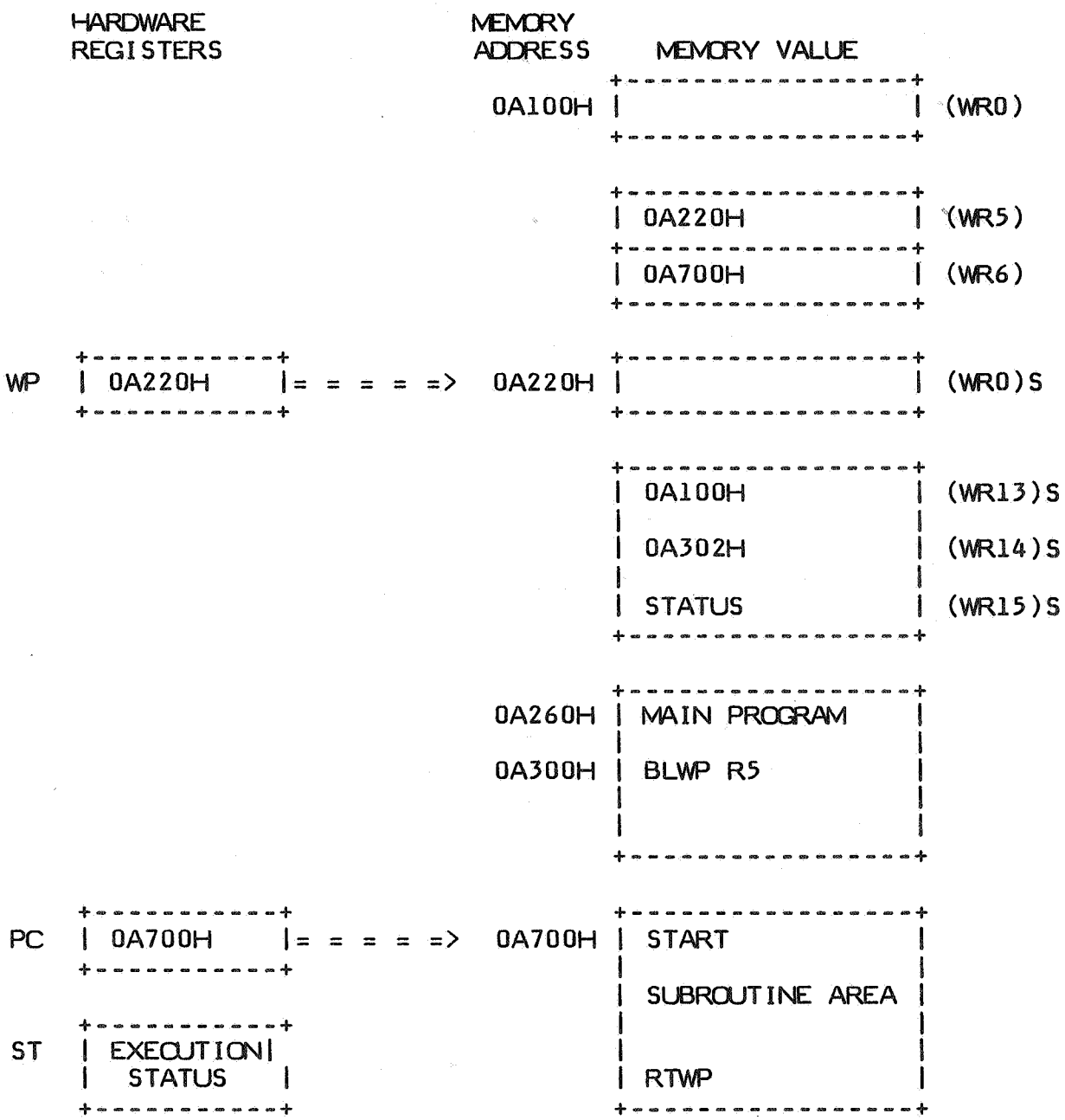


(WNr) = Workspace Register of Main Program
(WNr)S = Workspace Register of Subroutine

Context Switch Subroutine Example

After the instruction at location 0300H is executed, the Workspace Pointer points to the subroutine workspace and the Program Counter points to the first instruction of the subroutine. The contents of the Status Register are not reset prior to the

execution of the first instruction of the subroutine, so the status indicated will actually be the status of the main program execution. A subroutine may then execute depending on the status of the main program.

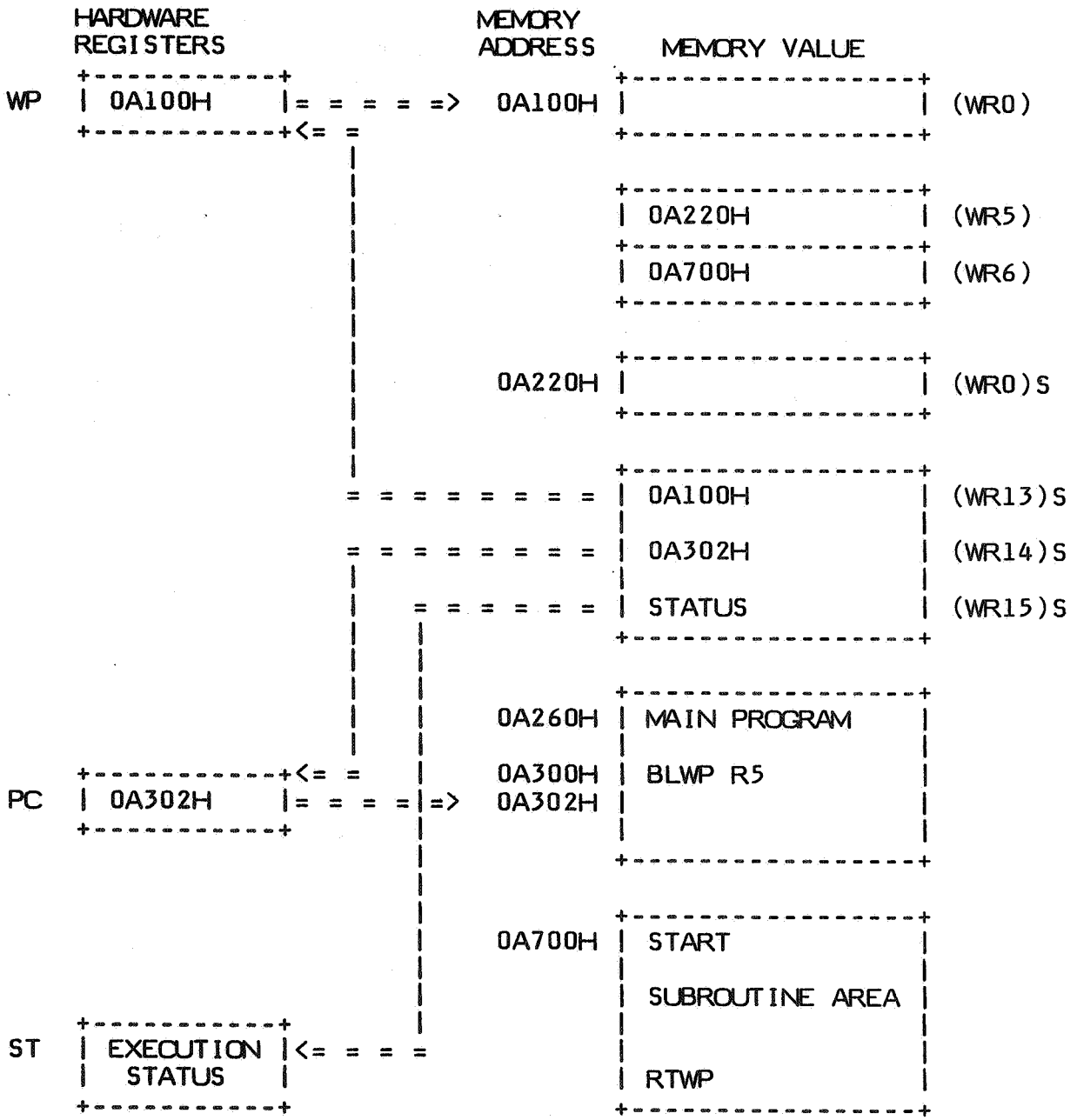


(WNr) = Workspace Register of Main Program
(WNr)S = Workspace Register of Subroutine

After Execution of BLWP Instruction

JUMP AND BRANCH INSTRUCTIONS

This example subroutine contains a RTWP return from the subroutine. Control is transferred to the main program at the instruction following the BLWP to the subroutine. The Status Register is restored from Workspace Register 15 and the Workspace Pointer points to the workspace of the main program.



(WNr) = Workspace Register of Main Program
(WNr)S = Workspace Register of Subroutine

After Return using the RTWP Instruction

7.20.3 Passing Data to Subroutines

When a subroutine is entered with a context switch (BLWP), data may be passed using either the contents of Workspace Register 13 or 14 of the subroutine workspace. Workspace Register 13 contains the memory address of the calling program's workspace, which may contain data to be passed to the subroutine. Workspace Register 14 contains the memory address of the next memory location following the BLWP instruction. This location and following locations may also contain data to be passed to the subroutine.

When the calling program's workspace contains data for the subroutine, this data may be obtained by using the indexed memory address mode indexed by Workspace Register 13. The address used is equal to twice the number of the Workspace Register that contains the desired data. The following instruction is an example.

```
MOV     @10(R13),R10
```

The contents of Workspace Register 5 of the calling program's workspace (bytes 10 and 11 relative to the workspace address) are placed in Workspace Register 10 of the subroutine workspace.

The examples on the next page show the passing of data to a subroutine by placing the data following the BLWP instruction.

```
        BLWP    @SUB            ;Subroutine call.
        .WORD   V1              ;Data.
        .WORD   V2              ;Data.
        .WORD   V3              ;Data.
        JEQ     ERROR           ;Return from subroutine, test for
        .                       error.  (The subroutine sets the
        .                       equal status bit to one for error.)
        .
SUB     .WORD   SUBWS,SUBPRG    ;Entry point for SUB and SUB
                                Workspace.
        .
        .
        .
SUBWS   .BLOCK  32
SUBPRG  MOV     *R14+,R1        ;Fetch V1 placed in Workspace
                                Register 1.
        MOV     *R14+,R2        ;Fetch V2 placed in Workspace
                                Register 2.
        MOV     *R14+,R3        ;Fetch V3 placed in Workspace
                                Register 3.
        .
        .
        .
        RTWP                    ;Return from subroutine.
```

The three MOV instructions retrieve the variables from the main program module and place them in Workspace Registers 1, 2, and 3 of the subroutine.

When the BLWP instruction is executed, the main program module status is stored in Workspace Register 15 of the subroutine. If the subroutine returns with a RTWP instruction, this status is placed in the Status Register after the RTWP instruction is executed. The subroutine may alter the Status Register contents prior to executing the RTWP instruction. The calling program can then test the appropriate bit of the status word (the equal bit in this example) with jump instructions.

A BL instruction can also be used to pass parameters to a subroutine. When using this instruction, the originating Program Counter value is placed in Workspace Register 11. Therefore, the subroutine must fetch the parameters relative to the contents of Workspace Register 11 rather than the contents of Workspace Register 14 as in the BLWP example. The example on the next page demonstrates parameter passing with a BL instruction.

```
        BL      @SUBR           ;Branch to subroutine.
        .WORD   PARM1,PARM2     ;Passed parameters stored in next
                                two memory words.
        JEQ     ERROR           ;Test for error. (Subroutine sets the
                                equal status bit to one for error.)
        .
        .
        .
SUBR    .EQU    $
        MOV     *R11+,R0        ;Get value of first parameter and
                                put in Workspace Register 0.
        MOV     *R11+,R1        ;Get value of second parameter and
                                put in Workspace Register 1. (R11
                                is incremented past the locations of
                                the two data words and now
                                indicates the address of the next
                                instruction in the main program.)
        .
        .
        B       *R11
```

7.20.4 Extended Operations

Extended operation instructions permit a limited extension of the existing instruction set to include additional instructions. In the computer, these additional instructions are implemented by software routines.

When the program module contains an XOP instruction that is software implemented, the computer locates the XOP Workspace Pointer and Program Counter words in the XOP reserved memory locations and loads the Workspace Pointer and Program Counter. When the Workspace Pointer and Program Counter are loaded, the computer transfers control to the XOP instruction set through a context switch. When the context switch is complete, the XOP workspace contains the calling routine return data in Workspace Registers 13, 14, and 15.

The XOP instruction passes one operand to the XOP (input to the XOP routine in Workspace Register 11 of the XOP workspace). At the completion of the software XOP, the XOP routine should return to the calling routine with an RTWP instruction that restores the execution environment of the calling routine to that in existence at the call to the XOP.

7.20.5 Execute Example

The execute instruction may be used to execute an instruction that is not in sequence without transferring control to the desired instruction. One useful application is to execute one of a table of instructions, selecting the desired instruction by using an index into the table. The computed value of the index determines which instruction is executed.

A table of shift instructions illustrates the use of the X instruction. Place the following instructions at location TBLE.

```
TBLE    SLA     R6,3            ;Shift Workspace Register 6.
        SLA     R7,3            ;Shift Workspace Register 7.
        SLA     R8,3            ;Shift Workspace Register 8.
TABEND  .EQU    $
```

A character is placed in the most significant byte of Workspace Register 5 to select the Workspace Register to be shifted to the left 3 bit positions. ASCII characters A, B, and C specify shifting Workspace Registers 6, 7, and 8, respectively. Other characters are ignored. The following code performs the selection of the shift desired.

```
        SRL     R5,8            ;Move to lower byte.
        AI      R5,-"A"         ;Subtract table bias.
        JLT     NOSHIFT         ;Illegal.
        SLA     R5,1            ;Make it a word index.
        CI      R5,TABEND-TBLE-2
        JGT     NOSHFT          ;Illegal.
        X       @TBLE(R5)
NOSHFT  .EQU    $
        .
        .
        .
```

When using the X instruction, if the substituted instruction contains a Ts field or a Td field that results in a two word instruction, the computer accesses the word following the X instruction as the second word, not the word following the substituted instruction. When the substituted instruction is a jump instruction with a displacement, the displacement must be computed from the X instruction, not from the substituted instruction.

SECTION 8: COMPARE INSTRUCTIONS

The following compare instructions are described in this section.

Instruction	Mnemonic	Section
Compare words	C	8.1
Compare Bytes	CB	8.2
Compare Immediate	CI	8.3
Compare Ones Corresponding	COC	8.4
Compare Zeros Corresponding	CZC	8.5

Compare instructions have no effect other than the setting or resetting of appropriate status bits in the Status Register. The compare instructions perform both arithmetic and logical comparisons. An arithmetic comparison is of the two operands as two's complement values, while a logical comparison is of the two operands as unsigned magnitude values.

Each instruction's description consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

COMPARE INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

8.1 COMPARE WORDS--C

Op-code: 8000 (Format I)

Syntax definition:

[<label>] b C b <gas>,<gad> b [<comment>]

Example:

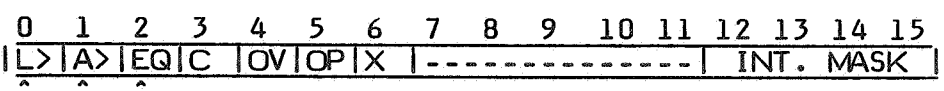
```
LABEL   C       R2,R3       ;Compares the contents of Workspace Register 2
                            and Workspace Register 3.
```

Definition:

Compares the source operand (word) with the destination operand (word) and sets/resets the status bits to indicate the results of the comparison. The arithmetic and equal comparisons compare the operand as signed, two's complement values. The logical comparison compares the two operands as unsigned, 16-bit magnitude values.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(gas) compared to (gad)

Application notes:

The C instruction compares the two operands as signed, two's complement values, affecting the A> status bit, and as unsigned integers, affecting the L> status bit. Some examples are:

		Status Bits Set		
Source	Destination	Logical>	Arithmetic>	Equal
0FFFFH	0000H	1	0	0
7FFFH	0000H	1	1	0
8000H	0000H	1	0	0
8000H	7FFFH	1	0	0
7FFFH	7FFFH	0	0	1
7FFFH	8000H	0	1	0

An alternate way to compare a word or byte to zero is to move the word or byte to itself. For example:

```
MOV     R0,R0
JEQ     OUT
```

jumps to OUT if R0 is equal to zero.

8.2 COMPARE BYTES--CB

Op-code: 9000 (Format I)

Syntax definition:

[<label>] b CB b <gas>,<gad> b [<comment>]

Example:

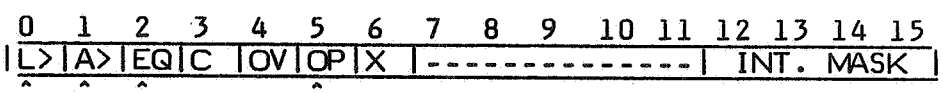
```
LABEL   CB      R2,R3       ;Compares the leftmost bytes of Workspace
                            Register 2 and Workspace Register 3.
```

Definition:

Compares the source operand (byte) with the destination operand (byte) and sets/resets the status bits according to the result of the comparison. The CB instruction uses the same comparison basis as does the C instruction. If the source operand contains an odd number of logic one bits, the odd parity status bit is set. The operands remain unchanged. If either operand is addressed in the Workspace Register mode, the byte addressed is the most significant byte.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

(gas) compared to (gad)

Application notes:

The CB instruction compares the two operands as signed, two's complement values, affecting the A> status bit, and as unsigned integers, affecting the L> status bit. Some examples are:

		Status Bits Set			
Source	Destination	Logical>	Arithmetic>	Equal	Odd Parity
0FFH	00H	1	0	0	0
7FH	00H	1	1	0	1
80H	7FH	1	0	0	1
7FH	7FH	0	0	1	1
7FH	80H	0	1	0	1

8.3 COMPARE IMMEDIATE--CI

Op-code: 0280 (Format VIII)

Syntax definition:

[<label>] b CI b <wa>,<iop> b [<comment>]

Example:

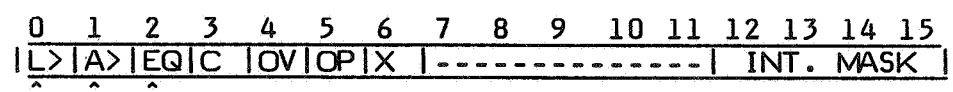
```
LABEL    CI      R3,7        ;Compares the contents of Workspace Register 3
                             to 0007H.
```

Definition:

Compares the contents of the specified Workspace Register with the word in memory immediately following the instruction and sets/resets the status bits according to the comparison. The CI instruction makes the same type of comparison as does the C instruction.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(wa) compared to iop

Application notes:

The CI instruction compares the Workspace Register to an immediate operand. For example, if the contents of Workspace Register 9 is 2183H, the instruction

```
CI      R9,F330H
```

results in the arithmetic greater than status bit being set and the logical greater than and equal status bits being reset.

8.4 COMPARE ONES CORRESPONDING--COC

Op-code: 2000 (Format III)

Syntax definition:

[<label>] b COC b <gas>,<wad> b [<comment>]

Example:

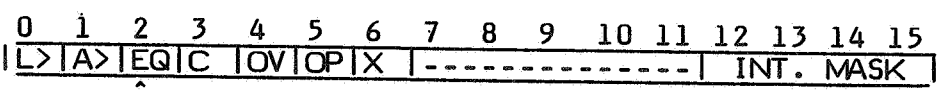
```
LABEL   COC     @MASK,R2  ;Compares the contents of Workspace Register 2
                          with the contents of MASK.
```

Definition:

When the bits in the destination operand Workspace Register that correspond to the logic one bits in the source operand are equal to logic one, sets the equal status bit. The source and destination operands are unchanged.

Status bits affected:

Equal.



Execution results:

The equal bit is set if all bits of <wad> that correspond to the bits of <gas> that are equal to 1 are also equal to 1.

Application notes:

The COC instruction tests single or multiple bits within a word in a Workspace Register. For example, if TESTBI contains the word 0C102H and Workspace Register 8 contains the value 0E306H, the instruction

```
COC     @TESTBI,R8
```

sets the equal status bit because for each 1 bit in the first operand there is a 1 bit in the corresponding bit position of the second operand as shown below.

```
0C102H = 1100 0001 0000 0010 and
0E306H = 1110 0011 0000 0110
```

If Workspace Register 8 contains 0E301H, the equal status bit is reset. Use this instruction to determine if a Workspace Register has 1s in the bit positions indicated by the 1s in a mask.

8.5 COMPARE ZEROS CORRESPONDING--CZC

Op-code: 2400 (Format III)

Syntax definition:

[<label>] b CZC b <gas>,<wad> b [<comment>]

Example:

```
LABEL    CZC     @MASK,R2  ;Compares the contents of Workspace Register 2
                           with the contents of MASK.
```

Definition:

When the bits in the destination operand Workspace Register that correspond to the one bits in the source operand are all equal to logic zero, sets the equal status bit. The source and destination operands are unchanged.

Status bits affected:

Equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
      ^
```

Execution results:

The equal bit is set if all bits of <wad> that correspond to the bits of <gas> that are equal to 1 are equal to 0.

Application notes:

The CZC instruction tests single or multiple bits within a word in a Workspace Register. For example, if the memory location labeled TESTBI contains the value 0C102H, and Workspace Register 8 contains 2301H, the instruction

```
CZC     @TESTBI,R8
```

resets the equal status bit because for each 1 bit in the first operand there is not a corresponding zero bit in the corresponding bit position of the second operand as shown below.

```
0C102H = 1100 0001 0000 0010 and
 2301H = 0010 0011 0000 0001
```

If Workspace Register 8 contains the value 2201H, then the equal status bit is set. Use the CZC instruction to determine if a Workspace Register has zeros in the positions indicated by ones in a mask.

SECTION 9: CONTROL AND CRU INSTRUCTIONS

The following control and CRU instructions are described in this section.

Instruction	Mnemonic	Section
LoaD CRU	LDCR	9.1
Set CRU Bit to One	SBO	9.2
Set CRU Bit to Zero	SBZ	9.3
STore CRU	STCR	9.4
Test Bit	TB	9.5

The following instructions are described in Section 9.6. All of them are properly assembled and are recognized by the TMS9900 microprocessor, but they should not be used on the Home Computer.

Instruction	Mnemonic
ClocK OFf	CKOF
ClocK ON	CKON
IDLE	IDLE
ReSET	RSET
Load or REstart eXecution	LREX

Examples are given in Section 9.7.

Control instructions affect the operation of the Arithmetic Unit (AU) and the associated portions of the computer or microprocessor. CRU instructions affect the modules connected to the Communications Register Unit.

For CRU bit instructions, the signed displacement is shifted one bit position to the left and added to the contents of Workspace Register 12. In other words, it is a displacement in bits from the contents of bits 3 through 14 of Workspace Register 12.

CONTROL AND CRU INSTRUCTIONS

Each instruction's description consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

9.1 LOAD CRU--LDCR

Op-code: 3000 (Format IV)

Syntax definition:

[<label>] b LDCR b <gas>,<cnt> b [<comment>]

Example:

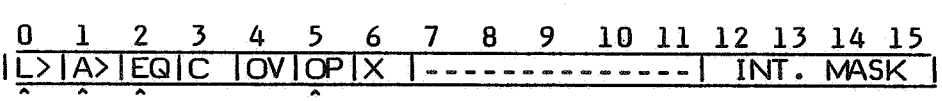
```
WRITE   LDCR   @BUFF,15   ;Sends 15 bits from BUFF to the CRU.
```

Definition:

Transfers the number of bits specified in the cnt field from the source operand to the CRU. The transfer begins with the least significant bit of the source operand. The CRU address is contained in bits 3 through 14 of Workspace Register 12. When the cnt field contains zero, the number of bits transferred is 16. If the number of bits to be transferred is from one to eight, the source operand address is a byte address. If the number of bits to be transferred is from 9 to 16, the source operand address is a word address. If the source operand address is odd, the address is truncated to an even address prior to data transfer. When the number of bits transferred is a byte or less, the source operand is compared to zero and the status bits are set/reset, according to the results of the comparison. The odd parity status bit is set when the bits in a byte (or less) to be transferred establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, and equal. When cnt is less than nine, odd parity is also set or reset. The odd parity status bit is set according to the full word or byte, not just the transferred bits.



Execution results:

The number of bits specified by cnt are transferred from memory at address gas to consecutive CRU lines beginning at the address in Workspace Register 12 (bits 3 through 14).

9.2 SET CRU BIT TO ONE--SBO

Op-code: 1D00 (Format II)

Syntax definition:

[<label>] b SBO b <disp> b [<comment>]

Example:

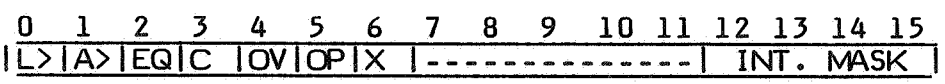
```
LABEL   SBO     7       ;Sets CRU bit 7, relative to the CRU base in
                        Workspace Register 12, to one.
```

Definition:

Sets the digital output bit to one on the CRU at the address derived from this instruction. The derived address is the sum of the signed displacement and the contents of Workspace Register 12, bits 3 through 14. The execution of this instruction does not affect the Status Register or the contents of Workspace Register 12.

Status bits affected:

None.



Execution results:

A CRU bit is set to one. The CRU bit equals the sum of the contents of Workspace Register 12 (bits 3 through 14) and the displacement.

9.3 SET CRU BIT TO ZERO--SBZ

Op-code: 1E00 (Format II)

Syntax definition:

[<label>] b SBZ b <disp> b [<comment>]

Example:

```
LABEL   SBZ     7           ;Sets CRU bit 7, relative to the CRU base in
                            Workspace Register 12, to zero.
```

Definition:

Sets the digital output bit to zero on the CRU at the address derived from this instruction. The derived address is the sum of the signed displacement and the contents of Workspace Register 12, bits 3 through 14. The execution of this instruction does not affect the Status Register or the contents of Workspace Register 12.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

A CRU bit is set to zero. The CRU bit equals the sum of the contents of Workspace Register 12 (bits 3 through 14) and the displacement.

9.4 STORE CRU--STCR

Op-code: 3400 (Format IV)

Syntax definition:

[<label>] b STCR b <gas>,<cnt> b [<comment>]

Example:

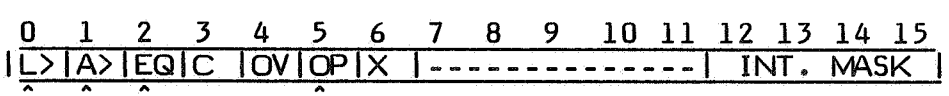
```
READ    STCR    @BUF,9      ;Reads 9 bits from the CRU and stores them at
                            location BUF.
```

Definition:

Transfers the number of bits specified in the cnt field from the CRU to the source operand. The transfer begins from the CRU address specified in bits 3 through 14 of Workspace Register 12 to the least significant bit of the source operand and fills the source operand toward the most significant bit. When the cnt field contains a zero, the number of bits to transfer is 16. If the number of bits to transfer is from one to eight, the source operand address is a byte address. Any bit in the memory byte not filled by the transfer is set to zero. When the number of bits to transfer is from 9 to 16, the source operand address is a word address. If the source operand address is odd, the address is truncated to an even address prior to data transfer. If the transfer does not fill the entire memory word, unfilled bits are set to zero. When the number of bits to transfer is a byte or less, the bits transferred are compared to zero and the status bits are set or reset to indicate the results of the comparison. Also, when the bits to be transferred are a byte or less, the odd parity bit is set when the bits establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, and equal. When cnt is less than 9, odd parity is also set or reset. Status is set according to the full word or byte, not just the transferred bits.



Execution results:

The number of bits specified by cnt are transferred from consecutive CRU lines beginning at the address in Workspace Register 12 (bits 3 through 14) to memory at address gas.

Application notes:

The STCR instruction transfers a specified number of CRU bits from the CRU to the memory location specified as the source operand. Note that the CRU base address must be in Workspace Register 12 (bits 3 through 14) prior to the execution of this instruction.

9.5 TEST BIT--TB

Op-code: 1F00 (Format II)

Syntax definition:

[<label>] b TB b <disp> b [<comment>]

Example:

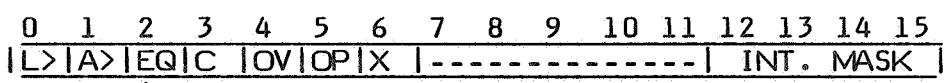
```
CHECK   TB      7       ;Reads CRU bit 7 relative to the CRU base
                        address in Workspace Register 12, and sets the
                        equal status bit to the value read.
```

Definition:

Reads the digital input bit on the CRU at the address specified by the sum of the signed displacement and the contents of Workspace Register 12, bits 3 through 14, and set the equal status bit to the value read. The digital input bit and the contents of Workspace Register 12 are unchanged.

Status bits affected:

Equal.



Execution results:

Equal bit is set to the value of the CRU bit addressed by the sum of the contents of Workspace Register 12 (bits 3 through 14) and the displacement.

Application notes:

The TB instruction transfers the level from the indicated CRU line to the equal status bit without modification. If the CRU line tested is set to one, the equal status bit is set to one; if the line is zero, it is set to zero. The JEQ instruction can then be used to transfer control when the CRU line is one and not transfer control when the line is zero. In addition, the JNE instruction transfers control under the opposite conditions.

9.6 OTHER INSTRUCTIONS

The following instructions are properly assembled and are recognized by the TMS9900 microprocessor, but they should not be used on the Home Computer. Their op-code and syntax definition are given below.

Instruction	Mnemonic	Op-code	Format	Syntax definition
ClocK OFf	CKOF	03C0	VII	[<label>] b CKOF b [<comment>]
ClocK ON	CKON	03A0	VII	[<label>] b CKON b [<comment>]
IDLE	IDLE	0340	VII	[<label>] b IDLE b [<comment>]
ReSET	RSET	0360	VII	[<label>] b RSET b [<comment>]
Load or REstart eXecution	LREX	03E0	VII	[<label>] b LREX b [<comment>]

9.7 CRU INPUT/OUTPUT

The Communications Register Unit (CRU) performs single and multiple bit programmed input/output. All input consists of reading CRU line logic levels into memory and output consists of setting CRU output lines to bit values from a word or byte of memory. The CRU provides a maximum of 4096 input and output lines that may be individually selected by a 12-bit address. The 12-bit address is located in bits 3 through 14 of Workspace Register 12 and is the base address for all CRU communications.

9.7.1 CRU I/O Instructions

There are five instructions for communications with CRU lines.

SBO Set CRU Bit to One. This instruction sets a CRU output line to one.

SBZ Set CRU Bit to Zero. This instruction sets a CRU output line to zero.

TB Test CRU Bit. This instruction reads the digital input bit and sets the equal status bit (bit 2) to the value of the digital input bit.

LDCR Load Communications Register. This instruction transfers the number of bits (1-16) specified by the cnt field of the instruction to the CRU from the source operand. When less than nine bits are specified, the source operand address is a byte address. When nine or more bits are specified, the source operand is a word address. The CRU address is the address of the first CRU digital output affected. The CRU address is determined by the contents of Workspace Register 12, bits 3 through 14.

STCR Store Communications Register. This instruction transfers the number of bits specified by the cnt field of the instruction from the CRU to the source operand. When less than nine bits are specified, the source operand address is a byte address. When nine or more bits are specified, the source operand address is a word address. The CRU address is determined by Workspace Register 12, bits 3 through 14.

9.7.2 Accessing Specific Bits

There are many different ways to access the same CRU bit. For instance, if Workspace Register 12 contains 0100H, making the base address in bits 3 through 14 equal to 80H, the following instruction sets CRU line 85H to one.

```
SBO     5
```

If Workspace Register 12 contains 010AH, making the base address in bits 3 through 14 equal to 85H, the following instruction also sets CRU line 85H to one.

```
SBO     0
```

9.7.3 SBO Example

Assume that a control device turns on a motor when the computer sets a one on CRU line 10FH and that Workspace Register 12 contains 0200H, making the base address in bits 3 through 14 equal to 100H. The following instruction sets CRU line 10FH to one.

```
SBO     15
```

9.7.4 SBZ Example

Assume that a control device shuts off a valve when the computer sets a zero on a CRU line that is connected to CRU line 2 and that Workspace Register 12 contains zero. The following instruction sets CRU line 2 to zero.

```
SBZ     2
```

9.7.5 TB Example

Assume that Workspace Register 12 contains 0140H, making the base address in bits 3 through 14 equal to 0A0H. The following instructions test the input on CRU line 0A4H and execute the instructions beginning at location RUN when the CRU line is set to one. When the CRU line is set to zero, the instructions beginning at location WAIT are executed.

```
        TB      4       ;Test CRU line 4.
        JEQ     RUN     ;If on, go to RUN.
WAIT    .               ;If off, continue.
        .
RUN     .
```

The TB instruction sets the equal bit of the Status Register to the level on line 4 of the CRU device.

SECTION 10: LOAD AND MOVE INSTRUCTIONS

The following load and move instructions are described in this section.

Instruction	Mnemonic	Section
Load Immediate	LI	10.1
Load Interrupt Mask Immediate	LIMI	10.2
Load Workspace Pointer Immediate	LWPI	10.3
MOVe words	MOV	10.4
MOVe Bytes	MOVB	10.5
STore STatus	STST	10.6
STore Workspace Pointer	STWP	10.7
SWaP Bytes	SWPB	10.8

An example is given in Section 10.9.

Load and move instructions permit you to establish the execution environment and the execution results. These instructions manipulate data between memory locations and between hardware registers and memory locations.

Each instruction's description consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

LOAD AND MOVE INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

10.1 LOAD IMMEDIATE--LI

Op-code: 0200 (Format VIII)

Syntax definition:

[<label>] b LI b <wa>,<iop> b [<comment>]

Example:

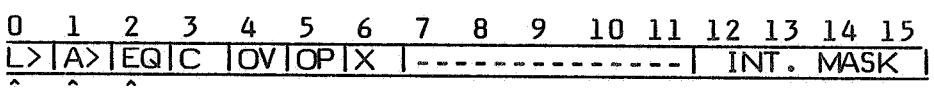
```
GETIT    LI      3,17H       ;Loads Workspace Register 3 with 0017H.
```

Definition:

Places the immediate operand (the word of memory immediately following the instruction) in the Workspace Register (W field). The immediate operand is not affected by the execution of this instruction. The immediate operand is compared to 0 and the logical greater than, arithmetic greater than, and equal status bits are set or reset according to the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(iop) => (wa)

Application notes:

The LI instruction places an immediate operand in a specified Workspace Register. This may be used to initialize a Workspace Register as a loop counter. For example, the instruction

```
        LI      7,5
```

initializes Workspace Register 7 with the value 5. In this example, the logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

10.2 LOAD INTERRUPT MASK IMMEDIATE--LIMI

Op-code: 0300 (Format VIII)

Syntax definition:

[<label>] b LIMI b <iop> b [<comment>]

Example:

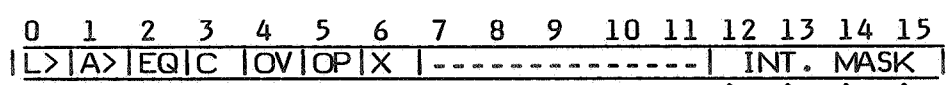
```
LABEL    LIMI    2              ;Masks level 2 and below.
```

Definition:

Places the least significant four bits (bits 12-15) of the contents of the immediate operand (the next word after the instruction) in the interrupt mask of the Status Register. The remaining bits of the Status Register (0 through 11) are not affected.

Status bits affected:

Interrupt mask.



Execution results:

Places the four least significant bits of iop into the interrupt mask.

Application notes:

The LIMI instruction initializes the interrupt mask so that a particular level of interrupt is accepted. For example, the instruction

```
LIMI    2
```

sets the interrupt mask to level two and enables interrupts at levels 0, 1, and 2.

The instruction

```
LIMI    0
```

Disables all interrupts and is the normal state of the computer.

Note: The p-System is not designed to allow interrupts, and assembly language programs that enable interrupts probably cannot return to the calling program or to the System.

10.3 LOAD WORKSPACE POINTER IMMEDIATE--LWPI

Op-code: 02E0 (Format VIII)

Syntax definition:

[<label>] b LWPI b <iop> b [<comment>]

Example:

```
NEWWP   LWPI    02F2H       ;Sets NEWWP equal to 02F2H.
```

Definition:

Replaces the contents of the Workspace Pointer with the immediate operand.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(iop) => (WP)

Application notes:

The LWPI instruction initializes or changes the Workspace Pointer Register to alter the Workspace environment of the program. You may use a BLWP or a LWPI instruction to load your own Workspace Registers.

10.4 MOVE WORD--MOV

Op-code: C000 (Format I)

Syntax definition:

[<label>] b MOV b <gas>,<gad> b [<comment>]

Example:

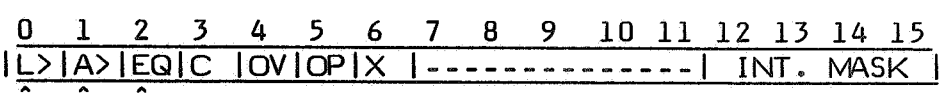
```
GET     MOV     @WD,R2      ;Moves a copy of WD into Workspace Register 2.
```

Definition:

Replaces the destination operand with a copy of the source operand. The computer compares the resulting destination operand to zero and sets/resets the status bits according to the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(gas) => (gad)

Application notes:

The MOV instruction moves 16-bit words as follows:

Memory-to-memory (non-register)
Load register (memory-to-register)
Register-to-register
Register-to-memory

The MOV instruction may also be used to compare a memory location to zero. For example,

```
MOV     R7,R7
JNE     TEST
```

moves Workspace Register 7 to itself and compares the contents of Workspace Register 7 to zero. If the contents are not equal to zero, the equal status bit is reset and control transfers to TEST.

As another example of the use of MOV, assume that Workspace Register 9 contains 3416H and location ONES contains 0FFFFH. Then

```
MOV     @ONES,R9
```

changes the contents of Workspace Register 9 to 0FFFFH, while the contents of location ONES is not changed. For this example, the logical greater than status bit is set and the arithmetic greater than and equal status bits are reset.

10.5 MOVE BYTE--MOVB

Op-code: D000 (Format I)

Syntax definition:

[<label>] b MOVB b (gas),(gad) b [<comment>]

Example:

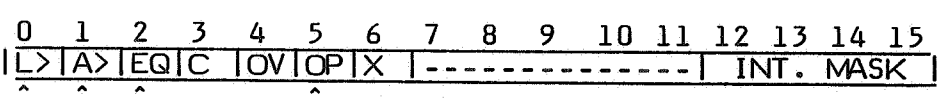
```
NEXT    MOVB   R2,2A41H   ;Stores the most significant byte of Workspace
                          Register 2 in address 2A41H.
```

Definition:

Replaces the destination operand (byte) with a copy of the source operand (byte). If either operand is addressed in the Workspace Register mode, the byte addressed is the most significant byte. The least significant byte is not affected. The computer compares the destination operand to zero and sets/resets the status bits to indicate the result of the comparison. The odd parity bit is set when the bits in the destination operand establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

(gas) => (gad)

Application notes:

The MOVB instruction moves bytes in the same combinations as the MOV instruction moves words. For example, if memory location 1C14H contains a value of 2016H and TEMP is located at 1C15H, and if Workspace Register 3 contains 542BH, the instruction

```
MOVB   @TEMP,R3
```

changes the contents of Workspace Register 3 to 162BH. The logical greater than, arithmetic greater than, and odd parity status bits are set, while the equal status bit is reset.

10.6 STORE STATUS--STST

Op-code: 02C0 (Format VIII)

Syntax definition:

[<label>] b STST b (wa) b [<comment>]

Example:

```
LABEL    STST    R7          ;Stores status in Workspace Register 7.
```

Definition:

Stores the Status Register contents in the specified Workspace Register.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(ST) => (wa)

Application notes:

The STST instruction stores the Status Register in the specified Workspace Register.

10.7 STORE WORKSPACE POINTER--STWP

Op-code: 02A0 (Format VIII)

Syntax definition:

[<label>] b STWP b (wa) b [<comment>]

Example:

```
LABEL   STWP   R6        ;Stores the Workspace Pointer in Workspace
                         Register 6.
```

Definition:

Places a copy of the Workspace Pointer contents in the specified Workspace Register.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(WP) => (wa)

Application notes:

The STWP instruction stores the contents of the Workspace Pointer in the specified Workspace Register.

10.8 SWAP BYTES--SWPB

Op-code: 06C0 (Format VI)

Syntax definition:

[<label>] b SWPB b (gas) b [<comment>]

Example:

```
SWITCH   SWPB    R3          ;Switches the most significant and least
                             significant bytes in Workspace Register 3.
```

Definition:

Replaces the most significant byte (bits 0-7) of the source operand with a copy of the least significant byte (bits 8-15) of the source operand and replaces the least significant byte with a copy of the most significant byte.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

Exchanges left and right bytes of word (gas).

Application notes:

Use the SWPB instruction to interchange bytes of an operand prior to executing various byte instructions. For example, if Workspace Register 0 contains 2144H and memory location 2144H contains the value 0F312H, the instruction

```
SWPB    *R0+
```

changes the contents of memory location 2144H to 12F3H and increments Workspace Register 0 to 2146H. The Status Register is unchanged.

10.9 INSTRUCTION EXAMPLE

The following program segment illustrates the use of many of the instructions discussed in this section. The first six instructions are a portion of a program that calls a subroutine labeled SUBR. The calling program performs some initialization and then calls the subroutine to check the contents of a 20-byte buffer. If the subroutine finds that the buffer contains byte values that are in numerically sequential order, then it returns 00H to the calling program in Workspace Register 4. If the bytes are not in numerically sequential order, the subroutine returns 01H in Workspace Register 4. The program and subroutine are described in greater detail after the program is listed.

```
        LWPI    0AC20H      ;Load Workspace Pointer.
        BL      @SUBR       ;Call subroutine.
        .WORD   BUFFER      ;Address of BUFFER.
        MOV     R4,R4       ;See if numbers are in sequence.
        JNE     NOSEQ       ;Jump if subroutine found non-sequential
                            numbers.
        .
        .
        .
SUBR    S       R4,R4       ;Clear Workspace Register 4.
        LI      R10,20      ;Put loop count in Workspace Register 10.
        MOV     *R11+,R7    ;Put address of BUFFER in Workspace Register
                            7.
        MOVB    *R7,R6      ;Put first number in the left byte of Workspace
                            Register 6.
CHECK   MOV     *R7+,RT8    ;Put two bytes in Workspace Register 8.
        SB      R6,R8       ;Check for sequence.
        JNE     OUT         ;Jump if out of sequence.
        AI      R6,100H     ;Add one to sequence checker.
        SWPB    R8          ;Put other byte in left half of register.
        SB      R6,R8       ;Check for sequence.
        JNE     OUT         ;Jump if out of sequence.
        AI      R6,100H     ;Add one to sequence checker.
        DECT    R10         ;Decrement loop counter.
        JGT     CHECK       ;Jump to check next two bytes.
        JMP     RETURN      ;Through checking, all in order.
OUT     INC     R4          ;Set Workspace Register 4 to a non-zero value.
RETURN  B       *R11        ;Return to calling program.
```

The BL instruction transfers program control to the subroutine with the address following the BL instruction placed in Workspace Register 11 to allow for return to the program. The location following the BL instruction contains the address of the 20-byte buffer to be checked by the subroutine. The subroutine returns control to the MOV instruction in the calling program, which then checks to see if the subroutine found the bytes in numerically sequential order and jumps to location NOSEQ (not shown) if they were not.

The subroutine clears Workspace Register 4 with the S instruction and puts a loop counter value of 20 in Workspace Register 10 with the LI instruction.

Since Workspace Register 11 contains the address of the location following the BL instruction in the calling program, the MOV *R11+,R7 instruction copies the address of BUFFER into Workspace Register 7 and increments the address in Workspace Register 11 to the location following the .WORD directive, setting the address to the MOV instruction for the return when the subroutine is finished. The MOVB *R7,R6 instruction copies the first byte value into the left byte of Workspace Register 6.

At label CHECK, the MOV instruction begins a loop that copies a word (two bytes) into Workspace Register 8 and auto-increments the address in Workspace Register 7 to the next word in the buffer. The left byte of Workspace Register 8 is subtracted from its right byte. A non-zero result indicates an out of sequence number and the JNE instruction transfers control to the instruction labeled OUT which places a 01H in Workspace Register 4.

If the subtraction produces a zero result, the AI 6,100H instruction increments the contents of Workspace Register 6 to the next byte to be checked. The following SWPB instructon swaps the bytes in Workspace Register 8 so the following SB and JNE instructions can check the sequence. If the sequence is correct, the next AI instruction updates Workspace Register 6 to the address of the next byte.

The DECT instruction decrements the loop counter in Workspace Register 10 by two since two bytes have been checked. If the result is non-zero, there are more bytes to be checked and the JGT instruction causes a reiteration of the loop. If the result is zero, all 20 bytes have been checked and the JMP instruction causes a jump to the subroutine's exit at RETURN. There the B *R11 instruction causes a return to the calling program.

SECTION 11: LOGICAL INSTRUCTIONS

The following logical instructions are described in this section.

Instruction	Mnemonic	Section
AND Immediate	ANDI	11.1
OR Immediate	ORI	11.2
EXclusive OR	XOR	11.3
INVert	INV	11.4
CLeaR	CLR	11.5
SET to One	SETO	11.6
Set Ones Corresponding	SOC	11.7
Set Ones Corresponding, Byte	SOCB	11.8
Set Zeros Corresponding	SZC	11.9
Set Zeros Corresponding, Byte	SZCB	11.10

Logical instructions permit you to perform various logical operations on memory locations and/or Workspace Registers.

Each instruction's description consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

LOGICAL INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field. Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

11.1 AND IMMEDIATE--ANDI

Op-code: 0240 (Format VIII)

Syntax definition:

[<label>] b ANDI b (wa),(iop) b [<comment>]

Example:

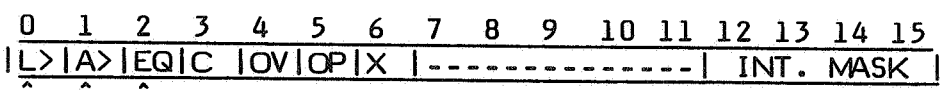
```
LABEL   ANDI   R3,0FFF0H   ;Sets least significant 4 bits of Workspace
                           Register 3 to zeros.
```

Definition:

Performs a bit-by-bit AND operation on the 16 bits in the immediate operand and the corresponding bits of the Workspace Register. The immediate operand is the word in memory immediately following the instruction word. Place the result in the Workspace Register. The computer compares the result to zero and sets/resets the status bits according to the results of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(wa) AND iop => (wa)

Application notes:

The ANDI instruction performs a logical AND with an immediate operand and a Workspace Register. Each bit of the 16-bit word of both operands follows the following table.

Immediate Operand Bit	Workspace Register Bit	AND Result
0	0	0
0	1	0
1	0	0
1	1	1

For example, if Workspace Register 0 contains 0D2ABH, the instruction

```
ANDI    R0,6D03H
```

results in Workspace Register 0 changing to 4003H. This AND operation on a bit-by-bit basis is

```
0110 1101 0000 0011   (Immediate operand--6D03)
1101 0010 1010 1011   (Workspace Register 0--D2AB)
---------------------
0100 0000 0000 0011   (Workspace Register 0 result--4003)
```

In this example, the logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

11.2 OR IMMEDIATE--ORI

Op-code: 0260 (Format VIII)

Syntax definition:

[<label>] b ORI b (wa),(iop) b [<comment>]

Example:

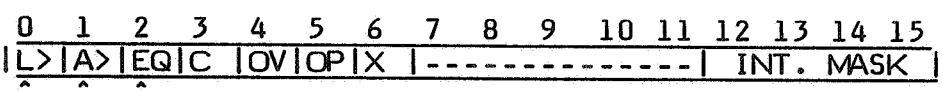
```
LABEL    ORI     R3,0F000H   ;Sets the most significant 4 bits of Workspace
                             Register 3 to ones.
```

Definition:

Performs a logical OR operation on the 16-bit immediate operand and the corresponding bits of the Workspace Register. The immediate operand is the memory word immediately following the ORI instruction. Place the result in the Workspace Register. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(wa) OR (iop) => (wa)

Application notes:

The ORI instruction performs a logical OR with the immediate operand and a specified Workspace Register. Each bit of the 16-bit word of both operands follows the following table.

Immediate Operand Bit	Workspace Register Bit	ORI Result
0	0	0
1	0	1
0	1	1
1	1	1

For example, if Workspace Register 5 contains 0D2ABH, the instruction

```
ORI     R5,6D03H
```

results in Workspace Register 5 changing to 0FFABH. This OR operation on a bit-by-bit basis is

```
0110 1101 0000 0011  (Immediate operand--6D03H)
1101 0010 1010 1011  (Workspace Register 5--0D2ABH)
--------------------
1111 1111 1010 1011  (Workspace Register 5 result--0FFABH)
```

In this example, the logical greater than status bit is set, and the arithmetic greater than and equal status bit are reset.

11.3 EXCLUSIVE OR--XOR

Op-code: 2800 (Format III)

Syntax definition:

[<label>] b XOR b (gas),(wad) b [<comment>]

Example:

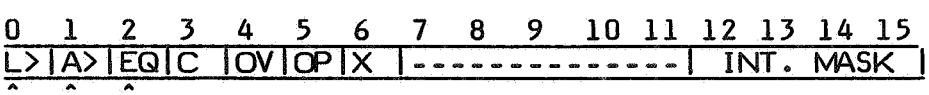
```
LABEL    XOR     @WORD,R3  ;Exclusive ORs the contents of WORD and
                           Workspace Register 3.
```

Definition:

Performs a bit-by-bit exclusive OR of the source and destination operands, and replaces the destination operand with the result. The exclusive OR is accomplished by setting the bits in the resultant destination operand to one when the corresponding bits of the two operands are not equal. The bits in the resultant destination operand are reset to zero when the corresponding bits of the two operands are equal. The computer compares the resultant destination operand to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(gas) XOR (wad) => (wad)

Application notes:

The XOR instruction performs an exclusive OR on two word operands. Each bit of the 16-bit word of both operands follows the table on the next page.

Immediate Operand Bit	Workspace Register Bit	XOR Result
0	0	0
0	1	1
1	0	1
1	1	0

For example, if Workspace Register 2 contains 0D2AAH and location CHANGE contains the value 6D03H, the instruction

```
XOR     @CHANGE,R2
```

results in the contents of Workspace Register 2 changing to 0BFA9H. Location CHANGE remains 6D03H. This is shown as follows.

```
0110 1101 0000 0011  (Source operand--6D03H)
1101 0010 1010 1010  (Destination operand--0D2AAH)
---------------------
1011 1111 1010 1001  (Destination operand result--0BFA9H)
```

In this example, the logical greater than status bit is set, while the arithmetic greater than and equal status bits are reset.

11.4 INVERT--INV

Op-code: 0540 (Format VI)

Syntax definition:

[<label>] b INV b (gas) b [<comment>]

Example:

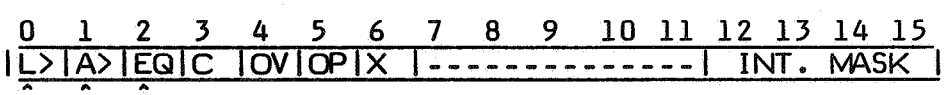
```
COMPL   INV     @BUFF(R2)   ;Replaces the value at the address found by
                            adding the value of Workspace Register 2 to the
                            contents of BUFF with the one's complement of
                            the data.
```

Definition:

Replaces the source operand with the one's complement of the source operand. The one's complement is equivalent to changing each zero in the source operand to one and each one in the source operand to zero. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

The one's complement of (gas) is placed in (gas).

Application notes:

The INV instruction changes each zero in the source operand to one and each one to zero. For example, if Workspace Register 11 contains 157AH, the instruction

```
INV     R11
```

changes the contents of Workspace Register 11 to 0EA85H. This INV operation on a bit-by-bit basis is

```
0001 0101 0111 1010(Workspace Register 11--157AH)
1110 1010 1000 0101(Workspace Register 11 result--0EA85H)
```

The logical greater than status bit is set and the arithmetic greater than and equal status bits are reset.

11.5 CLEAR--CLR

Op-code: 04C0 (Format VI)

Syntax definition:

[<label>] b CLR b (gas) b [<comment>]

Example:

```
PRELM   CLR     @BUFF(R2)   ;Clears the value at the address found by adding
                            the value of Workspace Register 2 to the
                            contents of BUFF.
```

Definition:

Replaces the source operand with a full 16-bit word of zeros.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

0 => (gas)

Application notes:

The CLR instruction sets a full, 16-bit, memory-addressable word to zero. For example, if Workspace Register 11 contains the value 2000H1, the instruction

```
        CLR     *0BH        ;0BH is equivalent to R11.
```

results in the contents of memory locations 2000H and 2001H being set to 0. Workspace Register 11 and the Status Register are unchanged. Word operations, such as CLR, operate on the next lower address when an odd address is given as the operand.

11.6 SET TO ONE--SETO

Op-code: 0700 (Format VI)

Syntax definition:

[<label>] b SETO b (gas) b [<comment>]

Example:

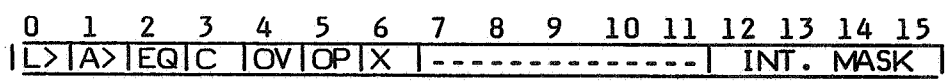
```
LABEL   SETO    R3          ;Sets Workspace Register 3 to 0FFFFH or
                            negative 1.
```

Definition:

Replaces the source operand with a full 16-bit word of ones.

Status bits affected:

None.



Execution results:

0FFFFH => <gas>

Application notes:

The SETO instruction initializes an addressable memory to a value of negative 1. For example, the instruction

```
        SETO    R3
```

initializes Workspace Register 3 to a value of 0FFFFH. The contents of the Status Register are unchanged. This is a useful means of setting flag words.

11.7 SET ONES CORRESPONDING--SOC

Op-code: E000 (Format I)

Syntax definition:

[<label>] b SOC b (gas),(gad) b [<comment>]

Example:

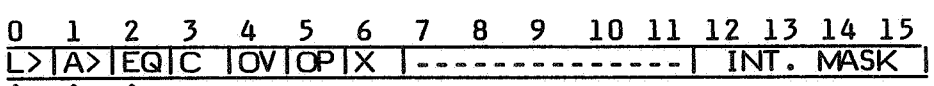
```
LABEL   SOC     R3,R2       ;ORs Workspace Register 3 into Workspace
                            Register 2.
```

Definition:

Sets to one the bits in the destination operand that correspond to the one bits in the source operand. Leaves unchanged the bits in the destination operand that are in the same bit positions as the zero bits in the source operand. This operation is an OR of the two operands. The changed destination operand replaces the original destination operand. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 1.

Application notes:

The SOC instruction ORs the 16-bit contents of two operands. For example, if Workspace Register 3 contains 0FF00H and location NEW contains 0AAAAH, the instruction

```
SOC     R3,@NEW
```

changes the contents of location NEW to 0FFAAH, while the contents of Workspace Register 3 are unchanged. This SOC operation on a bit-by-bit basis is

```
1111 1111 0000 0000  (Source operand--0FF00H)
1010 1010 1010 1010  (Destination operand--0AAAAH)
---------------------
1111 1111 1010 1010  (Destination operand result--0FFAAH)
```

In this example, the logical greater than status bit is set and the arithmetic greater than and equal status bits are reset.

11.8 SET ONES CORRESPONDING, BYTE--SOCB

Op-code: F000 (Format I)

Syntax definition:

[<label>] b SOCB b (gas),(gad) b [<comment>]

Example:

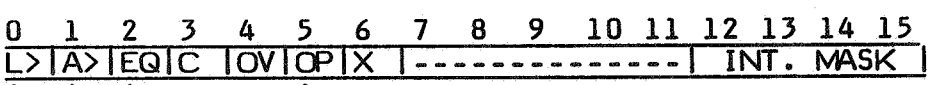
```
LABEL   SOCB   R3,@DET   ;ORs Workspace Register 3 into the byte at
                         location DET.
```

Definition:

Sets to one the bits in the destination operand that correspond to the one bits in the source operand byte. Leaves unchanged the bits in the destination operand that are in the same bit positions as the zero bits in the source operand byte. This operation is an OR of the two operand bytes. The changed destination operand byte replaces the original destination operand byte. The computer compares the resulting destination operand byte to zero and sets/resets the status bits to indicate the results of the comparison. The odd parity status bit is set when the bits in the resulting byte establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 1.

Application notes:

The SOCB instruction ORs two byte operands. For example, if Workspace Register 5 contains 0F013H and Workspace Register 8 contains the value 0AA24H, the instruction

```
SOCB    R3,R8
```

changes the contents of Workspace Register 8 to 0FA24H, while the contents of Workspace Register 5 are unchanged. This SOCB operation on a bit-by-bit basis is

```
1111 0000 0001 0011   (Source operand--0F013H)
1010 1010 0010 0100   (Destination operand--0AA24H)
---------------------
1111 1010 0010 0100   (Destination operand result--0FA24H)
 ----------
 (Unchanged)
```

In this example, the logical greater than status bit is set, while the arithmetic greater than, equal, and odd parity status bits are reset.

11.9 SET ZEROS CORRESPONDING--SZC

Op-code: 4000 (Format I)

Syntax definition:

[<label>] b SZC b (gas),(gad) b [<comment>]

Example:

```
LABEL   SZC     @MASK,R2  ;Resets the bits of Workspace Register 2 as
                          indicated by MASK.
```

Definition:

Sets to zero the bits in the destination operand that correspond to the bit positions equal to one in the source operand. This operation is effectively an AND operation of the destination operand and the one's complement of the source operand. The computer compares the resulting destination operand to zero and sets/resets the status bits to indicate the results of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^
```

Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 0.

Application notes:

The SZC instruction turns off flag bits or ANDs the destination operand. For example, if Workspace Register 5 contains 6D03H and Workspace Register 3 contains 0D2AAH, the instruction

```
SZC     R5,R3
```

changes the contents of Workspace Register 3 to 92A8H, while the contents of Workspace Register 5 remain unchanged. This SCZ operation on a bit-by-bit basis is

```
0110 1101 0000 0011   (Source operand--6D03H)
1101 0010 1010 1010   (Destination operand--0D2AAH)
---------------------
1001 0010 1010 1000   (Destination operand result--92A8H)
```

In this example, the logical greater than status bit is set, while the arithmetic greater than and equal status bits are reset.

11.10 SET ZEROS CORRESPONDING, BYTE--SZCB

Op-code: 5000 (Format I)

Syntax definition:

[<label>] b SZCB b (gas),(gad) b [<comment>]

Example:

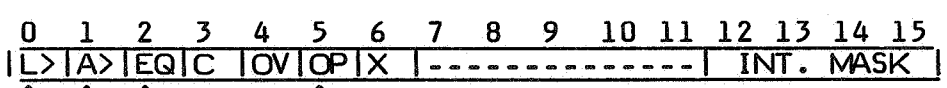
```
LABEL   SZCB    @MASK,@CHAR         ;Resets the bits of CHAR as
                                     indicated by MASK.
```

Definition:

Sets to zero the bits in the destination operand byte that correspond to the bit positions equal to one in the source operand byte. This operation is effectively an AND operation of the destination operand byte and the one's complement of the source operand byte. The computer compares the resulting destination operation to zero and sets/resets the status bits to indicate the results of the comparison. The odd parity status bit is set when the bits in the resulting destination operand byte establish odd parity. When the destination operand is given as a Workspace Register, the most significant byte is the one affected.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 0.

Application notes:

The SZCB instruction is used for the same applications as SZC except that bytes are used instead of words. For example, if location BITS contains the value 0F018H and location TESTVA contains the value 0AA24H, the instruction

```
SZCB    @BITS,@TESTVA
```

changes the contents of TESTVA to 0A24H, while BITS remains unchanged. This is shown as

```
1111 0000 0001 1000   (Source operand--0F018H)
1010 1010 0010 0100   (Destination operand--0AA24H)
---------------------
0000 1010 0010 0100   (Destination operand result--0A24H)
 ----------
 (Unchanged)
```

In this example, the logical greater than and arithmetic greater than status bits are set, while the equal and odd parity status bits are reset.

SECTION 12: WORKSPACE REGISTER SHIFT INSTRUCTIONS

The following Workspace Register shift instructions are described in this section.

Instruction	Mnemonic	Section
Shift Right Arithmetic	SRA	12.1
Shift Right Logical	SRL	12.2
Shift Left Arithmetic	SLA	12.3
Shift Right Circular	SRC	12.4

An example is given in Section 12.5.

Workspace Register shift instructions permit you to shift the contents of a specified Workspace Register from one to 16 bits. For each of these instructions, if the shift count in the instruction is zero, the shift count is taken from Workspace Register 0, bits 12 through 15. If the four bits of Workspace Register 0 are equal to zero, the shift count is 16 bit positions. The value of the last bit shifted out of the Workspace Register is placed in the carry bit of the Status Register. The result is compared to zero, and the results of the comparison are shown in the logical greater than, arithmetic greater than, and equal bits in the Status Register. If a shift count greater than 15 is supplied, the Assembler fills in the four-bit field with the least significant four bits of the shift count.

Each instruction's description consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

WORKSPACE REGISTER SHIFT INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

12.1 SHIFT RIGHT ARITHMETIC--SRA

Op-code: 0800 (Format V)

Syntax definition:

[<label>] b SRA b (wa),(scnt) b [<comment>]

Example:

```
LABEL   SRA     R2,3        ;Shifts Workspace Register 2 right 3 bit
                            locations.
```

Definition:

Shifts the contents of the specified Workspace Register to the right for the specified number of bit positions. Fills vacated bit positions with the sign bit.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and carry.

```
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
|L>|A>|EQ|C |OV|OP|X |----------| INT. MASK |
 ^   ^   ^   ^
```

Execution results:

Shifts the bits of (wa) to the right, extending the sign bit to fill vacated bit positions. When scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 positions.

Application notes:

The SRA instruction shifts the given Workspace Register to the right the given number of bit positions and fills vacated positions with the sign bit. If Workspace Register 5 contains the value 8224H, and Workspace Register 0 contains the value 0F326H, the instruction

```
SRA     R5,0
```

changes the contents of Workspace Register 5 to 0FE08H. This SRA operation on a bit-by-bit basis is

```
1111 0011 0010 0110  (Workspace Register 0--0F326H. Four least
                      significant bits are 0110, so shift 6 positions)
1000 0010 0010 0100  (Workspace Register 5--8224H)
--------------------
1111 1110 0000 1000  (Workspace Register 5 result--0FE08H)
```

The logical greater than and carry status bits are set, while the arithmetic greater than and equal status bits are reset.

12.2 SHIFT RIGHT LOGICAL--SRL

Op-code: 0900 (Format V)

Syntax definition:

[<label>] b SRL b (wa),(scnt) b [<comment>]

Example:

```
LABEL   SRL     R2,7        ;Shifts Workspace Register 2 right 7 bit
                            locations.
```

Definition:

Shifts the contents of the specified Workspace Register to the right the specified number of bits. Fills the vacated bit positions with zeros.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and carry.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^
```

Execution results:

Shifts the bits of (wa) to the right, filling the vacated bit positions with zeros. If scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 bit positions.

Application notes:

The SRL instruction shifts the given Workspace Register to the right the given number of bit positions and fills vacated positions with zeros. If Workspace Register zero contains the value 0FFEFH, the instruction

```
SRL     R0,3
```

changes the contents of Workspace Register 0 to 1FFDH. This SRL operation on a bit-by-bit basis is

```
1111 1111 1110 1111   (Workspace Register 0--0FFEFH)
---------------------
0001 1111 1111 1101   (Workspace Register 0 result--1FFDH)
```

The logical greater than, arithmetic greater than and carry status bits are set, while the equal status bit is reset.

12.3 SHIFT LEFT ARITHMETIC--SLA

Op-code: 0A00 (Format V)

Syntax definition:

[<label>] b SLA b (wa),(scnt) b [<comment>]

Example:

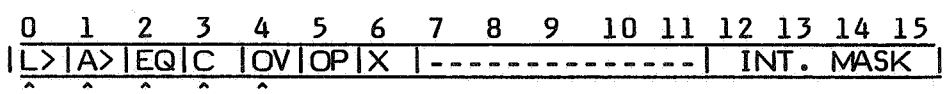
```
LABEL    SLA     R2,1          ;Shifts Workspace Register 2 left 1 bit location.
```

Definition:

Shifts the contents of the specified Workspace Register to the left the specified number of bit positions. Fills the vacated bit positions with zeros. Note that the overflow status bit is set when the sign of the word changes during the shift operation. The carry status bit contains the value shifted out of bit position zero.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

Shifts the bits of (wa) to the left, filling the vacated bit positions with zeros. When scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 bit positions.

Application notes:

The SLA instruction shifts the given Workspace Register to the left the given number of bit positions and fills vacated positions with zeros. If Workspace Register 10 contains the value 1357H, the instruction

```
SLA     R10,5
```

changes the contents of Workspace Register 10 to 6AE0H. This SLA operation on a bit-by-bit basis is

```
0001 0011 0101 0111  (Workspace Register 10--1357H)
---------------------
0110 1010 1110 0000  (Workspace Register 10 result--6AE0H)
```

The logical greater than, arithmetic greater than, and overflow status bits are set, while the equal and carry status bits are reset. Refer to Section 12.5 for another example.

12.4 SHIFT RIGHT CIRCULAR--SRC

Op-code: 0B00 (Format V)

Syntax definition:

[<label>] b SRC b (wa),(scnt) b [<comment>]

Example:

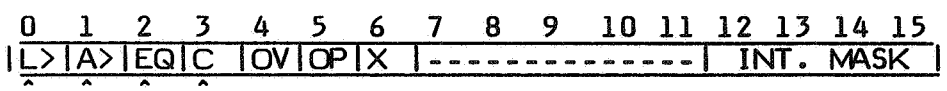
```
LABEL    SRC     R7,16-3     ;Shifts Workspace Register 7 circularly 13 bit
                             locations right.
```

Definition:

Shifts the specified Workspace Register to the right the specified number of bit positions. Fills vacated bit positions with the bit shifted out of position 15. The carry status bit contains the value shifted out of bit position zero.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and carry.



Execution results:

Shifts the bits of (wa) to the right, filling the vacated bit positions with the bits shifted out at the right. If scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 bit positions.

WORKSPACE REGISTER SHIFT INSTRUCTIONS

Application notes:

The SRC instruction shifts the given Workspace Register to the right the given number of bit positions and fills vacated positions with the bits shifted. If Workspace Register 2 contains the value 0FFEFH, the instruction

```
SRC     R2,7
```

changes the contents of Workspace Register 2 to 0DFFFH. This SRC operation on a bit-by-bit basis is

```
1111 1111 1110 1111  (Workspace Register 2--0FFEFH)
--------------------
1101 1111 1111 1111  (Workspace Register 2 result--0DFFFH)
```

The logical greater than and carry status bits are set, while the arithmetic greater than and equal status bits are reset.

There is no "shift left circular" instruction because the same effect can be obtained with SRC. To shift left a number of bits, instead shift right by a number equal to 16 minus the number. For example, to shift left 7 bits, shift right by 16 minus 7 or 9 bits.

12.5 INSTRUCTION EXAMPLE

This shift instruction shifts the indicated Workspace Register a specified number of bits to the left. For example, the instruction

```
SLA     R5,1
```

shifts the contents of Workspace Register five one bit to the left. The carry status bit contains the value shifted out of bit position zero. The jump instructions JOC and JNC permit you to test the shifted bit. The overflow status bit is set when the sign of the contents of the Workspace Register being shifted changes during the shift operation. If Workspace Register 5 contains

```
0100 1111 0000 1111
```

before the shift, the instruction changes Workspace Register 5 to

```
1001 1110 0001 1110
```

The carry status bit contains a zero and the overflow status bit is set because the contents changed from positive to negative (bit zero changed from 0 to 1). If this shift sign change is significant, you could insert a JNO instruction to test the overflow condition. If there is no overflow, control transfers to the normal program sequence. Otherwise, the next instruction is executed.

It is possible to construct double-length shifts with the SLA instruction to shift two or more words in a Workspace. The code on the next page shifts two consecutive Workspace Registers assuming that:

- The contents of Workspace Registers 1 and 2 are shifted one bit position.
- Additional code could be included to execute the code once for each bit shift required, when shifts of more than one bit position are required. The additional code must include a means of testing that the desired number of shifts are performed.
- Additional code tests for overflow from Workspace Register 1, to branch to an error routine at location ERR when overflow occurs.

WORKSPACE REGISTER SHIFT INSTRUCTIONS

```
        SLA     R1,1        ;Shift R1 one bit.
        JOC     ERR
        SLA     R2,1        ;Shift R2 one bit.
        JNC     EXIT        ;Transfer if no carry.
        INC     R1          ;Transfer bit from R2 to R1.
        .
        .
        .
EXIT    NOP                 ;Continue with program.
        .
        .
        .
ERR     NOP
```

SECTION 13: ASSEMBLER DIRECTIVES

Assembler directives let you supply data to be included in the program and exercise control over the assembly process. Assembler directives appear in the source code as predefined identifiers preceded by a period.

The following conventions are used in the Assembler directive syntax definitions given in the remainder of this section.

Special characters and items in capital letters must be entered as shown.

Items within angle brackets (<>) are defined by the user.

Items within square brackets ([]) are optional.

The word "or" indicates a choice between two items.

Items in lower-case letters are generic names for classes of items.

The following terms are names for classes of items as used in this section.

b = the occurrence of one or more blanks.

integer = any legal integer constant as defined in Section 3.3.4.

label = any legal label as defined in Section 3.4.1.

expression = any legal expression as defined in Section 3.3.5.

value = any label, constant, or expression. The default value is 0.

valuelist = a list of zero or more values separated by commas.

identifier = a legal identifier as defined in Section 3.3.2.

idlist = a list of one or more identifiers separated by commas.

id:integer list = a list of one or more identifier-integer pairs separated by a colon, with each pair separated by a comma. The colon-integer part is optional and has a default value of 1.

comment = any legal comment as defined in Section 3.4.4.

character string = any legal character string as defined in Section 3.3.3.

file identifier = any legal name for a p-System text file.

The following example illustrates the use of these conventions.

[<label>] [b] .ASCII b <character string> [<comment>]

The example indicates that a label can be included in the label field (but is not necessary) and that a character string must be included as an operand.

For your convenience, the directives discussed in this section are listed here in alphabetical order along with their forms and the section where they are discussed.

Directive	Form	Section
.ABSOLUTE	[b] .ABSOLUTE [<comment>]	13.8
.ALIGN	[b] .ALIGN b <value> [<comment>]	13.3
.ASCII	[<label>] [b] .ASCII b <character string> [<comment>]	13.2
.ASCIILIST	[b] .ASCIILIST [<comment>]	13.4
.ASECT	[b] .ASECT [<comment>]	13.8
.BLOCK	[<label>] [b] .BLOCK b <length> [,<value>] [<comment>]	13.2
.BYTE	[<label>] [b] .BYTE b [valuelist] [<comment>]	13.2
.CONDLIST	[b] .CONDLIST [<comment>]	13.4
.CONST	[b] .CONST [b] <idlist> [<comment>]	13.5
.DEF	[b] .DEF [b] <idlist> [<comment>]	13.5
.ELSE	[b] .ELSE [<comment>]	13.6
.END	[<label>] [b] .END	13.1
.ENDC	[b] .ENDC [<comment>]	13.6
.ENDM	[b] .ENDM [<comment>]	13.7
.EQU	<label> [b] .EQU b <value> [<comment>]	13.2
.FUNC	[b] .FUNC [b] <identifier> [,<integer>] [<comment>]	13.1
.IF	[b] .IF <expression> [= or <> <expression>] [<comment>]	13.6
.INCLUDE	[b] .INCLUDE [b] <file identifier> [b <comment>]	13.8
.INTERP	valid when used in <expression>	13.5
.LIST	[b] .LIST	13.4
.MACRO	[b] .MACRO [b] <identifier> [<comment>]	13.7
.MACROLIST	[b] .MACROLIST	13.4
.NARROWPAGE	[b] .NARROWPAGE [<comment>]	13.4
.NOASCIILIST	[b] .NOASCIILIST [<comment>]	13.4

Directive	Form	Section
.NOCONDLIST	[b] .NOCONDLIST [<comment>]	13.4
.NOLIST	[b] .NOLIST	13.4
.NOMACROLIST	[b] .NOMACROLIST	13.4
.NOPATCHLIST	[b] .NOPATCHLIST	13.4
.NOSYMTABLE	[b] .NOSYMTABLE [<comment>]	13.4
.ORG	[b] .ORG b <value> [<comment>]	13.3
.PAGE	[b] .PAGE	13.4
.PAGEHEIGHT	[b] .PAGEHEIGHT [b] <integer> [<comment>]	13.4
.PATCHLIST	[b] .PATCHLIST	13.4
.PRIVATE	[b] .PRIVATE [b] <id:integer list> [<comment>]	13.5
.PROC	[b] .PROC b <identifier> [,<integer>] [<comment>]	13.1
.PSECT	[b] .PSECT [<comment>]	13.8
.PUBLIC	[b] .PUBLIC [b] <idlist> [<comment>]	13.5
.RADIX	[b] .RADIX [b] <integer> [<comment>]	13.8
.REF	[b] .REF [b] <idlist> [<comment>]	13.5
.RELFUNC	[b] .RELFUNC [b] <identifier> [,<integer>] [<comment>]	13.1
.RELPROC	[b] .RELPROC b <identifier> [,<integer>] [<comment>]	13.1
.TITLE	[b] .TITLE b <character string> [<comment>]	13.4
.WORD	[<label>] [b] .WORD b <valuelist> [<comment>]	13.2

13.1 PROCEDURE DELIMITING DIRECTIVES

Every source program, including those to be used as stand-alone code files, must contain at least one set of procedure-delimiting directives as described below. The most frequent use of the Assembler is in assembling small routines intended to be linked with a host compilation unit. The directives .PROC and .FUNC identify and delimit assembly language procedures. .RELPROC and .RELFUNC identify and delimit dynamically relocatable procedures. Dynamically relocatable procedures can reside in the code pool and are subject to more of the System's memory management strategies. See the UCSD p-System Linker manual for a more detailed description of the use of these directives.

.END — The "end" directive marks the end of an assembly source file.

Form: [<label>] [b] .END

.FUNC — The "function" directive identifies the beginning of a statically relocatable assembly language function which is expected (by the host compilation unit) to return a function result on top of the stack. Otherwise, it is equivalent to the .PROC directive.

Form: [b] .FUNC [b] <identifier> [,<integer>] [<comment>]
<identifier> is the name associated with the assembly procedure.
<integer> indicates the number of words of parameters passed to this routine. The default for <integer> is 0.

Example: .FUNC RANDOM

.PROC — The "procedure" directive identifies the beginning of a statically relocatable assembly language procedure. The procedure is terminated by the occurrence of another delimiting directive in the source file.

Form: [b] .PROC b <identifier> [,<integer>] [<comment>]
<identifier> is the name associated with the assembly procedure.
<integer> indicates the number of words of parameters passed to this routine. The default for <integer> is 0.

Example: .PROC DLDRIVE,2

.RELFUNC — The "relocatable function" directive identifies the beginning of a dynamically relocatable assembly language function which is expected (by the host compilation unit) to return a function result on the stack. Otherwise, it is equivalent to the .RELPROC directive.

Form: [b] .RELFUNC [b] <identifier> [,<integer>] [<comment>]
<identifier> is the name associated with the assembly function.
<integer> indicates the number of words of parameters passed to this routine. The default for <integer> is 0.

Example: .RELFUNC DAE

.RELPROC — The "relocatable procedure" directive identifies the beginning of a dynamically relocatable assembly language procedure. Such assembly procedures must be position-independent (see the Linker manual). The procedure is terminated by the occurrence of another delimiting directive in the source file.

Form: [b] .RELPROC b <identifier> [,<integer>] [<comment>]
<identifier> is the name associated with the assembly procedure.
<integer> indicates the number of words of parameters passed to this routine. The default for <integer> is 0.

Example: .RELPROC POOF,3

13.2 DATA AND CONSTANT DEFINITION DIRECTIVES

These directives assign string and numerical values.

.ASCII — The "ASCII" directive converts character strings to a series of ASCII byte constants in memory. The bytes are allocated in the order that they appear in the string. An identifier in the label field is assigned the location of the first character allocated in memory.

Form: [<label>] [b] .ASCII b <character string> [<comment>]
<character string> is any string of printable ASCII characters enclosed in double quotes.

Example: .ASCII "HELLO"

.BLOCK — The "block" directive allocates and initializes a block of consecutive bytes in memory. A byte value must be an absolute quantity. The default value is 0. An identifier in the label field is assigned the location of the first byte allocated.

Form: [<label>] [b] .BLOCK b <length> [,<value>] [<comment>]
<length> is the the number of bytes to be allocated with the initial value <value>.

Example: TEMP .BLOCK 4,6H
The output code is
06 06 06 06 (Four bytes with value 06 hexadecimal)

.BYTE — The "byte" directive allocates and initializes values in one or more bytes of memory. The values must be absolute byte quantities. The default value is 0. An identifier in the label field is assigned the location of the first byte allocated in memory.

Form: [<label>] [b] .BYTE b [valuelist] [<comment>]

Example: TEMP .BYTE 4 ; The code is 04 hexadecimal

.EQU — The "equals" directive equates a value to a label. Labels can be equated to an expression containing relocatable labels, externally referenced labels, and/or absolute constants. The general rule is that labels equated to values must be defined before use. The exception to this rule is for labels equated to expressions containing another label. Local labels cannot appear in the label field of an equate statement.

Form: <label> [b] .EQU b <value> [<comment>]

Example: BASE .EQU R6

.WORD The "word" directive allocates and initializes values in one or more consecutive words of memory. The values can be relocatable quantities. The default value is 0. An identifier in the label field is assigned the location of the first word allocated.

Form: [<label>] [b] .WORD b <valuelist> [<comment>]

Example: TEMP .WORD 0,2,,4

The output code is

```
0000
0002
0000  ; This is a default value.
0004
```

Example: L1 .WORD L2

The output code is a word containing the address of the label L2.

13.3 LOCATION COUNTER MODIFICATION DIRECTIVES

These directives affect the value of the location counter (LC or ALC) and the location in memory of the code being generated.

.ALIGN — The "align" directive outputs enough 0 bytes to set the location counter to a value which is a multiple of the operand value.

Form: [b] .ALIGN b <value> [<comment>]

Example: .ALIGN 2
This aligns the LC on a word boundary.

.ORG — The "origin" directive initializes the location counter to <value> if used at the beginning of an absolute assembly program. Used anywhere else, .ORG generates 0 bytes until the value of the location counter equals <value>.

Form: [b] .ORG b <value> [<comment>]

Example: .ORG 1000H

13.4 LISTING CONTROL DIRECTIVES

These directives allow you to control the format of the assembled listing file generated by the Assembler. No code is generated by these directives, and their source lines do not appear on assembled listings. See Section 16 for a more detailed description of an assembled listing.

.ASCIILIST — The "ASCII list" directive prints all bytes generated by the .ASCII directive in the code field of the list file, creating multiple lines in the list file if necessary. Assembly begins with an implicit .ASCIILIST directive.

Form: [b] .ASCIILIST [<comment>]

.CONDLIST — The "conditional list" directive lists the source code contained in the unassembled sections of conditional assembly directives.

Form: [b] .CONDLIST [<comment>]

.LIST — The "list" directive enables output to the list file if a listing is not already being generated. .LIST and .NOLIST can be used to examine certain sections of source and object code without creating an assembled listing of the entire program. Assembly begins with an implicit .LIST directive.

Form: [b] .LIST

.MACROLIST — The "macro list" directive specifies that all following macro definitions are to have their macro bodies printed when they are invoked in the source program. Assembly begins with an implicit .NOMACROLIST directive. See Section 15 for a description of macro language.

Form: [b] .MACROLIST

.NARROWPAGE — The "narrow page" directive limits the width of an assembled listing to 80 columns. The symbol table is printed in a narrow format, source lines are truncated to a maximum of 49 characters, and title lines on the page headers are truncated to a maximum of 40 characters.

Form: [b] .NARROWPAGE [<comment>]

.NOASCIILIST The "no ASCII list" directive limits the printing of data generated by the .ASCII directive to the number of bytes that fit in the code field of one line in the list file.

Form: [b] .NOASCIILIST [<comment>]

.NOCONDLIST The "no conditional list" directive suppresses the listing of source code contained in the unassembled sections of conditional assembly directives. Assembly begins with an implicit .NOCONDLIST directive.

Form: [b] .NOCONDLIST [<comment>]

.NOLIST The "no list" directive suppresses output to the list file if it is not already off.

Form: [b] .NOLIST

.NOMACROLIST The "no macro list" directive specifies that all following macro definitions are not to have their macro bodies printed when they are invoked in the source program. Only the macro identifier and parameter list are included in the listing. Assembly begins with an implicit .NOMACROLIST directive.

Form: [b] .NOMACROLIST

.NOPATCHLIST The "no patch list" directive suppresses the listing of back patches of forward references. See Section 16 for a description of back patches.

Form: [b] .NOPATCHLIST

.NOSYMTABLE The "no symbol table" directive suppresses the printing of the symbol table after each assembly routine in an assembled listing.

Form: [b] .NOSYMTABLE [<comment>]

.PAGE The "page" directive continues the assembled listing on the next page by sending an ASCII form-feed character to the assembled listing.

Form: [b] .PAGE

.PAGEHEIGHT The "page height" directive controls the number of lines printed in an assembled listing between page breaks. Assembly begins with an implicit .PAGEHEIGHT 59 directive.

Form: [b] .PAGEHEIGHT [b] <integer> [<comment>]

Example: .PAGEHEIGHT 53

.PATCHLIST — The "patch list" directive lists occurrences of all back patches of forward-referenced labels in the list file. Assembly begins with an implicit .PATCHLIST directive. See Section 16 for a description of back patches.

Form: [b] .PATCHLIST

.TITLE — The "title" directive changes the title printed on the top of each page of the assembled listing. The title can be up to 80 characters long. The Assembler changes the title to "SYMBOLTABLE DUMP" when printing a symbol table. The title reverts to its former value after the symbol table is printed. The default value for the title is " ".

Form: [b] .TITLE b <character string> [<comment>]

Example: .TITLE "INTERPRETER"

13.5 PROGRAM LINKAGE DIRECTIVES

Linking directives allow communication between separately assembled and/or compiled programs. See the Linker manual for a description of program linking.

.CONST — The "constant" directive gives the assembly procedure access to globally declared constants in the host compilation unit.

Form: [b] .CONST [b] <idlist> [<comment>]
Each element of <idlist> is the name of a global constant declared in the host.

Example:
```
.CONST LENGTH
```

.DEF — The "define" directive makes one or more labels to be defined in the current routine available to other assembly language routines for reference.

Form: [b] .DEF [b] <idlist> [<comment>]

Example:
```
.DEF STOP,RUN
```

.INTERP — The "interpreter" directive allows an assembly language procedure to access code or data in the System interpreter. .INTERP is a predefined symbol for a processor-dependent location in the resident interpreter code. Offsets from this base location can be used to access any code in the interpreter. Correct usage of this feature requires a knowledge of the interpreter's jump vector for this location. The use of the .INTERP directive is generally restricted to systems applications.

Form: This directive is valid when used in an expression.

Example:
```
EXECERR   .EQU        12  ; Hypothetical routine offset.
BOMBINT   .EQU        .INTERP+EXECERR
          B           @BOMBINT
```

.PRIVATE — The "private" directive allows an assembly language routine to store variables that are accessible only to the assembly language routine. The directives are stored in the global data segment of the host compilation unit.

Form: [b] .PRIVATE [b] <id:integer list> [<comment>]
Each elment of <id:integer list> is treated as a label defined in the source code. <integer> determines the number of words of space allocated for <id>.

Example:
```
.PRIVATE PRINT,BARRAY:9
```

.PUBLIC — The "public" directive allows variables declared in the global data segment of the host compilation unit to be referred to by an assembly language routine.

Form: [b] .PUBLIC [b] <idlist> [<comment>]
Each element of <idlist> is the name of a global variable declared in the UCSD p-System host.

Example: .PUBLIC I,J,LENGTH

.REF — The "reference" directive provides access to one or more labels defined in other assembly language routines.

Form: [b] .REF [b] <idlist> [<comment>]

Example: .REF BRITT

13.6 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives allow the assembly of code based on previous results or conditions. See Section 14 for a description of conditional assembly features.

.ELSE — The "else" directive marks the start of an alternative section of source statements.

Form: [b] .ELSE [<comment>]

.ENDC — The "end conditional" directive marks the end of a conditional section of source statements.

Form: [b] .ENDC [<comment>]

.IF — The "if" directive marks the start of a conditional section of source statements.

Form: [b] .IF <expression> [= or <> <expression>] [<comment>]

Example: .IF TI994A

13.7 MACRO DEFINITION DIRECTIVES

Macro definition allows you to create sections of code that can be conveniently referred to later in the program. See Section 15 for a description of macro language.

.ENDM — The "end macro" directive marks the end of a macro definition.

Form: [b] .ENDM [<comment>]

.MACRO — The "macro" directive indicates the start of a macro definition.

Form: [b] .MACRO [b] <identifier> [<comment>]
<identifier> is used to invoke the macro being defined.

Example: .MACRO ADDWORDS

13.8 MISCELLANEOUS DIRECTIVES

These directives allow including other files in the code to be assembled, permit absolute as well as relative sections, and allow changing the default radix (base).

.ABSOLUTE — The "absolute" directive causes the assembly routine following the directive to be assembled without relocation information. Labels become absolute addresses and label arithmetic is allowed in expressions. Usage is valid only before the occurrence of the first procedure delimiting directive. .ABSOLUTE must not be used when creating a Pascal external procedure. See the Linker manual for a description of absolute code files.

Form: [b] .ABSOLUTE [<comment>]

.ASECT — The "absolute section" directive specifies the start of an absolute section. See Section 3.5.3 for a description of .ASECT.

Form: [b] .ASECT [<comment>]

.INCLUDE — The "include" directive causes the Assembler to assemble the file named as an argument of the directive. When the end of this file is reached, assembly resumes with the source code that follows the directive in the original file. This feature is useful for including a file of macro definitions or for splitting up a source program which is too large to be edited as a single text file. .INCLUDE cannot be used in an included source file (nested) and cannot be used in a macro definition.

Form: [b] .INCLUDE [b] <file identifier> [b <comment>]
The comment field of the .INCLUDE directive must be separated from the file identifier by at least one blank character.

Example: .INCLUDE MYDISK:MACROS

.PSECT — The "program section" directive specifies the start of a program section and terminates an absolute section. See Section 3.5.3 for a description of .PSECT.

Form: [b] .PSECT [<comment>]

.RADIX The "radix" directive sets the current default radix to the value of the operand. Allowable operands are 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal). See Section 3.3.4 for a description of radices. The initial defaults for the computer are decimal for the default constant radix and hexadecimal for the default list radix.

Form: [b] .RADIX [b] <integer> [<comment>]

Example: .RADIX 16 ; Hexadecimal default radix

SECTION 14: CONDITIONAL ASSEMBLY

Conditional assembly directives selectively exclude or include sections of source code at assembly time. Conditional sections are initiated with the .IF directive and terminated with the .ENDC directive and can contain the .ELSE directive. Control over the inclusion of conditional sections is determined by conditional expressions. Conditional sections can contain other conditional sections.

When the Assembler encounters an .IF directive, it evaluates the associated expression to determine the condition value. If the condition is true, the source code before the .ENDC or .ELSE directive is assembled and any code between .ELSE and .ENDC is discarded. If the condition value is false, the source statements following the directive are discarded until a matching .ENDC or .ELSE is reached. Then any code between a matching .ELSE and .ENDC is assembled.

The syntax for a conditional section, using the conventions described in Section 13, is as follows.

```
.IF <conditional expression>
<source statements>
[.ELSE
<source statements>]
.ENDC
```

14.1 CONDITIONAL EXPRESSIONS

A conditional expression can be a single expression or a comparison of two character strings or expressions. The single expression is false if it evaluates to zero and true otherwise. The comparison of two character strings or expressions is for equality or inequality (indicated by the symbols "=" and "<>", respectively).

The following example illustrates the use of conditional expressions. The indentations are for clarity. They are not necessary in the actual file.

```
.IF   LABEL1-LABEL2   ; Arithmetic expression.  True if
                      LABEL1-LABEL2 is nonzero and false if it
                      is zero.

      ...                   ; This and following code, up to
                            .ELSE, is assembled if the outer
                            condition is true and any other
                            conditions are met.

      .IF  %1 = "STUFF"     ; Comparison expression.  True if the
                            text of the first macro parameter is
                            equal to STUFF.

           ...                   ; This code is assembled if both
                                 the outer condition and inner
                                 condition are true.

      .ENDC                 ; Terminate inner section.

      ...                   ; This code is assembled if the outer
                            condition is true.

      .ELSE                 ; End section assembled if the outer
                            section is true. Begin section
                            assembled if the outer condition is
                            false.

      ...                   ; This code is assembled if the first
                            condition is false.

.ENDC                 ; Terminate outer section.
```

SECTION 15: MACRO LANGUAGE

The Assembler supports the use of a macro language in source programs. A macro language allows you to associate a set of source statements with an identifying symbol. When the Assembler encounters this symbol (known as a macro identifier) in the source code, it substitutes the corresponding set of source statements (known as the macro body) for the macro identifier and assembles the macro body as if it had been included directly in the source program. A carefully designed set of macro definitions can be used to simplify the development of assembly language routines.

You can enhance the macro language by including a mechanism for passing parameters (known as macro parameters) to the macro body while it is being expanded. In this way, a single macro definition can be used for an entire class of subtasks.

Here is a simple example.

```
                        ; Macro definition...
.MACRO  STRING          ; Macro identifier is STRING.
                        ; Macro body.
                        ; %1 and %2 are parameter declarations.
.BYTE   %2              ; Second parameter is length byte.
.ASCII  %1              ; First parameter is argument.
.ENDM                   ; End macro definition.

...

STRING  "WRITE",5               ; First macro-call parameters
                                are WRITE and 5.
STRING  "TYPE SPACE",10         ; Second macro-call parameters
                                are TYPE SPACE and 10.
```

This is what gets assembled.

```
.BYTE   5                       ; Data string declarations.
.ASCII  "WRITE"

.BYTE   10
.ASCII  "TYPE SPACE"
```

15.1 MACRO DEFINITIONS AND CALLS

Macro definitions can occur anywhere in a source program, delimited by the directives .MACRO and .ENDM. Macro definitions must appear before their macro calls are assembled. The macro identifier must be unique to the source program except when you are redefining a predefined machine instruction name as a macro identifier. A macro definition cannot include another macro definition. However, it can include macro calls.

Macro calls can occur anywhere in a source program where code can be generated. A macro call consists of a macro identifier followed by a list of parameters. The parameters are delimited by commas and terminated by a carriage return or semicolon.

When the Assembler encounters a call to a macro, it checks to see if the macro has been defined. If it has not, an error message is generated. If it has been defined, the body of the macro is copied into the source program. As part of the copy process, parameters and references are resolved.

After the copy is completed, the assembly process continues, starting with the copied macro body. When another macro call is encountered, whether or not it is within the previous macro (a nested macro call), the copy process is repeated.

The following demonstrates the use of macros. Macro ONE is defined, followed by the definition of macro TWO, followed by the main code.

```
.MACRO   ONE
MOV      R1,R1
TWO
MOV      R3,R3
.ENDM

.MACRO   TWO
MOV      R2,R2
.ENDM

.PROC    MACTEST,1
MOV      R0,R0
ONE
MOV      R4,R4
.END
```

The following is the code assembled with the portions of the macros identified by # followed by the number of the macro from which they were copied.

```
        .PROC   MACTEST,1
        MOV     R0,R0
        ONE
#1      MOV     R1,R1
#1      TWO
#2      MOV     R2,R2
#1      MOV     R3,R3
        MOV     R4,R4
        .END
```

15.2 PARAMETER PASSING

Macro parameters are referred to in a macro body by the symbol "%n", where "n" is a single nonzero decimal digit. Upon scanning this symbol, the Assembler replaces it with the text of the nth macro parameter. Macro parameters are not expanded within the quotes of an ASCII data string.

There are several possibilities when the Assembler is substituting the parameters passed to a macro into the parameter list defined by the macro. The simplest case is when the number of parameters passed in the macro call is the same as the number of parameters expected by the macro. Then a simple substitution is made. For example, if a macro is defined as

```
.MACRO  ONE
MOV     %2, %1
.ENDM
```

and is called with

```
ONE     R2,R1
```

then the macro assembles to

```
        ONE     R2,R1
#       MOV     R1,R2
```

If the macro call passes fewer parameters than the macro expects, then the unpassed macro is left blank. For example, if the macro ONE defined above is called with

```
ONE     R2
```

then the macro assembles to

```
        ONE     R2
#       MOV     ,R2
```

and an error occurs.

If more macro parameters are passed than are necessary, the extra parameters are ignored. For example, if the macro ONE defined above is called with

```
        ONE     R2,R1,R3
```

then the macro assembles to

```
        ONE     R2,R1
#       MOV     R1,R2
```

A parameter passed by a macro call may be used in a nested macro call. For example, suppose macros ONE and TWO are defined as follows.

```
        MACRO   ONE
        MOV     R0,R1
        TWO     %1,%2
        .ENDM

        .MACRO  TWO
        S       %2,%1
        .ENDM
```

If macro one is then called with

```
        ONE     R4,R5
```

then the following is the result of the assembly process.

```
        ONE     R4,R5
#1      MOV     R0,R1
#1      TWO     R4,R5
#2      S       R5,R4
```

15.3 SCOPE OF LABELS IN MACROS

Local label names declared in a macro body are local to that macro. Thus, a section of code that contains a local label $1 and a macro call whose body also has the local label $1 assembles without errors. (Compare this with what happens when two occurrences of $1 fall between two regular labels.) This feature allows local labels to be used freely in macros without fear of conflicting with the rest of the program.

Declaring a regular label in a macro body is incorrect if the macro is called more than once since the label would be substituted twice into the source program and flagged by the Assembler as a previously defined label. Location-counter-relative addressing can be used but is prone to errors in nontrivial applications. Instead, use labels that are local to the macro body. The maximum of 21 local labels active at any instant still applies.

15.3.1 Local Labels as Macro Parameters

The passing of local labels as parameters has a special property. Unlike other macro parameters, local labels are not passed as uninterpreted text. The scope of a local label passed in a macro call does not change as it is passed through increasing levels of macro nesting, regardless of naming conflicts along the way. One use of this property is passing an address to a macro which simulates a conditional branch instruction.

The following is an example of passing local labels as macro parameters.

```
        .MACRO    EIN
        JEQ       $1
        JMP       %1
$1
        .ENDM
```

In a program, the code

```
TWIE
        MOV     ICHI,NI
        EIN     $1
        B       *R11
$1
        CLR     SAN
```

assembles as

```
TWIE
        MOV     ICHI,NI     ; Looks confusing, but if the
                            listing were off, the result
                            is what the programmer meant
                            to occur.
        JEQ     $1          ; This refers to the macro
                            local label.
        JMP     $1          ; This refers to the outside
                            $1.
$1                          ; Macro local label.
        B       *R11
$1                          ; Outside $1.
        CLR     SAN
```

SECTION 16: ASSEMBLER OUTPUT

The Assembler can generate two types of output files. A code file is always produced, but you control whether an assembled listing of the source file is produced. A description of the code file format is beyond the scope of this manual.

An assembled listing displays each line of the source program, the machine code generated by that line, and the current value of the location counter. You can optionally have it display the expanded form of all macro calls in the source program. Any errors that occur during the assembly process have messages printed in the listing file, usually immediately following the line of source code that caused the error. A symbol table is printed at the end of the listing to serve as a directory for locating all labels declared in the source program.

16.1 SOURCE LISTING

A paginated, assembled listing is produced when you respond to the Assembler's listing prompt with a list file name. The default listing is 132 characters wide and 55 lines per page. Each line of a source program is included in the assembled listing, except for source lines that contain list directives. Source statements that contain the equate directive .EQU have the resulting value of the associated expression listed to the left of the source line.

Macro calls are always listed, including the list of macro parameters and the comment field, if any. The macro is expanded by listing the body, with all formal parameters replaced by their passed values, if the macro list option (.MACROLIST) was enabled when the macro was defined. Macro expansion text is marked in the assembled listing by the character "#" to the left of the source listing. Comment fields in the definition of the macro body are not listed in macro expansions.

Source lines with conditional assembly directives are listed. However, source statements in an unassembled part of a conditional section are not listed.

16.2 ERROR MESSAGES

Error messages in assembled listings have the same format as the error messages sent to the screen (see Section 2) except that the prompt is not included.

16.3 CODE LISTING

The code field lies to the left of the source program listing. It contains the value of the location counter, along with either the code generated by the matching source statement or the value of an expression occurring in a statement that includes the equate directive, .EQU. All values are printed in hexadecimal notation. Separately produced bytes and words of code on the same line are separated by spaces.

16.3.1 Forward References

When the Assembler produces a byte or word quantity that is the result of evaluating an expression that includes an undefined label, it lists an * for each digit of the quantity printed. For example, an unresolved hexadecimal byte is listed as **, while an unresolved octal word appears as ******. If the .PATCHLIST directive is used, the Assembler lists patch messages every time it encounters a label declaration that enables it to resolve all occurrences of a forward reference to that label. The messages (one for every backpatch performed) appear before the source statement that contains the label in question, and are of the form

<location in code file patched>* <patch value>

With this feature, the listing describes the contents of each byte or word of code. If neatness of the assembled listing is more desirable, the .NOPATCHLIST directive suppresses the patch messages.

16.3.2 External References

When the Assembler produces a word quantity that is the result of evaluating an expression that contains an externally referenced label, the value of that label (which cannot be determined until link time) is taken as zero. Therefore, the value reflects only the result of any assembly-time constants that were present in the expression.

16.3.3 Multiple Code Lines

Sometimes it is possible for a source statement to generate more code than fits in the code field. In most cases, the code is listed on successive lines of the code field with corresponding blank source listing fields. However, in the case of the .ORG, .ALIGN, and .BLOCK directives, the code field is limited to as many bytes as fit in the code field of one line because most uses of these directives generate large numbers of byte values that are not useful.

16.4 SYMBOL TABLE

The symbol table is an alphabetically sorted table of entries for all symbols declared in the source program. Each entry consists of the symbol identifier, the symbol type, and the value assigned to that symbol. The symbol identifiers are defined in a dictionary printed at the top of the symbol table. Symbols equated to constants have their constant values in the third field, while program labels are matched with their location counter offsets. All other symbols have dashes in their value field, as they possess no values relevant to the listing.

16.5 EXAMPLE

The following program calls an assembly language program named REVERSE.

```
PROGRAM TRYREVERSE;
VAR A:STRING;
PROCEDURE REVERSE(VAR S:STRING);EXTERNAL;
BEGIN
  A:='THIS IS A STRING';
  REVERSE(A);
  WRITELN(A);
END.
```

The program REVERSE is listed below.

```
        .PROC  REVERSE,1
        ;REVERSE A STRING.  CALLED AS REVERSE(S)
        MOV     *R10+,R1    ;GET THE POINTER TO THE STRING
        MOVB    *R1+,R2     ;LENGTH OF STRING
        SRL     R2,8        ;MAKE IT A FULL WORD
        MOV     R1,R3       ;A SECOND COPY OF THE POINTER
        A       R2,R3
        DEC     R3          ;POINT TO LAST BYTE OF STRING
        SRL     R2,1        ;NUMBER OF PAIRS OF CHARACTERS
$1      MOVB    *R1,R4      :LEFT BYTE TO TEMP
        MOVB    *R3,*R1+    ;RIGHT BYTE TO LEFT BYTE
        MOVB    R4,*R3      ;TEMP TO RIGHT BYTE
        DEC     R3          ;GET READY FOR NEXT
        DEC     R2          ;DONE?
        JNE     $1          ;NO, GO BACK
        B       *R11        ;RETURN TO CALLER
        .END
```

When assembled, REVERSE results in the following assembled listing.

```
Page -   0  File:SYSTEM.WRK.TEXT      9900 Assembler   II.0 [e.3]

0000|              .PROC   REVERSE,1
0000|          ;REVERSE A STRING.  CALLED AS REVERSE(S)
0000|  C07A       MOV      *R10+,R1    ;GET THE POINTER TO THE STRI
0002|  D0B1       MOVB     *R1+,R2     ;LENGTH OF STRING
0004|  0982       SRL      R2,8        ;MAKE IT A FULL WORD
0006|  C0C1       MOV      R1,R3       ;A SECOND COPY OF THE POINTE
0008|  A0C2       A        R2,R3
000A|  0603       DEC      R3          ;POINT TO LAST BYTE OF STRIN
000C|  0912       SRL      R2,1        ;NUMBER OF PAIRS OF CHARACTE
000E|  D111  $1   MOVB     *R1,R4      :LEFT BYTE TO TEMP
0010|  DC53       MOVB     *R3,*R1+    ;RIGHT BYTE TO LEFT BYTE
0012|  D4C4       MOVB     R4,*R3      ;TEMP TO RIGHT BYTE
0014|  0603       DEC      R3          ;GET READY FOR NEXT
0016|  0602       DEC      R2          ;DONE?
0018|  16FA       JNE      $1          ;NO, GO BACK
001A|  045B       B        *R11        ;RETURN TO CALLER
001C|              .END

Page -   1  REVERSE   File:SYSTEM.WRK.TEXT   Symbol Table

AB - Absolute      LB - Label     UD - Undefined     MC - Macro
RF   Ref           DF - Def       PR - Proc          FC - Func
PB - Public        PV - Private   CS - Consts

REVERSE  PR ----|

Page -   2  REVERSE   File:SYSTEM.WRK.TEXT   Symbol Table

Assembly complete:          17 lines
     0   errors flagged on this assembly
```

SECTION 17: APPENDICES

The following are the appendices contained in this section.

Appendix	Section
Memory Organization	17.1
CRU and Interrupt Structure	17.2
Character Set	17.3
Assembler Directive Table	17.4
Hexadecimal Instruction Table	17.5
Alphabetical Instruction Table	17.6
Program Development with Multi-Drive Systems	17.7

17.1 MEMORY ORGANIZATION

To understand memory organization, you must understand some basic terms and how they apply to the TI Home Computer.

The Central Processing Unit (CPU) of the computer contains all the circuitry for arithmetic functions, comparisons, hardware registers, and all other functions that actually process computer instructions. The CPU processes all commands and instructions fed into the computer and accesses all memory spaces. The CPU in the Home Computer is the TMS9900 Microprocessor.

One way to divide memory is into RAM (Random Access Memory) and ROM (Read Only Memory). RAM is a memory which can be written to, or read from, by any program. It stores programs and data. ROM is a memory which can only be read but not altered by any program. It is used to store information used by the computer itself, such as the built-in TI BASIC language and the makeup of the alphanumeric characters.

So that you can refer to any specific byte in the computer's memory, each byte is assigned a number. These sequential numbers, called the addresses of the bytes, are unique within each of the computer's memory spaces. They are usually referred to in hexadecimal notation.

The 9900 microprocessor has an address space of 64K bytes. On the Home Computer, some of this address space contains RAM and some contains ROM. In addition, some addresses are used for access to special devices, such as sound and speech, and to other areas of memory, such as VDP RAM and GROMs.

17.1.1 Directly Addressable Memory

When all possible devices are connected, 64K (65,536 or 10000H) bytes of memory are directly addressable.

Addresses 0000H through 1FFFH are built into the console. They contain 8K bytes of ROM that contain the TI BASIC language and other information necessary to the functioning of the computer.

Addresses 2000H through 3FFFH are the 8K bytes of RAM that make up the low memory of the Memory Expansion unit. They can only be used when the Memory Expansion unit is connected. The p-System uses this portion of RAM for its tables.

Addresses 4000H through 5FFFH are built into various peripherals. They contain up to 8K bytes of ROM for the Device Service Routine used to run peripheral devices such as disk drives and printers. Pascal also uses 12K bytes of ROM in this space. These ROMs are selected by CRU operations (see Section 9), so several ROMs can be at the same address.

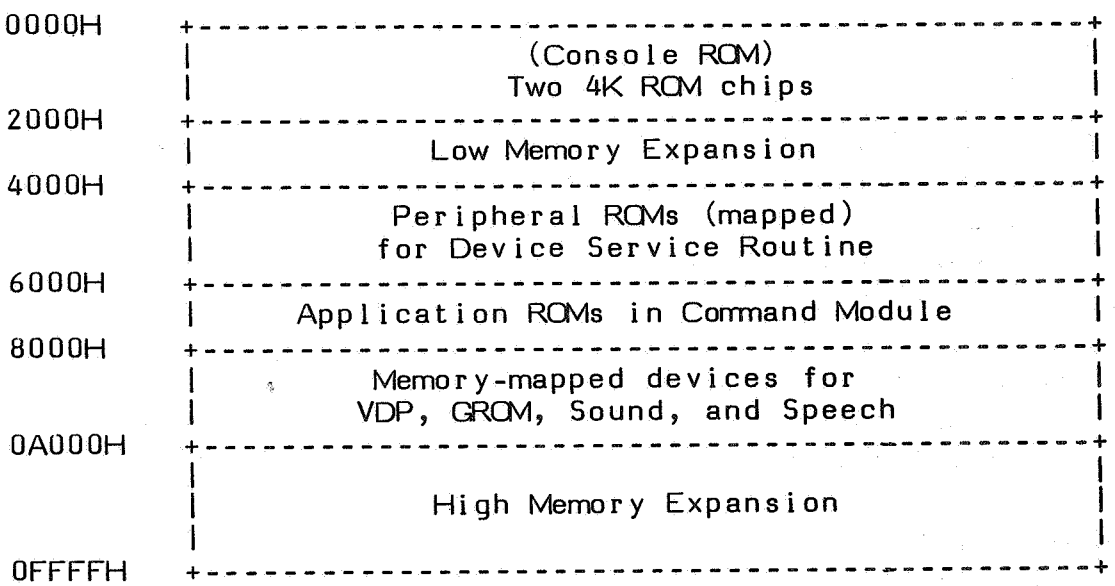
Addresses 6000H through 7FFFH are available on the command module port. Some command modules have ROM in this space.

Addresses 8000H through 9FFFH are built into the console. They contain all of the memory-mapped device locations.

Addresses 0A000H through 0FFFFH are the 24K bytes of RAM that make up the high memory of the Memory Expansion unit. They can only be used when the Memory Expansion unit is connected. The majority of this area is available for your use under the p-System.

The following memory map summarizes the above information.

CPU Memory Use
General Case



17.1.1.1 Expansion RAM

The Memory Expansion unit is a 32K-byte peripheral on an eight-bit bus. It has two blocks of memory, an 8K block from 2000H through 3FFFH and a 24K block from addresses 0A000H through 0FFFFH. Addresses 0FFD8H through 0FFFFH are used for XOP 1 on the TI-99/4A.

17.1.1.2 ROM

All the ROMs (Read Only Memory) are directly accessible by an assembly language program. Two 4K-byte console ROMs are located at addresses 0000H through 1FFFH. They contain the console operating system, the GPL interpreter, and part of the TI BASIC interpreter.

The memory block at addresses 4000H through 5FFFH is assigned to the peripheral ROMs and the p-Code ROMS which can be accessed by setting the bit assigned for the CRU (Communication Register Unit) to enable the particular ROM. These ROMs contain DSRs (Device Service Routines), and the p-Code interpreter. They are located in a peripheral. See Section 9 for more information.

Application ROMs are contained in command modules. They occupy address 6000H through 7FFFH.

17.1.1.3 GROM

A GROM (Graphics Read Only Memory) is another type of ROM. It is designed to contain GPL (Graphic Programming Language) programs which are executed by the GPL interpreter in the console. GPL is commonly used in applications software and can only be executed through a GROM. A GROM can also contain p-Code programs that are executed by the p-System.

A GROM is a memory-mapped device, just as VDP RAM is. A GROM's memory is addressed by writing its address to a specific CPU address and reading data from another specified CPU address.

GROM addresses are from 0000H through 0F7FFH. Each GROM has 6K bytes of memory that start from an even-numbered first-digit address. For example, GROM 0 is at addresses 0000H through 17FFH and GROM 1 is at addresses 2000H through 37FFH. The computer can access up to eight GROMs at a time.

GROMs 0, 1, and 2 are in the console and contain the monitor program, part of the console operating system, and most of the TI BASIC interpreter. The GROMs are used when the p-Code System is not in use on the Home Computer. Five additional GROMS can be located in a Command Module. The number of GROMs used in a Command Module depends on the size of the applications program.

17.1.2 Memory-Mapped Devices

The memory-mapped devices are VDP (Video Display Processor) RAM, the Speech Synthesizer, the sound processor, GROMs, and so forth. VDP RAM is discussed in this section.

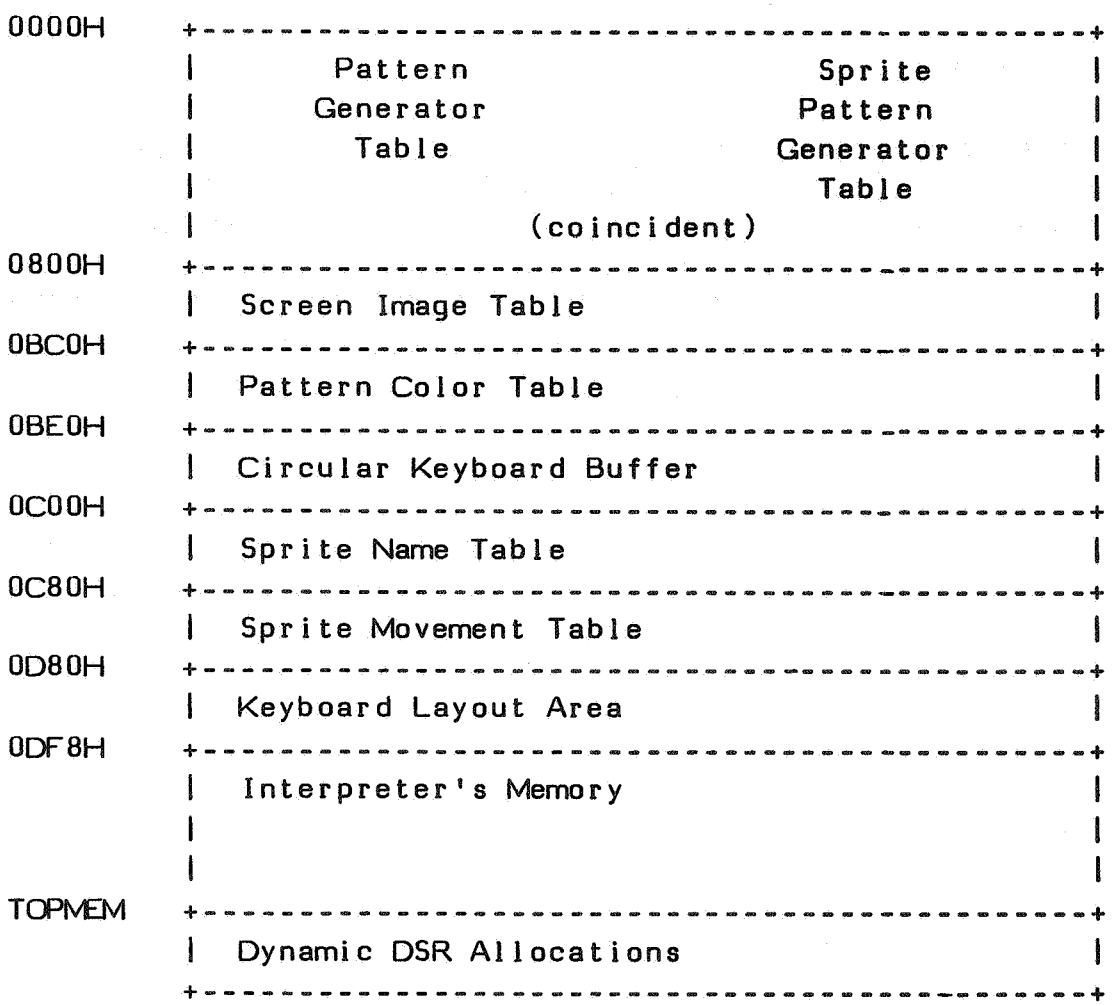
The VDP RAM, located in the console, is used chiefly for common video functions, such as screen images, character pattern tables, color tables, etc.

When TI BASIC is in use, VDP RAM also contains the TI BASIC program, the program symbol table, the value stack, and the string space. When the p-Code System is in use, VDP RAM is also used to hold some of the p-Code programs. Another part of VDP RAM functions as a PAB (Peripheral Access Block) to pass information from a file to the appropriate DSR (Device Service Routine). Assembly language programs cannot be executed from VDP RAM.

VDP RAM is a memory-mapped area of 16K (16,384 or 4000H) bytes numbered 0000H through 3FFFH. VDP RAM addresses are automatically incremented, so only one address in CPU RAM is required to read or write a specific block of data. There are assigned addresses for each I/O function in the RAM. For example, the VDP RAM read data Register is located at CPU RAM address 8800H, the VDP read status Register is found at CPU RAM address 8802H, the VDP write data Register is at CPU RAM address 8C00H, the VDP write address Register is at CPU RAM address 8C02H.

The diagram on the next page shows the memory of VDP RAM when it is being used by the p-System.

VDP RAM Memory Use
p-System



17.2 CRU AND INTERRUPT STRUCTURE

The following describes CRU use and interrupt handling.

17.2.1 CRU Allocation

The Communication Register Unit (CRU) is used for system access to peripherals. There are 4K CRU bits, numbered 0000H through 0FFFH. The CRU address loaded into Workspace Register 12 is twice the bit number. Thus, loading Workspace Register 12 with 1000H sets the base equal to CRU bit 800H. (See Section 9 for more information.) Of the available 4K of CRU bits, the first 1K, at addresses 0000H through 07FEH, are used internally by the console. This includes the TMS9901 I/O chip, which addresses the keyboard, joysticks, cassette, etc.

The second 1K, at addresses 0800H through 0FFEH, are reserved for future use.

The last 2K, at addresses 1000H through 1FFEH, are reserved for the peripherals that are attached to the console port. A block of 128 CRU bits is assigned to each peripheral as shown below. A0 through A15 are the CPU address bus lines.

CRU Addresses	A3	A4	A5	A6	A7	Use (Peripheral)	Device Number
0000H - 0FFEH	0	x	x	x	x	Internal use	
1000H - 10FEH	1	0	0	0	0	Reserved	0
1100H - 11FEH	1	0	0	0	1	Disk controller	1
1200H - 12FEH	1	0	0	1	0	Reserved	2
1300H - 13FEH	1	0	0	1	1	RS232, ports 1 and 2	3
1400H - 14FEH	1	0	1	0	0	Reserved	4
1500H - 15FEH	1	0	1	0	1	RS232, ports 3 and 4	5
1600H - 16FEH	1	0	1	1	0	Reserved	6
1700H - 17FEH	1	0	1	1	1	Reserved	7
1800H - 18FEH	1	1	0	0	0	Thermal Printer	8
1900H - 19FEH	1	1	0	0	1	Future expansion	9
1A00H - 1AFEH	1	1	0	1	0	Future expansion	10
1B00H - 1BFEH	1	1	0	1	1	Future expansion	11
1C00H - 1CFEH	1	1	1	0	0	Future expansion	12
1D00H - 1DFEH	1	1	1	0	1	Future expansion	13
1E00H - 1EFEH	1	1	1	1	0	Future expansion	14
1F00H - 1FFEH	1	1	1	1	1	P-Code peripheral	15

CRU address 0 at A8 through A14 is the memory enable bit in each device address space. Setting the bit to 1 turns the device ROM/RAM on, and resetting it to 0 turns it off. This enables the address space from 4000H through 5FFFH reserved for the peripheral ROM.

17.2.2 Interrupt Handling

The highest priority interrupts are the reset and load vectors with a priority of 0. The reset interrupt is used when the computer is turned on. Interrupt priority 1 connects through the TMS9901 Programmable Systems Interface for interrupt expansion. The following shows the interrupts available.

9900 Interrupts

Interrupt Level	Vector Memory Address	CPU Pin	Device Assignment
Highest	0000H	RESET	Reset
0	0FFECH	LOAD	Load
1	0004H	INT1	External Device (TMS9900)

Note that the lower priority CPU interrupts are not used. The following additional interrupts are implemented on the TMS9901.

9901 Interrupt Mapping

Address	CRU Bit	9901	Pin	Function
0000H	0	Control		Control.
0002H	1	INT1	17	External.
0004H	2	INT2	18	VDP vertical synchronization.
0006H	3	INT3	9	Clock interrupt, keyboard enter line, and joystick fire button.
0008H	4	INT4	8	Keyboard 1 line and joystick left.
000AH	5	INT5	7	Keyboard p line and joystick right.
000CH	6	INT6	6	Keyboard 0 line and joystick down.
000EH	7	INT7 (P15)	34	Keyboard shift line and joystick up.

Address	CRU Bit	9901	Pin	Function
0010H	8	INT8 (P14)	33	Keyboard space line.
0012H	9	INT9 (P13)	32	Keyboard q line.
0014H	10	INT10 (P12)	31	Keyboard 1 line.
0016H	11	INT11 (P11)	30	Not used.
0018H	12	INT12 (P10)	29	Reserved.
001AH - 001EH	13 - 15	INT13 - INT15	28, 27, 23	Not used.
0020H	16	P0	38	Reserved.
0022H	17	P1	37	Reserved.
0024H	18	P2	26	Bit 2 of keyboard select.
0026H	19	P3	22	Bit 1 of keyboard select.
0028H	20	P4	21	Bit 0 of keyboard select.
002AH	21	P5	20	Not used.
002CH	22	P6	19	Cassette control 1.
002EH	23	P7 (INT15)	23	Cassette control 2.
0030H	24	P8 (INT14)	27	Audio gate.
0032H	25	P10 (INT12)	28	Magnetic tape output.
0036H	27	P11 (INT11)	30	Magnetic tape input.
0038H - 003EH	28 - 31	P12 - P15	31 - 34	Not used.

17.3 CHARACTER SET

The p-system recognizes the ASCII characters listed in the following table. The table includes the ASCII code for each character represented as both a hexadecimal and decimal value. The p-System also recognizes the six special characters shown in the second table.

Primary Character Set

Hexadecimal Value	Decimal Value	Character
20	32	Space
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=

Hexadecimal Value	Decimal Value	Character
3E	62	>
3F	63	?
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j

Hexadecimal Value	Decimal Value	Character
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7D	125	}
7E	126	~

Special Characters

Hexadecimal Value	Decimal Value	Character
5B	91	[
5C	92	\
5D	93	]
5E	94	^
5F	95	_
60	96	Grave accent

17.4 ASSEMBLER DIRECTIVE TABLE

Assembler directives let you supply data to be included in the program and exercise control over the assembly process. The following conventions are used in the Assembler directive syntax definitions.

Special characters and items in capital letters must be entered as shown.

Items within angle brackets (<>) are defined by the user.

Items within square brackets ([]) are optional.

The word "or" indicates a choice between two items.

Items in lower-case letters are generic names for classes of items.

The following terms are names for classes of items as used in this section.

b = the occurrence of one or more blanks.

integer = any legal integer constant as defined in Section 3.3.4.

label = any legal label as defined in Section 3.4.1.

expression = any legal expression as defined in Section 3.3.5.

value = any label, constant, or expression. The default value is 0.

valuelist = a list of zero or more values separated by commas.

identifier = a legal identifier as defined in Section 3.3.2.

idlist = a list of one or more identifiers separated by commas.

id:integer list = a list of one or more identifier-integer pairs separated by a colon, with each pair separated by a comma. The colon-integer part is optional and has a default value of 1.

comment = any legal comment as defined in Section 3.4.4.

character string = any legal character string as defined in Section 3.3.3.

file identifier = any legal name for a p-System text file.

Directive	Form	Section
.ABSOLUTE	[b] .ABSOLUTE [<comment>]	13.8
.ALIGN	[b] .ALIGN b <value> [<comment>]	13.3
.ASCII	[<label>] [b] .ASCII b <character string> [<comment>]	13.2
.ASCIILIST	[b] .ASCIILIST [<comment>]	13.4
.ASECT	[b] .ASECT [<comment>]	13.8
.BLOCK	[<label>] [b] .BLOCK b <length> [,<value>] [<comment>]	13.2
.BYTE	[<label>] [b] .BYTE b [valuelist] [<comment>]	13.2
.CONDLIST	[b] .CONDLIST [<comment>]	13.4
.CONST	[b] .CONST [b] <idlist> [<comment>]	13.5
.DEF	[b] .DEF [b] <idlist> [<comment>]	13.5
.ELSE	[b] .ELSE [<comment>]	13.6
.END	[<label>] [b] .END	13.1
.ENDC	[b] .ENDC [<comment>]	13.6
.ENDM	[b] .ENDM [<comment>]	13.7
.EQU	<label> [b] .EQU b <value> [<comment>]	13.2
.FUNC	[b] .FUNC [b] <identifier> [,<integer>] [<comment>]	13.1
.IF	[b] .IF <expression> [= or <> <expression>] [<comment>]	13.6
.INCLUDE	[b] .INCLUDE [b] <file identifier> [b <comment>]	13.8
.INTERP	valid when used in <expression>	13.5
.LIST	[b] .LIST	13.4
.MACRO	[b] .MACRO [b] <identifier> [<comment>]	13.7
.MACROLIST	[b] .MACROLIST	13.4
.NARROWPAGE	[b] .NARROWPAGE [<comment>]	13.4
.NOASCIILIST	[b] .NOASCIILIST [<comment>]	13.4
.NOCONDLIST	[b] .NOCONDLIST [<comment>]	13.4
.NOLIST	[b] .NOLIST	13.4
.NOMACROLIST	[b] .NOMACROLIST	13.4
.NOPATCHLIST	[b] .NOPATCHLIST	13.4
.NOSYMTABLE	[b] .NOSYMTABLE [<comment>]	13.4
.ORG	[b] .ORG b <value> [<comment>]	13.3
.PAGE	[b] .PAGE	13.4
.PAGEHEIGHT	[b] .PAGEHEIGHT [b] <integer> [<comment>]	13.4
.PATCHLIST	[b] .PATCHLIST	13.4
.PRIVATE	[b] .PRIVATE [b] <id:integer list> [<comment>]	13.5
.PROC	[b] .PROC b <identifier> [,<integer>] [<comment>]	13.1
.PSECT	[b] .PSECT [<comment>]	13.8
.PUBLIC	[b] .PUBLIC [b] <idlist> [<comment>]	13.5

Directive	Form	Section
.RADIX	[b] .RADIX [b] <integer> [<comment>]	13.8
.REF	[b] .REF [b] <idlist> [<comment>]	13.5
.RELFUNC	[b] .RELFUNC [b] <identifier> [,<integer>] [<comment>]	13.1
.RELPROC	[b] .RELPROC b <identifier> [,<integer>] [<comment>]	13.1
.TITLE	[b] .TITLE b <character string> [<comment>]	13.4
.WORD	[<label>] [b] .WORD b <valuelist> [<comment>]	13.2

17.5 HEXADECIMAL INSTRUCTION TABLE

The following table lists the TMS9900 assembly language instructions, their format, and the section in which they are described. They are in order according to their hexadecimal operation code. For an alphabetical listing by their mnemomic operation code, see Section 17.6. See Section 5 for an explanation of the format.

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
0200	LI	Load Immediate	VIII	10.1
0220	AI	Add Immediate	VIII	6.4
0240	ANDI	AND Immediate	VIII	11.1
0260	ORI	OR Immediate	VIII	11.2
0280	CI	Compare Immediate	VIII	8.3
02A0	STWP	STore Workspace Pointer	VIII	10.7
02C0	STST	STore STatus	VIII	10.6
02E0	LWPI	Load Workspace Pointer Immediate	VIII	10.3
0300	LIMI	Load Interrupt Mask Immediate	VIII	10.2
0340	IDLE	IDLE	VII	9.6
0360	RSET	ReSET	VII	9.6
0380	RTWP	ReTurn with Workspace Pointer	VII	7.17
03A0	CKON	ClocK ON	VII	9.6
03C0	CKOF	ClocK OFf	VII	9.6
03E0	LREX	Load or REstart eXecution	VII	9.6
0400	BLWP	Branch And Load Workspace Pointer	VI	7.3
0440	B	Branch	VI	7.1
0480	X	EXecute	VI	7.18
04C0	CLR	CLeaR operand	VI	11.5
0500	NEG	NEGate	VI	6.11
0540	INV	INVert	VI	11.4
0580	INC	INCrement	VI	6.8
05C0	INCT	INCrement by Two	VI	6.9
0600	DEC	DECrement	VI	6.5
0640	DECT	DECrement by Two	VI	6.6
0680	BL	Branch and Link	VI	7.2
06C0	SWPB	SWaP Bytes	VI	10.8

APPENDICES

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
0700	SETO	SET to One	VI	11.6
0740	ABS	ABSolute value	VI	6.3
0800	SRA	Shift Right Arithmetic	V	12.1
0900	SRL	Shift Right Logical	V	12.2
0A00	SLA	Shift Left Arithmetic	V	12.3
0B00	SRC	Shift Right Circular	V	12.4
1000	JMP	Unconditional JuMP	II	7.11
1100	JLT	Jump Less Than	II	7.10
1200	JLE	Jump if Low Or Equal	II	7.9
1300	JEQ	Jump EQual	II	7.4
1400	JHE	Jump High Or Equal	II	7.6
1500	JGT	Jump Greater Than	II	7.5
1600	JNE	Jump Not Equal	II	7.13
1700	JNC	Jump No Carry	II	7.12
1800	JOC	Jump On Carry	II	7.16
1900	JNO	Jump No Overflow	II	7.14
1A00	JL	Jump if logical Low	II	7.8
1B00	JH	Jump if logical High	II	7.7
1C00	JOP	Jump Odd Parity	II	7.15
1D00	SBO	Set CRU Bit to One	II	9.2
1E00	SBZ	Set CRU Bit to Zero	II	9.3
1F00	TB	Test Bit	II	9.5
2000	COC	Compare Ones Corresponding	III	8.4
2400	CZC	Compare Zeros Corresponding	III	8.5
2800	XOR	EXclusive OR	III	11.3
2C00	XOP	EXtended OPeration	IX	7.19
3000	LDCR	LoaD CRU	IV	9.1
3400	STCR	STore CRU	IV	9.4
3800	MPY	MultiPlY	IX	6.10
3C00	DIV	DIVide	IX	6.7
4000	SZC	Set Zeros Corresponding	I	11.9
5000	SZCB	Set Zeros Corresponding, Byte	I	11.10
6000	S	Subtract words	I	6.12
7000	SB	Subtract Bytes	I	6.13
8000	C	Compare words	I	8.1
9000	CB	Compare Bytes	I	8.2
A000	A	Add words	I	6.1

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
B000	AB	Add Bytes	I	6.2
C000	MOV	MOVe words	I	10.4
D000	MOVB	MOVe Bytes	I	10.5
E000	SOC	Set Ones Corresponding	I	11.7
F000	SOCB	Set Ones Corresponding, Byte	I	11.8

17.6 ALPHABETICAL INSTRUCTION TABLE

The following table lists the TMS9900 assembly language instructions, their format, and the section in which they are described. They are in alphabetical order by their mnemonic operation code. For a listing in order according to their hexadecimal operation code, see Section 17.5. See Section 5 for an explanation of the format.

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
A000	A	Add words	I	6.1
B000	AB	Add Bytes	I	6.2
0740	ABS	ABSolute value	VI	6.3
0220	AI	Add Immediate	VIII	6.4
0240	ANDI	AND Immediate	VIII	11.1
0440	B	Branch	VI	7.1
0680	BL	Branch and Link	VI	7.2
0400	BLWP	Branch And Load Workspace Pointer	VI	7.3
8000	C	Compare words	I	8.1
9000	CB	Compare Bytes	I	8.2
0280	CI	Compare Immediate	VIII	8.3
03C0	CKOF	ClocK OFf	VII	9.6
03A0	CKON	ClocK ON	VII	9.6
04C0	CLR	CLeaR operand	VI	11.5
2000	COC	Compare Ones Corresponding	III	8.4
2400	CZC	Compare Zeros Corresponding	III	8.5
0600	DEC	DECrement	VI	6.5
0640	DECT	DECrement by Two	VI	6.6
3C00	DIV	DIVide	IX	6.7
0340	IDLE	IDLE	VII	9.6
0580	INC	INCrement	VI	6.8
05C0	INCT	INCrement by Two	VI	6.9
0540	INV	INVert	VI	11.4
1300	JEQ	Jump EQual	II	7.4
1500	JGT	Jump Greater Than	II	7.5
1B00	JH	Jump if logical High	II	7.7
1400	JHE	Jump High Or Equal	II	7.6
1A00	JL	Jump if logical Low	II	7.8
1200	JLE	Jump if Low Or Equal	II	7.9
1100	JLT	Jump Less Than	II	7.10

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
1000	JMP	Unconditional JuMP	II	7.11
1700	JNC	Jump No Carry	II	7.12
1600	JNE	Jump Not Equal	II	7.13
1900	JNO	Jump No Overflow	II	7.14
1800	JOC	Jump On Carry	II	7.16
1C00	JOP	Jump Odd Parity	II	7.15
3000	LDCR	LoaD CRU	IV	9.1
0200	LI	Load Immediate	VIII	10.1
0300	LIMI	Load Interrupt Mask Immediate	VIII	10.2
03E0	LREX	Load or REstart eXecution	VII	9.6
02E0	LWPI	Load Workspace Pointer Immediate	VIII	10.3
C000	MOV	MOVe words	I	10.4
D000	MOVB	MOVe Bytes	I	10.5
3800	MPY	MultiPlY	IX	6.10
0500	NEG	NEGate	VI	6.11
0260	ORI	OR Immediate	VIII	11.2
0360	RSET	ReSET	VII	9.6
0380	RTWP	ReTurn with Workspace Pointer	VII	7.17
6000	S	Subtract words	I	6.12
7000	SB	Subtract Bytes	I	6.13
1D00	SBO	Set CRU Bit to One	II	9.2
1E00	SBZ	Set CRU Bit to Zero	II	9.3
0700	SETO	SET to One	VI	11.6
0A00	SLA	Shift Left Arithmetic	V	12.3
E000	SOC	Set Ones Corresponding	I	11.7
F000	SOCB	Set Ones Corresponding, Byte	I	11.8
0800	SRA	Shift Right Arithmetic	V	12.1
0B00	SRC	Shift Right Circular	V	12.4
0900	SRL	Shift Right Logical	V	12.2
3400	STCR	STore CRU	IV	9.4
02C0	STST	STore STatus	VIII	10.6
02A0	STWP	STore Workspace Pointer	VIII	10.7
06C0	SWPB	SWaP Bytes	VI	10.8
4000	SZC	Set Zeros Corresponding	I	11.9

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
5000	SZCB	Set Zeros Corresponding, Byte	I	11.10
1F00	TB	Test Bit	II	9.5
0480	X	EXecute	VI	7.18
2C00	XOP	EXtended OPeration	IX	7.19
2800	XOR	EXclusive OR	III	11.3

17.7 PROGRAM DEVELOPMENT WITH MULTI-DRIVE SYSTEMS

Section 1 describes the use of the Assembler with a single-drive system. With a single drive, the Assembler diskette must be on-line during the entire process, which limits the size of the programs which you may assemble. The following describe using the System with two or three drives.

17.7.1 Two-Drive System

Two disk drives allow you much more flexibility than a single-drive system. To efficiently use two drives, first place a diskette containing the Pascal Compiler and Editor in #5 and a diskette with the Filer in #4. Create the Pascal program that is to call the assembly language program on a diskette in #4. Then place the Assembler, Linker, and Editor programs on one diskette, place that diskette in #5, and create the assembly language program on the diskette in #4. Then use the Linker program to link the Pascal and assembly language programs as described in Section 1. This allows you to develop quite large programs, with the software needed always on line.

Once the development is complete, the source and object code files can be copied to an applications diskette and deleted from the diskette which contains the Filer.

17.7.2 Three-Drive System

Three drives provide the most convenient and flexible development system. The Assembler, Linker, and Editor should be placed on one diskette and placed in #5. The Filer diskette should be placed in #4. The source and object code of the program you are developing can then be put on the diskette in #9.

SECTION 18: IN CASE OF DIFFICULTY

1. Be sure that the diskette you are using is the correct one. Use the L(dir (list directory) command in the Filer to check for the correct diskette or program.

2. Ensure that your Memory Expansion unit, P-Code peripheral, and Disk System are properly connected and turned on. Be certain that you have turned on all peripheral devices and have inserted the appropriate diskette before you turn on the computer.

3. If your program does not appear to be working correctly, end the session and remove the diskette from the disk drive. Reinsert the diskette, and follow the "Set-Up Instructions" carefully. If the program still does not appear to be working properly, remove the diskette from the disk drive, turn the computer and all peripherals off, wait 10 seconds, and turn them on again in the order described above. Then load the program again.

4. If you are having difficulty in operating your computer or are receiving error messages, refer to the "Maintenance and Service Information" and "Error Messages" appendices in your User's Reference Guide or UCSD p-System P-Code manual for additional help.

5. If you continue to have difficulty with your Texas Instruments computer or the UCSD p-System Pascal Compiler package, please contact the dealer from whom you purchased the unit or program for service directions.

THREE-MONTH LIMITED WARRANTY
HOME COMPUTER SOFTWARE MEDIA

Texas Instruments Incorporated extends this consumer warranty only to the original consumer purchaser.

WARRANTY COVERAGE

This warranty covers the case components of the software package. The components include all cassette tapes, diskettes, plastics, containers, and all other hardware contained in this software package ("the Hardware"). This limited warranty does not extend to the programs contained in the software media and in the accompanying book materials ("the Programs").

The Hardware is warranted against malfunction due to defective materials or construction. **THIS WARRANTY IS VOID IF THE HARDWARE HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIAL OR WORKMANSHIP.**

WARRANTY DURATION

The Hardware is warranted for a period of three months from the date of original purchase by the consumer.

WARRANTY DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE PRODUCT OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

PERFORMANCE BY TI UNDER WARRANTY

During the three-month warranty period, defective Hardware will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below. The replacement Hardware will be warranted for a period of three months from the date of replacement. TI strongly recommends that you insure the Hardware for value prior to mailing.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U. S. Residents:
Texas Instruments Service Facility
P. O. Box 2500
Lubbock, Texas 79408

Canadian Residents only:
Geophysical Services Incorporated
41 Shelley Road
Richmond Hill, Ontario, Canada L4C5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments Consumer Service
6700 Southwest 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

Texas Instruments Consumer Service
831 South Douglas Street
El Segundo, California 90245
(213) 973-1803

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAMS

The following should be read and understood before purchasing and/or using the software media.

TI does not warrant the Programs will be free from error or will meet the specific requirements of the consumer. The consumer assumes complete responsibility for any decisions made or actions taken based on information obtained using the Programs. Any statements made concerning the utility of the Programs are not to be construed as express or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE PROGRAMS AND MAKES ALL PROGRAMS 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 THE PROGRAMS AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE SOFTWARE MEDIA. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE USER OF THE PROGRAMS.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

INDEX

9900.opcodes 19

A
Absolute expressions 34
Absolute sections 42
Addressing modes 43
Assembler directives 191, 236
Assembler output 217
Assembly, conditional 208

B
Binary constants 32

C
Character constants 34
Character set 31, 233
Character strings 32
Code listing 220
Comment field 40
Conditional assembly 208
Conditional expressions 209
Console: 21
Constants 32
CRU addressing 48
CRU structure 230

D
Decimal constants 32
Difficulty, in case of 246
Directives, assembler 191, 236

E
Errors 23, 219
Example 222
Example, absolute expressions 42
Example, arithmetic 85
Example, context switching 115
Example, CRU use 143
Example, execute 122
Example, expressions 37
Example, extended operations 121
Example, labels 39
Example, load and move instructions 157
Example, passing data to subroutines 119
Example, SBO 144
Example, SBZ 144
Example, shift instructions 189
Example, subroutines 113
Example, TB 144
Expansion RAM 227
Expressions 34
Expressions, conditional 209
External references 220

F
Fields 38
Format, source file 41
Format, source statement 38
Formats, instruction 52
Forward references 220

G
Global declarations 41
GROM 227

H
Hexadecimal constants 33

I
Input/output files 20
Instruction formats 52
Instruction table 239, 242
Interrupt handling 231

K
Keys, special 17

L
Label field 38
Labels in macros 215
Linker.info 23
Listing prompt 21
Listing.text 21

M
Macro language 210
Memory organization 225
Multiple code lines 220

O
Object code format 30
Octal constants 33
Op-code field 39
Op-codes 19
Operand field 40
Operators 36
Organization, memory 225
Output modes 22
Output, assembler 217

P
Parameter passing 213
Printer: 21
Program counter register 24
Program development 245

R
References 220
Registers 24
Relocatable expressions 34
Remout: 21
ROM 227

S
Source code format 31
Source file format 41
Source listing 218
Source statement format 38
Special keys 17
Status bits 26
Status register 24
Symbol table 221
System.wrk.code 20

V
Video Display Processor (VDP) memory 228

W
Workspace pointer register 24

TEXAS INSTRUMENTS
INCORPORATED

Texas Instruments invented the integrated circuit,
the microprocessor, and the microcomputer.
Being first is our tradition.