SPNU002B

TMS7000 Assembly Language Programmer's Guide

8-Bit Microcomputer Family

TEXAS INSTRUMENTS

Manual Update

MANUAL TITLE: TMS7000 Assembly Language Programmer's Guide

REVISION CHANGE: B to C MANUAL UPDATE NUMBER: SPBZ011

ECN NUMBER: 517192 PRINTING DATE: February 1984 DATE OF CHANGE: July 1984

This sheet accompanies the manual which has the following part numbers:

ENGINEERING P/N: 1602127-9701 SP NUMBER: SPNU002B

PAGE	CHANGE OR ADD
2-13	When the immediate value is greater than >7F and the user precedes this immediate value with %# (immediate and negate unary operations), the assembler correctly calculates the value but issues an error message. The error message "EXPRESSION OUT OF BOUNDS" should be ignored. See following example:

```
TEST        TMS7000   MACRO ASSEMBLER  VAX/VMS  2.1 83.088     14:07:07   6/13/84
                                                                 PAGE 0001

 0001                *
 0002                *  VAX X-SUPPORT TEST SOFTWARE
 0003                *
 0004                       IDT    'TEST'
 0005 F000                  AORG    >F000
 0006 F000   52             MOV    %>10,B
      F001   10
 0007 F002   0D             LDSP
 0008 F003   01             IDLE
 0009 F004   28             ADD    %#>40,A
      F005   BF
 0010 F006   28             ADD    %#>7F,A
      F007   80
 0011 F008   28             ADD    %#>80,A
      F009   7F
 *********  EXPRESSION OUT OF BOUNDS
 0012                       END
0001 ERROR, 0000 WARNINGS, LAST ERROR AT  0011
```

3-5	Insert the attached sheet for page 3-5.

IMPORTANT NOTICES

TMS 7000 ASSEMBLY LANGUAGE PROGRAMMER'S GUIDE

TABLE OF CONTENTS

SECTION 3: ASSEMBLY INSTRUCTIONS

SECTION 6: PROGRAM LINKING

SECTION 7: ASSEMBLER OUTPUT

SECTION 8: MACRO ASSEMBLER LANGUAGE

APPENDICES

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1

INTRODUCTION

1.1 GENERAL

Assembly Language is a computer-oriented language for writing programs, consisting of mnemonic instructions and assembler directives. In assembly instructions, symbolic addresses are assigned to memory locations and specify instructions by means of symbolic operation codes called mnemonic operation codes. Instruction operands are specified by means of symbolic addresses, numbers, and expressions consisting of symbolic addresses and numbers.

Assembler directives control the process of converting an Assembly Language program into a machine language program, place data in the program, and assign symbols to values to be used in the program. Assembler directives that place data in memory locations allow the user to assignment of symbolic addresses to those locations.

Assembly Language is computer-oriented in that the mnemonic operation correspond directly with machine instructions. The chief advantage of an Assembly Language over machine language is that the mnemonic symbols are easier to use and easier to remember than the binary zeros and ones of machine language. Other advantages are the use of expressions as operands and the use of decimal numbers in expressions and as operands.

This manual describes the construction of Assembly Language programs for Texas Instrument's TMS 7000 family of 8-bit microcomputers. Topics covered include general programming information, discussion of addressing modes and instruction types, a definition of instructions, discussion of user application techniques, and descriptions of source and cross-reference listings, object code, and normal and abnormal errors.

1.2 ASSEMBLY LANGUAGE APPLICATION

An Assembly Language program (the source program) must be processed by an Assembler to obtain a machine language program that can be execute by the computer. Changing a source program to object code is called assembling because the process converts the mnemonic instruction to binary values, then associates them with absolute or relocatable binary addresses to form a machine language instruction.

To illustrate the function of Assembly Language in the development of programs, consider the following steps in program development:

1) Define the problem.

2) Flowchart the solution to the problem.

3) Code the solution by writing Assembly Language statements (machine instructions and assembler directives) that correspond to the steps of the flowchart.

4) Prepare the source program by writing the statements on the medium appropriate to the installation; e.g., enter a file on a disk, keypunch the statements, etc.

5) Execute the Assembler to assemble the machine language object code corresponding to the source program.

6) Debug the resulting object code by loading and executing the object code and making corrections indicated by the results of executing the object code.

7) Repeat steps 5 and 6 until no further correction is required.

The use of Assembly Language in program development relieves the programmer of the tedious task of writing machine language instructions and keeping track of binary machine addresses within the program. Figure 1-1 also illustrates this procedure.

```
        +--------+
        | BEGIN  |
        +--------+
            |
            V
       +----------+
       | DEFINE   |
       | PROBLEM  |
       +----------+
            |
            V
      +------------+
      | FLOWCHART  |
      | SOLUTION   |
      +------------+
            |<----------------+
            V                 |
      +------------+          |
      |    CODE    |          |
      | SOLUTION   |          |
      +------------+          |
            |                 |
            V                 |
      +------------+          |
      |   INPUT    |          |
      |   CODE     |          |
      +------------+      +--------+
            |             | DEBUG  |
            V             +--------+
      +------------+          |
      |  EXECUTE   |          |
      | ASSEMBLER  |          |
      +------------+          |
            |                 |
            V                 |
      +------------+          |
      | LOAD AND   |          |
      | EXECUTE    |          |
      | OBJ CODE   |          |
      +------------+          |
            |                 |
            V                 |
      +------------+   YES    |
      | ANY        |          +
      | ERRORS ?   |__________
      +------------+
            |  NO
            V
         +------+
         | END  |
         +------+
```

FIGURE 1-1 - DEVELOPMENT PROCESS

SECTION 2

GENERAL PROGRAMMING INFORMATION

2.1 GENERAL

The TMS7000 Assembly Language is a powerful set of instructions consisting of mnemonic operation codes (herein called mnemonics) that correspond directly to binary machine instructions. The assembly language program (the source program) must be converted to a machine language program (the object program) by a process called assembling before it can be executed by the computer. Assembling consists of converting the mnemonics to binary values and associating those values with binary addresses to create machine language instructions. Assembler directives, discussed in Section 5, control the process, place data in the object program, and assign values to the symbols to be used in the object program.

2.2 DATA AREAS

The data manipulated by the TMS7000 are organized into three areas:

- Register areas, including up to 128 general-purpose registers for data storage. In addition, the TMS7000 CPU has access to three special-purpose registers: the 16-bit Program Counter (PC), the 8-bit Status Register (ST), and the 8-bit Stack Pointer (SP).
- Program areas containing the main program and subroutines.
- The Peripheral File (PF) area used for I/O purposes.

Detailed information and illustrations of these data areas are presented in Appendix B.

2.3 THE TMS7000 INSTRUCTION SET

The TMS7000 instruction set is composed of 54 instructions that provide for the input, output, manipulation, and comparison of data. The instruction set is divided into eight functional categories. They are:

Arithmetic instructions

Branch and Jump instructions

Compare instructions

Control instructions

Load and Move instructions

Logical instructions

Shift instructions

I/O instructions

The instructions making up each category are discussed in the following paragraphs. Detailed information concerning all of these instructions is presented in Section 3.

2.3.1 Arithmetic Instructions

TMS7000 arithmetic instructions perform basic arithmetic operations on byte values. They are:

INSTRUCTION	MNEMONIC
Add With Carry	ADC
Add	ADD
Decimal Add with Carry	DAC
Decrement	DEC
Decrement Double	DECD
Decimal Subtract with Borrow	DSB
Increment	INC
Invert (Complement)	INV
Multiply	MPY
Subtract with Borrow	SBB
Subtract	SUB

2.3.2 Branch and Jump Instructions

TMS7000 branch and jump instructions transfer control to specified locations in program memory. Branch instructions are unconditional; the destination specified may be anywhere within the 64K-byte program space. Jump instructions may be conditional or unconditional; the destination specified is limited to a displacement of +127 to -128 bytes relative to the address of the next instruction in the program. Conditional jump instructions transfer control according to the state of one or more bits in the Status Register, a file register, or peripheral port.

INSTRUCTION	MNEMONIC
Branch	BR
Bit Test and Jump if One Peripheral	BTJOP
Bit Test and Jump if Zero Peripheral	BTJZP
Bit Test and Jump if One	BTJO
Bit Test and Jump if Zero	BTJZ
Call Subroutine	CALL
Decrement Register and Jump if Non Zero	DJNZ
Jump if Carry/Jump if Higher or Same	JC/JHS
Jump Unconditionally	JMP
Jump if Negative	JN
Jump if No Carry/Jump if Lower	JNC/JL
Jump if Not Zero/Jump if Not Equal	JNZ/JNE
Jump if Positive	JP
Jump if Positive or Zero	JPZ
Jump if Zero/Jump if Equal	JZ /JEQ
Return from Interrupt	RETI
Return from Subroutine	RETS
Trap to Subroutine	TRAP

2.3.3 Compare Instructions

TMS7000 compare instructions set or reset bits in the Status Register, usually in preparation for a conditional jump instruction. The compare instructions perform arithmetic comparisons on signed and unsigned 8-bit values.

INSTRUCTION	MNEMONIC
Compare	CMP
Compare A to memory	CMPA
Test A register	TSTA
Test B register	TSTB

2.3.4 Control Instructions

Control instructions affect the operation of the TMS7000. These instructions are concerned with control of the carry status bit and interrupt flag in the Status Register, and execution of IDLE and NOP directives.

INSTRUCTION	MNEMONIC
Clear Carry Bit	CLRC
Set Carry Bit	SETC
Disable Interrupts	DINT
Enable Interrupts	EINT
Idle until Interrupt	IDLE
No Operation	NOP

2.3.5 Load and Move Instructions

Load and move instructions form a comprehensive set of data movement operations, with single instructions to implement register to register, memory to register and I/O to register transfers.

INSTRUCTION	MNEMONIC
Load A register	LDA
Load Stack Pointer	LDSP
Move	MOV
Move Double	MOVD
Move to/from Peripheral	MOVP
Pop from Stack	POP
Push on Stack	PUSH
Store A register	STA
Store Stack Pointer	STSP
Swap Nibble	SWAP
Exchange with B register	XCHB

2.3.6 Logical Instructions

Logical instructions provide the capability to perform various Boolean operations on system data, memory locations and registers.

INSTRUCTION	MNEMONIC
AND	AND
Clear	CLR
Invert	INV
OR	OR
Exclusive OR	XOR

2.3.7 Rotate/Shift Instructions

Rotate/Shift instructions shift the contents of a specified register by one bit. The value of the last bit shifted out of register is placed in the carry bit of the Status Register. The resulting value is compared to zero and the results of that comparison are reflected in the zero and sign bits of the Status Register.

INSTRUCTION	MNEMONIC
Rotate Left	RL
Rotate Left through Carry	RLC
Rotate Right	RR
Rotate Right through Carry	RRC

2.3.8 I/O Instructions

Input/output instructions manipulate data in any one of the peripheral file (PF) registers. Since certain PF registers correspond to the I/O pins of the TMS7000, these instructions are used to set, reset, and test the I/O pins of the device.

INSTRUCTIONS	MNEMONIC
Move to/from Peripheral Register	MOVP
OR Peripheral File Register	ORP
AND Peripheral File Register	ANDP
XOR Peripheral File Register	XORP
Bit Test and Jump if One-Peripheral	BTJOP
Bit Test and Jump if Zero-Peripheral	BTJZP

NOTE

The particular use of peripheral file registers varies among TMS7000 family microcomputers. See the User's Guide for that particular device for details.

2.4 SOURCE STATEMENT FORMAT

An Assembly Language source program consists of source statements that may contain assembler directives, machine instructions, pseudo-instructions, or comments. Source statements scanned by the Assembler may contain four ordered fields separated by one or more blanks. These fields, label, command, operand, and comment, are discussed in the following paragraphs. Source statements containing an asterisk (*) in the first character position are comment statements, and as such, they have no affect on the assembly.

The character set accepted by the TMS7000 Assembler consists of the ASCII character set as well as special characters that are undefined in ASCII. Appendix A contains tables that list the TMS7000 Assembler character set, along with associated ASCII and Hollerith codes.

The syntax for source statements other than comment fields is:

```
[<label>] ...<mnemonic> ...[<operand>]......[<comment>]
```

- A source statement may have a label that is defined by the user.

- One or more blanks separate the label from the command mnemonic. Instruction operation codes, assembler directives, and user-defined operation codes are all included in the generic term mnemonic.

- One or more blanks separate the mnemonic from the operand (when an operand is required).

- One or more blanks separate the operand(s) from the comment field. Comments are ignored by the Assembler.

The following conventions are required:

- Items in capital letters and special characters must be entered as shown.

- Items within angle brackets (< >) are defined by the user.

- Items in lowercase letters are classes (generic names) of items.

- Items within brackets ([ ]) are optional.

- Items within braces ({ }) are alternative items; one must be entered.

- All ellipses (...) indicate that the preceding item may be repeated.

- Blanks (indicated by carets (¬)) in the definition or syntax are significant.

The last source statement of a source program is followed by the end-of-file statement for the source medium (e.g., for punched cards, a card with a slash, (/) punched in column one and an asterisk (*) in column two).

Figure 2-1 illustrates a method of entering source statements. In each of the first four statements, the label begins in column 1, the opcode in column 8, the operands in column 14, and comments in column 26.

```
EXAMPLE  TMS7000 FAMILY MACRO ASSEMBLER  DX2.1 83.074     9:15:53 7/19/83
                                                               PAGE 0001
    0001                      *---------------------------------------*
    0002                      *   EXAMPLE OF SOURCE PROGRAM INPUT       *
    0003                      *---------------------------------------*
    0005
    0005                             IDT   'EXAMPLE'
    0006  0000     C5                CLR   B
    0007  0001     80         LABEL1 MOVP  P4,A
          0002     04
    0008  0003     67                BTJZ  1,A,LABEL1
          0004     FC
    0009                             END
   NO ERRORS, NO WARNINGS
```

FIGURE 2-1 - SOURCE STATEMENT FORMAT

2.4.1 Label Field

The label field begins in character position one of the source record, extends to the first blank, and contains a symbol of up to six significant characters. The first character of the symbol must be alphabetic. Additional characters may be any alphanumeric characters. A label is optional for machine instructions and for many assembler directives. When the label is omitted, the first character position must contain a blank. A source statement consisting of only a label field is a valid statement. It has the effect of assigning the current value of the location counter to the label; this is equivalent to the following directive statement:

```
<label>  EQU  $
```

2.4.2 Command Field

The command field begins after the blank that terminates the label field, or in the first nonblank character past the first character position (which must be blank when the label is omitted). The command field is terminated by one or more blanks and may not extend past the right margin. The command field may contain one of the following opcodes:

- Mnemonic operation code of a machine instruction
- Mnemonic operation code of user defined instructions
- Assembler directive.

2.4.3 Operand Field

The operand field begins following the blank that terminates the command field and may not extend past the right margin of the source record. The operand field may contain one or more constants or expressions (described in paragraphs 2.5 and 2.7) separated by commas. The operand field is terminated by one or more blanks.

2.4.4 Comment Field

The comment field begins after the blank that terminates the operand field or the blank that terminates the command field, in the case of commands that have no operands. The comment field may extend to the end of the source record, if required, and may contain any ASCII character including blank(s). The contents of the comment field up to the end of the input record are listed in the source portion of the assembly listing and have no other effect on the assembly.

2.5 CONSTANTS

The Assembler recognizes five types of constants, each internally maintained as a 16-bit quantity: decimal integer constants, binary integer constants, hexadecimal integer constants, character constants, and assembly-time constants. They are described in the following paragraphs.

2.5.1 Decimal Integer Constants

A decimal integer constant is written as a string of decimal digits. The range of values of decimal integers is -32,768 to +65,535. Positive decimal integer constants in the range 32,768 to 65,535 are considered negative when interpreted as two's complement values.

The following are valid decimal constants:

1000	Constant equal to 1000 or >3E8
-32768	Constant equal to -32768 or >8000
25	Constant equal to 25 or >19
65535	Constant equal to 65535 ot >FFFF

2.5.2 Binary Integer Constants

A binary integer constant is written as a string of up to 16 binary digits (0/1) preceded by a question mark, "?". If less than sixteen digits are specified, the Assembler will right justify the given bits in the resulting constant.

The following are valid binary constants:

?0000000000010011	Constant equal to 19 or >0013
?0111111111111111	Constant equal to 32767 or >7FFF
?11110	Constant equal to 30 or >001E

2.5.3 Hexadecimal Integer Constants

A hexadecimal integer constant is written as a string of up to four hexadecimal digits preceded by a greater than sign, '>'. Hexadecimal digits include the decimal values '0' through '9' and the letters 'A' through 'F'.

The following are valid hexadecimal constants:

>78	Constant equal to 120
>F	Constant equal to 15
>37AC	Constant equal to 14252

2.5.4 Character Constants

A character constant is written as a string of one or two alphabetic characters enclosed in single quotes. Two consecutive single quotes are required to represent each single quote contained within a character constant. The characters are represented internally as eight-bit ASCII characters. A character constant consisting only of two single quotes (no character) is valid and is assigned the value 0000(Hex).

The following are valid character constants:

'AB'	Represented internally as >4142
'C'	Represented internally as >43 or >0043
'N'	Represented internally as >4E or >004E
'''D'	Represented internally as >2744

2.5.5 Assembly-Time Constants

An assembly-time constant is a symbol given a value by an EQU directive (see paragraph 2.4.1). The value of the symbol is determined at assembly time and is considered to be absolute or relocatable according to the relocatability of the expression, not according to the relocatability of the location counter value. Absolute value symbols may be assigned values with expressions using any of the above constant types.

2.6 SYMBOLS

Symbols are used in the label field and the operand field. A symbol is a string of alphanumeric characters, ('A' through 'Z', '0' through '9' and '$'). The first character in a symbol must be 'A' through 'Z' or '$'. No character may be blank. When more than six characters are used in a symbol, the Assembler prints all the characters, but accepts only the first six characters for processing (the Assembler also prints a warning indicating that the symbol has been truncated). Therefore, symbols must be unique in the first six characters. User-defined symbols are valid only during the assembly in which they are defined.

Symbols used in the label field become symbolic addresses. They are associated with locations in the program and must not be used in the label field of other statements. Mnemonic operation codes and assembler directive names may also be used as valid user-defined symbols when placed in the label field.

Symbols used in the operand field must be defined in the assembly, usually by appearing in the label field of a statement or in the operand field of a REF or SREF directive.

The following are examples of valid symbols:

START	Assigned the value of the location at which it appears in the label field.
ADD	Assigned the value of the location at which it appears in the label field.
OPERATION	OPERAT is assigned the value of the location where it appears in the label field.

Symbols are discussed in the paragraphs that follow.

2.6.1 Predefined Symbols

The predefined symbols are the dollar sign character ($) and the Register and Port symbols. The dollar sign character is used to represent the current location within the program. The register symbols are of the form "Rn" where 'n' is a constant in the range 0 to 255.

The peripheral file symbols are of the form Pn, where n ranges from 0 to 255.

The following are examples of a valid predefined symbols:

$	Represents the current location
R0	Represents Register 0
P0	Represents Peripheral Register 0

The symbol ST is reserved and may not be defined by the user.

2.6.2 Terms

Terms are used in the operand field of machine instruction and assembler directive. A term may be a binary, character, decimal or hexadecimal constant, an absolute assembly-time constant or a label having an absolute value.

2.6.3 Character Strings

Several assembler directives require character strings in the operand field. A character string is written as a string of characters enclosed in single quotes. For each single quote in a character string, two consecutive single quotes are required to represent the single quote. The maximum length of the string is defined for each directive that requires a character string. The characters are represented internally as eight-bit ASCII. Appendix A gives a complete list of valid characters within character strings.

The following are valid character strings:

'SAMPLE PROGRAM'	Defines a 14-character string consisting of SAMPLE PROGRAM
'PLAN ''C'''	Defines an 8-character string consisting of PLAN `C'
'OPERATOR MESSAGE : PRESS START SWITCH'	Defines a 37-character string consisting of the expression enclosed in in single quotes.

2.7 EXPRESSIONS

Expressions are used in the operand fields of assembler directives and machine instructions. An expression is a constant or symbol, a series of constants or symbols, or a series of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a minus sign (unary minus), a plus sign (unary plus), or the # symbol (unary invert). The # symbol causes the value of the logical complement of the following constant or symbol to be used. An expression may not contain embedded blanks. Symbols that are defined as external references may be operands of arithmetic instructions within certain limits, as described in paragraph 2.7.1.

2.7.1 Arithmetic Operators In Expressions

The arithmetic operators used in expressions are as follows:

+ for addition
- for subtraction
* for multiplication
/ for signed division
# for logical not

In evaluating an expression, the Assembler first negates any constant or symbol preceded by a unary minus and then performs the arithmetic operations from left to right. The Assembler does not assign precedence to any operation other than unary plus or unary minus. All operations are integer operations. The Assembler truncates the fraction in division.

For example, the expression 4+5*2 would be evaluated 18, not 14 and the expression 7+1/2 would be evaluated four, not seven.

The Assembler checks for overflow conditions when arithmetic operations are performed at assembly time and gives a warning message whenever an overflow occurs, or when the sign of the result is not as expected in respect to the operands and the operation performed. Examples where a "VALUE TRUNCATED" message is given are as follows:

```
-2*>4000      >FFFE+2     -1*>8001

>8000*2       ->8000-1    -2*>8000
```

2.7.2 Logical Operand In Expressions

If a pound sign (#) precedes a number or an expression, the number or expression is changed to its complement. All other arithmetic operations have precedence over the Logical Not (#) operation, except where modified by parenthesis.

2.7.3 Parentheses In Expressions

The Assembler supports the use of parentheses in expressions to alter the order of evaluation of the expression. Nesting of pairs of parentheses within expressions is also supported. When parentheses are used, the portion of the expression within the innermost parentheses is evaluated first. Then the portion of the expression within the next innermost pair is evaluated. When evaluation of the portions of the expression within the parentheses has been completed, the evaluation is completed from left to right. Evaluation of portions of an expression within parentheses at the same nesting level may be considered to be simultaneous. Parenthetical expressions may not be nested more than eight deep.

For example, the use of parentheses in the expression LAB1+((4+3)*7) will result in the following operation: add four to three; multiply the resulting sum by seven; add the resulting product to the value of LAB1.

2.7.4 Well-Defined Expressions

Some assembler directives require well-defined expressions in operand fields. For an expression to be well-defined, any symbols or assembly-time constants in the expression must have been previously defined. Also, the evaluation of a well-defined expression must be absolute and a well-defined expression must not contain a character constant.

2.7.5 Relocatable Symbols In Expressions

An expression that contains a relocatable symbol or relocatable constant immediately following a multiplication or division operator is illegal. Also, when the result of evaluating an expression up to a multiplication or division operator is relocatable, the expression is illegal.

If the current value of an expression is relocatable with respect to one relocatable section, a symbol of another section may not be included until the value of the expression becomes absolute. Some examples of relocatable symbols used in expressions are:

BLUE+1	The sum of the value of symbol BLUE plus one.
GREEN-4	The result of subtracting four from the value of symbol GREEN.
2*16+RED	The sum of the value of symbol RED plus the product of two times 16.
440/2-RED	The result of dividing 440 by two and subtracting the value of symbol RED from the quotient. RED must be absolute.

Table 2-1 defines the relocatability of the result for each type of operator.

TABLE 2-1 - RESULTS OF OPERATIONS ON ABSOLUTE AND RELOCATABLE ITEMS IN EXPRESSIONS

A	B	A+B	A-B	A*B	A/B
ABS	ABS	ABS	ABS	ABS	ABS(B<>0)
ABS	RELOC	RELOC	illegal	Note1	illegal
RELOC	ABS	RELOC	RELOC	Note2	Note3
RELOC	RELOC	illegal	Note4	illegal	illegal

Note 1: Illegal unless A equals zero or one. If A is one, the result is relocatable. If A is zero, the result is an absolute zero.

Note 2: Illegal unless B equals zero or one. If B is one, the result is relocatable. If B is zero, the result is an absolute zero.

Note 3: Illegal unless B equals one. If B equals one, the result is relocatable.

Note 4: Illegal unless A and B are in the same section. If A and B are in the same section, the result is absolute.

2.7.6 Externally Defined Symbols In Expressions

The Assembler allows externally defined symbols (defined in REF and SREF directives) in expressions under the following conditions:

1) Only one externally referenced symbol may be used in an expression.

2) The character preceding the referenced symbol must be a plus sign, a blank, or a comma (the @ sign is not considered). The portion of the expression preceding the symbol, if any, must be added to the symbol.

3) The portion of the expression following the referenced symbol must not include multiplication, division, or logical operations on the symbol (as for a relocatable symbol described in Subsection 2.7.4).

4) The remainder of the expression following the referenced symbol must be absolute.

The Assembler limits the user to a total of 255 external referenced symbols per module. Modules using more than 255 external symbols must be broken into smaller modules for assembly and linked using the link editor.

SECTION 3

ASSEMBLY INSTRUCTIONS

3.1 GENERAL

This section describes the mnemonic instructions of the TMS7000 Assembly Language. Detailed assembly instruction descriptions and descriptions of the addressing modes used in the Assembly Language and the instruction formats of the assembly instructions are provided. Also included are examples of programming the instructions.

To understand the material presented in this section, the user must be familiar with the data organization required by the Assembler. Detailed information on byte and word organization, the status register, peripheral files, and register files is presented in Appendix B.

3.2 OPERAND ADDRESSING MODES

The TMS7000 Assembly Language supports seven operand addressing modes. Four of these modes specify 8-bit operands only and are classified as Special Addressing Modes. The remaining three are used to generate a full 16-bit address and are classified as Extended Addressing Modes. Table 3-1 defines all seven modes.

TABLE 3-1 - ADDRESSING MODES

CLASS	ADDRESSING MODE	EXAMPLE	SEE SECTION
DIRECT	Register File Addressing	R3 3	3.2.1.1
	Peripheral File Addressing	P10	3.2.1.2
	Immediate Addressing	ANDP %>98,010	3.2.1.3
	Program Counter Relative Addressing	JMP LABEL	3.2.1.4
EXTENDED	Direct Memory Addressing	@>F476 @>THERE	3.2.2.1
	Register File Indirect Addressing	*R0 *10	3.2.2.2
	Indexed Addressing	@TABLE (B)	3.2.2.3

3.2.1 Special Addressing Modes

The Special addressing modes specify 8-bit source and destination operands. Each of these modes is discussed in the paragraphs that follow.

3.2.1.1 Register File Addressing: Register file addressing specifies a file register that contains the operand. Any register may be referenced by the expression. For example:

```
        Rn
              or
        n
```

where n is the register file number (0 <= x <= 127). In general, instructions which reference the register file include a byte which contains the register number. The following examples show the coding of instructions that have register file addressing:

```
LINE     ADDR    OBJECT   STATEMENT
                                                Note: this is a 3-
  1      0000      48     LINE1  ADD   R3,R4    byte instruction;
         0001      03                           the opcode is >48,
         0002      04                           source register >03
                                                and the destination
                                                register is >04.
```

Register R0 is called the A register, and register R1 is called the B register. The fact that A or B is an operand in the instruction is usually encoded in the opcode byte. Thus, instructions which reference the A or B registers are usually shorter. For example, the following line lists an instruction to add register R3 to A:

```
LINE     ADDR    OBJECT   STATEMENT

  2      0003      18     LINE2  ADD   R3,A
         0004      03
```

This example is only a two-byte instruction: the opcode is >18 and the source register number is >03. The various dual-operand addressing types which imply the A or B register are described in paragraph 3.3.

3.2.1.2 Peripheral File Addressing: Peripheral file addressing is used to perform I/O on the TMS7000. The Peripheral File (PF) is a 256-byte block of memory address space dedicated to the I/O interface and other on-chip peripheral functions (such as a programmable timer). Each PF register, or port, has an 8-bit PF register number. Peripheral file addressing specifies a PF register number in one byte of a multi-byte instruction. A PF number is written as:

```
Pn          where n is the port number (0 <= x <=255).
```

The PF is accessed by special peripheral file instructions, which have a P postfix on the instruction name. Each is a dual operand instruction in which the PF is the destination. The source operand is limited to the A or B registers and immediate data. Examples of the PF instructions are given below. The code also demonstrates immediate addressing. Note the use of the % sign.

```
APORT EQU   P4               PF register for 8-bit A port (input).
BPORT EQU   P6               PF register for 8-bit B port (output).
CPORT EQU   P8               PF register for 8-bit C port (I/O).
CDIR  EQU   P9               C port data direction register.
DPORT EQU   P10              PF register for 8-bit D port (I/O).
DDIR  EQU   P11              D port data direction register.
      ANDP  A,P3             Replace contents of P3 (Timer Cntl)
*                            with AND of contents of A and P3.
      MOVP  B,P0             Copy B to I/O control register (P0).
      BTJOP %>40,APORT,LAB   Test bit 6 of port A and jump to LAB
*                            if it is a '1'.
      ORP   %>01,BPORT       Set bit 0 of port B to '1'.
      ANDP  %>FE,BPORT       Set bit 0 of port B to '0'.
      MOVP  %>F0,CDIR        Setup C(3-0) input, C(7-4) output.
      XORP  %>80,CPORT       Toggle bit 7 of C port.
      MOVP  %>00,DDIR        Setup all D port bits as input.
```

An exception to the PF destination-only rule is made for the MOVP instruction, by which PF register contents may be copied to the A or B register.

```
      MOVP  DPORT,A          Copy current inputs on D to A register
```

3.2.1.3 Immediate Addressing: Immediate instructions use the contents of a byte following the opcode byte as an operand. The immediate value operand is an expression, and the value of the expression is placed in a byte following the opcode byte. The immediate operand is written as an expression preceded by a percent sign. The following examples illustrate immediate addressing:

```
      MOV   %>98,R123        Replace the contents of R123
                             with >98

      ANDP  %MASK,P10        Logically AND the value of MASK
                             and the contents of P10; copy
                             the results to P10.
```

Immediate operands may be used as the source operand in all dual operand instructions, including those with a peripheral file acting as the destination. Immediate operands will be denoted <iop> in this document.

3.2.1.4 Program Counter Relative Addressing: Program counter relative addressing is used by all jump instructions. The Assembler subtracts the target address (ta) specified from the location (pcn) of the next instruction to form a signed 8-bit relative address (ra). For example:

ra = ta - pcn

where ra must be in the range of -128 through +127. When the instruction is in relocatable code (that is, when the location counter is relocatable), then the relocation type of the evaluated address expression must match the relocation type of the current location counter. When the instruction is in absolute code, the expression must be absolute. The following example illustrates the use of program counter relative addressing:

```
        JNC     THERE           Jump to THERE if the carry status bit
                                is equal to zero.

        DJNZ    R3,LOOP         Decrement R3 and jump to LOOP if  the
                                result is non-zero.

        BTJZP   %>01,APORT,$    Keep looping as long as bit  0  of  A
                                port is a  0.
```

The Assembler will generate an error message if the -128 to +127 range is exceeded.

3.2.2 Extended Addressing Modes

Three addressing modes may be used to generate a full 16-bit address to memory. These addressing modes are Direct, Register Indirect, and Indexed. Because the TMS7000's on-chip register, peripheral files, and ROM are mapped into its 16-bit memory address space, these addressing modes may be used to reference the register file, peripheral file, and program memory data areas as well as off-chip memory.

3.2.2.1 Direct Memory Addressing: specifies a 16-bit address that contains the operand. A direct memory address is written as:

```
          <addr>
```

where <addr> is a program label or other 16-bit expression. The following are examples of instructions using direct memory addressing:

```
            LDA  @>F47D        Copy contents of memory location
                               >F47D to register A

            BR   @THERE        Branch to location THERE
```

3.2.2.2 Register File Indirect Addressing: specifies the address of a pair of register file locations which contain the address of the operand. An indirect register file address is written as

```
            *Rn
                 or
            *expr
```

where the decimal constant n or the expression (*expr) is the number of the register containing the least-significant byte of the 16-bit address. The most-significant byte of the address is contained in register n-1. For example, if an address is contained in registers R4 and R5, "*R5" must be specified to use that address. If R0 (register

A) is specified, then R255 is used for the most significant half. (*R0 is undefined for TMS7000 family devices that do not implement R255.) The following example illustrates the use of register file indirect addressing in the STA (Store A) instructon:

```
MOVD %>4358,R45     Load address into (R44,R45) pair.

STA  *R45           Copy the  contents of register  A
                    into address >4358.
```

3.2.2.3 Indexed Addressing: specifies a memory address that contains the 8-bit operand. The address is the sum of the contents of the B register and a 16-bit direct address. An indexed address is written as an expression preceded by an at sign, @, and followed by a B in parentheses:

```
@<expr>(B)
```

where <expr> is a program label or 16-bit expression. The following example illustrates the use of indexed addressing:

```
     STA  @TABLE(B)     Copy the contents of A into the
*                       memory location specified by the
*                       contents of B and the value of
*                       symbol TABLE.
```

This addressing mode is particulary suited for table lookup algorithms. When tables start at a higher address and run to a lower address, the two-byte DJNZ B instruction may be used in a loop to step through the table until the desired element is found. For example, the following subroutine searches through a table for the byte contained in A, returning with the index of that byte in B. The calling program should initialize A to the search value and B to the total size of the table. For example:

```
* LOOKUP -- TABLE LOOKUP ROUTINE
* ON ENTRANCE,   A IS SET BY CALLER TO SEARCH VALUE
*                B IS SET BY CALLER TO TABLE SIZE
* ON EXIT, B IS SET TO 1-BASED INDEX OF SEARCH VALUE
*    IN TABLE, OR ZERO IF IT IS NOT FOUND.
*
LOOKUP EQU   $
LOOP   CMPA  TABLE-1(B)  COMPARE TABLE ELEMENT TO A
       JZ    EXIT        IF EQUAL, EXIT
       DJNZ  B,LOOP      IF NOT, DECREMENT B AND LOOP
EXIT   RETS              RETURN FROM SUBROUTINE
```

3.3 INSTRUCTION TYPES

Instruction types are the combinations of operand addressing modes that are used by TMS7000 instructions. The instruction types supported are Single Register, Dual Register, Peripheral File, Simple Relative, Dual Relative, Extended Address, and Special. Each is described in the paragraphs that follow.

3.3.1 Single Register Instruction Type

The Single register instruction addressing type is used by all instructions that specify only one register in the operand field. The operand of the instruction is usually the register specified. Some instructions, however, may affect a register pair, in which case the register specified contains the least significant byte. Single register instructions generally require a byte in the instruction to specify the register number. When the A or B register is used, however, the operand is implied in the opcode; thus the register number byte is not required.

Several examples of valid single register instructions follow:

```
          INC   A       Increment A register
          DEC   R3      Decrement register 3
          RR    43      Rotate register 43 right
COUNT     EQU   R14
          DEC   COUNT   Decrement register 14
          DECD  R10     Decrement the 16-bit value
*                       in R9 and R10
```

Some TMS7000 family devices will not implement the full 256 bytes of the register file. The results obtained by executing instructions specifying non-existent registers are undefined.

The machine instruction format for each single register addressing mode is described in Table 3-2.

TABLE 3-2 - SINGLE REGISTER MACHINE INSTRUCTION FORMATS

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT
<inst> A	opcode
<inst> B	
<inst> Rd	opcode d

3.3.2 Dual Register Instruction Type

Dual register instructions specify two operands in the operand field: a source and a destination. The source may be a register or an immediate 8-bit operand. The destination is always a register. In the most general case, such instructions require 3 bytes: one for the opcode, one for the source register number (or immediate operand) and one for the destination register number. When the destination is the A or B register, the destination operand is implied in the opcode. In this case, two bytes are required: one for the opcode and one for the source operand. When B is the source and A is the destination (B to A addressing mode), only an opcode byte is required (however when A is the source and B is the destination, both an opcode byte and a source byte are required).

Table 3-3 lists the directly supported addressing modes for dual operand instructions. The MOV instruction is expanded to include A to B, A to RF, and B to RF addressing mode combinations.

TABLE 3-3 - DUAL REGISTER OPERAND ADDRESSING COMBINATIONS

SOURCE	DESTINATION		
	A	B	RF
A	N	M	M
B	X	N	M
RF	X	X	X
%<op>	X	X	X

X -- Supported for all instructions
M -- Supported for MOV instruction only
N -- Not supported

For ease and clarity of programming, combinations of operand addressing modes not directly supported may be specified in an TMS7000 Assembly Language statement. The Assembler will automatically translate them to a directly supported combination. For instance, ADD A,R3 will be translated to ADD R0,R3, which uses the RF to RF instruction type. When this translation occurs, it will take an extra byte of memory that may not have been anticipated. The machine instruction formats for the various dual register operand addressing forms are shown in Table 3-4.

TABLE 3-4 - DUAL REGISTER MACHINE INSTRUCTION FORMATS

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT		
<inst> B,A	opcode		
<inst> Rs,A <inst> Rs,B	opcode	s	
<inst> Rs,Rd	opcode	s	d
<inst> %<iop>,A <inst> %<iop>,B	opcode	iop	
<inst> %<iop>,Rd	opcode	iop	d
MOV A,B	opcode		
MOV A,Rd MOV B,Rd	opcode	D	

3.3.3 Peripheral File Instruction Type

Peripheral file instructions are the I/O instructions of the TMS7000. They are dual operand instructions in which the source is A, B, or an immediate operand and the destination is a peripheral file register. Peripheral file instructions include MOVP, ANDP, ORP, XORP, BTJOP, and BTJZP. The MOVP instruction additionally may be used to read from a PF register and copy the contents into A or B.

The peripheral file operand addressing mode combinations are summarized in Table 3-5.

TABLE 3-5 - PERIPHERAL FILE OPERAND ADDRESSING MODES

	DESTINATION		
SOURCE	A	B	PF
A	N	N	X
B	N	N	X
PF	M	M	N
%<iop>	N	N	X

X-Supported for all instructions
M-Only supported for MOVP instruc.
N-Not directly supported

The machine instruction formats for peripheral type instructions are shown in Table 3-6.

TABLE 3-6 - PERIPHERAL FILE MACHINE INSTRUCTION FORMATS

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT
<inst> A,Pn	¦ opcode ¦ ¦ n ¦
<inst> B,Pn	
<inst> %<iop>,Pn	¦ opcode ¦ ¦ iop ¦ ¦ n ¦
MOVP Pn,A	¦ opcode ¦ ¦ n ¦
MOVP Pn,B	

3.3.4 Relative Address Instruction Types

Relative address instruction type is used by most instructions that alter the flow of control (instructions not included in this type are the BRanch, CALL, TRAP, RETI, and RETS). One operand in the assembly statement for relative branch instructions is the target address (ta) to which control is transferred. The assembler calculates an 8-bit signed relative address (ra) as follows:

ra = ta - pcn

where "pcn" is the program counter for the next instruction. The target address must be in the same control section (i.e., relocatable section number or absolute) as the program counter. The relative address types can be classified as Simple Relative, Single Relative, Dual Relative, and Peripheral Relative instruction types, as described in the following subsections.

3.3.4.1 Simple Relative Address Instruction Type: requires only the target address (ta) in the operand field. These include the JMP and J<cnd> instructions where <cnd> completes the mnemonic according to the condition evaluated (e.g., JC for Jump if Carry).

The machine instruction format for simple relative addressing type is shown in Table 3-7.

TABLE 3-7 - SIMPLE RELATIVE MACHINE INSTRUCTION FORMATS

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT
<inst> <ta>	| opcode | | ra |

3.3.4.2 Single Relative Address Instruction Type: this instruction is a combination of single register and simple relative address types. There are two operands: a register number and a target address. This addressing type is used by the DJNZ (Decrement and Jump if Nonzero) instruction. The machine instruction format for single relative instructions is shown below in Table 3.8.

TABLE 3-8 - SINGLE RELATIVE MACHINE INSTRUCTION FORMATS

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT
<inst> A,<ta> <inst> B,<ta>	| opcode | | ra |
<inst> Rn,<ta>	| opcode | | n | | ra |

3.3.4.3 Dual Relative Instruction Type: a combination of dual register and simple relative instruction types. Dual relative instructions, such as BTJO (Bit Test and Jump if One), contain a destination register and a target address. The supported source and destination register combinations are the same as those specified for dual register instructions. The machine instruction format for dual relative instructions is described in Table 3-9.

TABLE 3-9 - DUAL RELATIVE MACHINE INSTRUCTION FORMATS

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT
<inst> B,A,<ta>	¦ opcode ¦ ¦ ra ¦
<inst> Rs,A,<ta> <inst> Rs,B,<ta>	¦ opcode ¦ ¦ s ¦ ¦ ra ¦
<inst> Rs,Rd,<ta>	¦ opcode ¦ ¦ s ¦ ¦ d ¦ ¦ ra ¦
<inst> %<iop>,A,<ta> <inst> %<iop>,B,<ta>	¦ opcode ¦ ¦ iop ¦ ¦ ra ¦
<inst> %<iop>,Rd,ta>	¦ opcode ¦ ¦ iop ¦ ¦ d ¦ ¦ ra ¦

3.3.4.4 Peripheral Relative Instruction Type: this instruction type is a combination of the peripheral file and simple relative instruction types. Peripheral relative instructions, such as BTJOP (Bit Test and Jump if One--Peripheral), specify three operands: an A, B, or immediate source; a PF register destination; and a target address.

The machine instruction format for peripheral relative instructions is shown in Table 3-10.

TABLE 3-10 - PERIPHERAL RELATIVE MACHINE INSTRUCTION FORMATS

```
_______________________________________________________________________
|     ASSEMBLY LANGUAGE     |                                          |
|        STATEMENT          |       MACHINE INSTRUCTION FORMAT         |
|---------------------------|------------------------------------------|
|                           |                                          |
| <inst> A,Pd,<ta>          |  +---------+ +---------+ +---------+     |
| <inst> B,Pd,<ta>          |  | opcode  | |    n    | |   ra    |     |
|                           |  +---------+ +---------+ +---------+     |
|                           |                                          |
|                           |  +---------+ +---------+ +---------+ +-----+ |
| <inst> %<iop>,Pd,<ta>     |  | opcode  | |   iop   | |    n    | | ra  | |
|                           |  +---------+ +---------+ +---------+ +-----+ |
|___________________________|__________________________________________|
```

3.3.5 Extended Address Instruction Type

Extended address type instructions are those which reference a byte via its 16-bit address in the memory space of the TMS7000. These instructions have a single operand in either direct, register indirect, or indexed operand addressing mode.

Extended address instructions include CALL (Call Subroutine) and BR (BRanch) which transfer control to any instruction in memory.

The machine instruction formats for extended address instructions are given in Table 3-11. The most significant byte of a 16-bit address in the instruction is stored first.

TABLE 3-11 - EXTENDED ADDRESS MACHINE INSTRUCTION FORMATS

```
_______________________________________________________________________
|     ASSEMBLY LANGUAGE     |                                          |
|        STATEMENT          |       MACHINE INSTRUCTION FORMAT         |
|---------------------------|------------------------------------------|
|                           |                                          |
|                           |  +----------+ +----------+ +----------+  |
|     <inst> @<addr>        |  | opcode   | | addr msb | | addr lsb |  |
|                           |  +----------+ +----------+ +----------+  |
|                           |                                          |
|                           |  +---------+ +----------+                |
|    <inst> *Rd             |  | opcode  | |    d     |                |
|                           |  +---------+ +----------+                |
|                           |                                          |
|                           |  +---------+ +----------+ +----------+   |
|    <inst> @<addr>(B)      |  |opcode   | | addr msb | | addr lsb |   |
|                           |  +---------+ +----------+ +----------+   |
|___________________________|__________________________________________|
```

3.3.6 Implied Operand Type Instructions

Implied operand type instructions are one-byte instructions whose operands, if any, are implied by the opcode itself. These include several miscellaneous instructions such as EINT (Enable Interrupts) and POP ST (Pop Status). The machine instruction format for implied operand instructions is shown in Table 3-12.

TABLE 3-12 - IMPLIED OPERAND TYPE INSTRUCTION

ASSEMBLY LANGUAGE STATEMENT	MACHINE INSTRUCTION FORMAT
<inst>	opcode

3.3.7 Special Address Type Instructions

Special address type instructions (e.g., MOVE DOUBLE) are those whose operands do not fit any of the above instruction types. The machine instruction formats for these instructions are listed with the instruction's description in Section 3.4.

3.4 INSTRUCTION DESCRIPTIONS

The following paragraphs describe the instruction set of the TMS7000. The Assembler for each TMS7000 family device will accept these instructions (in the indicated Assembly Language format). The byte count for each instruction may be determined from its instruction type and its operands. The binary opcodes and cycle counts for each instruction may vary among family members. Refer to the individual family member specification for opcode assignment and instruction timing information.

The instruction descriptions are presented in alphabetic order. The discussion of each instruction includes mnemonic, syntax, instruction type, example, definition, status bit, and application note information.

3.4.1 Add With Carry Instruction (ADC)

SYNTAX: [<label>] ...ADC ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL ADC R66,R117

DEFINITION: Add the source operand to the destination operand with carry in and store the result at the destination address.

EXECUTION RESULTS: (s) + (d) + C -> (d)

```
STATUS BITS AFFECTED:  C set to '1' on carry-out of (s)+(d) +C
                       Z set on result
                       N:   set on result
```

APPLICATION NOTES: ADC may be used to implement multi-precision addition of signed or unsigned integers. ADC with an immediate operand of zero value is equivalent to a conditional increment of the destination operand. For example, the 16-bit integer in register pair (R2,R3) may be added to the 16-bit integer in (A,B) as follows:

```
ADD  R3,B              Low order bytes added
ADC  R2,A              High order bytes added
```

3.4.2 Add Instruction (ADD)

SYNTAX: [<label>] ...ADD ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL ADD A,B

Definition: Add the source operand to the destination operand and store the result at the destination address.

EXECUTION RESULTS: (s) + (d) -> (d)

STATUS BITS AFFECTED:
C: '1' on carry-out of (s) + (d)
Z: set on result
N: set on result

APPLICATION NOTES: ADD is used to add two bytes, and may be used for signed two's complement or unsigned addition.

3.4.3 And Instruction (AND)

SYNTAX: [<label>] ...AND ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL AND A,B

DEFINITION: Logically AND the source operand to the destination operand and store the result at the destination address.

EXECUTION RESULTS: (s) .AND. (d) -> (d)

STATUS BITS AFFECTED:
C: set to '0'
N,Z: set on result

APPLICATION NOTES: AND is used to perform a logical AND of the two operands. Each bit of the 8-bit result follows the truth table:

SOURCE Operand Bit	DESTINATION Operand Bit	DESTINATION Result Bit
0	0	0
0	1	0
1	0	0
1	1	1

3.4.4 AND Peripheral File Register Instruction (ANDP)

SYNTAX: [label>] ... ANDP ...<s>,<d> [<comment>]

TYPE: Peripheral File

EXAMPLE: LABEL ANDP %>DF,P6 Clear bit 5 of B port

DEFINITION: Logically AND the source and the peripheral file register specified in the destination, and place the result in the PF register. The source may be the A or B registers, or an immediate value.

STATUS BITS AFFECTED:
C: set to '0'
Z,N: set on result

APPLICATION NOTES: ANDP may be used to clear one or more bits in the peripheral file. Thus, it may be used to reset an individual output line to zero. This may be done with an ANDP instruction where the source is an immediate operand that serves as a mask field. The example above shows how bit 5 of the I/O control register (P0) is cleared, thus disabling level-3 interrupts.

3.4.5 Bit Test And Jump If One Instruction (BTJO)

SYNTAX: [<label>] ...BTJO ...<s>,<d>,<offset> ...[<comment>]

TYPE: Dual Relative

EXAMPLE: LABEL BTJO %>41,B,THERE Jump if bit B(6) or bit B(0) is set.

DEFINITION: Logically AND the source and destination operands and do not copy the result. If the result is non-zero, then perform a program counter relative jump using the offset operand. The program counter is set to the first byte AFTER the BTJO instruction before the offset is added.

EXECUTION RESULTS: if (s).AND.(D)<>0, then PC+(offset)->PC

STATUS BITS AFFECTED:

C: set to zero
Z,N: set on (s).AND.(D)

APPLICATION NOTES: Use the BTJO instruction to test for at least one bit which has a corresponding '1' bit in each operand. ~~For example~~, the source operand can be used as a bit mask to test for '1' bits in the destination address.

3.4.6 Bit Test And Jump If One-Peripheral Instruction (BTJOP)

SYNTAX: [<label>] ..BTJOP ..<s>,<d>,<offset> ..[<comment>]

TYPE: Peripheral-Relative

EXAMPLE: LABEL BTJOP %>01,P4,THERE Test Port A(0) bit

DEFINITION: Logically AND the source and destination operands and do not copy the result. If the result is non-zero, then perform a program counter relative jump using the offset operand. The program counter is set to the first byte after the BTJOP instruction before the offset is added.

EXECUTION RESULTS: if (s).AND.(D)<>0, then PC+(offset)->PC

STATUS BITS AFFECTED:

```
                    C:   set to zero
                    Z,N: set on (s).AND.(D)
```

APPLICATION NOTES: Use the BTJOP instruction to test for at least one bit position which has a corresponding 1 in each operand. For example, the source operand can be used as a bit mask to test for one bits in the destination peripheral file register. The example above tests bit 0 of the input A port, and jumps if it is a 1.

3.4.7 Bit Test And Jump If Zero Instruction (BTJZ)

SYNTAX: [<label>] ...BTJZ ...<s>,<d>,<offset> ...[<comment>]

TYPE: Dual Relative

EXAMPLE: LABEL BTJZ %>10,23,HERE IF R3(4)='0', JUMP

DEFINITION: Logically AND the source and the inverted destination operand; do not copy the result. If the result is not equal to zero, then perform a program counter relative jump using the offset operand. The program counter is set to the first byte after the BTJZ instruction before the offset is added.

EXECUTION RESULTS: if (s).AND.(NOT d)<>0, then PC+(offset)->PC

STATUS BITS AFFECTED:

C: set to '0'
Z,N: set on (s).AND.(NOT d)

APPLICATION NOTES: Use the BTJZ instruction to test for at least one 0 bit in the destination operand which has a corresponding 1 bit in the source operand.

3.4.8 Bit Test And Jump If Zero-Peripheral Instruction (BTJZP)

SYNTAX: [<label>] ..BTJZP ..<s>,<d>,<offset> ..[<comment>]

TYPE: Peripheral Relative

EXAMPLE: LABEL BTJZP %>81,P4,THERE If Port A(0) or A(7) are 0, then jump.

DEFINITION: Logically AND the source and inverted destination operands, and do not copy the result. If the result is non-zero, then perform a program counter relative jump using the offset operand. The program counter is set to the first byte after the BTJZP instruction before the offset is added.

EXECUTION RESULTS: if (s).AND.(NOT d)<>0, then PC+(offset)->PC

STATUS BITS AFFECTED:

C: set to zero
Z,N: set on (s).AND.(NOT d)

Application notes: Use the BTJO instruction to test for at least one bit position which has a 1 in the source and an 0 in the peripheral file register. For example, the source operand can be used as a bit mask to test for zero bits in the destination peripheral file register. The example above tests bit 0 of the input A port, and jumps if it is a 0.

3.4.9 Branch Instruction (BR)

SYNTAX: [<label>] ...BR ...<d> ...[<comment>]

TYPE: Extended Address

EXAMPLES:	LABEL BR @THERE	Direct addressing
	BR @TABLE(B)	Indexed addressing
	BR *R14	Indirect addressing

DEFINITION: Branch directly to location specified by the 16-bit addressing mode. The effective address is obtained using any one of the three extended addressing modes.

EXECUTION RESULTS: (d)->PC

STATUS BITS AFFECTED: none

APPLICATION NOTES: BR may be used to branch to any location in the the program. The powerful concept of computed GOTO's is supported by the BR *Rn instruction.

An indexed branch instruction of the form BR @TABLE(B) is an extremely efficient way of executing one of several actions on the basis of some control input. This is similar to the CASE statement of Pascal and other high-level languages. For example, suppose register R3 contains a control value. The program˙ branches to label ACTION0 if R3=0, ACTION1 if R3=1, etc, for up to 128 different actions.

For Example:

```
ENTER     EQU  $     START EXECUTION HERE
     MOV  R3,B       MOVE CONTROL INPUT TO B
     RL   B          MPY BY 2 TO GET TABLE OFFSET
     BR   @TABLE(B)  BRANCH TO CORRECT "JMP ACTION"
*                    STATEMENT

DISPATCH  EQU  $     DISPATCH TABLE
     JMP  ACTION0
     JMP  ACTION1
     ...
     JMP  ACTIONn
ACTION0   EQU  $
     <code for action 0>
ACTION1   EQU  $
     <code for action 1>
     ...
```

This technique may be used to transfer control on character inputs, error codes, etc.

3.4.10 Call Instruction (CALL)

SYNTAX: [<label>] ...CALL ...<a> ...[<comment>]

TYPE: Extended Address

EXAMPLES:
```
LABEL1 CALL @LABEL4
LABEL2 CALL @LABEL5(B)
LABEL2 CALL *R12
```

DEFINITION: Push the Current PC on the stack and branch to the effective operand address.

EXECUTION RESULTS:
```
SP + 1 -> SP
PC MSByte -> stack
SP + 1 -> SP
PC LSByte -> stack
operand address -> PC
```

STATUS BITS AFFECTED: none

Application notes: CALL is used to invoke a subroutine. The PUSH and POP instructions can be used to save, pass, or restore status or register values.

3.4.11 Clear Instruction (CLR)

SYNTAX: [<label>] ...CLR ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL CLR B

DEFINITION: Replace the operand value with a zero.

EXECUTION RESULTS: 0 -> (d)

STATUS BITS AFFECTED:

C: set to '0'
N: set to '0'
Z: set to '1'

APPLICATION NOTES: CLR is used to clear or initialize any file register including the A and B registers.

3.4.12 Clear Carry Instruction (CLRC)

SYNTAX: [<label>] ...CLRC ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL CLRC

DEFINITION: Clear the carry status; the sign and zero flags are determined by the contents of the A register.

STATUS BITS AFFECTED:

```
                    C:   set to '0'
                    N,Z: set on value of A register
```

EXECUTION RESULTS: status bits set

Application notes: CLRC is used to clear the carry flag if required before an arithmetic or rotate instruction. Note that the logical and move instructions typically clear the carry status. The CLRC instruction is equivalent to the TSTA instruction.

3.4.13 Compare Instruction (CMP)

SYNTAX: [<label>] ...CMP ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL CMP R13,R89

DEFINITION: Subtract the source operand from the destination operand; do not store the result.

EXECUTION RESULTS: (d) - (s) computed

STATUS BITS AFFECTED:

C: '1' if (d) is logically greater than or equal to (s)
N: Sign of result
Z: '1' if (d) is equal to (s)

APPLICATION NOTES: CMP is used to compare the destination operand to the source operand. The N bit is set to the sign of the result of subtracting (s) from (d). The C bit is set to '1' if (d) is greater than or equal to (s), interpreting (d) and (s) as unsigned integers. For either signed or unsigned interpretations, the Z bit is set to '1' if (d) and (s) are equal.

The status bits are set upon the result of computing (d) - (s). N and Z are set on the result of this subtraction. The carry bit C is a "borrow" bit--i.e., it is '0' if (d) is logically less than (s). The difference between logical and arithmetic compares is demonstrated below:

DESTINATION	SOURCE	C	N	Z
FF	00	1	1	0
7F	00	1	0	0
80	00	1	1	0
80	7F	1	0	0
7F	7F	1	0	1
7F	80	0	1	0

As shown above, negative numbers are considered arithmetically less than, but logically greater than, positive numbers. Note that the state of the n bit does not necessarily reflect a comparison of s and d interpreted as signed two's complement 8-bit numbers.

The CMP instruction can be used with the conditional branch instructions to branch on the comparison between the destination operand (d) and the source operand(s), as shown on the next page:

JUMP INSTRUCTION	CONDITION ON WHICH JUMP IS TAKEN
JC/JHS	D logically >= S
JN	D arithmetically < S
JNC/JL	D logically < S
JNZ/JNE	D not equal to S
JP	D arithmetically > S
JZ/JEQ	D equal to S
JPZ	D arithmetically >= S

3.4.14 Compare With An Extended Instruction (CMPA)

SYNTAX: [<label>] ...CMPA ...<s>...[<comment>]

TYPE: Extended Address

EXAMPLE: LABEL CMPA @TABLE(B)

DEFINITION: Subtract the contents of the byte addressed by the operand from the contents of the A register.

EXECUTION RESULTS: (A) - <s> computed

STATUS BITS AFFECTED:

C:	'1' if (A) is logically greater than or equal to <s>.
N:	'1' if (A) is arithmetically less than <s>
Z:	'1' if (A) is equal to <s>

APPLICATION NOTES: CMPA may be used to compare a long-addressed operand (e.g., via direct, indirect, or indexed addressing modes) to the A register. It is especially useful in table lookup programs in which the table is stored either in extended memory or in the program ROM. The status bits are set exactly as if the register A were the destination (d) and the addressed byte the source (s). See the CMP instruction for programming techniques using the CMPA instruction.

3.4.15 Decimal Add With Carry Instruction (DAC)

SYNTAX: [<label>] ...DAC ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL DAC %24,A

DEFINITION: Add the source operand to the destination operand with carry in and store the result at the destination address. Each operand is a two-digit integer using BCD format.

EXECUTION RESULTS: (s) + (d) + C -> (d)

STATUS BITS AFFECTED:

C:	'1' if value of (s) + (d) + C >= 100
N:	set on result
Z:	set on result

APPLICATION NOTES: DAC is used to add bytes in binary-coded decimal (BCD) form. Each byte is assumed to contain two BCD digits. Operation of DAC is undefined for non-BCD operands. DAC with an immediate operand of zero value is equivalent to a conditional increment of the destination operand. The DAC instruction automatically performs a decimal adjust of the binary sum of (s)+(d)+C. The carry bit is added to facilitate adding multi-byte BCD strings, and so the carry bit must be cleared before execution of the first DAC instruction.

EXAMPLE: Add '1234' (STR1) to '5678' (STR2) in binary coded decimal form. Each operand is stored as a 2-byte BCD string with the most significant digits first. Assume operand STR1 is stored in R3 and R4, containing values >12 and >34 respectively. STR2 is stored in R5 and R6 as >56 and >78. The result would be the string >69,>12, representing the number 6,912. Assume STR1 is stored in registers R3,R4 and STR2 in R5,R6. The code to add STR1 and STR2 is:

```
CLRC          CLEAR CARRY IF NOT ALREADY CLEAR
DAC  R4,R6    ADD LOW BYTES
DAC  R3,R5    ADD HIGH BYTES PLUS CARRY
```

The result will be left in STR2 (i.e., register pair R5,R6).

The following subroutine adds packed decimal strings of less than 256 bytes (512 digits) stored at memory locations STR1 and STR2 together, placing the result in STR2. The strings must be stored with the most significant byte first.

```
* Decimal Addition Subroutine
*   On input: B = length of string (number of bytes)
*             Stack must have 3 available bytes.
*   On output: STR2 = STR1+STR2
*
*
         CLRC              CLEAR CARRY BIT
         PUSH ST           SAVE STATUS ON STACK
LOOP     LDA  @STR1-1(B)   LOAD CURRENT BYTE OF STR1
         MOV  A,R2         SAVE IT IN R2
         LDA  @STR2-1(B)   LOAD NEXT BYTE OF STR2
         POP  ST           RESTORE CARRY FROM LAST ADD
         DAC  R2,A         ADD DECIMAL BYTES
         PUSH ST           SAVE CARRY FROM THIS ADD
         STA  @STR2(B)     STORE RESULT
         DJNZ B,LOOP       LOOP UNTIL DONE
         POP  ST           RESTORE STACK
         RETS
```

Notice the use of indexed addressing mode to reference the bytes of the decimal strings. Notice also the need to push the status register between decimal additions, to save the decimal carry bit. The B register is used to keep count of the number of bytes that have been added.

3.4.16 Decrement Instruction (DEC) DEC

SYNTAX: [<label>] ...DEC ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL DEC R102

DEFINITION: Subtract one from a copy of the operand and store the result in the operand address.

EXECUTION RESULTS: (d) - 1 -> (d)

STATUS BITS AFFECTED:

C:	'0' if (d) decrements from #00 to #FF; '1' otherwise.
N:	set on result
Z:	set on result

APPLICATION NOTES: The DEC instruction is used to subtract a value of one from any addressable operand. The DEC instruction is also useful in counting and addressing byte arrays.

3.4.17 Decrement Double Instruction (DECD)

SYNTAX: [<label>] ...DECD ...<rp> ...[<comment>]

TYPE: Single Register

```
EXAMPLE:  LABEL   DECD  R51  Decrement (R50,R51)
                 *               register pair
```

DEFINITION: Subtract one from the 16-bit value contained in the destination register pair. The operand is the register number of the least significant byte.

EXECUTION RESULTS: (rp) - 1 -> (rp)

STATUS BITS AFFECTED:

C:	'0' if most significant byte decrements from >00 to >FF. Otherwise, C = '1'.
N:	set on most significant byte of result
Z:	set on most significant byte of result

APPLICATION NOTES: DECD may be used to decrement 16-bit indirect addresses stored in the register file. Tables longer than 256 bytes may be scanned using this instruction. The subroutine shown on the next page searches a 500 byte table for a given byte, and returns with the (R4,R5) register pair containing the address of that byte. Register pair (R2,R3) should be initialized to the last address (i.e. highest-addressed element) of the table:

```
* LONGLOOK: LONG TABLE LOOKUP ROUTINE
*   CALLING SEQUENCE:
*      MOVD <TABLE LAST ADDR>,R3 TABLE ADDRESS -> (R2,R3)
*      MOVD <TABLE SIZE>,R5      TABLE SIZE -> (R4,R5)
*      MOV  <SEARCH VALUE>,A     SEARCH VALUE -> A
*      CALL @LONGLOOK
*
*   ON EXIT, (R2,R3) WILL CONTAIN ADDRESS OF SEARCH VALUE
*            (R4,R5) WILL CONTAIN 1-BASED INDEX OF VALUE
*            CARRY BIT WILL BE SET TO '1' IF NOT FOUND,
*            OTHERWISE IT WILL BE RESET TO '0'
LONGLOOK  EQU   $

LOOP      CMPA  *R3     CHECK CURRENT BYTE
          JZ    FOUND   IF EQUAL, WE FOUND IT (CARRY CLEARED)
          DECD  R3     IF NOT, DECREMENT TABLE ADDRESS
          DECD  R5     DECREMENT TABLE COUNT
          JNZ   LOOP    IF HIGH BYTE <> 0, CONTINUE
          CMP   %0,R5  IF LOW BYTE <> 0, CONTINUE
          JNZ   LOOP
          SETC          IF COUNT = 0, SET CARRY FOR ERROR
FOUND     RETS          RETURN FROM SUBROUTINE LONGLOOK
```

3.4.18 Disable Interrupts Instruction (DINT)

SYNTAX: [<label>] ...DINT ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL DINT

DEFINITION: Clear the interrupt enable flag in the status thus disabling further interrupts.

STATUS BITS AFFECTED:
I: set to '0'
C,N,Z: set to '0'

EXECUTION RESULTS: 0 -> interrupt enable status bit

APPLICATION NOTES: DINT is used to turn off all interrupts simultaneously. Since the interrupt enable flag is stored in the status register, the POP ST, and RETI instructions may reenable interrupts even though a DINT instruction has been executed. During the interrupt service, the interrupt enable bit is automatically cleared after the old status register value has been pushed onto the stack.

3.4.19 Decrement Register And Jump If Non-Zero Instruction (DJNZ)

SYNTAX: [<label>] ...DJNZ ...<d>,<offset> ...[<comment>]

TYPE: Single-Relative

EXAMPLE: LABEL DJNZ R15,THERE

DEFINITION: Decrement the operand and copy result to operand address. If result is non-zero, then take relative jump.

EXECUTION RESULTS: (d)-1->(d); if (d)<>0, then PC+(offset)->PC

STATUS BITS AFFECTED: None

APPLICATION NOTES: The DJNZ instruction is used for looping control.

3.4.20 Decimal Subtract With Borrow Instruction (DSB)

SYNTAX: [<label>] ...DSB ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL DSB R15,R76

DEFINITION: Subtract the source operand and borrow in from the destination operand and store the result at the destination address. Each operand is a two digit integer in packed BCD (binary coded decimal) format.

EXECUTION RESULTS: (d) - (s) - 1 + C -> (d)

STATUS BITS AFFECTED:
C: '1' no borrow required, '0' if borrow required.
N,Z: set on result

APPLICATION NOTES: DSB is used for multiprecision decimal BCD subtraction. A DSB instruction with an immediate operand of zero value is equivalent to a conditional decrement of the destination operand.

The carry status bit functions as a borrow bit, so if no borrow in is required, the carry bit should be set to '1'. This can be accomplished by executing the SETC instruction.

3.4.21 Enable Interrupts Instruction (EINT)

SYNTAX: [<label>] ...EINT ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL EINT

DEFINITION: Set the interrupt enable flag in the status thus enabling interrupts.

STATUS BITS AFFECTED:
I: set to '1'
C,N,Z: set to '1'

EXECUTION RESULTS: 1 -> interrupt enable

APPLICATION NOTES: EINT is used to turn on all enabled interrupts simultaneously. Since the interrupt enable flag is stored in the status register, the POP ST, LDST, and RETI instructions may disable interrupts even though a EINT instruction has been executed. During the interrupt service, the interrupt enable bit is automatically cleared after the old status register value has been pushed onto the stack. Thus, the EINT instruction must be included inside the interrupt service routine to permit nested or multilevel interrupts.

3.4.22 Idle Until Interrupt Instruction (IDLE)

SYNTAX: [<label>] ...IDLE ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL IDLE

DEFINITION Suspend further instruction execution until an interrupt or a reset occurs. Upon return from an interrupt, control passes to the instruction following the IDLE instruction.

STATUS BITS AFFECTED: none

APPLICATION NOTES: IDLE is used to allow the program to suspend operation until either an interrupt or reset occurs. It is the programmer's responsibility to assure that the Timer Control Register bit for Halt and Wake-up Modes (and individual interrupt enable bits in the I/O control register) are set before executing the IDLE instruction.

The IDLE instruction has a differenct affect on the TMS70C00 CMOS family devices. The IDLE inruction will cause the CMOS device to enter one of two low power modes which use a fraction of the normal operating power. In the Wake-Up Mode, the on-chip oscillator remains active, and activation of the timer interrupt or the external interrupts (RESET-, INT1-, INT3-) can be used to release the device from the low power mode. In the Halt Mode, the oscillator and time are disabled and only activation of an external interrupt will release the device from the Halt Mode.

When any TMS7000 family device is released from an IDLE instruction, program control passes to the next instruction.

3.4.23 Increment Instruction (INC)

SYNTAX: [<label>] ...INC ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL INC A

DEFINITION: ADD one to a register.

EXECUTION RESULTS: (d) + 1 -> (d)

STATUS BITS AFFECTED:

```
                C:   '1' if (d) incremented from #FF to #00;
                     '0' otherwise.
                N,Z: set on result
```

APPLICATION NOTES: INC is used to increment the value of any register.

3.4.24 Invert Instruction (INV)

SYNTAX: [<label>] ...INV ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL INV A

DEFINITION: Invert or complement all bits in the operand.

EXECUTION RESULTS: NOT (d) -> (d)

STATUS BITS AFFECTED:
C: set to '0'
N/Z : set on result

APPLICATION NOTES: INV performs a logical or one's complement of the operand. A two's complement of the operand can be made by following the INV instruction with an increment (INC).

3.4.25 Jump Unconditional Instruction (JMP)

SYNTAX: [<label>] ...JMP ...<offset> ...[<comment>]

TYPE: Simple Relative

EXAMPLE: LABEL JMP THERE

DEFINITION: Jump unconditionally to the address specified in the operand. The second byte of the JMP instruction is loaded with the 8-bit relative address of the operand. The operand address must therefore be within -128 to +127 bytes of the location of the instruction following the JMP instruction.

STATUS BITS AFFECTED: none

EXECUTION RESULTS: PC + (offset) -> PC

APPLICATION NOTES: The Assembler will indicate an error if the target address is beyond -128 to +127 bytes from the next instruction.

3.4.26 Jump On Condition Instruction (J<cnd>)

SYNTAX: [<label>] ...J<cnd> ...<offset> ...[<comment>]

TYPE: Simple Relative

EXAMPLES:
```
LABEL   JNC   THERE
LABEL   JP    HERE
```

DEFINITION: The assembler recognizes two mnemonics for each of the conditional jump instructions. One set of mnemonics reflects the actual conditon of the status bits tested. The other set reflects the particular condition of the status bits after a compare instruction (CMP or CMPA). The destination is considered compared to the source. For example, assume the A register contains >FE hex. The following instruction:

```
CMP %3,A
```

is read "Compare A to 3". The instruction:

```
JGT LABEL1
```

is equivalent to "JP LABEL1" and will not jump, because A is not greater than 3 (i.e., as a signed value). The instruction:

```
JHS LABEL2
```

is equivalent to "JC LABEL2", and will jump because A is higher than 3 (i.e., as an unsigned number).

Table 3-13 on the next page lists each conditional jump instruction, and the condition in which it will cause a jump to the location specified in the operand field:

TABLE 3-13 - CONDITIONAL JUMP INSTRUCTIONS

INSTRUCTION	MNEMONIC	CONDITION FOR JUMP (STATUS BIT VALUES)		
		CARRY	NEGATIVE	ZERO
Jump If Carry	JC	1	X	X
Jump If Equal	JEQ	X	X	1
Jump If Higher Or Same	JHS	1	X	X
Jump If Lower	JL	0	X	X
Jump If Negative	JN	X	1	X
Jump If No Carry	JNC	0	X	X
Jump If Not Equal	JNE	X	X	0
Jump If Non-zero	JNZ	X	X	0
Jump If Positive	JP	X	0	0
Jump If Positive Or Zero	JPZ	X	0	X
Jump If Zero	JZ	X	X	1

EXECUTION RESULTS: If tested condition is true, PC+offset->PC

STATUS BITS AFFECTED: none

APPLICATION NOTES: The J<cnd> instructions may be used after a CMP instruction to branch according to the relative values of the operands tested. After MOV, MOVP, LDA, or STA operations, a JZ or JNZ may be used to test if the value moved was equal to zero. JN and JPZ may be used in this case to test the sign bit of the value moved.

3.4.27 Load A Register Instruction (LDA)

SYNTAX: [<label>] ...LDA ...<s> ...[<comment>]

TYPE: Extended Address

```
EXAMPLES: LABEL1  LDA  @LABEL4        DIRECT
          LABEL2  LDA  @LABEL5(B)     INDEXED
          LABEL3  LDA  *R13           INDIRECT
```

DEFINITION: Copy the contents of the source operand address to the A register; addressing modes include direct, indexed, and indirect.

EXECUTION RESULTS: (s) -> A

STATUS BITS AFFECTED:

C: set to '0'
Z,N: set on value loaded

APPLICATION NOTES: The LDA instruction is used to read values stored in extended memory. The direct addressing provides an efficient means of directly accessing a variable in general memory. Indexed addressing provides an efficient table look-up capability for most applications. Indirect addressing allows the use of very large look-up tables and the use of multiple memory pointers since any pair of file registers can be used as the pointer. The DJNZ (Decrement and Jump if Nonzero) instruction can be used with either indexed or indirect addressing to create fast and efficient program loops or table searches.

3.4.28 Load Stack Pointer Instruction (LDSP)

SYNTAX: [<label>] ...LDSP ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL LDSP

DEFINITION: Copy the contents of the B register to the stack pointer register.

EXECUTION RESULTS: (B) -> SP

STATUS BITS AFFECTED:
C,N,Z: no effect

APPLICATION NOTES: LDSP is used to initialize the stack pointer.

3.4.29 MOVE Instruction (MOV)

```
SYNTAX:    [<label>] ...MOV ...<s>,<d> ...[<comment>]

TYPE:      Dual Register

EXAMPLES: LABEL1  MOV  A,B
          LABEL2  MOV  R32,R234
          LABEL3  MOV  %10,R3
```

DEFINITION: Copy the source operand to the destination operand address.

EXECUTION RESULTS: (s) -> (d)

```
STATUS BITS AFFECTED:
                    C:   set to '0'
                    Z,N: set on value loaded
```

APPLICATION NOTES: MOV is used to transfer values in the register file. Immediate values may be loaded into registers directly from the instruction. The fact that the A or B register is a source is implied in the MOV opcode, resulting in shorter and quicker moves from the A or B register. See the Dual Register instruction type description.

3.4.30 Move Double Instruction (MOVD)

```
SYNTAX:    [label>] ... MOVD ...<s>,<rp>  [<comment>]

TYPE:      Special, see below

EXAMPLE:   LABEL   MOVD  %>1234,R3    LOAD (R2,R3) REGISTER PAIR
                   MOVD  R5,R3        COPY (R4,R5) TO (R2,R3)
                   MOVD  %TAB(B),R3   COPY INDEXED ADDRESS TO
                                                (R2,R3)
```

DEFINITION: MOVD moves a two-byte value to the register pair indicated by the destination register number. The destination is the higher-addressed register of the register pair. The source may be a 16-bit constant, another register pair, or an indexed address. For the latter case, the source must be of the form "%ADDR(B)" where ADDR is a 16-bit constant or address. This 16-bit value is added to the contents of the B register, and the result placed in the destination register pair.

STATUS BITS AFFECTED:

```
                C:    set to '0';
                N,Z:  set on most significant byte moved
```

APPLICATION NOTES: "MOVD %ADDR,Rn" is useful for initializing register pairs to be used in indirect addressing mode. "MOVD Rs,Rd" will transfer two registers at a time. "MOVD %ADDR(B),Rn" will store an indexed address into a register pair, for use later in indirect addressing mode. That is, the contents of B are added to the 16-bit value of ADDR and the result placed in the register pair (Rn-1,Rn).

INSTRUCTION FORMAT: The instruction format of the MOVD instruction is a combination of the Extended Address and Single Register Formats, as shown below:

```
  ASSEMBLY LANGUAGE
     STATEMENT            MACHINE INSTRUCTION FORMAT

                      +--------+ +---------+ +---------+ +---------+
  MOVD %ADDR,Rd       | opcode | | addr msb| | addr lsb| |    d    |
                      +--------+ +---------+ +---------+ +---------+
                      +-------+ +---------+ +---------+
  MOVD Rs,Rd          |opcode | |    s    | |    d    |
                      +-------+ +---------+ +---------+
                      +-------+ +---------+ +---------+ +---------+
MOVD %ADDR(B),Rd      |opcode | | addr msb| | addr lsb| |    d    |
                      +-------+ +---------+ +---------+ +---------+
```

3.4.31 Move To/From Peripheral File (MOVP)

SYNTAX: [label>] ... MOVP ...<s>,<d> [<comment>]

TYPE: Peripheral File

EXAMPLE:
```
LABEL MOVP A,P2 SETUP TIMER VALUE
LABEL MOVP P4,B READ PORT A DATA
```

DEFINITION: Read or write data to the peripheral file. The destination is read before the source is written into it.

STATUS BITS AFFECTED:

C: set to '0'
Z,N: set on value moved

APPLICATION NOTES: MOVP is used to transfer values to and from the peripheral file. This may be used to input or output 8-bit quantities on the I/O ports. For example:

```
MOVP  P4,A
```

reads the data from input port 4. The instruction

```
MOVP  B,P6
```

puts the contents of the B register into I/O register 6, which is the B output port.

The peripheral file also contains control registers for the interrupt lines, the I/O ports, and the timer controls. For a full description of the peripheral file register consult the individual MLP family member specification.

During peripheral file instructions, a peripheral file port is read. The read can include output operations such as 'MOV A,P6'. If this read is undesirable because of hardware configuration, an STA (Store A) instruction with the memory-mapped address of the peripheral register can be used.

3.4.32 Multiply Instruction (MPY)

SYNTAX: [<label>] ...MPY ...<s>,<d> ...[<comment>]

TYPE: Dual Register

```
EXAMPLE: LABEL MPY R3,A MULTIPLY R3 AND A
         LABEL2 MPY %32,B SHIFT B 5 PLACES LEFT
```

DEFINITION: MPY performs an 8-bit multiply for a general source and destination operand. The 16-bit result is placed in the 'A,B' register pair with the most significant byte in A.

EXECUTION RESULTS: (s) * (d) -> (A,B)

STATUS BITS AFFECTED:
C: set to '0'
N,Z: set on most significant byte of result

APPLICATION NOTES: MPY is used to perform an 8-bit multiply. Multiplying by a power of two is a convenient means of performing double-byte shifts.

Multiple-precision multiply routines may be implemented easily with the MPY instruction. The subroutine shown on the next page implements a 16 by 8 bit multiply:

```
* MPY16X8: MULTIPLY 16-BIT NUMBER IN (R2,R3) BY
* 8 BIT NUMBER IN A. ON RETURN,
* LOW ORDER 16-BITS OF RESULT IS (R2,R3). HIGH 8 BITS
* ARE IN A.
* IF NO OVERFLOW, ZERO BIT IS '1'
* IF OVERFLOW ERROR, ZERO BIT IS '0'
* USES 3 BYTES OF STACK (NOT INCLUDING RETURN PC)
*
* CALLING SEQUENCE:
*    MOVD <16-BIT>,R3          R2 R3
*    MOV  <8-BIT>,A             x  A
*    CALL MPY16X8             ------
*                    A*R3(MSB) A*R3(LSB)
*        +           A*R2(LSB)
*        + A*R2(MSB)
*        ------------------------------
* RESULT:    A         R2        R3
*
MPY16X8   EQU  $
          PUSH B         SAVE TEMPORARY REGISTERS
          PUSH R4
          MOV  A,R4      COPY A
          MPY  R3,A      A=A*R3(MSB) B=A*R3(LSB)
          PUSH A         SAVE A*R3(MSB)
          MOV  B,R3      RESULT LSB = A*R3(LSB)
          MOV  R4,A      RESTORE A
          MPY  R2,A      A=A*R2(MSB) B=A*R2(LSB)
          POP  R2        POP A*R3(MSB) TO RESULT(MSB)
          ADD  B,R2      RESULT(MSB)= A*R3(MSB)+A*R2(LSB)
          ADC  %0,A      RIPPLE CARRY TO OVERFLOW BYTE
          POP  R4        RESTORE TEMPS
          POP  B
          RETS           RETURN
```

3.4.33 No Operation Instruction (NOP)

SYNTAX: [<label>] ...NOP ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL NOP

DEFINITION: Perform no operation.

EXECUTION RESULTS: PC + 1 -> PC

STATUS BITS AFFECTED: none

APPLICATION NOTES: NOP is useful as a pad instruction during program development, to "patch out" unwanted or erroneous instructions.

3.4.34 Or Instruction (OR)

SYNTAX: [<label>] ...OR ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL OR A,R12

DEFINITION: Logically OR the source operand to the destination operand and store the result at the destination address.

EXECUTION RESULTS: (s) .OR. (d) -> (d)

STATUS BITS AFFECTED:

C:	set to '0'
N,Z:	set on result

APPLICATION NOTES: OR is used to perform a logical OR of the two operands. Each bit of the 8-bit result follows the truth table:

SOURCE OPERAND BIT	DESTINATION OPERAND BIT	DESTINATION RESULT BIT
0	0	0
0	1	1
1	0	1
1	1	1

3.4.35 OR Peripheral File Register Instruction (ORP)

SYNTAX: [<label>] ...ORP ...<s>,<d> ...[<comment>]

TYPE: Peripheral File

EXAMPLE: LABEL ORP A,P12

DEFINITION: Logically OR the source operand to the destination peripheral file register and write the result to the peripheral file register. The source may be the A or B registers, or an immediate value.

EXECUTION RESULTS: (s) .OR. (d) -> (d)

STATUS BITS AFFECTED:

C:	set to '0'
N,Z:	set on result

APPLICATION NOTES: OR is used to perform a logical OR of the source operand with a peripheral file location, and write the result back to the peripheral file. This may be used to set an individual I/O bit, as follows:

```
        ORP       %>01,.P6       SET BIT 0 OF PF REGISTER 6 (B Port)
```

3.4.36 POP From Stack Instruction (POP)

SYNTAX: [<label>] ...POP ...<d> ...[<comment>]

TYPE: Single Register
"POP ST" Special, see below

EXAMPLES:
```
LABEL1 POP R32
LABEL2 POP ST
```

DEFINITION: Remove the top byte from the stack and copy to the operand address. Decrement the stack pointer to point to the new top-of-stack byte.

EXECUTION RESULTS: Stack top -> (d)
SP - 1 -> SP

STATUS BITS AFFECTED:

C:	set to '0'
N,Z:	set on value POPped

APPLICATION NOTES: The data stack can be used to save or to pass operands, especially during subroutines and interrupt service routines. The POP instruction pulls an operand from the stack.

The status register may be replaced with the contents on the stack by the statement:

```
POP  ST
```

This one-byte instruction is usually executed in conjunction with a previously performed "PUSH ST" instruction.

3.4.37 Push On Stack Instruction (PUSH)

SYNTAX: [<label>] ...PUSH ...<d> ...[<comment>]

```
TYPE:   Single Register
        "PUSH ST" Special, see below
```

```
EXAMPLES: LABEL1 PUSH A
          LABEL2 PUSH ST
```

DEFINITION: Increment the stack pointer and place the operand value on the stack as the new top-of-stack.

```
EXECUTION RESULTS: SP + 1 -> SP;
                   (d) -> (stack top)
```

```
STATUS BITS AFFECTED:
                   C:    set to '0'
                   N,Z:  set on value pushed
```

APPLICATION NOTES: The data stack can be used to save or pass operands, especially during subroutines and interrupt service routines. The PUSH instruction places an operand on the stack. The Status register may be pushed on the stack with the statement:

```
          PUSH ST
```

This one-byte instruction is usually executed in conjunction with a subsequently performed "POP ST" instruction. The status register is unaffected.

3.4.38 Return From Interrupt Instruction (RETI)

SYNTAX: [<label>] ...RETI ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL RETI

DEFINITION: POP the top two bytes from the stack to form the return address, POP the status from the top of stack, and branch to the return address.

```
EXECUTION RESULTS: Stack  -> PC LSByte
                   SP - 1 -> SP
                   Stack  -> PC MSByte
                   SP - 1 -> SP
                   Stack  -> ST
                   SP - 1 -> SP
```

STATUS BITS AFFECTED:
ST register loaded from stack

APPLICATION NOTES: RETI is typically the last instruction in an interrupt service routine. RETI restores the status register to its state immediately before the interrupt occurred and branches back to the program at the instruction boundary where the interrupt occurred.

3.4.39 Return From Subroutine Instruction (RETS)

SYNTAX: [<label>] ...RETS ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL RETS

DEFINITION: POP the top two bytes from the stack and branch to the resulting 16-bit address.

```
EXECUTION RESULTS: Stack  -> PC LSByte
                   SP - 1 -> SP
                   Stack  -> PC MSByte
                   SP - 1 -> SP
```

STATUS BITS AFFECTED: no effect

APPLICATION NOTES: RETS is typically the last instruction in a subroutine. RETS results in a branch to the location immediately following the subroutine call instruction.

3.4.40 Rotate Left Instruction (RL)

SYNTAX: [<label>] ...RL ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL RL R102

DEFINITION: Shift the operand one position to the left and fill the least significant bit and the carry status bit with the value of the original most significant bit; copy the result to destination addresss

EXECUTION RESULTS: Bit(n) -> Bit(n+1)
Bit(7) -> Bit(0) and Carry

STATUS BITS AFFECTED:

C:	set to bit(7) of original operand
N,Z:	set on result

APPLICATION NOTES: An example of the RL instruction is: If the B register contains the value >93, then the RL instruction changes the contents of B to >27 and sets the carry status bit.

```
+---+      +-----------------------------------+
| C | <-+--| MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB | <--+
+---+   |  +-----------------------------------+    |
        |___________________________________________|
```

3.4.41 Rotate Left Through Carry Instruction (RLC)

SYNTAX: [<label>] ...RLC ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL RLC R102

DEFINITION: Shift the operand to the left one bit position and fill the least significant bit with the original value of the carry status bit; copy the result to the destination address. Move the original operand most significant bit to the carry status bit.

```
EXECUTION RESULTS: Bit(n)->Bit(n+1)
                   Carry->Bit(0)
                   Bit(7)->Carry

STATUS BITS AFFECTED:
                   C:    set to bit(7) of original operand
                   N,Z:  set on result
```

APPLICATION NOTES: An example of the RLC instruction is: if the B register contains the value >93 and the carry status bit is a zero, then the RLC instruction changes the operand value to >26 and carry to one.

```
     +---+         +----------------------------------+
+--<--| C |<--------| MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB | <-+
|     +---+         +----------------------------------+   |
|                                                          |
|__________________________________________________________|
```

3.4.42 Rotate Right Instruction (RR)

SYNTAX: [<label>] ...RR ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL RR A

DEFINITION: Shift the operand to the right one bit position and fill the most-significant bit and the carry status bit with the value of the original least significant bit. Copy the result to operand address.

```
EXECUTION RESULTS:  Bit(n+1) -> Bit(n)
                    Bit(0)   -> Bit and Carry
```

```
STATUS BITS AFFECTED:
                   C:   set to bit(0) of original operand
                   N,Z: set on result
```

APPLICATION NOTES: An example of the RR instruction is: If the B register contains the value >93, then the "RR B" instruction changes the contents of B to >C9 and sets the carry status bit.

```
+---+     +-----------------------------------+
| C |<--+-->| MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB |<-+
+---+   |   +-----------------------------------+  |
        |__________________________________________|
```

3.4.43 Rotate Right Through Carry (RRC)

SYNTAX: [<label>] ...RRC ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL RRC R32

DEFINITION: Shift the operand to the right one bit position and fill the most significant bit from the carry status bit. Fill the carry status bit with the value of the original least significant bit.

```
EXECUTION RESULTS:  Bit(n+1)->Bit(n)
                    Carry->Bit(7)
                    Bit(0)->Carry

STATUS BITS AFFECTED:
                   C:    set to bit(0) of original operand
                   N,Z:  set on result
```

APPLICATION NOTES: An example of the RRC instruction is: If the B register contains the value >93 and the carry status bit is zero, then the 'RRC B' instruction changes the operand value to >49 and sets the carry status bit.

```
      +---+     +----------------------------------+
  +-->| C |---->| MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB |-->-+
  |   +---+     +----------------------------------+    |
  |                                                     |
  |_____________________________________________________|
```

3.4.44 Subtract With Borrow Instruction (SBB)

SYNTAX: [<label>] ...SBB ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL SBB %23,B

DEFINITION: Subtract the source operand and borrow in from the destination operand and store the result at the destination address.

EXECUTION RESULTS: (d) - (s) -1 + C -> (d)

STATUS BITS AFFECTED:

C:	set to '1' if no borrow; '0' otherwise
N,Z:	set on result.

APPLICATION NOTES: SBB is used for multiprecision two's complement subtraction. An SBB instruction with an immediate operand of zero value is equivalent to a conditional decrement of the destination operand. With (s)=0, if C='0', then (d) is decremented, otherwise it is unchanged. A borrow is required if the result is negative. In this case, the carry bit is set to '0'.

3.4.45 Set Carry Instruction (SETC)

SYNTAX: [<label>] ...SETC ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL SETC

DEFINITION: Set the carry and zero status flags and clear the sign status flag.

EXECUTION RESULTS: status bits affected

STATUS BITS AFFECTED:

C:	set to '1'
N:	set to '0'
Z:	set to '1'

APPLICATION NOTE: SETC is used to set the carry flag if required before an arithmetic or rotate instruction.

3.4.46 Store A Register Instruction (STA)

SYNTAX: [<label>] ...STA ...<d> ...[<comment>]

TYPE: Extended Address

```
EXAMPLES: LABEL1  STA  @LABEL4        DIRECT
          LABEL2  STA  @LABEL5(B)     INDEXED
          LABEL3  STA  *R13           INDIRECT
```

DEFINITION: Copy the contents of the A register to the operand address. The addressing modes are Direct, Indexed, and Indirect.

EXECUTION RESULTS: (A) -> (D)

STATUS BITS AFFECTED:

C:	set to '0'
N/Z:	set on value loaded

APPLICATION NOTES: The STA instruction is used to store values anywhere in the memory address space. The direct addressing provides an efficient means of directly accessing a variable in general memory. The indexed addressing provides an efficient table look-up capability. Indirect addressing allows the use of very large look-up tables and the use of multiple memory pointers since any pair of file registers can be used as the pointer. The 'Decrement Register and Jump if Non-Zero' instruction (DJNZ) can be used with either indexed or indirect addressing to create fast and efficient program loops or table searches.

3.4.47 Store Stack Pointer Instruction (STSP)

SYNTAX: [<label>] ...STSP ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL STSP

DEFINITION: Copy the contents of the stack pointer register to the B register.

EXECUTION RESULTS: (SP) -> (B)

STATUS BITS AFFECTED: none

APPLICATION NOTES: STSP is used to make a copy of the SP if required. This instruction can be used to test the stack size. The indexed addressing mode may be used to reference operands on the stack.

3.4.48 Subtract Instruction (SUB)

SYNTAX: [<label>] ...SUB ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL SUB R19,B

DEFINITION: Subtract the source operand from the destination operand and store the result at the destination address.

EXECUTION RESULTS: (d) - (s) -> (d)

STATUS BITS AFFECTED:

C:	set to '1' if result >=0, '0' otherwise
N,Z:	set on result

APPLICATION NOTES: SUB is used for two's complement subtraction The carry bit is set to '0' if a borrow is required, i.e. if the result is negative.

3.4.49 Swap Nibbles Instruction (SWAP)

SYNTAX: [<label>] ...SWAP ...<d> ...[<comment>]

TYPE: Single Register

EXAMPLE: LABEL SWAP R45

DEFINITION: Swap the least significant nibble of the operand with the most significant nibble and copy the result to the operand address. The SWAP instruction is equivalent to four consecutive rotate left (RL) instructions with the carry status bit set equal to the least significant bit of the result.

EXECUTION RESULTS: Bits 7:6:5:4 <-> Bits 3:2:1:0

STATUS BITS AFFECTED:

C:	set to Bit(0) of result or Bit(4) of original
Z,N:	set on result

APPLICATION NOTES: SWAP is used to manipulate four bit operands, especially during packed BCD operations.

3.4.50 Trap To Subroutine Instruction (TRAP)

SYNTAX: [<label>] ...TRAP ...<n> ...[<comment>]

TYPE: Special

EXAMPLE: LABEL TRAP 15

DEFINITION: The operand <n> is a trap number which identifies a location in the Trap Vector Table, addresses >FFD0 to >FFFF in memory. The contents of the two-byte vector location form a 16-bit trap vector to which a subroutine call is performed.

STATUS BITS AFFECTED: none

```
EXECUTION RESULTS: SP + 1       -> SP
                   PC MSByte    -> stack
                   SP + 1       -> SP
                   PC LSByte    -> stack
                   Entry vector -> PC
```

APPLICATION NOTES: TRAP is an efficient way to invoke a subroutine. The uppermost block of memory is the Trap Vector Table, and contains as many subroutine addresses as available traps for the TMS7000 family member. The subroutine addresses are stored like any other address in memory, with the least significant byte in the higher-addressed location, as shown below.

TRAP VECTOR TABLE

```
       ----------------------
>FFFF | TRAP 0 Address LSB  |
>FFFE |    "           MSB  |
>FFFD | TRAP 1 Address LSB  |
>FFFC |    "           MSB  |
>FFFB | TRAP 2 Address LSB  |
>FFFA |    "           MSB  |
 ...  .      ...            .
 ...  .      ...            .
 ...  .      ...            .
>FFE1 | TRAP 15 Address LSB |
>FFE0 |    "            MSB |
 ...  .      ...            .
 ...  .      ...            .
 ...  .      ...            .
>FFD1 | TRAP 23 Address LSB |
>FFD0 |    "            MSB |
      |_____________________|
```

A Trap subroutine address 'TRAPn' may be used as follows:

```
        ORG  FFF-2N-1
        DATA TRAPn      TRAP n SUBROUTINE ADDRESS
```

Note that TRAP 1, TRAP 2, AND TRAP 3 correspond to the hardware-invoked interrupts 1,2, and 3 respectively. The hardware-invoked interrupts, however, push the program counter and the status register before branching to the interrupt routine, while the TRAP instruction pushes only the program counter. TRAP 0 will branch to the same code executed for a system reset.

The number of traps allowed depends on the individual family member. On the TMS7000 and TMS7020 the maximum number of traps allowed is 24.

3.4.51 Test A Register Instruction (TSTA)

SYNTAX: [<label>] ...TSTA ...[<comment>]

TYPE: Implied Operand

EXAMPLE: LABEL TSTA TEST A REGISTER

DEFINITION Set the status bits on the value of the A register

EXECUTION RESULTS: C,N,Z bits set

STATUS BITS AFFECTED:

C:	set to '0'
Z,N:	set on value in A register

APPLICATION NOTES: This instruction can be used to set the status bits according to the value in the A register.

3.4.52 Test B Register Instruction (TSTB)

SYNTAX: [<label>] ...TSTB ...[<comment>]

TYPE: Implied Operand

```
EXAMPLE:  LABEL     TSTB      TEST B REGISTER
```

DEFINITION: Set the status bits on the value of the B register

EXECUTION RESULTS: C,N,Z bits set

STATUS BITS AFFECTED:

```
                    C:   set to '0'
                    Z,N: set on value in B register
```

APPLICATION NOTES: This instruction can be used to set the status bits according to the value in the B register. It may be used to clear the carry bit.

3.4.53 Exchange With B Register Instruction (XCHB)

```
SYNTAX:    [<label>] ...XCHB ...<d> ...[<comment>]

TYPE:      Single Register

EXAMPLE:   LABEL   XCHB   A
                   XCHB   R3
```

DEFINITION: Copy the destination operand to the B register; then copy the B value to the destination register.

EXECUTION RESULTS: Bits (B) <-> (d)

```
STATUS BITS AFFECTED:
                    C:   set to '0'
                    N,Z: set on original contents of B.
```

APPLICATION NOTES: XCHB is used to exchange a file register with the B register without going through an intermediate location. The XCHB instruction with the B register as the operand is equivalent to the TSTB instruction.

3.4.54 Exclusive Or Instruction (XOR)

SYNTAX: [<label>] ...XOR ...<s>,<d> ...[<comment>]

TYPE: Dual Register

EXAMPLE: LABEL XOR %>98,R125

DEFINITION: Logically exclusive OR the source operand to the destination operand and store the result at the destination address.

EXECUTION RESULTS: (s) .XOR. (d) -> (d)

STATUS BITS AFFECTED:
C: set to '0'
N,Z: set on result

APPLICATION NOTES: XOR is used to perform a bit-wise exclusive OR of the operands. The XOR instruction can be used to complement bits in the destination operand. Each bit of the 8-bit result the following truth table:

SOURCE OPERAND BIT	DESTINATION OPERAND BIT	DESTINATION RESULT BIT
0	0	0
0	1	1
1	0	1
1	1	0

3.4.55 Exclusive Or Peripheral File Register Instruction (XORP)

SYNTAX: [<label>] ...XORP ...<s>,<d> ...[<comment>]

TYPE: Peripheral File

EXAMPLE: LABEL XORP %>01,P9 REVERSE BIT C(0) DIRECTION

DEFINITION: Logically exclusive OR the source operand to the peripheral file register specified, and write the result to the peripheral file register.

EXECUTION RESULTS: (s) .XOR. (d) -> (d)

STATUS BITS AFFECTED:

C:	set to '0'
N,Z:	set on result

APPLICATION NOTES: XORP is used to perform a bit-wise exclusive OR of the operands. The XORP instruction can be used to complement bits in the destination PF register. The example above inverts bit 0 of P9, which is the port C data direction register, thus reversing the direction of the bit.

SECTION 4

USER APPLICATION NOTES

4.1 GENERAL

This section provides information and specific examples that supplement the application notes in the instruction descriptions. Programming examples are included for those instructions for which the application notes require additional explanation.

4.2 ARITHMETIC INSTRUCTIONS

The TMS7000 Instruction Set supports all arithmetic operations, as well as array indexing, loop control, and bit shifting.

4.2.1 Incrementing Instructions (INC/DAC/ADC)

The TMS7000 instruction set supports single and double-precision incrementing of any file register. These instructions can be used for arithmetic purposes, for array indexing, or for loop control.

Any file register can be directly incremented using the INC instruction. For example, the instruction sequence:

```
        INC     A
        INC     R14
```

increments the contents of both register A and register 14. The INC instruction can also be used to control a flag value as follows:

```
*          150 BYTE BUBBLE SORT

*
FLAG       EQU    R2                 COUNT
*
SORT       CLR    FLAG               RESET SWAP FLAG
           MOV    %149,B             SORT COUNT
LOOP1      LDA    @TABLE(B)
           CMPA   @TABLE-1(B)        COMPARE ADJACENT VALUES
           JL     LOOP2              IF LOWER, SKIP
           INC    FLAG               SET DETECT FLAG TO NON-ZERO
           PUSH   A                  USE STACK FOR TEMPORARY STORAGE
           LDA    @TABLE-1(B)
           STA    @TABLE(B)          SWAP OPERANDS
           POP    A
           STA    @TABLE-1(B)
LOOP2      EQU    $
           DJNZ   B,LOOP1            LOOP UNTIL ALL TABLE IS SWEPT
           BTJO   %>FF,FLAG,SORT     IF SWAP MADE, RESWEEP TABLE
```

In this bubble sort routine, the INC instruction is used to guarantee a non-zero value in the FLAG register if an operand swap is made. The FLAG value is initialized using the CLR instruction. The BTJO instruction is used to test the FLAG value to determine if a swap has been made.

A register containing Binary Coded Data (BCD) can be incremented with the DAC instruction as shown below:

```
CLRC                          Clear Carry
DAC        %1,R18             Add "1" to register 18
                              using BCD format
```

The CLRC instruction is required only if the carry status bit is not already cleared by the preceding instruction execution sequence.

Any file register can be conditionally incremented by a single instruction using the ADC instruction with a zero value immediate source operand. For example, the instruction sequence:

```
INC        R19
ADC        %0,R18
```

performs a double precision increment by unconditionally incrementing register R19 and then conditionally incrementing register R18.

4.2.2 Decrementing Instructions (DEC/DSB/DECD)

The TMS7000 instruction set supports single and double-precision decrementing of any file register. These instructions can be used for arithmetic purposes, for array indexing, or for loop control.

Any file register can be directly decremented using the DEC instruction or conditionally decremented using the SBB instruction. For example, the instruction sequence:

```
            DEC      B
            SUB      R103,R3
            SBB      %0,A
```

unconditionally decrements the B register, subtracts register R103 from register R3, and then decrements the A register if a borrow-out is generated by the subtraction.

A register containing BCD data can be decremented with the DSB instruction as shown below:

```
            SETC                    Clear Borrow
            DSB     - %1,B          BCD Decrement
```

The SETC instruction is required only if the carry status bit is not already set by the preceding instruction execution sequence. Remember that the subtraction "borrow-in" is the complement of the carry status bit.

Any pair of contiguous registers can be decremented using the double precision DECD instruction. DECD is especially powerful when used in conjunction with instructions with register indirect adddressing. For example, the instruction sequence:

```
* 'TABLE'                                START OF TABLE OF DATA
POINT       EQU       R81                TABLE POINTER
START       EQU       TABLE+1000         TABLE START
            MOVD      START,POINT       INITIALIZE POINTER
LOOP        EQU       $
            CMPA      *POINT             MATCH
            JEQ       MATCH              IF SO, EXIT
            DECD      POINT              NEXT TEST VALUE
            JC        LOOP
*
NOMTCH
*
MATCH       EQU       $
```

searches a 1000-byte table for an entry matching the contents of the A register. If a match is found, then register pair R80::R81 points at the matching table entry. The DECD instruction is used to step the pointer through the table.

4.2.3 Addition Instructions (ADD/ADC/DAC)

The TMS7000 instruction set supports both single and double-precision addition for either binary or BCD data.

The ADD and ADC instructions are used for single and multi-precision binary addition respectivly. The ADD instruction adds the two specified operands with a zero value carry-in. The ADC instruction adds the two operands with a carry-in value equal to the value of the carry status bit. Thus, the following instruction sequence:

```
ADD        R30,R120
ADC        R29,R119
ADC        R28,R118
ADC        R27,R117
```

adds the 32-bit value in registers R27-R30 to the 32 bit value in registers R117-R120. The initial instruction is ADD since no carry-in is desired. ADC is used in the next three instructions in order to ripple a carry through all 32 bits.

The DAC instruction is used to add BCD values. The carry-in value is equal to the carry status bit value. Consequently, the carry status bit must be cleared if no carry-in is desired. For example, the following sequence:

```
MOVP       P4,A
MOVP       P8,B
DAC        %>13,B
DAC        %>47,A
```

adds a BCD constant equal to 4713 to the contents of the A and B registers. Note that the MOVP instruction automatically clears the carry status bit thus eliminating the need for a CLRC instruction.

4.2.4 Subtraction Instructions (SUB/SBB)

The TMS7000 instruction set supports both single and double-precision subtraction for either binary or BCD data. The SUB and SBB instructions are used for single and multi-precision binary subtraction respectively. The SUB instruction adds the two specified operands with no borrow-in. The SBB instruction uses a borrow-in which is equal to the complement of the carry status bit. For example, the following sequence:

```
SUB   R2,R125
SBB   R3,R124
SBB   R4,R123
```

subtracts the 24-bit value in registers R2 through R4 from the 24-bit value in registers R125 through R127. Because the borrow in is the complement of the carry status bit, the following examples clears the A register:

```
MOV   %1,A
SBB   %0,A
```

Normally, however, the one-byte 'CLR A' instruction is used to place a zero value in the A register.

4.2.5 Multiply Instruction (MPY)

The MPY instruction performs an 8-bit by 8-bit multiply with a 16-bit result that is stored in the A and B registers. The most significant byte of the result is in A, and the least significant byte in B. The MPY instruction can also be used for multi-bit right or left shifts as shown below:

```
        MPY    %8,B
```

This instruction takes the value of B and multiplies it by 8. After the instruction executes, B contains the previous value, left shifted 3 bits (2**3=8) with zero fill bits. The A register can be considered to contain the previous value right shifted 5 bits (2**[8-5]=8) with zero fill bits. Thus, left or right multi-bit shifts can be performed as shown below:

IMMEDIATE MULTIPLIER	BITS RIGHT SHIFTED	BITS LEFT SHIFTED
2	7	1
4	6	2
8	5	3
16	4	4
32	3	5
64	2	6
128	1	7

Multi-precision multiplications can be conveniently performed by breaking the multiplier and multiplicand into scaled 8-bit quantities as shown below:

```
*     16 BIT MPY:                XH    XL            X VECTOR
*                           X    YH    YL            Y COEFFICIENT
*                           ------------------
*                                XLYLm  XLYL1        1 = lsb
*                         XHYLm  XHYL1               m = msb
*                         XLYHm  XLYH1
*                 + XHYHm XHYH1
*             ------------------------------
*                   RSLT3  RSLT2  RSLT1  RSLT0
*
XH       EQU   R2                Higher operand of X
XL       EQU   R3                Lower operand of X
YH       EQU   R4                Higher operand of Y
YL       EQU   R5                Lower operand of Y
RSLT3    EQU   R6                MSB of the final result
RSLT2    EQU   R7
RSLT1    EQU   R8
RSLT0    EQU   R9                LSB of the final result
*
MPY32    CLR   ACC2              Clear the present value
         CLR   ACC3
         MPY   XL,YL             Multiply LSBs
         MOV   B,RSLT0           Store LSB in Result Register 0
         MOV   A,RSLT1           Store MSB in Result Register 1
         MPY   XH,YL             Get XHYL
         ADD   R1,RSLT1          Add to existing result XLYL
         ADC   R0,RSLT2          Add carry if present
         MPY   XL,YH             Multiply to get XLYH
         ADD   R1,RSLT1          Add to existing result XLYL+XHYL
         ADC   R0,RSLT2          Add to existing results and carry
         ADC   %0,RSLT3          Add if carry present
         MPY   XH,YH             Multiply MSBs
         ADD   R1,RSLT2          Add once again to the Result Register
         ADC   R0,RSLT3          Do the final add to the Result Register
```

The 16-bit operands in registers R2-R3 and R4-R5 are multiplied to yield the 32-bit result in registers R6-R9. Note: the A and B registers are, in general cases, referred to as R0 and R1 in order to emphasize that the dual register addressing mode with A or B as the source operand is not directly supported (except for MOV instruction). The R0 or R1 address is substituted for the A or B register respectively. The assembler normally, however, automatically performs this translation.

4.3 DATA MOVEMENT INSTRUCTIONS

The TMS7000 Instruction Set supports instructions which permit simple data movement, exchange or swapping of register contents, copy register contents from one continguous register pair to another, calculations of indexed addresses, transfer peripheral file port values, and move values to an I/O port, read and store values, table look-up, searching, and loop control (of table or block transfers).

4.3.1 Register Move Instructions (MOV/XCHB/MOVD)

For simple data movement between registers, the MOV, XCHB, and MOVD instructions are used. The MOV instruction is used to copy the contents of one register into another register or load a register with an 8-bit immediate value. For example, the instruction sequence:

```
MOV   %87,A
MOV   A,R93
MOV   R93,R112
```

results in R0, R93, and R112, each containing the decimal value "87".

The XCHB (exchange with B register) instruction is used to exchange or swap the contents of the B register and any other register. Thus, the A and B registers can be exchanged with the instruction:

```
XCHB   A
```

The MOVD instruction is used to copy the contents of any two contiguous registers into any other register pair, or else load a 16-bit immediate value into a register pair. The higher-numbered register of the register pair is specified as shown below:

```
MOVD   %>18FA,R4
MOVD   R4,R117
```

In the above example, R3 and R4 are first loaded with immediate values >18 and >FA respectively. R3 and R4 are then copied into R116 and R117 respectively. The MOVD instructions are often used to initialize or to copy register pairs to be used in the indirect register addressing mode. In the following instruction sequence:

```
MOVD   %>F900,R100
LDA    *R100
```

The A register is loaded with the value of memory location >F900.

The MOVD instruction also supports a special "indexed immediate" mode. This mode adds the 8-bit B register value to a 16-bit immediate operand and stores the result in a register pair. This is equivalent to calculating an indexed address but in this case the resulting address is stored rather than used to fetch an operand. For example, the following examples place the values of >97 and >1E into registers R17 and R18 respectively:

```
MOV   %20,B
MOVD  %>971E(B),R18
```

4.3.2 I/O Move Instruction (MOVP)

The peripheral move (MOVP) instruction is used to transfer a peripheral file port value to or from the A or B register, or move an immediate value to an I/O port. There are consequently five different MOVP address combinations as shown below:

```
        MOVP    P4,A        INPUT I/O TO A
        MOVP    P4,B        INPUT I/O TO B
        MOVP    A,P6        OUTPUT A TO I/O
        MOVP    B,P6        OUTPUT B TO I/O
        MOVP    %>13,P6     OUTPUT IMMEDIATE TO I/O
```

A peripheral file port is read during ALL peripheral file instructions including output operations (e.g., "MOVP A,P4"). If this read is undesirable as a result of hardware concerns, then a (store A)STA instruction should be used with the memory-mapped address of the peripheral port.

4.3.3 Load and Store Instructions (LDA/STA/DJNZ)

The LOAD A register (LDA) and STORE A register (STA) instructions are used to read or store values anywhere in the full 64K byte address space. There are three extended addressing modes: direct, indirect, and indexed. Direct addressing provides an immediate 16-bit address which directly points to any byte in the TMS7000 memory space. The following instructions result in the transfer of the byte in location >F819 to location >7193 and the A register.

```
        LABEL     EQU       >F819
                  LDA       @LABEL
                  STA       @>7193
```

Indexed addressing also uses a 16-bit direct address. The effective 16-bit memory address, however, is formed by adding the 8-bit B register value to the 16-bit direct address. Thus, the following instructions copy the A register value into memory location >1927:

```
        MOV     %>27,B
        STA     @>1900(B)
```

Indexed addressing is normally used in table lookup, transfer, or search algorithms The 8-bit B register index value provides a range of up to 256 bytes, which is sufficient for most applications. Register indirect addressing, however, is available for applications requiring a larger table size.

A table lookup can be performed by simply placing the table index into the B register and using an indexed LDA instruction as shown below:

```
        MOVP    P37,B        INPUT   B REG
        LDA     @TABLE(B)    LOOKUP VALUE   B
```

The DJNZ instruction is especially powerful in table or block transfers. This loop control instruction decrements the specified register and transfers control if the result is non-zero. Thus, the table index can be stepped with automatic looping until the transfer is completed. For example, an 80-byte block transfer is performed by the following sequence:

```
            MOV     %80,B
LOOP        LDA     @SRC-1(B)
            STA     @DEST-1(B)
            DJNZ    B,LOOP
```

Table searches are efficiently performed through the use of the compare A register extended (CMPA) instruction. In the following example, a 150-byte table is searched for a match with a 6-byte string:

```
SEARCH      MOV     %150+1,R2       TABLE LENGTH = 150 bytes
LOOP1       MOV     %6,B            STRING LENGTH = 6 bytes
LOOP2       XCHB    R2              SWAP POINTERS, LONG STRING IN B
            DEC     B               TABLE END?  IF SO, NO MATCH FOUND
*           JZ      NOFIND
            LDA     @TABLE-1(B)     LOAD TEST CHARACTER
            XCHB    R2              SWAP POINTERS, STRNG POINTER IN B
            CMPA    @STRING-1(B)    MATCH?
            JNE     LOOP1           IF NOT, RESET STRING PTR.
            DJNZ    B,LOOP2         ELSE TEST NEXT CHARACTER
MATCH       EQU     $               MATCH FOUND
*
NOFIND      EQU     $               NO MATCH FOUND
```

The indexed addressing mode is used in the above example and has the capability to search a 256 byte string if necessary. The B Register alternates between a pointer into the six-byte test string and a pointer in to the longer table string.

Register indirect addressing is normally used when the 256-byte indexing range is not adequate. For example, a 1000-byte table move is performed in the following example:

```
            MOVD    %1000,R6
            MOVD    %SRC+999,R4
            MOVD    %DEST+999,R6
LOOP        LDA     *R4
            DECD    R4
            STA     *R6
            DECD    R6
            DJNZ    RZ,LOOP
            DJNZ    B,LOOP
```

4.4 LOGICAL INSTRUCTIONS

The TMS7000 instruction set provides powerful and flexible register and I/O logical bit manipulation and test support.

4.4.1 Register Logical Instructions (INV/XOR/OR/AND)

The invert (INV), exclusive OR (XOR), AND, and OR instructions are used for register logical or Boolean bit manipulation. The bit test and jump instructions (BTJO,BTJZ) provide efficient and flexible single or multi-bit testing.

Any register can be complemented via the INV instruction. Each bit is replaced with its boolean NOT thus resulting in a one's complement of the register. Individual bits may be complemented with the XOR instruction as shown below:

```
          XOR     %>81,R24
```

In this example, the most significant and least significant bits of R24 are complemented since the immediate operand has "1" bits in these bit positions. The remaining bits of R24 are unaffected since the immediate operand has "0" bits in the corresponding bit positions. The XOR bit mask source can be specified to be a register operand as well as an immediate operand. In the following example, the contents of R24 are selectively complemented after a table lookup to find the desired bit made:

```
         MOVP    P6,B
         LDA     @TABLE(B)
         XOR     R0,R24
```

Note the use of R0 to specify the A register in the final instruction.

The AND and OR instructions can be used to clear or to set selected bits in a register. The instruction sequence:

```
        BIT0     EQU     >01
        BIT6     EQU     >40
                 AND     %BIT0,A
                 OR      %BIT0+BIT6,B
```

clears the least significant bit (Bit 0) of the register A and sets bit 6 and bit 0 of the B register.

The BTJO and BTJZ instructions provide register single or multiple bit test capability. The Bit Test and Jump if One (BTJO) instruction tests for "1" bits in the destination operand. A relative jump is made if at least one such match is found. The BTJO instruction can be thought of as a logical AND instruction in which the relative jump is taken if the result is non-zero. The result, however, is not stored thus leaving the destination operand unchanged. The instruction:

```
          BTJO     %>FF,A,NZERO
```

causes a relative jump to location NZERO if the A register contains at least one "1" bit. The instruction sequence:

```
          BIT4     EQU      >10
                   BTJO     %BIT4,R19,>FD19
```

results in a relative jump to location >FD19 if bit 4 of R19 is a "1" independent of other R19 bit values.

The Bit Test and Jump if zero (BTJZ) instruction is similar to BTJO except that it tests for "0" bits in the destination operand. BTJZ test for "0" bits in the destination corresponding to "1" bits in the source operand and jumps if at least one match occurs. For example, the instruction:

```
          BTJZ     %>0F,B,ZEROS
```

results in a conditional jump to location ZEROS if any bit in the least significant nibble (bits 0-3) of the B register is a "0".

4.4.2 I/O Logical Instructions (XORP/ANDP/ORP/BJOP/BTJZP)

The exclusive OR (XORP), ANDP, ORP, BTJOP, and BTJZP provide I/O Boolean logical support. The Boolean bit mask can be specified to be either an immediate operand or else derived from the A or B registers. The destination operand is any one of the 256 peripheral file I/O ports.

The XORP, ANDP, and ORP allow any bit in an I/O port to be toggled, cleared, or set by a single instruction. For example, the following instruction sequence toggles bit 7 of PORTB, clears bit 0 of PORTC, sets bits 1 and 6 of PORTC, and then restores bit 7 of PORTB to its original value:

```
          BIT0     EQU      >01
          BIT1     EQU      >02
          BIT6     EQU      >40
          BIT7     EQU      >80
          CPORT    EQU      P8
          BPORT    EQU      P6
                   XOR      %BIT7,BPORT
                   AND      %BIT0,CPORT
                   OR       %BIT1+BIT6,CPORT
                   XOR      %BIT7,BPORT
```

The BTJOP and BTJZP instructions are similar to the BTJO and BTJZ instructions. They provide flexible and efficient I/O bit testing. For example, the instruction:

```
          BTJZP    %>0F,P18,$
```

loops on itself as long as any bit in the lower nibble (bits 3-0) of I/O port P18 is a "0". The Bit Test Jump instructions can also be used to test a single I/O bit as shown below:

```
BIT5    EQU   >20
        BTJOP %BIT5,P0,>FEA7
```

Execution of the above code causes a conditional jump to location >FEA7 if bit 5 of I/O port P0 is a "1".

The conditional jump instructions can also be used to test for I/O values. For example, the following instruction sequence:

```
AND     A,P200
JZ      LABEL
```

logically ANDs the value of the A register to I/O port P200 and then jumps to location LABEL if the resulting P200 value is zero. The Jump if Negative (JN) and Jump if Positive or Zero (JPZ) instructions can also be used if the most significant bit (i.e., sign bit as bit 7) is to be tested. For example, the instructions:

```
OR      B,P97
JPZ     LABEL1
```

Or the contents of the B register to I/O port P97 and jump if bit 7 of the result is a "1" bit.

4.5 BRANCH INSTRUCTION (BR)

The Branch instruction (BR) is used to unconditionally transfer program control to any desired location in the 64K-byte memory space. The BR instruction supports direct, indexed, and indirect addressing. Direct addressing is used for simple "GOTO" programming. Indexed addressing allows table jumps. In the example below, indexed addressing is used to access a relative jump table:

```
           MOVP    P4,B
           RL      B
           BR      @CTABLE (B)
           .
           .
           .
CTABLE     JMP     CASE0                  IF P4=0
           JMP     CASE1                  IF P4=1
           JMP     CASE2                  IF P4=2
           .
           .
           .
```

This indexed branch technique is similar to the Pascal "CASE" statement. Program control is transferred to location CASE0 if the input is '0', to CASE1 if it is a '1', etc. Up to 128 cases can be implemented. The case table entries can, of course, be longer entries simply by adjusting the B register index to a different alignment value.

The branch instruction can also be used with indirect addressing in order to branch to a computed address. For example, suppose that a computed branch address has been constructed in R19 and R20. The desired program control transfer is made by:

```
        BR      *R20
```

4.6 SUBROUTINE INSTRUCTIONS (CALL/TRAP/RETS)

TMS7000 Instruction Set provides several simple and flexible means of invoking and transferring control between subroutines, and for implementing complex algorithms.

The TMS7000 has two means of invoking subroutines: CALL and TRAP. Both instructions save the current value of the program counter on the stack before transferring control to the subroutine. Since the return address is stored on the stack, subroutines can be easily nested. The two instructions differ only in the way in which the subroutine address is determined.

The CALL instruction uses direct, indirect, or indexed addressing to specify the subroutine address. This permits both simple calls with a fully specified address or more complex calls with a calculated address. Thus, a table driven subroutine call similar to the branch "CASE" statement can be implemented with indexed addressing.

The TRAP instruction is the most efficient way to invoke a subroutine. There can be up to twenty-four TRAP instructions. The precise number supported is specified in the appropriate TMS7000 family member data manual. For example, the TMS7000 and TMS7020 both support twenty-four different TRAPs.

An individual subroutine address is associated with each of the TRAPs. These addresses are contained in a TRAP vector table which is in the upper-most block of memory. Each vector table entry contains the 16-bit starting address of the corresponding subroutine as shown below.

```
        TRAP      4
        .
        .
        .
        ORG       >FFF8          TRAP 4 VECTOR TABLE ENTRY
        DATA      BITTEST        TRAP 4 SUBROUTINE ADDRESS
        .
        .
        .
```

The trap subroutine address may be placed into the table as shown below:

```
          ORG       >FFFF-2N-1
          DATA      TRAPn             TRAP n subroutine address
```

Thus, for example, the subroutine starting at location BITTEST can be called either by a CALL instruction:

```
          CALL      @BITTEST
```

or by a TRAP instruction.

In each instance, a CALL requires three bytes: the opcode and two subroutine address bytes. If the subroutine is required at six locations, eighteen program bytes are used in total to implement the CALLs. The first use of a TRAP instruction also requires three bytes: the opcode and the two bytes in the vector table. However, only the opcode byte is required for successive use of the same TRAP. Thus, six uses of a TRAP require eight bytes (ten less than required by the equivalent CALLs).

The Return from Subroutine (RETS) instruction restores program control to the instruction immediately following the CALL or TRAP instruction. The return address is "POPped" off the stack and placed into the program counter. The stack is restored to its original state. If desired, the subroutine return can be aborted as demonstrated in the following code:

```
                JC        ERROR          DETECTED ERROR%
                RETS                     IF NOT, NORMAL RETURN
        ERROR   POP       ST             ELSE POP OFF RETURN
                POP       ST             ADDRESS
                .
                .
                .
```

In this example, the return address is removed from the stack since it is no longer desired.

The value of a file register and the status register can be pushed on or POPped from the stack. This is often done to pass data between routines or to temporarily store data during loops. For example, the following instruction sequence restores the value of the A register to its value before a table lookup instruction occurs:

```
            PUSH     A
            MOVP     P19,B
            LDA      @TABLE(B)
            MOVP     A,P20
            POP      A
```

4.7 THE STACK

The stack is located in RAM and can be tailored to the specific needs of the user. One powerful application of the stack is the establishment of tables. For example, the following program illustrates a dispatch table with an Interpretive Program Counter (IPC). An IPC is used in some high-level languages, such as Pascal, to give the proper execution sequence. The IPC can be contained in any register, and it points to an interpretive pseudo code (PCODE) byte that in turn specifies one of 256 dispatch routines. The overall effect of this function is that a program can execute one of a large number of different routines, depending on the single value stored in a register. Two separate 256-byte sections are required for the high and low address bytes of each dispatch routine. The first entry of each section (ROV0) corresponds to PCODE=0, and the second entry (ROV1) corresponds to PCODE=1, etc.

```
*
IPC       EQU       R3                INTERPRETIVE PROGRAM COUNTER
          LDA       *IPC              GET INPUT CODE. RANGE = 0-255
          DECD      IPC               POINT TO NEXT INPUT CODE
          MOV       A,B               PCODE  INDEX REGISTER
          LDA       @DTABLE(B)        LOOKUP ADDRESS MSB
          PUSH      A                 PUT MSB ON STACK
          LDA       @DTABLE+256(B)    LOOKUP ADDRESS LSB
          PUSH      A                 PUT LSB ON STACK
          RETS                        JUMP TO ADDRESS ON THE STACK
          .
*
*
DTABLE    BYTE      ROV0/256          BEGINNING OF MSB TABLE
          BYTE      ROV1/256
          .
          .
          BYTE      ROV255/256
*                                    BEGINNING OF LSB TABLE. WARNING
          BYTE      ROV0             MESSAGES MAY APPEAR HERE, BUT
          BYTE      ROV1             THEY DO NOT AFFECT RESULTS.
          .
          .
          .
          BYTE      ROV255
```

It should be noted that the assembler expressions have 16-bit values.

For those instructions requiring an 8-bit operand, the expression is truncated to the least significant eight bits. A warning message may result from this truncation, but the value will be correct. Thus, the following instructions place byte values >AA, >55, >55 at memory locations >8000, >8001, and >8002 respectively:

```
        AA55    LABEL    EQU     >AA55
8000                     AORG    8000
8000    AA55             DATA    LABEL
8002    AA               BYTE    LABEL       LSB only
```

The most significant byte (MSB) of an expression can be obtained by dividing the value by 256 as shown below:

```
        AA55    LABEL    EQU     >AA55
8000    AA55             AORG    8000
8000    AA               DATA    LABEL
                         BYTE    LABEL/256   MSB only
```

In this example, byte values >AA, >55, >AA are placed at memory locations 8000, 8001, and 8002.

4.8 INTERRUPTS

The number of interrupts for an TMS7000 family device is specified by the appropriate device data manual. The TMS7020, for example, has three interrupts in addition to RESET.

RESET and the interrupts are vectored through predetermined memory locations. RESET uses the "TRAP 0" vector which is stored at memory locations >FFFE->FFFF. The interrupts also use the TRAP vector table with INT1 using the "TRAP 1" vector, etc. Thus, the "TRAP 2" instruction involves the same code as the interrupt INT2 instruction.

The interrupts differ from the TRAPs, however, in that they also push the Status Register value on the stack, clear the interrupt enable bit in the Status Register, and reset the corresponding interrupt flag bit. Thus, the EINT instruction must be used if nested interrupts are desired. The return from interrupt (RETI) instruction restores the Status Register and the Program Counter, re-enabling interrupts.

Many interrupt service routines alter the status of key registers such as the A and B registers. These routines should use the stack to restore the machine state to the desired value. For example, the following interrupt routine performs an I/O driven table lookup. The A and B registers are used, but their values are saved and restored:

```
*
INT     PUSH    A             STORE A AND B REGISTERS ON STK
        PUSH    B
        MOVP    P4,B          GET INPUT FROM A PORT
        LDA     @LOOKUP(B)    DO TABLE LOOKUP TO GET NEW VAL
        MOVP    A,P6          OUTPUT NEW VALUE ON B PORT
        POP     B             RESTORE A AND B REGISTER IN
        POP     A               THE REVERSE ORDER THAT THEY
                                WERE PUT ON THE STACK
        RETI                  RETURN TO MAIN PROGRAM
*
```

Normally, all interrupts are disabled during an interrupt service routine. If an interrupt needs to be able to occur while the processor is servicing another interrupt, then the interrupt enable bit in the Status Register should be set to a 1. The number of interrupts that can be serviced at any one time is determined by the size of the stack, which is always a maximum of 128 bytes because the stack resides in the register file. Since other registers and data will most likely share the same space, the stack size is usually much less.

When doing nested interrupts, great care must be taken to avoid corrupting the data in the registers used by the most recent routine. If INT1 interrupts an ongoing INT1 service routine, then the registers used by the INT1 routine are used in two different contexts. If provisions are not made for these types of situations, such as disabling all interrupts at critical times, then data errors will result.

Sometimes a program will have distinct parts which require different responses to the same interrupt call. Since the interrupt vector is always set in nonchangeable ROM, another method must be used to change the vector for each part. One way of accomplishing this is to store a second vector in a RAM register pair and then let the first instruction in the interrupt routine execute an indirect branch on that register. The example below shows how this is done:

```
*      PROGRAM TO DEMONSTRATE MULTIPLE INTERRUPT SERVICE ROUTINE
*      LOCATIONS
*             (Main Program)
           MOVD   %SERVIC,R127        PUT INT1 SERVICE ROUTINE IN
           EINT                         REGISTER, TURN ON AND WAIT
           IDLE                         FOR INTERRUPTS.
           MOVD   %SERVI2,R127        CHANGE INT1 ROUTINE TO SERV12
           ...
*
              (First INT1 Service Routine)
*
SERVIC     PUSH  A                    BEGIN INT1 SERVICE ROUTINE
           PUSH  B                      FOR THIS PART OF PROGRAM.
           ...
*
*             (Second INT1 Service Routine)
SERVI2     PUSH  A                    BEGIN ANOTHER INT1 SERVICE
           DEC   R4                     ROUTINE.
           ...
*
INT1       BR    *R127                THE ENTIRE INT1 SERVICE
*                                       ROUTINE TRANSFERS CONTROL
*                                       TO THE ADDRESS IN R126 AND
*                                       R127.
*             (Interrupt Vector Table At End Of Memory)
           AORG   >FFFC
           DATA   INT1                ADDR OF INT1 SERVICE ROUTINE
           DATA   >F806               RESET VECTOR START OF PROGRAM
```

The next routine is an example of a bubble-type sorting program. The routine demonstrates the utility of Indexed Mode addressing. Table up to 256 bytes in length can be sorted using the routine. Longer tables can be sorted using the Indirect Addressing Mode.

```
*      150-BYTE BUBBLE SORT
*
FLAG                                 SWAP HAD BEEN MADE FLAG
*
SORT     CLR   FLAG                  RESET SWAP FLAG
         MOV   %149,B                NUMBER OF BYTES TO BE SORTED
LOOP1    LDA   @TABLE(B)             LOOK AT ENTRY IN TABLE
         CMPA  @TABLE-1(B)           LOOK AT NEXT LOWER BYTE
         JL    LOOP2                 IF LOWER, SKIP TO NEXT VALUE
         INC   FLAG                  ENTRY IS NOT LOWER; SET SWAP FLAG
         PUSH  A                     STORE UPPER BYTE
         LDA   @TABLE-1(B)           TAKE LOWER BYTE
         STA   @TABLE(B)             PUT WHERE UPPER BYTE WAS
         POP   A                     GET OLD UPPER BYTE
         STA   @TABLE-1(B)           PUT WHERE LOWER BYTE WAS
LOOP2    DJNZ  B,LOOP1               LOOP TIL ALL TABLE IS EXAMINED
         BTJO  %>FF,FLAG,SORT        IF SWAP MADE, THEN RESWEEP TABLE;
*                                    IF NO SWAP MADE, TABLE DONE.
```

SECTION 5

ASSEMBLER DIRECTIVES

5.1 GENERAL

The TMS7000 assembly language is processed by the Macro Assembler, executing in a host computer. This section describes the assembler and its directives.

5.2 THE MACRO ASSEMBLER

The Macro Assembler generates object code for the TMS7000 microcomputer. The Assembler processes source code twice. On the first pass, the assembler maintains the Location Counter, builds a symbol table, and produces a copy of the source code for processing during the second pass. On the second pass, the assembler reads the copy of the source and assembles the object code using the operation codes and the symbol table produced during the first pass.

5.3 ASSEMBLER DIRECTIVES

Assembler directives and machine instructions in source programs supply data to be included in the program and control the assembly process. The assembler supports a number of directives in the following categories:

- Directives that affect the location counter
- Directives that affect the assembler output
- Directives that initialize constants
- Directives that provide linkage between programs
- Miscellaneous directives.

5.3.1 Directives That Affect The Location Counter

As an assembler reads the source statements of a program, a component of the assembler called the Location Counter advances to correspond to the memory locations assigned to the resulting object code. The first nine of the assembler directives listed below initialize the Location Counter and define the value as relocatable, absolute, or dummy. The last three directives advance the Location Counter to provide a block or an area of memory for the object code to follow. The word boundary directive also ensures a word boundary (even address). The directives are listed in Table 5-1. The following paragraphs provide a detailed discussion of each.

TABLE 5-1 - ASSEMBLER DIRECTIVES THAT AFFECT THE LOCATION COUNTER

DIRECTIVES	MNEMONICS
* Absolute origin	AORG
* Relocatable origin	RORG
* Dummy origin	DORG
* Data segment	DSEG
* Data segment end	DEND
* Common segment	CSEG
* Common segment end	CEND
* Program segment	PSEG
* Program segment end	PEND
* Block starting with symbol	BSS
* Block ending with symbol	BES

5.3.1.1. Absolute Origin Directive (AORG): AORG places a value in the location counter and defines the succeeding locations as absolute. Use of the label field is optional. When a label is used, it is assigned the value that the directive places in the location counter. The command field contains AORG. The operand field is optional, but when used, contains a well-defined expression (wd-exp). The assembler places the value of the well-defined expression in the location counter. The comment field is optional and may be used only when the operand field is also used. When no AORG directive is entered, no absolute addresses are included in the object program. When the operand field is not used, the length of all preceding absolute code replaces the value of the location counter.

SYNTAX:

```
        [<label>]...AORG...[<wd-exp>...[<comment>]]
```

EXAMPLE:

```
        AORG >1000+X
```

Symbol X must be absolute and must have been previously defined. If X has a value of 6, the location counter is set to >1006" by this directive. Had a label been included, the label would have been assigned the value >1006.

5.3.1.2 Relocatable Origin Directive (RORG):
RORG places a value in the location counter; if encountered in absolute code, it also defines succeeding locations as program-relocatable When a label is used, it is assigned the value that the directive places into the location counter. The command field contains RORG. The operand field is optional; when it is used, the operand must be an absolute or relocatable expression (exp) that contains only previously defined symbols. The comment field may be used only when the operand field is used.

SYNTAX:

```
        [<label>] ...RORG ...[<exp] ...[<comment>]
```

When the operand field is not used, the length of the program segment, data segment, or specific common segment of a program replaces the value of the location counter. For a given relocation type X (data-, common-, or program-relocatable), the length of the X-relocatable segment at any time during an assembly is either of the following values:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of X-relocatable code

- Zero, if no X-relocatable code has been, previously assembled

Since the location counter begins at zero, the length of a segment and the next available address within that segment are identical.

If the RORG directive appears in absolute or program-relocatable code and the operand field is not used, the location counter value is replaced by the current maximum length of the program segment of that program. If the directive appears in data-relocatable code without an operand, the location counter value is replaced by the maximum length of the data segment. Likewise, in common-relocatable code, the RORG directive without an operand causes the maximum length of the appropriate common segment to be loaded into the location counter.

When the operand field is used, the operand must be an absolute or relocatable expression (exp) that contains only previously defined symbols. If the directive is encountered in absolute code, a relocatable operand must be program-relocatable; in relocatable code, the relocation type of the operand must match that of the current location counter. When it appears in absolute code, the RORG directive changes the location counter to program-relocatable and replaces its value with the operand value. In relocatable code, the operand value replaces the current location counter value, and the relocation type of the location counter remains unchanged.

EXAMPLE:

```
        RORG $-20 OVERLAY TEN WORDS
```

The $ symbol refers to the location following the preceding relocatable location of the program. This has the effect of backing up the location counter by ten words. The instructions and directives following the RORG directive replace the ten previously assembled words of relocatable code, permitting correction of the program without removing source records. If a label had been included, the label would have been assigned the value placed in the location counter.

```
          SEG2 RORG
```

The location counter contents depend upon preceding source statements. Assume that after defining data for a program that occupied >44 bytes, an AORG directive initiated an absolute block of code. The absolute block is followed by the RORG directive from the preceding example. This places >0044 in the location counter and defines the location counter as relocatable. Symbol SEG2 is a relocatable value, >0044. The RORG directive from the above example would have no effect except at the end of an absolute block or a dummy block.

5.3.1.3 Dummy Origin Directive (DORG): DORG places a value in the location counter and defines the succeeding locations as a dummy block or section. When assembling a dummy section, the assembler does not generate object code but operates normally in all other respects. The result is that the symbols that describe the layout of the dummy section are available to the assembler during assembly of the remainder of the program. The label is assigned the value that the directive places in the location coûnter. The operation field contains DORG. The operand field contains an expression <exp> which may be either absolute or relocatable. Any symbol in the expression must have been previously defined.

SYNTAX:

```
          [<label>] ...DORG ...<exp> ...[<comment>]
```

When the operand field is absolute, the location counter is assigned the absolute value. When the operand is relocatable, the location counter is assigned the relocatable value and the same relocation type as the operand. When this occurs, space is reserved in the section that has that relocation type.

EXAMPLE:

```
          DORG 0
```

The effect of this directive is to cause the assembler to assign values relative to the start of the dummy section to the labels within the dummy section. The example directive is appropriate to define a data structure. The executable portion of the module (following a RORG directive) should use the labels of the dummy section as relative addresses. In this manner, the data is available to the procedure regardless of the memory area into which the data is loaded.

EXAMPLE:

```
        RORG  0
         .
         .
         .    (code as desired)
         .
        DORG  $
         .
         .
         .    (data segment)
         .
        END
         .
```

The example of the DORG directive is appropriate for the executable portion (procedure division) of a procedure that is common to more than one task. The code corresponding to the dummy section must be assembled in another program module. In this manner, separate data portions (dummy sections) are available to the procedure portion.

The DORG directive may also be used with data-relocatable or common-relocatable operands to specify dummy data or common segments. The following example illustrates this usage:

```
        CSEG 'COM1'

        DORG $     "$" HAS A COMMON-RELOCATABLE VALUE
         .
         .
         .
         .
      LAB1 DATA $

      MASK DATA >F000
         .
         .
         .
         .
        CEND
```

In the example, no object code is generated to initialize the common segment COM1, but space is reserved and all common-relocatable labels describing the structure of the common block (including LAB1 and MASK) are available for use throughout the program.

5.3.1.4 Block Starting With Symbol Directive (BSS): BSS advances the location counter by the value of the well-defined expression (wd-exp) in the operand field. Use of the label field is optional. When used, a label is assigned the value of the location of the first byte in the block. The operation field contains BSS. The operand field contains a well-defined expression that represents the number of bytes to be added to the location counter. The comment field is optional.

SYNTAX:

```
          [<label>] ...BSS ...<wd-exp> [<comment>]
```

EXAMPLE:

```
        BUFF1 BSS 80 CARD INPUT BUFFER
```

This directive reserves an 80-byte buffer at location BUFF1.

5.3.1.5 Block Ending With Symbol Directive (BES): BES advances the location counter by the value in the operand field. Use of the label field is optional. When used, a label is assigned the value of the location following the block. The operation field contains BES. The operand field contains a well-defined expression that represents the number of bytes to be added to the location counter. The comment field is optional.

SYNTAX:

```
          [<label>] ...BES ...<wd-exp> ...[<comment>]
```

The following example shows a BES directive:

```
          BUFF2 BES >10
```

The directive reserves a 16-byte buffer. Had the location counter contained >100 when the assembler processed this directive, BUFF2 would have been assigned the value >110.

5.3.1.6 Even Boundary Directive (EVEN): EVEN places the location counter on the next word boundary (even byte address). When the location counter is already on an even boundary, the location counter is not altered. Use of the label field is optional. When used, a label is assigned the value in the location counter after the directive is processed. The command field contains EVEN. The operand field is not used, and the comment field is optional. SYNTAX:

```
          [<label>] ...EVEN ...[<comment>]
```

EXAMPLE:

```
          WRF1 EVEN
```

The directive assures that the location counter contains an even boundary address and assigns the location counter address to label WRF1.

5.3.1.7 Data Segment Directive (DSEG): DSEG places a value in the location counter and defines succeeding locations as data-relocatable. Use of the label field is optional. When a label is used, it is assigned the data-relocatable value that the directive places in the location counter. The command field contains DSEG. The operand field is not used, and the comment field is optional.

SYNTAX:

```
[<label>] ...DSEG ...[<comment>]
```

Initially, the location counter is set to zero. A RORG directive may be used to adjust the location counter values.

The DSEG directive defines the beginning of a block of data-relocatable code. The block is normally terminated with a DEND directive. If several such blocks appear throughout the program, they comprise the data segment of the program. The entire data segment may be relocated independently of the program segment at link-edit time. This provides a convenient means of separating modifiable data from executable code.

In addition to the DEND directive, the PSEG, CSEG, AORG, and END also properly terminate the definition of a block of data-relocatable code. The PSEG directive, like DEND, indicates that succeeding locations are program-relocatable. The CSEG and AORG directives effectively terminate the data segment by beginning a common segment (CSEG) or an absolute segment (AORG). The END directive terminates the data segment as well as the program.

EXAMPLE:

```
RAM    DSEG          START OF DATA AREA
.
.
.
.
<Data-relocatable code>
.
.
.
.
ERAM   DEND

LRAM   EQU ERAM-RAM
```

The block of code between the DSEG and DEND directives is data-relocatable. RAM is the symbolic address of the first word of this block; ERAM is the data-relocatable byte address of the location following the code block. The value of the symbol LRAM is the length in bytes of the block.

5.3.1.8 Data Segment End Directive (DEND): DEND terminates the definition of a block of data-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. Use of the label field is optional. When used, a label is assigned the value of the location counter prior to modification. The command field contains DEND. The operand field is not used, and the comment field is optional. As a result of this directive, the location counter is set to one of these values:

- The maximum value attained by the location counter as a result of the assembly of any preceding block of program-relocatable code.
- Zero, if no program-relocatable code has been previously assembled.

If encountered in common-relocatable or program-relocatable code, DEND functions as a CEND or PEND, and a warning message is issued. Like CEND and PEND, it is invalid when used in absolute code.

SYNTAX:

```
[<data>] ....DEND ...[<comment>]
```

5.3.1.9 Common Segment Directive (CSEG): CSEG places a value in the location counter and defines succeeding locations as common-relocatable (i.e., relocatable with respect to a common segment). Use of the label field is optional. When used, a label is assigned the value placed by the directive in the location counter. The operation field contains CSEG, and the operand field is optional. The comment field may only be used when the operand field is used.

If the operand field is not used, the CSEG directive defines the beginning of (or continuation of) the blank common segment of the program. When used, the operand field contains a character string of up to six characters enclosed in quotes. (If the string length exceeds six characters, the assembler prints a truncation error message and retains the first six characters of the string.) If this string has not previously appeared as the operand of a CSEG directive, the assembler associates a new relocation section number with the operand, sets the location counter to zero, and defines succeeding locations as relocatable with respect to the new relocatable section. When the operand string has been previously used in a CSEG, the succeeding code represents a continuation of the particular common segment associated with the operand. The location counter is restored to the maximum value attained during the previous assembly of any portion of that particular common segment. The second operand, <exp>, specifies the memory alignment for the beginning of the Section.

SYNTAX:

```
[<label>] ...CSEG ...['<string>' .... [<comment>]]
```

The following directives will properly terminate the definition of a block of common-relocatable code: CEND, PSEG, DSEG, AORG, and END. The block is normally terminated with a CEND directive. The PSEG directive, like CEND, indicates that succeeding locations are program-relocatable. The DSEG and AORG directives effectively terminate the common segment by beginning a data segment or an absolute segment. The END directive terminates the common segment as well as the program.

The CSEG directive permits the construction and definition of independently relocatable segments of data that several programs may access or reference at execution time. The segments are the assembly language counterparts of FORTRAN blank COMMON and labeled COMMON, and in fact, permit assembly language programs to communicate with FORTRAN programs which use COMMON. Information placed in the object code by the assembler permits the link editor to relocate all common segments independently and make appropriate adjustments to all adresses that reference locations within common segments. Locations within a particular common segment may be referenced by several different programs if each program contains a CSEG directive with the same operand or no operand.

The following example illustrates the use of both the CSEG and the CEND directives:

```
COM1A   CSEG    'ONE'

   <Common-relocatable section, type 'ONE'>
        .
        .
        .
        .
        CEND

COM2A   CSEG    'TWO'
        .
        .
        .
   <Common-relocatable section, type 'TWO'>
        .
        .
        .

COM2B   CEND
COM1C   CSEG   'ONE'
        .
        .
        .
   <Common-relocatable section, type 'ONE'>
        .
        .
        .

COM1B   CEND

COM1L   DATA COM1B-COM1A   LENGTH OF SEGMENT 'ONE'
COM2L   DATA COM2B-COM2A   LENGTH OF SEGMENT 'TWO'
```

The three blocks of code between the CSEG and the CEND directives are common-relocatable. The first and third blocks are relocatable with respect to one common relocation type; the second is relocatable with respect to another. The first and third blocks comprise the common segment 'ONE'; the value of the symbol COM1L is the length in bytes of this segment. The symbol COM2A is the symbolic address of the first word of the first word of common segment 'TWO'; COM2B is the common-relocatable (type 'TWO') byte address of the location following the segment. (Note that the symbols COM2B and COM1C are of different relocation types and possibly different values.) The value of the symbol COM2L is the length in bytes of common segment 'TWO'.

5.3.1.10 Common Segment End Directive (CEND): CEND terminates the definition of a block of common-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable. Use of the label field is optional. When used, a label is assigned the value of the location counter prior to modification. The command field contains CEND. The operand field is not used, and the comment field is optional. As a result of this directive, the location counter is set to one of the following values:

- The maximum value the location counter has ever attained as a result of the assembly of any preceding block of program-relocatable code.

- Zero, if no program-relocatable code had been previously assembled.

SYNTAX:

```
[<label>] ...CEND ...[<comment>]
```

If encountered in data- or program-relocatable code, this directive functions as a DEND or PEND. As is the case for DEND and PEND, CEND is invalid when used in absolute code. See Subsection 5.3.1.9 for an example of the use of the CEND directive.

5.3.1.11 Program Segment Directive (PSEG): PSEG places a value in the location counter and defines succeeding locations as a program-relocatable. When used, a label is assigned the value that the directive places in the location counter. The command field contains PSEG. The operand field and the comment field is optional. The location counter is set to one of the following values:

- The maximum value the location counter had attained as a result of the assembly of any preceding block of program-relocatable code.

- Relocatable zero, if no program-relocatable code had been previously assembled.

SYNTAX:

```
[<label>] ...PSEG ...[<comment>]
```

The PSEG directive is provided as the program-segment counterpart to the DSEG and CSEG directives. Together, the three directives provide a consistent method of defining the various types of relocatable segments. The following sequences of directives are functionally identical:

```
SEQUENCE 1                     SEQUENCE 2
__________                     __________

DSEG                           DSEG
 .                              .
 .                              .
 .                              .
<Data-relocatable code>        <Data-relocatable code>
 .                              .
 .                              .
 .                              .
DEND                            .
CSEG                           CSEG
 .                              .
 .                              .
 .                              .
<Common-relocatable code>      <Common-relocatable code>
 .                              .
 .                              .
 .                              .
CEND                            .
PSEG                           PSEG
 .                              .
 .                              .
 .                              .
<Program-relocatable code>     <Program-relocatable code>
 .                              .
 .                              .
PEND                            .
 .                              .
END                            END
```

5.3.1.12 Program Segment End Directive (PEND): The PEND directive is provided as the program-segment counterpart to the PEND and CEND directives. Like those directives, it places a value in the location counter and defines succeeding locations as program-relocatable (however, since PEND properly appears only in program-relocatable code, the relocation type of succeeding locations remains unchanged). Use of the label field is optional. When used, a label is assigned the value of the location counter prior to modification. The command field contains PEND. The operand field is not used, and the comment field is optional. The value placed in the location counter by this directive is simply the maximum value attained by the location counter as a result of the assembly of all preceding program-relocatable code, this directive functions as a DEND or CEND. Like DEND and CEND, it is invalid when used in absolute code.

SYNTAX:

```
        [<label>] ...PEND ...[<comment>]
```

5.3.2 Directives That Affect Assembler Output

This category contains the directive supplying a program identifier in the object code and five directives affecting the source listing. Table 5-2 lists those Directives. The paragraphs following discuss the Directives in detail.

TABLE 5-2 - DIRECTIVES THAT AFFECT ASSEMBLER OUTPUT

DIRECTIVES	MNEMONICS
Output Options	OPTION
Program Identifier	IDT
Page Title	TITL
Restart Source Listing	LIST
Stop Source Listing	UNL
Eject Page	PAGE

5.3.2.1 Output Options Directive (OPTION): OPTION selects several options for the assembler listing output. The <option-list> operand is a list of keywords, separated by commas, where each keyword selects a listing feature. The available <option-list> features are:

- BUNLST:	Limit the listing of BYTE Directives to one line
- DUNLST:	Limit the listing of DATA Directives to one line
- FUNLST:	Turn off all unlist options
- NOLIST:	Inhibit all listing output. (This overrides the LIST Directive)
- SYMLST:	Produce a symbol listing in the object file
- TUNLST:	Limit the listing of TEXT Directives to one line
- XREF:	Produce a symbol cross-reference listing

SYNTAX:

```
... OPTION    <option-list>
```

5.3.2.2 Program Identifier Directive (IDT): IDT assigns a name to the object module produced. Use of the label field is optional. When used, a label assumes the current value of the location counter. The command field contains IDT. The operand field contains the module name <string>, a character string of up to eight characters within single quotes. When a character string of more than eight characters is entered, the assembler prints a truncation error message and retains the first eight characters as the program name.

SYNTAX:

```
     [<label>] ...IDT ...'<string>' ...[<comment>]
```

EXAMPLE:

```
          IDT    'CONVERT'
```

This example directive assigns the name CONVERT to the module being assembled. The module name is printed in the source listing as the operand of the IDT directive and appears in the page heading of the source listing. the module name is also placed in the object code and is used by the link editor for automatic entry-point resolution. A routine whose entry point is to be automatically resolved by the link editor must be declared as the 'string' on the IDT statement for that module. The entry point must also be REF'd in this case.

NOTE

Although the Assembler will accept lowercase letters and special characters within the quotes, ROM loaders, (for example) will not. Therefore, only uppercase letters and numerals are recommended.

5.3.2.3 Page Title Directive (TITL): TITL supplies a title to be printed in the heading of each page of the source listing. When a title is desired in the heading of the listing's first page, a TITL directive must be the first source statement submitted to the assembler. This directive is not printed in the source listing. Use of the label field is optional. When used, a label field assumes the current value of the location counter. The command field contains TITL. The operand field contains the title (string), a character string of up to 50 characters enclosed in single quotes. When more than 50 characters are entered, the assembler retains the first 50 characters as the title and prints a truncation error message. The comment field is optional; the assembler does not print the comment but does increment the line counter.

SYNTAX:

```
          [<label>] ...TITL ... '<string>'  ...[<comment>]
```

EXAMPLE:

```
          TITL  '**REPORT GENERATOR**'
```

This directive causes the title **REPORT GENERATOR** to be printed in the page headings of the source listing. When a TITL directive is the first source statement in a program, the title is printed on all pages until another TITL directive is processed. Otherwise, the title is printed on the next page after the directive is processed, and on subsequent pages until another TITL directive is processed.

5.3.2.4 Restart Source Listing Directive (LIST): LIST restores printing of the source listing. This directive is required only when a no source listing (UNL) directive is in effect and causes the assembler to resume listing. This directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When used, a label assumes the current value of the location counter. The command field contains LIST. The operand field is not used. Use of the comment field is optional but the assembler does not print the comment.

SYNTAX:

```
        [<label>] ...LIST ...[<comment>]
```

EXAMPLE:

```
        UNL
```

The UNL directive inhibits printing of the source listing, and can be used to reduce assembly time and the size of the source listing.

5.3.2.5 Stop Source Listing Directive (UNL): UNL halts the source listing output until the occurrence of a LIST Directive. It is not printed in the source listing, but the source line counter is incremented. This directive is frequently used in MACRO definitions to inhibit the listing of the macro expansion.

Use of the label field is optional, but when used, the label assumes the value of the Location Counter. The command field contains the symbol UNL. The operand field is not used. The comment field is optional, but the Assembler does not print the comment.

SYNTAX:

```
        [<label>] ...UNL ...[<comments>]
```

5.3.2.6 Eject Page Directive (PAGE): PAGE causes the Assembler to continue the source program listing on a new page. The PAGE directive is not printed in the source listing, but the line counter increments. Use of the label field is optional. When used, a label assumes the current value of the location counter. The command field contains PAGE. The operand field is not used. Use of the comment field is optional, but the assembler does not print the comment.

SYNTAX:

```
        [<page>] ...PAGE ...[<comment>]
```

EXAMPLE:

```
          PAGE
```

The directive causes the assembler to begin a new page of the source listing. The next source statement is the first statement listed on the new page. Use of the page directive to source listing into logical divisions improves program documentation.

5.3.3 Directives That Initialize Constants

This category consists of directives assigning values in successive bytes or words of the object code, a directive placing characters of text in the object code for display or print purposes, and a directive initializing a constant for use during the assembly process. Table 5-3 lists these Directives. The following paragraphs discuss each directive in detail.

TABLE 5-3 - DIRECTIVES THAT INITIALIZE CONSTANTS

DIRECTIVE	MNEMONIC
Initialize Byte	BYTE
Initialize Word	DATA
Initialize Text	TEXT
Define Assembly-Time Constant	EQU

5.3.3.1 Initialize Byte Directive (BYTE): BYTE places one or more values in one or more successive bytes of memory. Use of the label field is optional. When used, a label is assigned the location in which the assembler places the first byte. The command field contains BYTE. The operand field contains one or more expressions separated by commas. The expressions must contain no external references. The assembler evaluates each expression and places the value in a byte as an eight-bit two's complement number. When truncation is required, the assembler prints a truncation warning message and places the right-hand portion of the value in the byte. The comment field is optional.

SYNTAX:

```
        [<label>] ...BYTE ...<exp>[,<exp>]......[<comment>]
```

EXAMPLE:

```
          KONS BYTE >F+1,-1,'D'-'=',0,'AB'-'AA'
```

The directive initializes five bytes, starting with a byte at location

KONS. The contents of the resulting bytes are 00010000, 11111111, 00000111, 00000000, and 00000001.

5.3.3.2 Initialize Word Directive (DATA): DATA places one or more values in one or more successive two-byte words memory. Use of the label field is optional. When used, a label is assigned the location at which the assembler places the first word. The command field contains DATA. The operand field contains one or more expressions separated by commas. The assembler evaluates each expression and places the value in a word as a 16-bit two's complement number. Words are stored most significant byte first, i.e. at the lower address. The comment field is optional.

SYNTAX:

```
        [<label>] ...DATA ...<exp>[,<exp>]...b...[<comment>]
```

EXAMPLE:

```
        KONS1 DATA 3200,1+'AB',-'AF',>F4A0,'A'
```

The directive initializes five words, starting with a word at location KONS1. The contents of the resulting words are >0C80, >4143, >BEBA, >F4A0, and >0041.

5.3.3.3 Initialize Text Directive (TEXT): TEXT places one or more characters in successive bytes of memory. The assembler negates the last character of the string when the string is preceded by a minus (-) sign (unary minus). Use of the label field is optional. When used, a label is assigned the location at which the assembler places the first character. The command field contains TEXT. The operand field contains a character string of up to 52 characters enclosed in single quotes, which may be preceded by a unary minus sign. The comment field is optional.

SYNTAX:

```
        [<label>] ...TEXT ..[-]'<string>' ...[<comment>]
```

EXAMPLE:

```
          MSG1  TEXT  'EXAMPLE'  MESSAGE HEADING
```

The directive places the eight-bit ASCII representations of the characters in successive bytes. When the location counter is on an even address, the result is >4558, >414D, >504C, and >45XX. XX, the contents of the rightmost byte of the fourth word, are determined by the next source statement. The label MSG1 is assigned the value of te first byte address containing >45. Another example, showing the use of a unary minus, follows:

```
          MSG2  TEXT - 'NUMBER'
```

When the location counter is on an even address, the result is >4E55, >4D42, and >45AE. The label MSG2 is assigned the value of the byte address in which >4E is placed.

5.3.3.4 Define Assembly-Time Constant Directive (EQU):

EQU assigns a value to a symbol. The label field contains the symbol to be given a value. The command field contains EQU. The operand field contains an expression. Use of the comment field is optional.

SYNTAX:

```
<label> ...EQU ...<exp> ...[<comment>]
```

NOTE

<exp> may not contain a REF'd symbol and may not contain forward references.

EXAMPLE:

```
SUM      EQU      R5
```

The directive assigns an absolute value to the symbol SUM, making SUM available to use as a register address. A second example of an EQU directive follows:

```
TIME     EQU      HOURS
```

The above example directive assigns the value of the previously defined symbol HOURS to the symbol TIME. When HOURS appears in the label field of a machine instruction in a relocatable block of te program, the value is a relocatable value. The two symbols may be used interchangeably. Symbols in the operand field must be previously defined.

5.3.4 Directives That Provide Linkage Between Programs

This category contains two directives that enable program modules to be assembled separately and integrated into an executable program. One directive places one or more symbols defined in the module into the object code making them available for linking. The other directive places symbols used in the module but defined in another module into the object code, allowing them to be linked. Table 5-4 lists these directives. The following paragraphs discuss each in detail.

TABLE 5-4 - DIRECTIVES THAT PROVIDE LINKAGE BETWEEN PROGRAMS

DIRECTIVE	MNEMONIC
External Definition	DEF
External Reference	REF
Secondary External Reference	SREF
Force Load	LOAD

5.3.4.1 External Definition Directive (DEF): DEF makes one or more symbols available to other programs for reference. The use of the label field is optional. When used, a label is assigned the current value of the location counter. The command field contains DEF. The operand field contains one or more symbols, separated by commas, to be defined in the program being assembled. The commend field is optional.

SYNTAX:

```
[<label>] ...DEF ... ...<symbol>[,symbol>]......[<comment>]
```

EXAMPLE:

```
DEF  ENTER,ANS
```

The directive causes the assembler to include symbols ENTER and ANS in the object code; these symbols are available to other programs.

5.3.4.2 External Reference Directive (REF): REF provides access to one or more symbols defined in other programs. The use of the label field is optional. When used, a label is assigned the current value of the location counter. The command field contains REF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

SYNTAX:

```
[<label>] ...REF ...<symbol>[,<symbol>]......[<comment>]
```

EXAMPLE:

```
REF  ARG1,ARG2
```

The directive causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

If a symbol is listed in the REF statement, then a corresponding symbol must also be present in a DEF statement in another source module. If a one-to-one matching of symbols does not occur, then an error occurs at link edit time. The system will generate a summary list of all "unresolved references".

5.3.4.3 Secondary External Reference Directive (SREF): SREF provides access to one or more symbols defined in other programs. The use of the label field is optional. When a label is used, the current value of the location counter is assigned to the label. The command field contains SREF. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement. The comment field is optional.

SYNTAX:

```
        [<label>] ...SREF ...<symbol>[,<symbol>]......[<comment>]
```

EXAMPLE:

```
        SREF   ARG1,ARG2
```

The directive causes the link editor to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

Unlike REF, SREF does not require a symbol to have a corresponding symbol listed in a DEF statement of another source module. The SREFed symbol will be an unresolved reference, but no error message will be given.

5.3.4.4 Force Load Directive (LOAD): The load directive is like a REF, but the symbol does not need to be used in the module containing the LOAD. The symbol used in the LOAD must be defined in some other module. LOADs are used with SREFs. If one-to-one matching of LOAD and DEF symbols does not occur, then unresolved references will occur during link editing.

SYNTAX:

```
        [<label>] ...LOAD ...<symbol>[,<symbol>]......[<comment>]
```

EXAMPLE:

```
      Module A1              Module A2              Module A3
      +------------+         +-------------+        +-------------+
      !            !         !             !        !             !
      ! LOAD C, D  !         !   LOAD C    !        ! LOAD E, F   !
      !            !         !             !        !             !
      +------------+         +-------------+        +-------------+
            !                       !                      !
            !                       !                      !
            V                       V                      V
            ------------------------------------------------
                                    !
                                    !
                       Module B     V
                       +----------------------+
                       !                      !
                       ! SREF C, D, E, F      !
                       !                      !
                       ! DATA C               !
                       ! DATA D               !
                       ! DATA E               !
                       ! DATA F               !
                       +----------------------+
                                    !
                                    !
                                    V
            ----------------------------------------------------------
            !                   !                      !                 !
            !                   !                      !                 !
Module C    V       Module D    V       Module E       V    Module F     V
+-----------+       +-----------+       +------------+      +----------+
!           !       !           !       !            !      !          !
!   DEF C   !       !  DEF D    !       !  DEF E     !      !  DEF F   !
!           !       !           !       !            !      !          !
+-----------+       +-----------+       +------------+      +----------+
```

Module A1 uses a branch table in module B to obtain one module C, D, E, or F. Module A1 knows which of module C, D, E, and F it requires. Module B has an SREF for C, D, E, and F. Module C has a DEF for C. Module D has a DEF for D. Module E has a DEF for E. Module F has a DEF for F. Module A1 has a LOAD for the modules C and D it needs. Module A2 has a LOAD for the module C it needs. Module A3 has a LOAD for the modules E and F it needs.

The LOAD and SREF directives permit module B to be written to handle a highly involved case and still be linked together without unnecessary modules since A1 only has LOAD directives for the modules it needs.

When a link edit is performed, automatic symbol resolutions will pull in the modules appearing in the LOAD directives.

If the link control file included A1 and A2, modules C and D would be pulled in while modules E and F would not be pulled in. If the link control file included A3, modules E and F would be pulled in while modules C and D would not be pulled in. If the link control file included A2, module C would be pulled in while modules D, E, and F would not be pulled in.

5.3.5 Miscellaneous Directives

This category includes those assembler directives not applicable to the other categories. Table 5-5 lists the directives and the following paragraphs discuss them.

TABLE 5-5 - MISCELLANEOUS DIRECTIVES

DIRECTIVE	MNEMONIC
Program End	END
Copy Source File	COPY
Define MACRO Library	MLIB

5.3.5.1 Program End Directive (END): END terminates the assembly. The last source statement of a program is the END directive. Any source statements following the END directive are considered part of the next assembly. Use of the label field is optional. When used, a label is assigned the current value of the location counter. The command field contains END. Use of the operand field is optional. When used, the operand field contains a program-relocatable or absolute symbol that specifies the entry point of the program. When the operand field is not used, no entry point is placed in the object code. The comment field may be used only with an operand field.

SYNTAX:

```
        [<label>] ...END ...[<symbol> ...[<comment>]]
```

EXAMPLE:

```
         END   START
```

The directive causes the assembler to terminate the assembly of this program. The assembler also places the value of START in the object code as an entry point.

5.3.5.2 Copy Source File Directive (COPY): COPY changes the source input for the assembler. Use of the label field is optional. The command field contains COPY. The operand field contains a file name from which the source statements are to be read. The file name may be the following:

- An access name recognized by the operating system
- A synonym form of an access name

The comment field is optional.

SYNTAX:

```
        [<label>] ...COPY ...<file name> ...[<comment>]
```

EXAMPLE:

```
          COPY   SFILE
```

The directive in the example causes the assembler to take its source statements from a file SFILE. At the end-of-file for SFILE, the assembler resumes processing source statements from the file or device previous to the COPY directive. A COPY directive may be placed in a file being copied. Nested copying of files can be performed by placing a COPY directive in a file being copied. Such nesting is limited by the assembler to eight levels; additional restrictions may be placed by the host operating system.

5.3.5.3 Define MACRO Library Directive (MLIB): The MLIB directive is used to provide the Assembler with the name of a library containing macro definitions. The operand of this directive is a directory pathname constructed according to the conventions of the host operating system and enclosed in single quotes. (See IDT and TITL directives) This directive is defined only for hosts which support libraries on hard disks.

SYNTAX:

```
          [<label>] ...MLIB ...'<pathname>' ...[<comment>]
```

Use of the label field is optional. When used, a label assumes the current value of the Location Counter. The command field contains MLIB. The operand field contains the pathname, a character string to up to 48 characters enclosed in single quotes; longer strings will cause a truncation error message. The comment field is optional.

NOTE

Neither the Assembler nor its run-time support has access to the operating system's synonym table, and so cannot expand pathnames. The use of synonyms will prevent finding any macros in that library.

EXAMPLE:

```
        MLIB 'MYVOLUME.MACDIR.CMPXMACS.NEWMACS'     (9900)
        MLIB 'USER32.BIGPROJ.MYTASK.MACROS'         (9900)
        MLIB 'DRCO:[MOORE.ASM32]'                   (VAX)
```

The above example would cause the macro function, when the program finds a macro call SUBMAC (not previously defined), to search first for a file named USER32.BIGPROJ.MYTASK.MACROS.SUBMAC, and then if that file isn't found, to search for a file named MYVOLUME.MACDIR.CMPXMACS.NEWMACS.SUBMAC, in that order.

5.4 SYMBOLIC ADDRESSING TECHNIQUES

The assembler processes symbolic memory addresses allowing the user to address a register by its symbolic memory address.

The following example illustrates this type of coding:

```
      SUM        EQU     33              ASSIGN SUM FOR
                                         REGISTER 33
      QUAN       EQU     34              ASSIGN QUAN FOR
                                         REGISTER 34
                    .
                 ADD     SUM,QUAN        ADD R33 to R34
                  .
                  .
                  .
```

The two initial EQU directives assign meaningful labels to be used as register addresses in the subroutine.

SECTION 6

PROGRAM LINKING

6.1 GENERAL

The TMS7000 Assembler supplies both absolute and relocatable object code that may be linked as required to form executable programs from separately assembled modules. This section contains guidelines to assist the user in taking full advantage of these capabilities.

6.2 RELOCATION CAPABILITY

Relocatable code includes information that allows a bootstrap loader to place the code in any available area of memory. This relocation capability allows the most efficient use of available memory and is required for disk-resident programs executed under an operating system.

Absolute code must be loaded into a specified area of memory. Absolute code is appropriate for code that must be placed in dedicated areas of memory and may be used for memory-resident programs executing under operating systems.

Object code generated by an assembler consists of machine language instructions, addresses, and data comprising the assembled program. The code may include absolute segments, program-relocatable segments, data-relocatable segments, and numerous common-relocatable segments. In assembly language source programs, symbolic references to locations within a relocatable segment are called relocatable addresses. These addresses are represented in the object code as displacements from the beginning of a specified segment. A program-relocatable address, for example, is a displacement into the program segment. At load time, all program-relocatable addresses are adjusted by a value equal to the load address. Data-relocatable addresses are represented by a displacement into the data segment. There may be several types of common-relocatable addresses in the same program, since distinct common segments may be relocated independently of each other. A subsequent section of this manual describes the representation of these relocatable addresses in the object code.

The elements of source statements are expressions, constants, and symbols. The relocatability of an expression is a function of the relocatability of the symbols and constants that make up the expression. An expression is relocatable when the number of relocatable symbols or constants added to the expression is one greater than the number of relocatable symbols or constants subtracted from the expression. (All other valid expressions are absolute.) When the first symbol or constant is unsigned, it is considered to be added to the expression. When a unary minus follows a subtraction operator, the effective operation is addition. The unary negation operator may not be applied to a relocatable expression or subexpression (see Subsection 2.6.4). For example, when all symbols in the following expressions are relocatable, the expressions are relocatable:

```
LABEL + 1
LABEL+TABLE+-INC
-LABEL+TABLE+INC
```

Decimal, hexadecimal, and character constants are absolute. Assembly-time constants defined by absolute expressions are absolute, and assembly-time constants defined by relocatable expressions are relocatable.

Any symbol that appears in the label field of a source statement other than an EQU directive is absolute when the statement is in an absolute block of the program. Any symbol that appears in the label field of a source statement other than an EQU directive is relocatable when the statement is in a relocatable block of the program. The type of the label or an EQU directive is the type of an expression in an operand field.

To summarize, a location is either absolute or relocatable and may contain either absolute or relocatable values. The example program in Appendix G includes absolute locations with relocatable contents and relocatable locations with absolute contents.

6.3 LINKING PROGRAM MODULES

Since the assembler includes directives that generate the information required to link program modules, it is not necessary to assemble an entire program in the same assembly. A long program may be divided into separately assembled modules to avoid a long assembly or to reduce the symbol table size. Also, modules common to several programs may be combined as required. Program modules may be linked by the link editor to form a linked object module that may be stored on a library and/or loaded as required. The following paragraphs define the linking information that must be included in a program module.

6.3.1 External Reference Directives

Each symbol from another program module must be placed in the operand field of an REF or SREF directive in the program module that requires the symbol. The example below shows a program named 'MAIN' whichs REFs a routine named 'SUBR1'. SUBR1 is not defined in File A.

```
        (FILE A)

                  IDT     'MAIN'
                  REF     SUBR1
                  .
                  .
                  .
                  CALL    @SUBR1
                  .
                  .
                  .
                  END
```

6.3.2 External Definition Directive

Each symbol defined in a program module and required by one or more other program modules must be placed in the operand field of a DEF directive. The example below show a program named 'ROUTINES' when DEFs a routine named 'SUBR1'. The label 'SUBR1' must be defined in the program.

```
        (FILE B)

                  IDT     'ROUTINES'
                  DEF     SUBR1,SUBR2
                  .
                  .
          SUBR1   EQU     $
                  .
                  .
                  RETS
          SUBR2   EQU     $
                  .
                  .
                  RETS
                  END
```

When program 'MAIN' in FILE A is linked with program 'ROUTINES' in FILE B, the linkage is automatically resolved.

6.3.3 Program Identifier Directive

Program modules that are to be linked by the link editor should include an IDT directive. The module names in the character strings of the IDT directives should be unique. The <string> on the IDT directive is not automatically a DEF'd symbol.

6.3.4 Linking

The link editor builds a list of symbols from REF directives as it links the program modules. The link editor matches symbols from DEF directives to the symbols in the reference list. The link editor follows linking commands to determine the modules to be linked. If the module in which a routine is defined has the same name as the routine entry points, the link editor can automatically locate the required module in a designated library.

SECTION 7

ASSEMBLER OUTPUT

7.1 GENERAL

This section presents information concerning the various data output by the assembler. The assembler output discussed includes source listings, error messages, a cross reference listing, and object code.

7.2 SOURCE LISTING

The source listings show the source statements and the resulting object code.

Each page of the source listing has a title line at the top. Any title supplied by a TITL directive is printed on this line. A page number is printed to the right of the title. The printer inserts a blank line below the title line and prints a line for each source statement listed. The line for each source statement contains a source statement number, a location counter value, the object code assembled, and the source statement as entered. A source statement may result in more than one byte of object code. The assembler prints the location counter value and object code on a separate line for each additional byte. Each added line is printed following the source statement line.

EXAMPLE:

```
        0018      0156      42        MOV       R10,R5
                  0157      0A
                  0158      05
```

The source statement number, 0018 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered including those source records that are not listed (e.g., TITL, LIST, UNL, and PAGE directives are not listed; source records between a UNL directive and a LIST directive are not listed). The difference between two source record numbers printed immediately in line indicates source records entered and not listed.

The next field on a line of the listing contains the location counter value, a hexadecimal value. In the example, 0156 is the location counter value. Not all directives affect the location counter; the field is blank for those directives that do not affect it. Of the directives that the assembler lists, the IDT, REF, DEF, EQU, SREF, and END directives leave the location counter field blank.

The third field normally contains a single blank. However, the assembler places a dash in this field when warning errors are detected.

The fourth field contains the hexadecimal representation of the object code, 420A05 in the above example. All machine instructions and the BYTE, DATA, and TEXT directives use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

The fifth field contains the characters of the source statement as they were scanned by the assembler. Spacing in this field is determined by the spacing in the source statement. The four fields contained in source statements will be aligned in the listing only when they are aligned in the source statements or when tab characters are used.

7.3 ASSEMBLER ERROR MESSAGES

The assembler issues two types of error messages: normal completion messages and abnormal completion messages. Each of these types is described in the following paragraphs.

7.3.1 Normal Completion Error Messages

When the assembler completes an assembly, it indicates any errors it encounters in the assembly listing. The assembler indicates errors following the source line in which they occur. At the end of a module (IDT-END pair), the corresponding messages are printed.

Table 7-1 lists error, warning, and information messages.

TABLE 7-1 - ASSEMBLY LISTING ERRORS

MESSAGE	EXPLANATION/RESPONSE
	NONFATAL ERRORS
WARNING - 'CEND' ASSUMED	
WARNING - 'DEND' ASSUMED	
WARNING - 'PEND' ASSUMED	
WARNING - 'DSEG' ASSUMED	This is a warning that the following two statements have the same result: CSEG 'DATA' DSEG

(CONTINUED)

TABLE 7-1 - ASSEMBLY LISTING ERRORS (CONTINUED)

MESSAGE	EXPLANATION/RESPONSE
	NON-FATAL ERRORS, Continued
WARNING - SYMBOL TRUNCATED	The maximum length for a symbol is six characters.
WARNING - STRING TRUNCATED	Check the syntax for the directive in question to determine the maximum length for the string.
WARNING - TRAILING OPERAND(S)	
WARNING - BYTE VALUE TRUNCATED	A value that is to be used as a byte value was larger than can be loaded into a byte.
**LAST WARNING	
	FATAL ERRORS
ABSOLUTE VALUE REQUIRED	
DISPLACEMENT TOO BIG	An instruction with an operand with a fixed upper limit was encountered that overflowed this limit.
INVALID EXPRESSION	This may indicate invalid use of a relocatable symbol in arithmetic.
EXPRESSION OUT OF BOUNDS	There is a range limit for the value being used that was exceeded.
DUPLICATE DEFINITION	The symbol appears as an operand of a REF statement, as well as in the label field of the source, OR, the symbol appears more than once in the label field of the source.
INVALID RELOCATION TYPE	The type of variable isn't relocatable
INVALID OPCODE	The second field of the source record contained an entry that is not a defined instruction, directive, pseudo-op, DXOP, DFOP, or macro name.
	(CONTINUED)

TABLE 7-1 - ASSEMBLY LISTING ERRORS (CONTINUED)

MESSAGE	EXPLANATION/RESPONSE
FATAL ERRORS, Continued	
INVALID OPTION	The option given in the OPTION directive are invalid.
INVALID REGISTER VALUE	The given register value is too large or too small.
INVALID SYMBOL	The symbol being used has invalid characters in it.
VALUE TRUNCATED	The value used was too big for the field, so it has been truncated.
SYMBOL USED IN BOTH REF AND DEF	
COPY FILE OPEN ERROR	File does not exist or is already being used.
EXPRESSION SYNTAX ERROR	Unbalanced parentheses OR invalid operations on relocatable symbols.
INVALID ABSOLUTE CODE DIRECTV	The directive PEND, DEND and CEND have no meaning in absolute code.
LABEL REQUIRED	
BLANK MISSING	A blank is needed but one was not found.
COMMA MISSING	Expected a comma but did not find one. Usually means that more operands were expected.
COPY FILENAME MISSING	
INDIRECT (*) MISSING	The indirect addressing (*) was needed
SYMBOL REQUIRED	
OPERAND MISSING	There was no operand field
REGISTER REQUIRED	A register should be used rather than a label or an absolute number.
CLOSE (') MISSING	

(CONTINUED)

TABLE 7-1 - ASSEMBLY LISTING ERRORS (CONTINUED)

MESSAGE	EXPLANATION/RESPONSE
FATAL ERRORS, Continued	
STRING REQUIRED	TEXT directive used with no text following.
PASS1/PASS2 OPERAND CONFLICT	The symbols in the symbol table did not have the same value in PASS1 and PASS2. Usually a problem with the Assembler.
SYNTAX ERROR	
UNDEFINED SYMBOL	The symbol being used has not been REF'ed or it has been DEF'ed but not used.
DIVIDE BY ZERO	
ILLEGAL SHIFT COUNT	The shift count being asked for is not valid.
CANNOT INDEX BY REGISTER ZERO	
INFORMATION MESSAGES	
OPCODES REDEFINED	As a result of an MLIB directive, one or more assembler opcodes has been redifined by a MACRO within a MACRO directory. The user should take action if this is not intended.
MACROS REDEFINED	As a result of an MLIB directive, one or more currently defined macros has been redefined by a MACRO (of the same name) with a MACRO directory. The user should take action if this is not intended.

7.3.2 Abnormal Completion Error Messages

Most abnormal completion error messages are issued by the operating system under which the assembly runs (messages in this category include those concerned with file I/O errors). The user should refer to the applicable operating system reference manual for detailed information.

Many abnormal completion messages are caused by transient error conditions that do not persist. For this reason, the user should attempt to execute the assembler a second time. If the abnormal termination persists, the user may load a backup copy of the assembler. If the error still persists, the user may wish to contact a customer representative.

Table 7-2 lists the abnormal error messages.

TABLE 7-2 - ABNORMAL COMPLETION ERROR MESSAGES

UNEXPECTED END OF PARSE
ERROR MAPPING PARSE - ASSEMBLER BUG
INVALID OPERATION ENCOUNTERED
NO OP CODE
INVALID LISTING ERROR ENCOUNTERED
SYMBOL TABLE ERROR
INVALID LIB COMMAND ID
UNKNOWN ERROR PASSED, CODE = XXXX

7.4 CROSS-REFERENCE LISTING

The assembler prints an optional cross-reference listing following the source listing. The format of the listing is shown in Figure 7-1.

```
LABEL      VALUE     DEFN     REFERENCES
-----      -----     ----     ----------

ADDT       01A8      0325     0314

ADSR  D    01A0      0316     0342     0343     0348     0349

GT         0006      0997

OBTCHN R             0088

SQUIB U                       0127     0233
```

FIGURE 7-1 - CROSS-REFERENCE LISTING FORMAT

As shown in the illustration above, in the label column the assembler prints each symbol defined or referenced in the assembly. If a single character followins the symbol, it represents the attribute of the symbol. These symbol-attribute characters and their meanings are listed in Table 7-3. The second (Value) column contains a four-digit hexadecimal number, the value assigned to the symbol. The number of the statement that defines the symbol appears in the third (Definition) column. For undefined symbols, this column is left blank. The right-most (Reference) column lists the numbers of statements that reference the symbol. A blank in this column indicates the symbol is never used.

TABLE 7-3 - SYMBOL ATTRIBUTES

CHARACTER	MEANING
R	External reference (REF)
D	External definition (DEF)
U	Undefined
M	Macro name
S	Secondary reference (SREF)
L	Force load (LOAD)

7.5 OBJECT CODE

The assembler produces object code that may be linked to other code modules or programs, and loaded directly into the computer. Object code consists of records containing up to 71 ASCII characters. The user can correct record data via a keyboard device. Reassembly would then be unnecessary. Figure 7-2 presents an example of object code.

```
K0000SAMPROG 90040C0000A0020BC06DB000290042C0020A0024BC81BC002A7F219F
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC072B0281B3A00A00ECB02217F151F
A00EEB0900B06C1A00EAB1102A00F2B0543B11F8B2C20C0032BC101B0B44BE0447F18EF
A0100BDD66B0003B0282C00A2B11EDB03407F832F
200CE0010C        7FCABF
:
```

FIGURE 7-2 - SAMPLE OBJECT CODE

7.5.1 Object Code Format

Object code is formatted to contain records made up of fields sandwiched betwwen tag characters. Table 7-4 below lists field and tag character information.

A tag character occupies the first position on each line of object code and identifies the fields it precedes to the loader. The specific tag character used depends on the function of the fields with which it is associated. The paragraphs that follow detail the various tag characters and their associated fields.

Tag character K is placed at the beginning of each program and is followed by two fields. Field one contains the number of bytes of program relocatable code; field two contains the program identifier assigned to the program by an IDT directive. When no IDT directive is entered, the field is blank. The linker uses the program identifier to identify the program, and the number of bytes of program-relocatable code to determine the load bias for the next module or program.

The tag character M is used when data or common segments are defined in the program and is followed by three fields. Field one contains the length, in bytes, of data- or common-relocatable code, field two contains the data or common segment identifier, and field three contains a "common number." The identifier is a six-character field containing the name $DATA (padded on the right by one blank) for data segments and $BLANK for blank common segments. If a named common segment appears in the program, an M tag will appear in the object code with an identifier field corresponding to the operand in the defining CSEG directive(s). Field three of the M tag consists of a four-character hexadecimal number defining a unique common number to be used by other tags that reference or initialize data of that particular segment. For data segments, this common number is always zero. For common segments (including blank common), the common numbers are assigned in increasing order, beginning at one and ending with the number of different common segments. The maximum number of common segments that a program may contain is 127.

Tag characters 1 and 2 are used with entry addresses. The associated field is used by the linker to determine the entry point in which execution starts when linking is complete. Tag character 1 is used when the entry address is absolute; tag character 2 when the address is relocatable. The field lists the address in hexadecimal form.

Tag characters 3, 4, and X are used for external references. Tag character 3 is used when the last appearance of the externally referenced symbol is in program-relocatable code; tag character 4 when it is in absolute code; and the X tag when it is in data-or common relocatable code. Tag characters 3 and 4 are associated with two fields. Tag character X may identifiy one additional field. Field one contains the location of the last appearance of the symbol. Field two contains the symbol itself. Field three is only used to supply the common number for the X tag.

Tag character E is used for external references. An E tag is used when a nonzero quantity is to be added to a reference. Field 1 identifies the reference by occurrence in the object code (0, 1, 2, ...). In other words, the value in field one is an index into references identified by 3, 4, V, X, Y and Z tags in the object code. The list is maintained by order of occurrence (i.e., the first entry in the list is the symbol located in field two of the first 3, 4, V, X, Y, or Z tag). Field 2 contains the value to be added to the reference after the reference is resolved.

Tag character @ is used for external references of an 8-bit value. It serves the same purpose for 8-bit values that the E-tag serves for 16-bit values.

Tag characters 5, 6, and W are used for external definitions. Tag character 5 is used when the location is program-relocatable. Tag character 6 is used when the location is absolute. Tag character W is used when the location is data- or common-relocatable. The fields are used by the linker to provide the desired linking to the external definition. Field one contains the location of the last appearance of the symbol. Field two contains the symbol of the external definition. Field three of tag character W contains the common number.

Tag character 7 precedes the checksum, and is placed at the end of the set of fields in the record. The checksum is an error detection word and is formed as the record is being written. It is the two's complement of the sum of the eight-bit ASCII values of the characters of the record from the first tag of the record through the checksum tag, 7.

Tag characters 9, A, S, and P are used with load addresses required for data words that are to be placed at other than the next immediate memory addresses. Tag character 9 is used when the load address is absolute. Tag character A is used when the load address is program-relocatable. Tag character S is used when the load address is data-relocatable. Tag character P is used when the load address is common-relocatable. Field one contains the load address. Field two is only present for tag character P and contains the common number.

Tag characters *, B, C, T, and N are used with data words. Tag characters * and B are used when the data is absolute (i.e., an instruction word or a word that contains text characters or absolute constants). Tag * is used for absolute byte data (8 bits) and B is used for absolute word data (16 bits). Tag character C is used for a word that contains a program-relocatable address. Tag character T is used for a word that contains a data-relocatable address. Tag character N is used for a word that contains a common-relocatable address. Field one contains the data word. The linker places the data word in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word. Field two is only used with N and contains the common number.

Tag characters G, H, and J are used when the symbol table option is specified. Tag character G is used when the location or value of the symbol is program-relocatable, tag character H is used when the location or value of the symbol is absolute, and tag character J is used when the location or value of the symbol is data- or common-relocatable. Field one contains the location or value of the symbol. Field two contains the symbol to which the location is assigned. Field three is used with tag character J only and contains the common number.

Tag character U is generated by the LOAD directive. The symbol specified is treated as if it were the value specified in an INCLUDE command to the linker. Field one contains zeros. Field two contains the symbol for which the loader will search for a definition.

Tag characters V, Y, and Z are used for secondary external references. Tag character V is used when the last appearance of the externally referenced symbol is in program-relocatable code; tag character Y when it is in absolute code; and the Z tag when it is in data- or common-relocatable code. Tag characters V and Y are associated with two fields. Tag character Z may identifiy one additional field. Field one contains the location of the last appearance of the symbol. Field two contains the symbol itself. Field three is only used to supply the common number for the Z tag.

Tag character 8 is also associated with the checksum field but is used when the checksum field is to be ignored.

Tag character D is used to specify a load bias. Its lone associated field contains the absolute address that will be used by a loader to relocate object code. The Link Editor does not accept the D tag.

Tag character F is placed at the end of the record. It may be followed by blanks.

The end of each record is identified by the tag character 7 followed by the checksum field and the tag character F (this data is described above). The assembler fills the rest of the record with blanks and a sequence number and begins a new record with the appropriate tag character.

The last record of an object module has a colon(:) in the first character position of the record, followed by blanks or time and date identifying data.

Table 7-4 defines the object record format and tags.

TABLE 7-4 - OBJECT RECORD FORMAT AND TAGS

TAG	1ST FIELD	2ND FIELD	3RD FIELD
(MODULE DEFINITION)			
K	PSEG LENGTH	PROGRAM ID(8)	
M	DSEG LENGTH	$DATA	0000
M	BLANK COMMON LENGTH	$BLANK	COMMON #
M	CSEG LENGTH	COMMON NAME(6)	COMMON #
(ENTRY POINT DEFINITION)			
1	ABSOLUTE ADDRESS		
2	P-R ADDRESS		
(LOAD ADDRESS)			
9	ABSOLUTE ADDRESS		
A	P-R ADDRESS		
S	D-R ADDRESS		
P	C-R ADDRESS	COMMON OR CBSEG #	
(DATA)			
*	ABSOLUTE 8-BIT VALUE (2)		
B	ABSOLUTE 16-BIT VALUE		
C	P-R ADDRESS		
T	D-R ADDRESS		
N	C-R ADDRESS	COMMON OR CBSEG #	
(EXTERNAL DEFINITIONS)			
6	ABSOLUTE VALUE	SYMBOL(6)	
5	P-R ADDRESS	SYMBOL(6)	
W	D-R/C-R ADDRESS	SYMBOL(6)	COMMON #
(EXTERNAL REFERENCES)			
3	P-R ADDRESS OF CHAIN	SYMBOL(6)	
4	ABSOLUTE ADDRESS OF CHAIN	SYMBOL (6)	
X	D-R/C-R ADDRESS OF CHAIN	SYMBOL (6)	COMMON *
E	SYMBOL INDEX NUMBER	ABSOLUTE OFFSET	
@	SYMBOL INDEX NUMBER	OFFSET (2)	MASK (2)
(SYMBOL DEFINITIONS)			
G	P-R ADDRESS	SYMBOL(6)	
H	ABSOLUTE VALUE	SYMBOL(6)	
J	D-R/C-R ADDRESS	SYMBOL(6)	COMMON #
(FORCE EXTERNAL LINK)			
U	0000	SYMBOL(6)	
(SECONDARY EXTERNAL REFERENCE)			
V	P-R ADDRESS OF CHAIN ENTRY	SYMBOL(6)	
Y	ABSOLUTE ADDRESS OF CHAIN	SYMBOL(6)	
Z	D-R/C-R ADDRESS OF CHAIN	SYMBOL(6)	COMMON #

(CONTINUED)

TABLE 7-4 - OBJECT RECORD FORMAT AND TAGS (Continued)

TAG	1ST FIELD	2ND FIELD	3RD FIELD
(CHECK SUM)			
7	VALUE		
(IGNORE CHECK SUM)			
8	ANY VALUE		
(LOAD BIAS)			
D	ABSOLUTE ADDRESS		
(END OF RECORD)			
F			
(END OF OBJECT MODULE)			
:			

NOTES

1. ALL FIELD WIDTHS ARE FOUR CHARACTERS UNLESS OTHERWISE SPECIFIED BY NUMBERS IN PARENTHESES

2. IF THE FIRST TAG IS 01 (HEX), THE FILE IS IN COMPRESSED OBJECT FORMAT.

3. P-R PROGRAM SEGMENT RELATIVE (ADDRESS)
   D-R DATA SEGMENT RELATIVE (ADDRESS)
   C-R COMMON SEGMENT RELATIVE (ADDRESS)

7.5.2 External References In Object Code

External references are possible. The Link Editor will resolve all external references automatically.

7.5.3 Changing Object Code

In most cases, changing the object code is not the best way to correct errors in a program. All changes or corrections to a program should be made in the source code, then the program should be reassembled. Failure to follow this principle can make subsequent correction or maintenance of the program impossible. The information in the following paragraphs is intended for those rare instances when reassembly is not possible. Any changes made directly to the object code should be thoroughly documented so that the programmers who come later can see what the program actually does, not what the source code says that it does.

To correct the object code without reassembling a program, change the object code by changing or adding one or more records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character, D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of the D tag character is to specify that area of memory into which the loader loads the program. The tag character D and the associated field must be placed ahead of the object code generated by the assembler.

Correction of the object code may require only changing a character or a word in an object code record. The user may duplicate the record up to the character or word in error, replace the incorrect data with the correct data, and duplicate the remainder of the record up to the seven tag character. Because the changes the user has made will cause a checksum error when the checksum is verified as the record is loaded, the user must change the seven tag character to eight.

When more extensive changes are required, the user may write an additional object code record or records. Begin each record with a tag character 9, A, S, or P, followed by an absolute load address or a relocatable load address. This may be an address into which an existing object code record places a different value. The new value on the new record will override the other value when the new record follows the other record in the loading sequence. Follow the load address with a tag character *, B, C, T, or N and an absolute data word or a relocatable data word. Additional data words preceded by appropriate tag characters may follow. When additional data is to be placed at a nonsequential address, write another load address tag character followed by the load address and data words preceded by tag characters. When the record is full, or all changes have been written, write tag character F to end the record.

When additional memory locations are loaded as a result of changes, the user must change field one of tag character zero, which contains the number of bytes of relocatable code. For example, if the object field written by the assembler contained 1000 Hex bytes of relocatable code and the has added eight bytes in a new object record, additional memory locations will be loaded. The user must find the zero tag character in the object code file and change the value following the tag character from 1000 to 1008; he must also change the tag character 7 to 8 in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium, and the last record that affects a given memory location determines the contents of that location at execution time.

The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field must follow all other object records. An additional field or record may be added to include reference to a program identifier. The tag character is 4, and the hexadecimal field contains zeros. The second field contains the first six characters of the IDT character string. External definitions may be added using tag character 5 or 6 followed by the relocatable or absolute address, respectively. The second field contains the defined symbol, filled to the right with blanks when the symbol contains less than six characters.

NOTE

> Both object code that will be linked and object code that will be loaded by the bootstrap loader can be changed without reassembling the program. The Link Editor, though, will not accept tag character D in changed or added object records.

SECTION 8

MACRO ASSEMBLER LANGUAGE

8.1 GENERAL

The Macro Assembler supports a macro definition language which may be used to simplify programming. A macro definition is a set of source statements (machine instructions, macro language statements, and assembler directives) which generate other source statements within the source program.

When the Assembler processes a macro call, it substitutes the predefined source statements of the macro definition for the macro call source statement, and assembles the substituted statements as if they had been included in the source program. This section describes the macro language and the verbs used to define macros.

8.2 DEFINING MACROS

The creation of macro definitions is normally done by including within the assembler source file lines of code in a predefined format, which is detailed in the paragraphs that follow. In general, the definition requires a line marking the start of a macro definition, putting the macro name in the label field of the symbolic line, the string '$MACRO' in the op-code field, and possibly a list of formal parameters separated by commas in the op-code field.

Macros may be defined in-line with the normal assembler input, except that a macro definition must appear prior to an invocation of that macro. Good documentation practice is to place all macro definitions at the top of the assembler source file. Placing the macro definitions at the top of the assembler source file also allows easy reference to all the definitions because they are in one location.

In addition, macros may be defined in external files. These files are simply text files, like the assembler source file which contains macros defined in the same manner as those defined in-line. Only one macro may be defined in a file. The Assembler is informed of the existence of a macro library (i.e., a collection of macro files) by means of the 'MLIB' assembler directive. The syntax of the MLIB directive is:

```
MLIB 'VOLUME.DIRECTORY.MACLIB'
```

The string enclosed in the quotes represents a directory name in the format required by the conventions set by the host operating system.

The use of a macro library is as follows: Assume that a library of macro definitions is contained in a directory named 'VOLUME.DIRECTORY.MACLIB', and that a file named 'CPXADD' is a member of that directory. If the macro call

```
LABEL  CPXADD CX1,CX2
```

is found in the assembler source, the in-memory macro table is first searched for the definition of CPXADD. CPXADD will be in the macro table if CPXADD was previously defined in the assembler source file or was previously encountered and has already been read from a macro file. If the definition is not found in the macro table, a search of the normal assembler op-code/directive table is made. If found there, the op-code will be assembled as a normal machine instruction. If not, an attempt is made to find the file whose name is formed, by appending the macro name to the MLIB name. If more than one MLIB directive has been encountered, the most recently defined library is searched first, then all remaining libraries are searched. If the file is found, the macro definition is copied into the Assembler's macro file (in a compressed format), and an entry is made in the macro table for later use.

Because of the sequence of the search for matching definitions (library search following op-code table search), a macro defined in a library will not automatically redefine a machine instruction, although this is easily done using an in-line macro definition. To extend this capability to the macro library, the user should include in that library, a text file named 'MLIST', in which is contained (one per line, starting in column one) the names of the opcodes and currently defined macros the user wishes to have redefined by macros contained in a macro library.

A typical MLIST file might be constructed as follows, using the appropriate system text editor:

```
file named  <MLIB directory name>.MLIST
record 1    ADD              (opcode)
record 2    LACK             (opcode)
record 3    MOV              (opcode)
record 4    FSUB             (macro)
   eof(MLIST)
```

This file (MLIST) is read, if provided, when the MLIB directive is processed. If a name found there matches a currently defined opcode or a name in the macro table, the matching entry is removed from its table. This forces a search of the libraries, since the name will not be found elsewhere. When a name is found matching an opcode, the message:

```
' **** OPCODES REDEFINED'
```

is printed in the assembler listing following the printing of the MLIB statement. A similar message:

```
' **** MACROS REDEFINED'
```

will appear when currently defined macros are redefined. If this is the user's intent, then no action is required; if not, then some action is required, such as the deletion of some or all of the records in the file MLIST.

The name of a macro in file should be the same as the file name, otherwise, some inefficiency in macro usage will result. If the file named CPXADD contains a definition line such as

```
CPXMUL $MACRO  MR, MD
```

an entry for a macro named CPXMUL will be made in the internal macro table, and the next call to CPXADD will be recognized as undefined, and again, reentered as CPXMUL into the internal macro table.

NOTE

The MLIB directive and the macro library concept are supported only by host systems which allow libraries on hard disks.

8.2.1 Sample Macros

The following is a simple example of a macro definition:

```
INCX    $MACRO
        LDA   @X
        INC   A
        STA   @X
        $END
```

The above code defines a macro named INC. $MACRO identifies the beginning of the macro definition, and $END identifies the end of the macro definition. LDA @X, INC A, and STA @X are model statements which will be placed into the source program upon a macro call. (A model statement is a statement that "models" an assembler language statement. Such a statement is or will form, after macro substitution, a legal language statement.) The macro INC may now be used in the source program as often as necessary. The macro may be called by simply placing the line

```
INCX
```

within the source file. The Macro Assembler will replace this line with the remainder of the definition, i.e.:

```
     LDA   @X
     INC   A
     STA   @X
```

X must be a symbol representing a memory address in the source program assigned by the EQU directive. INCX is limited because the macro can only be used with a single memory location. The following macro, however, can be used with any memory location:

```
     INC   $MACRO M
           LDA   @:M.S:
           INC   A
           STA   @:M.S:
           $END
```

M is a macro parameter which is replaced by the actual parameter when the macro is called. M.S is the string component of this variable, i.e., the symbol representation of the variable. For example, the line:

```
     INC Y
```

will be replaced by:

```
     LDA   @Y
     INC   A
     STA   @Y
```

but

```
     INC Z
```

will be replaced by:

```
     LDA   @Z
     INC   A
     STA   @Z
```

Another component of a macro variable is the value component. An example of the use of this component is:

```
     ADDK   $MACRO X,NUM          (X and NUM are parameters. See
            LDA    @:X.S:         paragraph 8.3.3.1)
            ADD    %:NUM.V:,A
            STA    @:X.S:
            $END
```

NUM.V is the value component of the parameter NUM. The call:

```
     ADDK Y,3
```

will result in:

```
     LDA  @Y
     ADD  %3,A
     STA  @Y
```

These and other macro commands will be explained in the following in greater detail in the following paragraphs.

8.3 MACRO LANGUAGE ELEMENTS

The elements of the macro language are strings, constants, operators, variables, keywords, and verbs. A macro definition consists of model statements and statements containing macro language verbs. A model statement results in an assembly language source statement. The elements of the macro language and model statements are explained fully in the following paragraphs.

8.3.1 Strings

The literal strings of the macro language consists of one or more characters enclosed in single quotes. They are identical to the character string used in the assembly language.

An example of a string is: 'ONE'.

Another example is: '   ' (a blank).

8.3.2 Constants And Operators

Constants for macro language are defined in the same manner as for assembly language. The following are examples of constants:

```
      >9F3C
      $              (current PC value)
```

Arithmetic operators are also valid in the macro assembler. Functions of +, -, * (multiply), and / (divide) can be used to generate operand values. The following are examples of the use of arithmetic operators:

```
      LABEL  EQU  $+4                (current PC value + 4)
```

Relational operators are also available for use in the macro assembler. The relational operators compare the values of two variables or constants and return the answer of TRUE or FALSE. The relational operators are:

=	Equal
>	Greater than
<	Less than
#=	Not equal

The following are examples of the use of relational operators:

```
$IF   A.V>3            (Process succeeding  clock if value
                        component of variable A is >3.

$IF   B.L#=A.L         (Process succeeding  block if length
                       component of variable B is not equal
                       to length component of variable A.
```

Boolean operators are another feature offered by the macro assembler. They perform the desired operation and return either TRUE or FALSE. The boolean operators are:

&	AND
++	OR
--	NOT

The following is an example of the use of the boolean operators:

```
$IF --((A.V>3)&(B.L#=A.L)
```

The macro language permits concatenation of macro symbol components with literal strings, characters of model statements, and other macro variables. Concatenation is indicated by writing character strings side by side with string mode references.

8.3.3 Variables

A macro definition may include variables which are represented in the same manner as symbols in the assembler symbol table with the restriction that they may be a maximum of two characters in length (see Macro Symbol Table Section 8.3.3.2). The following are examples of variables:

```
VA          P4          SC          F2          A
```

NOTE

Macro variables are strictly local; they are available only to the macro which defines them. Access to symbols in the AST is through the symbol components.

8.3.3.1 Parameters: Parameters are a special class of macro variables. They are declared in the $MACRO statement at the beginning of the macro definition. The sequence of parameters in the operand field of the $MACRO statement corresponds to the sequence of operands in the operand field of the macro call. In the expansion of a macro call, the parameters have values which are associated with the corresponding operands in the macro call in a manner to be described in the following section. The following are examples of $MACRO statements with parameters:

```
LABEL  $MACRO  A,B3

NAME  $MACRO  O,RC,AM
```

8.3.3.2 Macro Symbol Table: The macro translator maintains a macro symbol table (MST) similar to the symbol table of the Assembler, the AST. Each entry consists of the string, value, length, and attributes of a variable or parameter. The macro expander module places parameters in the MST as it processes a macro call, and places variables in the MST as it processes the macro language statements that declare variables.

The string component of an MST entry contains a character string assigned to the macro variable/parameter by the macro expander.

The value component of an MST entry contains the binary equivalent of the string component, if the string component is an integer. The value component can also contain the value of the symbol, if the string component is a symbol in the AST. If a parameter is an operand list, the value is the length of the list. The length component contains the number of characters in the string component. The attribute component of the MST is a bit vector, the bits of which correspond to the attributes of the variable or parameter.

For example, the statement:

```
ADDK $MACRO X,NUM
```

identifies a macro, ADDK having parameters AU and AD.

A macro call to activate that macro definition could be coded as follows:

```
ADDK VAR1,3
```

The MST would now contain parameters X and NUM. The string component of parameter X would be the character string VAR1. The attribute component would indicate that the parameter is supplied in a macro call. The length component would be four. The string component of parameter NUM would be the character 3. The value component would be three,(expressed as a binary number) and the length component would be one. The attribute component would indicate that the parameter is supplied in the macro call.

Each component of a macro variable may be accessed individually.to the Reference to a variable component is made in either binary mode of string mode. In the binary mode, the referenced macro variable component is treated as a signed 16-bit integer. Binary mode access is made by wiring the variable name and component. A reference to the string component of a macro variable in binary mode is the 16-bit integer value of the ASCII representation of the first two characters of the string. For example, the binary mode value of the string component of X is >5641, which is the ASCII representation for VA.

String mode access of macro variable components is signified by enclosing the variable in a pair of colon characters (:). For example: :X: .

NOTE

Colons are always used in pairs to enclose a variable name. If a component qualifier is used, the pair of colons enclose the entire qualified name.

8.3.3.3 Variable Qualifiers: The components of a parameter or variable may be specified using the specific names as shown in Table 8-1. The variable name is followed by a period (.) and the single letter qualifier. The following examples show qualified variables:

X.S String component of variable VAR1. For example, in the macro call: ADDK, X.S equals the binary equivalent for VA, or >5641. If a string mode is indicated, as in :X.S:, the string component is the character string "VAR1".

X.A Attribute component of variable VAR1. This component may be accessed by use of logical operators and keywords which are described in tables 8-2, 8-3 and 8-4 which follow.

X.V Value component of variable VAR1.

X.L Length component of variable VAR1. For example, in the macro call: ADD VAR1,3 , :X.L is the character string 4.

TABLE 8-1 - VARIABLE QUALIFIERS

QUALIFIER	MEANING
S	The string component of the variable.
A	The attribute component of the variable.
V	The value component of the variable.
L	The length component of the variable.

Except in an $ASG statement , an unqualified variable means the string component of the variable. In the two following examples, the concatenated strings are equivalent:

:CT.S: WAY	Variable CT qualified: string component = WAY.
:CT: WAY	Variable CT unqualified: string component = WAY.

NOTE

In model statements, binary references to macro variables MUST be qualified.

All symbols in the AST have symbol components. (All components of macro parameters and the values of all AST symbols are directly accessible.) In order for other components to be accessed in a macro, the symbol must be assigned to the string component of a macro variable using $ASG. The additional qualifiers shown in Table 8-2 may be used with the macro variable to access the symbol components of the AST symbol.

The following are examples of qualified variables that specify symbol components of string components of variables. (Assume that V1.S has been defined as MASK, and the statement: MASK EQU >FF has been previously encountered in the assembly language source program.)

B1.SS	String component of the symbol MASK. This is null unless a macro instruction has caused a string to be associated with it by using a $ASG statement.
V1.SV	Value component of the symbol MASK, i.e., >FF. In the string mode, V1.SV equals the characters: 255.
V1.SA	Attribute component of the symbol MASK. This component may be accessed by use of logical operators and keywords, as described below.
V1.SL	Length component of the symbol MASK. If a string has been assigned to MASK, then V1.SL is the length of that string.

Concatenation is especially useful when a previously defined string is augmented with additional characters. The string ONE could be represented by a qualified variable such as CT.S. In that case, concatenation is expressed as follows:

```
:CT.S:' WAY'
```

would provide the same result as writing

```
        'ONE WAY'
```

If the qualified variable CT.S represents the characters: TWO , the result of the concatenation in the example would be TWO WAY. Strings and qualified variables may be concatenated as required and the variable need not be first. Components of variables that are represented by a binary value (e.g., CT.V and CT.L) are converted to their ASCII decimal equivalent before concatenation.

For example:

```
        :CT.S' WAY ':CT.L:
```

is expanded as

```
        ONE WAY 3
```

since the length component of the variable CT is three.

TABLE 8-2 - VARIABLE QUALIFIERS FOR SYMBOL COMPONENTS

QUALIFIER	MEANING
SS	String component of a symbol that is the string component of a variable.
SV	Value component of a symbol that is the string component of a variable.
SA	Attribute component of a symbolthat is the string component of a variable.
SL	Length component of a symbol that is the string component of a variable.

8.3.4 Keywords

The macro language recognizes certain keywords that specify the attributes of assembler symbols and macro parameters. Each keyword represents a bit position within a word that contains all attributes of the symbol or parameter. A keyword may be used with a logical operator and the attribute component to test or set a specific attribute of a symbol or parameter. The following paragraphs describe how keywords are used with symbols and parameters.

8.3.4.1 Symbol Attribute Component Keywords: The keywords listed in Table 8-3 may be used with a logical operator and the symbol attribute component: .SA to test or set the corresponding attribute component in the AST. The following example shows an expression that uses a symbol attribute component keyword:

V1.SA&$STR	This is the result of an AND operation between the attribute component of the symbol MASK and a flag corresponding to keyword $STR. The expression is TRUE when the contents of the string component of MASK is not null; otherwise, the expression is FALSE.

Another example shows an expression that uses a symbol attribute keyword:

V1.SA++$REL	This is the result of an OR operation between the attribute component of the symbol MASK and the flag corresponding to keyword $REL.

TABLE 8-3 - SYMBOL ATTRIBUTE KEYWORDS

KEYWORD	MEANING
$REL	Symbol is relocatable
$REF	Symbol is an operand of an REF directive
$DEF	Symbol is an operand of a DEF directive
$STR	Symbol has been assigned a component string.
$MAC	Symbol is defined as a macro name.
$UNDF	Symbol is not defined.

NOTE: The use of these attributes in conditional assembly (See $IF) can lead to pass conflict errors if the symbol has not been defined prior to the macro call.

8.3.4.2 Parameter Attribute Keywords: The keywords listed in Table 8-4 may be used with a logical operator and the macro symbol attribute component to test or set the corresponding attribute in the MST attribute component. These attribute keywords may be used to test or set attributes of all variables in the MST. The following examples show expressions that use parameter attribute component keywords:

P6.A&$PCALL	This is the result of an AND operation between the attribute component of variable P6 and the flag corresponding to keyword $PCALL. The expression is TRUE when variable P6 is a parameter supplied in a macro call. Otherwise the expression is FALSE.
RA.A++$PSYM	This is the result of an OR operation between the attribute component of variable RA and the flag corresponding to keyword $PSYM.

TABLE 8-4 - PARAMETER ATTRIBUTE KEYWORDS

KEYWORD	MEANING
$PCALL	Parameter appears as a macro-instruction operand.
$POPL	Parameter is an operand list. The value component contains the number of operands in the list.
$PSYM	Parameter is a symbolic memory address. (NOTE: a symbolic memory address is recognized when the variable is preceded by an @ character.

8.3.5 Verbs

The macro language supports 7 verbs that are used in macro language statements. Any statement in a macro definition that does not contain a macro language verb in the operation field is processed as a model statement.

8.3.5.1 $MACRO Statement: The $MACRO statement must be the first statement of a macro definition. It assigns a name to the macro and declares the parameters for the macro. The macro name consists of one to six alphanumeric characters, the first of which must be alphabetic. Each <parm> is a parameter for the definition. Parameters are described in paragraph 8.3.4.1. The operand field may contain as many parameters as the size of the field allows and must contain all parameters used in the macro definition. The comment field may not be used if there are no parameters.

SYNTAX:

```
    <macro name> ...$MACRO ...[<parm>][,<parm>]... ...[<comment>]
```

The macro definition is used in the expansion of macro calls that have the macro name as an operation code. The syntax for a call is as follows:

```
..<macro name> ..[<operand list>],]<operand/list>]... ...[<comment>]
```

The macro name specifies the macro definition to be used. Each operand may be any expression or address type recognized by the Assembler, or a character string enclosed in quotes. Alternatively, a list which is a group of operands enclosed in parentheses and separated by commas (when two or more operands are in list) may be used. An operand list is processed as a set after removal of the outer parentheses during macro expansion.

Operands (or operand lists) may be nested in parentheses in the macro call for use within macro definitions.

For example:

```
     ONE $MACRO  P1,P2
```

specifies 2 parameters.

A call such as

```
     ONE PAR1,PAR2
```

will result in

PAR1 being associated with P1 and PAR2 being associated with P2. However, a call such as:

```
     ONE PAR1,(PAR21,PAR22)
```

will result in PAR1 being associated with P1 and PAR21,PAR22 being associated with P2. Now :P2: or :P2.S: can be used as a pair of operands in a model statement.

Processing of each macro call in a source program causes the macro expander to associate the first parameter in the $MACRO statement with the first operand or operand list on the macro call line and the second parameter with the second operand or operand list, etc. Each parameter receiving a value has the $PCALL attribute set in the MST. When the macro definition has more parameters specified than the number of operands in the macro call, the $PCALL attribute is not set for the excess parameters. The $PCALL attribute is also not set if an operand is "null",i.e., the call line has two commas adjacent or an operand list of zero operands. Expansion of the macro can be controlled by the number of operands by using the $PCALL attribute and $IF statements. For example, a macro definition containing AMAC $MACRO P1,P2,P3 when called by AMAC AB1,AB2 sets $PCALL in parameters P1 and P2 but not for P3. Similarly, AMAC XY,,XY3 causes $PCALL to be set for P1 and P3 but not for

P2.

When the macro instruction has more operands than the number of parameters in the $MACRO statement, the excess operands are combined with the operand or operand list corresponding to the last parameter to form an operand list (or a longer operand list). For example, with the macro statements shown below, the operands of the two macro calls in the following c would be assigned to the parameters in the same ways:

```
(1)  ONE     EQU                  9
     TWO     EQU                  43
     THREE   EQU                  86
     FIX     $MACRO               P1,P2               MACRO FIX
               .
               .
               .
             FIX                  ONE,TWO,THREE       MACRO INSTRUCTION
             FIX                  ONE,(TWO,THREE)     MACRO INSTRUCTION
```

```
(2)     A           EQU           7
        B           EQU          15
        C           DATA         17
        D           DATA         63
        E           EQU          95
        F           EQU          47
        G           EQU          58
        H           EQU         101
        I           EQU         119
        PARM        $MACRO      P1,P2,P3,P4,P5,P6,P7,P8,P9
                      .
                      .
                      .
                    PARM        @A,,B,(),C,(D),E,(G,(H,I))
```

Parameter assignments:

```
P8.S = A              P2.S =                   (no string)
P8.A = $PCALL         P2.A =                   (all false)
P8.L = 1              P2.L = 0
P8.V = 7              P2.V = 0

P3.S = B              P4.S =                   (no string)
P3.A = $PCALL         P4.A = $POPL
P3.L = 1              P4.L = 0
P3.V = 15             P4.V = 0

P5.S = C              P6.S = D
P5.A = $PCALL         P6.A = $PCALL,$POPL
P5.L = 1              P6.L = 1
P5.V = 17             P6.V = 1
```

```
P7.S = E                 P8.S = G,(H,I)
P7.A = $PCALL            P8.A = $PCALL,$POPL
P7.L = 1                 P8.L = 7
P7.V = 95                P8.V = 2

P9.S =                                          (no string)
P9.A = 0                                        (all false)
P9.L = 0
P9.V = 0
```

NOTE

A macro definition will supercede previous macro definitions and native instructions with the same name. Symbolic operands which appear in a macro call are treated as symbolic operands in native instructions, i.e., if they are not defined with the program in which they appear, whey will be listed as undefined symbols.

8.3.5.2 $VAR Statement: The $VAR statement declares the variables for a macro definition. The $VAR statement is required only if the macro definition contains one or more variables other than parameters. More than one $VAR statement may be included and each $VAR statement may declare more than one variable. Each <var> in the operand is a variable as previously described, see Section 8.3.4.

SYNTAX;

```
        ...$VAR ...<var>[,<var>]... ...[<comment>]
```

The following is an example of a $VAR statement:

```
     $VAR A,CT,V3 Three variables for a macro
```

The example declares variables A, CT, and V3. A, CT, and V3 must not have been declared as parameters. The $VAR statement does not assign values to any components of the variables. $VAR statements may appear anywhere in the macro definition to which they apply, provided each variable is declared before the first statement that uses the variable. Placing $VAR statements immediately following the $MACRO statement is recommended.

8.3.5.3 $ASG Statement: The $ASG statement assigns values to the components of a variable. Variables that are not parameters do not have values for any components until values are assigned using $ASG statements. Components of variables with previously assigned values may be assigned new values with $ASG statements.

SYNTAX:

```
..$ASG ...<expression/string> TO <var> ...[<comment>]
```

The expression operand may be any expression valid to the assembler and may contain binary mode variable references and the keywords in Tables 8-3 and 8-4.

NOTE

> The binary mode value of a string component or symbol string component used in an expression is the binary value of the first two characters of the string. Thus, if GP.S has the string LAST, the value used for GP.S is an expression in the <string> hexadecimal number >4C41 which is the ASCII representation for LA.

A string may be one or more characters enclosed in single quotes, or the linking (concatenation) of such a literal string with the string mode value of a qualified variable. The <var> may be either an unqualified variable or a qualified variable.

When the operands are both unqualified variables, all components are transferred to target variables. When the destination variable is qualified, only the specified component receives the corresponding component of the expression or string. An exception to this is when a string is assigned to the string component of a variable or symbol, the length component of that variable or symbol is set to the number of characters in the assigned string. If the attribute component of the destination variable is to be changed, only those attributes which can be tested using keywords are changed. Other attributes maintained by the macro assembler may or may not be changed as appropriate.

NOTE

> A qualified variable that specifies the length component is illegal as a destination in a $ASG statement and will NOT set the length component.

The following examples show the use of the $ASG statement:

$ASG P3 TO V3	Assign all the components of variable P3 to variable V3.

$ASG :P3.S:'ES' TO P3.S	Concatenate string 'ES' to the string component of variable P3, and set the string component to the result. Also, add 2 to the value of the new length component.
$ASG :CT.A++PSYM TO CT.A	Set the flag in the attribute component of variable CT to indicate the symbolic address attribute.

Variables P3, V3, and CT must have been previously declared either as parameters in a $MACRO statement or as variables in a $VAR statement.

The $ASG statement may be used to modify symbol components as shown in the following examples. Assume the P3.V = 6 and P3.S = SUB.

$ASG 'TEN' TO G.S	Assigns 'TEN' as the string component of variable G. When 'TEN' is a symbol in the AST, this statement allows the use of indirect component qualifiers to modify the components of symbol TE
$ASG P3.V TO G.SV	Sets the value component of the symbol in the string component of variable G to the value component of variable P3. In this case, the value component of TEN is set to 6.
$ASG 'A':P3.S:'S' TO G.SS	Concatenates string 'A', the string component of variable P3, and string 'S' and places the result in the indirect string component of variable G. Also sets the length component of the same symbol. Thus, the string component of TEN is ASUBS and the length component is five.

NOTE

Keywords in a $ASG statement MUST be used with a Boolean operator and an attribute component of a variable in the source field. The attribute component must come first.

8.3.5.4 $IF Statement: The $IF statement provides conditional processing in a macro definition.

SYNTAX:

```
          ...$IF ...<expression> ...[<comment>]
```

An $IF statement is followed by a block of macro language statements terminated by an $ELSE statement or an $ENDIF statement. When the $ELSE statement is used, the $ELSE statement is followed by another block of macro language statements terminated by an $ENDIF statement. When the expression in the $IF statement has a nonzero value (or evaluated as TRUE), the block of statements following the $IF statement is processed. When the expression in the $IF statement has a zero value (or evaluated as FALSE), the block of statements following the $IF statement is skipped. When the $ELSE statement is used and the expression in the $IF statement has a nonzero value, the block of statements following the $ELSE statement and terminated by the $ENDIF statement is skipped. Thus, the condition of the $IF statement may determine whether or not a block of statements is processed, or which of two blocks of statements is processed. A block may consist of zero or more statements. The <expression> may be any expression as defined for the $ASG statement and may include qualified variables and keywords. The expression defines the condition for the $IF statement.

NOTE

The expression is always evaluated in binary mode. Specifically, the relational operations (<,>,=,#=) operate only on the binary mode values of macro variables. Boolean operators may be nested. See Section 8.3.3. In addition, $IF blocks may be nested, at most, 44 levels deep.

The following examples show conditional processing in macro definition:

```
    .
    .
    .
$IF         KY.SV      Process the  statement of  BLOCK A when the
    .                  indirect value component of the variable KY
    .       BLOCK A    contains  a  non-zero  value.  Process  the
    .                  statements of  BLOCK B  when the  component
$ELSE                  contains  zero.  After  processing  either
    .                  block of statements. continue processing at
    .       BLOCK B    the statement following the $ENDIF statement.
    .
$ENDIF
    .
    .
```

```
$IF --(T.A&$PCALL)        Process the statements of BLOCK A when the
   .                      attribute  component  of parameter T indi-
   .        BLOCK A       cates that parameter T was not supplied in
   .                      the macro instruction.  If parameter T was
   .                      supplied, do  not  process  the statements
$ENDIF                    of  BLOCK A.  Continue  processing  at the
   .                      statement following the  $ENDIF statements
   .                      in either case.
$IF         T.L=5         Process the statements of BLOCK A when the
   .                      length component of variable T is equal to
   .                      5,  do not process the statements of BLOCK
   .        BLOCK A       A.  Continue  processing at the  statement
$ENDIF                    following the $ENDIF statement.
```

8.3.5.5 $ELSE Statement: The $ELSE statement begins an alternate block to be processed if the preceding $IF expression was false. (See Section 8.3.5.6)

SYNTAX:

```
          ...$ELSE ...[<comment>]
```

8.3.5.6 $ENDIF Statement: The $ENDIF statement terminates the conditional processing initiated by an $IF statement in a macro definition. Examples of $ENDIF statements and their use are shown in a preceding paragraph. (See Section 8.3.5.6)

SYNTAX:

```
          ...$ENDIF ...[<comment>]
```

8.3.5.7 $END Statement: The $END statement marks the end of the group of statements that the macro definition named in the operand. When executed, the $END statement terminates the processing of the macro definition. The <macro name> parameter is optional.

SYNTAX:

```
          ...$END ...[<macro name>][<comment>]
```

The following is an example of an $END statement:

```
   $END FIX                  Terminates the definition of macro FIX.
```

8.3.6 Model Statements

As previously mentioned, a macro definition consists of model statements and statements that contain macro language verbs. A model statement always results in an assembly language statement. This statement may be composed of the usual elements of an assembly language statement combined with string mode qualified variable components, see Section 8.3.4.3. The resulting source statement must be a legal assembly language statement. The following examples show model statements:

MOV %6,R12 — This model statement is itself an assembly language source statement that contains a machine instruction.

:P7.S: MPY   :P2.S:,R8   :V4.S: — This model statement begins with the string component of variable P7. Three blanks, MPY, and three more blanks are concatinated to the string. The string component of variable P2 is concatenated to the result, to which R8 and three blanks are concatenated. A final concatenation places the string component of variable V4 in the model statement. The result is an assembly language machine instruction having the label and comment fields and part of the operand field supplied as string components.

:MS.S: — This model statement is the string component of variable MS. Preceding statements in the macro definition must place a valid assembly language source statement in the string component to prevent assembly errors.

NOTE

Conditional assembly directives may not appear as operations in a model statement. Comments supplied in model statements may not contain periods (.) since the macro assembler scans comments in the same way as model statements and improper use of punctuation may cause syntax errors.

8.4 MACRO EXAMPLES

Macros may simply substitute a machine instruction for a macro instruction, or they may include conditional processing, access the assembler symbol table, and employ recursion. Several examples of macro definitions are described in the following paragraphs.

8.4.1 Macro ID

Macro ID is an example of a macro with a default value. The macro supplies two DATA directives to the source program. It consists of nine macro language statements, four of which are model statements. The definition is as follows:

```
ID      $MACRO  WS,PC          Defines ID with parameters  WS  and
                               PC.

        DATA    :WS.S:         Model  statement  -  places  a DATA
                               directive with the  string  of  the
                               first  parameter  as the operand in
                               the source program.

        $IF     PC.A&$PCALL    Tests for presence of parameter PC.

        DATA    :PC.S:,15      Model statement  -  places  a  DATA
                               directive  in  the  source program.
                               The first operand is the string  of
                               the   second   parameter,  and  the
                               second   operand   is   15.    This
                               statement   is   processed  if  the
                               second parameter is present.
        $ELSE                  State  of  alternate   portion   of
                               definition.

        DATA    START,15       Model  statement  -  places  a DATA
                               directive in  the  source  program.
                               The  first  operand is label START,
                               and the second operand is 15.  This
                               statement   is   processed  if  the
                               second parameter is omitted.

START   EQU     $              Model statement -  places  a  label
                               START  in  the source program. This
                               statement  is  processed   if   the
                               second parameter is omitted.

        $ENDIF                 End of conditional processing.

        $END                   End of macro.
```

SYNTAX:

```
        [<LABEL>] ...ID ...<address>[,<address>] ...[<comment>]
```

The addresses may be expressions or symbols.

The following is an example of a macro instruction for macro ID:

```
ID WORK1,BEGIN
```

The resulting source code would be:

```
DATA WORK1
DATA BEGIN,15
```

If only one operand is supplied, the macro instruction could be coded as follows:

```
     ID  WORK2
```

This would result in the following source code:

```
          DATA WORK2
          DATA START,15
     START EQU $
```

This form of the macro instruction imposes two restrictions on the source program. The source program may not use the label START and may not call macro ID more than once. Problems with labels supplied in macros may be prevented by reserving certain characters for use in macro-generated labels. A macro definition may maintain a count of the number of times it is called and use this count in each label generated by the macro.

8.4.2 Macro GENCMT

This Macro GENCMT example shows how to implement both those comments which appear in the macro definition only, and those comments which appear in the expansion of the macro. When this macro is called, the statement in line six generates a comment.

```
 0001                  IDT    'GENCMT'
 0002           GENCMT $MACRO
 0003                  $VAR V
 0004           * THIS IS A MACRO DEFINITION COMMENT *
 0005                  $ASG '*' TO V.S
 0006           :V.S: THIS IS A MACRO EXPANSION COMMENT *
 0007                  $END
 0008                  GENCMT
 0001           * THIS IS A MACRO EXPANSION COMMENT *
 0009 0000 0000        DATA 0,1
      0002 0001
 0010                  GENCMT
 0001           * THIS IS A MACRO EXPANSION COMMENT *
 0011                  GENCMT
 0001           * THIS IS A MACRO EXPANSION COMMENT *
 0012 0004 0004        DATA 4
 0013                  END
NO ERRORS, NO WARNINGS
```

8.4.3 Macro FACT

Macro FACT is an example of the recursive use of macros. FACT produces the assembly code necessary to calculate the factorial of N, and store that value at data memory address LOC. Macro FACT accomplishes this by calling FACT1, which calls itself recursively.

```
FACT   $MACRO N,LOC
       $IF N.V<2
       MOV %1,A                     *  1! = 0! =1
       STA @:LOC:
       $ELSE
       MOV %:N.V:,A                 *  N greater than/equal 2, so store
       STA @:LOC:                   *     N at LOC
       $ASG N.V-1 TO N.V            *  DECREMENT N
       FACT1 :N.V:,:LOC:            *  DO FACTORIAL OF N-1
       $ENDIF
       $END
*
FACT1  $MACRO M,AREA
       $IF M.V>1
       LDA @:AREA:                  * Multiply factorial so far by
       MPY %:M.V:,A                 *    current position
       MOV B,A
       STA @:AREA:                  * Save result
       $ASG M.V-1 TO M.V            * Decrement position
       FACT1 :M.V:,:AREA:           * Recursively calls itself
       $ENDIF
       $END
```

8.4.4 Macro PULSE

This is a set of macros in which the name describes an addressing mode expected by the macro. The example assigns Register A to a port, Register B to a port, and an immediate value to a port. These macros can be usefull in programming I/O routines.

```
PULSEA $MACRO PX
       ORP  A,:PX.S:
       $END
*
PULSEB $MACRO PX
       ORP  B,:PX.S:
       $END
*
PULSEI $MACRO I,PX
       ORP  %:I.S:,:PX.S:
       $END
```

8.5 MACRO ERROR MESSAGES

Table 8-5 lists and defines the Macro error messages which may be generated.

TABLE 8-5 - MACRO ERROR MESSAGES

MACRO ERROR MESSAGE	DESCRIPTION
MACRO LINE TOO LONG	In a macro definition, macro directive lines may only be 58 characters long, and model statements, when fully expanded, may only be 60 characters long.
LONG MACRO VARIABLE QUALIFIER	Macro variable qualifiers may only be one or two characters in length.
TOO MANY MANY VARIABLES	The total number of macro parameters, variables and labels in one macro definition may not exceed 128.
INVALID MACRO QUALIFIER	The only valid macro qualifiers are: S,V, L, A, SS, SV, SL and SA.
VARIABLE ALREADY DEFINED	A macro variable cannot be redefined within a macro.
IF LEVEL EXCEEDED	The maximum nesting level of $IF directives is 44.
MACRO ASSEMBLER PROGRAM ERROR	The Macro Assembler has detected an internal error. These can be caused by incorrect syntax.

APPENDIX A

CHARACTER SETS RECOGNIZED BY THE ASSEMBLER

The TMS7000 Assembler recognizes the ASCII character listed in Table A-1. It also accepts the characters listed in Table A-2, if they occur within quoted strings or in comment fields. The special characters in Table A-3 are not accepted by the assembler but may be recognized and acted upon appropriately by other programs. The device service routine for the card reader accepts (and stores into the calling program''s buffer) all the characters listed in Tables A-1, A-2, and A-3.

All of the tables include the ASCII code for each character represented a a hexadecimal value and a decimal value. The tables also include the keypunch (Hollerith Code) for each character.

TABLE A-1 - ASCII CHARACTER SET

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) HOLLERITH CODE
20	32	Space	Blank
21	33	!	11-8-2
22	34	"	8-7
23	35	#	8-3
24	36	$	11-8-3
25	37	%	0-8-4
26	38	&	12
27	39	'	8-5
28	40	(	12-8-5
29	41	)	11-8-5
2A	42	*	11-8-4
2B	43	+	12-8-6
2C	44	,	0-8-3
2D	45	-	11
2E	46	.	12-8-3
2F	47	/	0-1
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5

(CONTINUED)

TABLE A-1 - ASCII CHARACTER SET

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) HOLLERITH CODE
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	8-2
3B	59	;	11-8-6
3C	60	<	12-8-4
3D	61	=	8-6
3E	62	>	0-8-6
3F	63	?	0-8-7
40	64	@	8-4
41	65	A	12-1
42	66	B	12-2
43	67	C	12-3
44	68	D	12-4
45	69	E	12-5
46	70	F	12-6
47	71	G	12-7
48	72	H	12-8
49	73	I	12-9
4A	74	J	11-1
4B	75	K	11-2
4C	76	L	11-3
4D	77	M	11-4
4E	78	N	11-5
4F	79	O	11-6
50	80	P	11-7
51	81	Q	11-8
52	82	R	11-9
53	83	S	0-2
54	84	T	0-3
55	85	U	0-4
56	86	V	0-5
57	87	W	0-6
58	88	X	0-7
59	89	Y	0-8
5A	90	Z	0-9
5B	91	[	12-2-8
5C	92	\	0-2-8 0-8-2
5D	93	]	11-1-8
5E	94	¬	11-7-8
5F	95	-	0-5-8

TABLE A-2 - SPECIAL CHARACTERS RECOGNIZED IN
QUOTED STRINGS AND COMMENT FIELDS

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	HOLLERITH CODE
60	96	`	8-1
61	97	a	12-0-1
62	98	b	12-0-2
63	99	c	12-0-3
64	100	d	12-0-4
65	101	e	12-0-5
66	102	f	12-0-6
67	103	g	12-0-7
68	104	h	12-0-8
69	105	i	12-0-9
6A	106	j	12-11-1
6B	107	k	12-11-2
6C	108	l	12-11-3
6D	109	m	12-11-4
6E	110	n	12-11-5
6F	111	o	12-11-6
70	112	p	12-11-7
71	113	q	12-11-8
72	114	r	12-11-9
73	115	s	11-0-2
74	116	t	11-0-3
75	117	u	11-0-4
76	118	v	11-0-5
77	119	w	11-0-6
78	120	x	11-0-7
79	121	y	11-0-8
7A	122	z	11-0-9
7B	123	{	12-0
7C	124	:	12-11
7D	125	}	11-0
7E	126	~	11-0-1
7F	127		12-9-7
80	128		11-0-9-8-1
81	129		0-9-1
82	130		0-9-2
83	131		0-9-3
84	132		0-9-4
85	133		11-9-5
86	134		12-9-6
87	135		11-9-7
88	136		0-9-8
89	137		0-9-8-1
8A	138		0-9-8-2
8B	139		0-9-8-3
8C	140		0-9-8-4

(Continued)

TABLE A-2 - SPECIAL CHARACTERS RECOGNIZED IN QUOTED STRINGS AND COMMENT FIELDS

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	HOLLERITH CODE
8D	141		12-9-8-1
8E	142		12-9-8-2
8F	143		11-9-8-3
90	144		12-11-0-9-8-1
91	145		9-1
92	146		11-9-8-2
93	147		9-3
94	148		9-4
95	149		9-5
96	150		9-6
97	151		12-9-8
98	152		9-8
99	153		9-8-1
9A	154		9-8-2
9B	155		9-8-3
9C	156		12-9-4
9D	157		11-9-4
9E	158		9-8-0
9F	159		11-0-9-1
A0	160		12-0-9-1
A1	161		12-0-9-2
A2	162		12-0-9-3
A3	163		12-0-9-4
A4	164		12-0-9-5
A5	165		12-0-9-6
A6	166		12-0-9-7
A7	167		12-0-9-8
A8	168		12-8-1
A9	169		12-11-9-1
AA	170		12-11-9-2
AB	171		12-11-9-3
AC	172		12-11-9-4
AD	173		12-11-9-5
AE	174		12-11-9-6
AF	175		12-11-9-7
B0	176		12-11-9-8
B1	177		11-8-1
B2	178		11-0-9-2
B3	179		11-0-9-3
B4	180		11-0-9-4
B5	181		11-0-9-5
B6	182		11-0-9-6
B7	183		11-0-9-7
B8	184		11-0-9-8
B9	185		0-8-1

(Continued)

TABLE A-2 - SPECIAL CHARACTERS RECOGNIZED IN QUOTED STRINGS AND COMMENT FIELDS

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	HOLLERITH CODE
BA	186		12-11-0
BB	187		12-11-0-9-1
BC	188		12-11-0-9-2
BD	189		12-11-0-9-3
BE	190		12-11-0-9-4
BF	191		12-11-0-9-5
C0	192		12-11-0-9-6
C1	193		12-11-0-9-7
C2	194		12-11-0-9-8
C3	195		12-0-8-1
C4	196		12-0-8-2
C5	197		12-0-8-3
C6	198		12-0-8-4
C7	199		12-0-8-5
C8	200		12-0-8-6
C9	201		12-0-8-7
CA	202		12-11-8-1
CB	203		12-11-8-2
CC	204		12-11-8-3
CD	205		12-11-8-4
CE	206		12-11-8-5
CF	207		12-11-8-6
D0	208		12-11-8-7
D1	209		11-0-8-1
D2	210		11-0-8-2
D3	211		11-0-8-3
D4	212		11-0-8-4
D5	213		11-0-8-5
D6	214		11-0-8-6
D7	215		11-0-8-7
D8	216		12-11-0-8-1
D9	217		12-11-0-1
DA	218		12-11-0-2
DB	219		12-11-0-3
DC	220		12-11-0-4
DD	221		12-11-0-5
DE	222		12-11-0-6
DF	223		12-11-0-7

TABLE A-3 ON NEXT A-

TABLE A-3 - ADDITIONAL CHARACTERS RECOGNIZED BY THE OPERATIVE SYSTEM DEVICE SERVICE ROUTINE

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	HOLLERITH CODE
00	0	NUL	12-0-9-8-1
01	1	SOH	12-9-1
02	2	STX	12-9-2
03	3	ETX	12-9-3
04	4	EOT	9-7
05	5	ENQ	0-9-8-5
06	6	ACK	0-9-8-6
07	7	BEL	0-9-8-7
08	8	BS	11-9-6
09	9	HT	12-9-5
0A	10	LF	0-9-5
0B	11	VT	12-9-8-3
0C	12	FF	12-9-8-4
0D	13	CR	12-9-8-5
0E	14	SO	12-9-8-6
0F	15	SI	12-9-8-7
10	16	DLE	12-11-9-8-1
11	17	DC1	11-9-1
12	18	DC2	11-9-2
13	19	DC3	11-9-3
14	20	DC4	11-9-4
15	21	NAK	9-8-5
16	22	SYN	9-2
17	23	ETB	0-9-6
18	24	CAN	11-9-8
19	25	EM	11-9-8-1
1A	26	SUB	9-8-7
1B	27	ESC	0-9-7
1C	28	FS	11-9-8-4
1D	29	GS	11-9-8-5
1E	30	RS	11-9-8-6
1F	31	US	11-9-8-7
7F	127	DEL	12-9-7

APPENDIX B

TMS7000 DATA ORGANIZATION

B.1 GENERAL

The TMS7000 is an 8-bit processor manipulating data organized into register areas, program areas, and file areas. The paragraphs that follow describe byte organization, 16-bit data organization, and the three general data areas present in the TMS7000 environment.

B.2 BYTE ORGANIZATION

The byte consists of eight bits of memory. The least significant bit (LSB) is designated bit 0; the most significant bit (MSB) is bit 7. Figure B-1 illustrates byte organization.

```
 7                                  0
_________________________________________
|    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |
|____|____|____|____|____|____|____|____|
MSB                                  LSB
```

FIGURE B-1 - BYTE ORGANIZATION

B.3 16-BIT ORGANIZATION

Some TMS7000 instructions produce 16-bit results. A 16-bit data area is organized with the MSB as bit 15 and the LSB as bit 0. Figure B-2 illustrates 16-bit data organization.

```
 15                               8    7                                 0
_________________________________________________________________________________
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|____|____|____|____|____|____|____|____|____|____|____|____|____|____|____|____|
MSB                                                                           LSB
```

FIGURE B-2 - SIXTEEN BIT DATA ORGANIZATION

B.4 REGISTER AREAS

The TMS7000 has access to three, special purpose hardware registers and 128 general purpose registers. Both types of register areas are described below.

B.4.1 Hardware Register Areas

The hardware registers accessed by the TMS7000 are as follows:

- o The 16-bit program counter (PC) containing the address of the next instruction to be executed. The address in the PC is used by the processor to fetch the next instruction from the program area. Following the fetch, the PC is incremented.
- o The 8-bit status register (ST). Three conditional status bits and the current state of the interrupt enable flag are present in the ST. Figure B-3 illustrates the status register.
- o The 8-bit stack pointer (SP) pointing to the last entry on the data stack. The data stack provides subroutine and interrupt capability as well as temporary data storage.

7	6	5	4	3	2	1	0
C	N	Z	I	0	0	0	0

N= sign bit	- most significant bit of result.
C= carry out	- 1 if carry out results; zero otherwise.
Z= zero	- 1 if result if zero; zero otherwise.
I= interrupt enable	- 1 if interrupts enbabled; zero otherwise.

FIGURE B-3 - STATUS REGISTER CONFIGURATION .

B.4.2 General Purpose Registers

The TMS7000 has 128 general purpose registers forming the registers file (RF). The RF is located in random access memory addresses >0000 through >007F. The registers are consecutively numbered from R0 to R127 and respectively located at >0000 to >007F. Registers R0, and R1 are also referred to as A and B registers. Registers in the register file may be addressed as registers by instructions having register file addressing modes or as memory locations by instructions having general memory addressing mode.

B.5 PROGRAM AREAS

The main body of a program is contained in the program area. Programs and subroutines are coded to solve an equation, run a motor, determine the status of a process, set or reset control lines, etc. The program area consists of 2048 bytes of ROM located at address >F800 to >FFFF. External memory can be added to expand the total program area to 63,226 bytes.

B.6 PHERIPHERAL FILE.

The on-chip input/output (I/O) resources are mapped into the peripheral file (PF) as illustrated in Figure B-5. The PF is memory-mapped into locations >0100 to >01FF. I/O Ports, I/O status, and I/O control registers are included with the peripheral file. P0 is mapped into >0100 and P255 into >01FF. A peripheral file location may be addressed using the I/O addressing modes or as a memory location by instructions with general memory addressing nodes.

```
                     **************************
>0000 - >007F        *   RAM register file    *
                     **************************
>007F - >00FF        *       future use       *
                     **************************
>0100 - >01FF        *    peripheral file     *
                     **************************
>0200 - >F7FF        *   memory expansion     *
                     **************************
>F800 - >FFFF        *  ROM program memory    *
                     **************************
```

FIGURE B-4 - TMS7020 PERIPHERAL FILE MEMORY MAP

APPENDIX C

ASSEMBLER DIRECTIVE TABLE

The assembler directives for the assembly language are listed in Table C-1. All directives may include a comment field following the operand field. Those directives that do not require an operand field may have a comment field following the operator field. Those directives that have optional operand fields (RORG) and (END) may have comment fields only when they have operand fields.

The following symbols and conventions are used in defining the syntax of assembler directives:

Angle brackets: < > :	Enclose items supplied by the user.
Brackets: [ ]:	Enclose optional items.
An Ellipsis: (...):	Indicates that the preceding item may be repeated.
Braces: { }:	Enclose two or more items of which one must be chosen.

The following words are used in defining the items used in assembler directives:

Symbol:	A character string composed of letters/digits representing a specific concept/label/value/expression.
String:	A character string of a length defined for each directive.
Exp:	An expression.
WD-Exp:	Well-defined expression.
WA:	Absolute expression in the range from 0 to 15.
Operation:	Mnemonic operation code, macro name, or previously defined operation or extended operation.

TABLE C-1 - ASSEMBLER DIRECTIVES

DIRECTIVE	SYNTAX
Output Options*	OPTION <keyword>[,<keyword>]...
Page Title	[<label>] TITL<string>
Program Identifier	[<label>] IDT<string>
Copy Source File	[<label>] COPY<file name>
External Definition	[<label>] DEF<symbol>[,<symbol>]
Secondary Reference	[<label>] SREF<symbol>[,<symbol>]
Absolute Origin	[<label>] AORG<wd expr>
Relocatable Origin	[<label>] RORG [<expr>]
Dummy Origin .	[<label>] DORG<expr>
Block Starting With Symbol	[<label>] BSS<wd expr>
Block Ending With Symbol	[<label>] BES<wd expr>
Initialize Word	[<label>] DATA<expr>[,<expr>]...
Initialize Text Operation	[<label>] TEXT [-] <string>
Define Assembly-Time Constant	<label> EQU<expr>
Word Boundary	[<label>] EVEN
No Source List	[<label>] UNL
List Source	[<label>] LIST
Page Eject	[<label>] C-
Initialize Byte	[<label>] BYTE<,wd expr>[,<wd expre>]...
Program End	[<label>] END [<symbol>]
Program Segment	[<label>] PSEG
Program Segment End	[<label>] PEND
Data Segment .	[<label>] DSEG
Data Segment End	[<label>] DEND
Common Segment	[<label>] CSEG[<string>b..[<comment>]]
Common Segment End	[<label>] CEND
Load Object	[<label>] LOAD <symbol>[,<symbol>][<comment>]
END	[<label>] END[Symbol]
Macro Library	[<label>] MLIB<string>

* VALID KEYWORDS (Output Options):

B:	Limit the listing of BYTE directives to one line.
D:	Limit the listing of BYTE directives to one line.
F:	Turn off all unlist options.
N:	Do not produce a listing.
S:	Produce a symbol table listing in the object file.
T:	Limit the listing of TEXT directives to one line.
X:	Produce a cross-reference listing.

APPENDIX D

TMS7000 HEXADECIMAL INSTRUCTION TABLE/OPCODE MAP

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Lo \ Hi	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000 0	NOP								MOVP Pn,A			TSTA/ CLRC	MOV A,B	MOV A,Rn	JMP	TRAP 15
0001 1	IDLE									MOVP Pn,B			TSTB	MOV B,Rn	JN/ JLT	TRAP 14
0010 2		MOV Rn,A	MOV %n,A	MOV Rn,B	MOV Rn,Rn	MOV %n,B	MOV B,A	MOV %n,Pn	MOVP A,Pn	MOVP B,Pn	MOVP %n,Pn	DEC A	DEC B	DEC Rn	JZ/ JEQ	TRAP 13
0011 3		AND Rn,A	AND %n,A	AND Rn,B	AND Rn,Rn	AND %n,B	AND B,A	AND %n,R	ANDP A,Pn	ANDP B,Pn	ANDP %n,Pn	INC A	INC B	INC Rn	JC/ JHS	TRAP 12
0100 4		OR Rn,B	OR %n,A	OR Rn,B	OR Rn,Rn	OR %n,B	OR B,A	OR %n,R	ORP A,Pn	ORP B,Pn	ORP %n,Pn	INV A	INV B	INV Rn	JP/ JGT	TRAP 11
0101 5	EINT	XOR Rn,A	XOR %n,A	XOR Rn,B	XOR Rn,Rn	XOR %n,B	XOR B,A	XOR %n,R	XORP A,Pn	XORP B,Pn	XORP %n,Pn	CLR A	CLR B	CLR Rn	JPZ/ JGE	TRAP 10
0110 6	DINT	BTJO Rn,A	BTJO %n,A	BTJO Rn,B	BTJO Rn,Rn	BTJO %n,B	BTJO B,A	BTJO %n,R	BTJOP A,Pn	BTJOP B,Pn	BTJOP %n,Pn	XCHB A	XCHB B	XCHB Rn	JNZ/ JNE	TRAP 9
0111 7	SETC	BTJZ Rn,A	BTJZ %n,A	BTJZ Rn,B	BTJZ Rn,Rn	BTJZ %n,B	BTJZ B,A	BTJZ %n,R	BTJZP A,Pn	BTJZP B,Pn	BTJZP %n,Pn	SWAP A	SWAP B	SWAP Rn	JNC/ JL	TRAP 8
1000 8	POP ST	ADD Rn,A	ADD %n,A	ADD Rn,B	ADD Rn,Rn	ADD %n,B	ADD B,A	ADD %n,R	MOVD %n,Rn	MOVD Rn,Rn	MOVD %n(B),Rn	PUSH A	PUSH B	PUSH Rn	TRAP 23	TRAP 7
1001 9	STSP	ADC Rn,A	ADC %n,A	ADC Rn,B	ADC Rn,Rn	ADC %n,B	ADC B,A	ADC %n,R				POP A	POP B	POP Rn	TRAP 22	TRAP 6
1010 A	RETS	SUB Rn,A	SUB %n,A	SUB Rn,B	SUB Rn,Rn	SUB %n,B	SUB B,A	SUB %n,R	LDA @n	LDA •Rn	LDA @n(B)	DJNZ A	DJNZ B	DJNZ Rn	TRAP 21	TRAP 5
1011 B	RETI	SBB Rn,A	SBB %n,A	SBB Rn,B	SBB Rn,Rn	SBB %n,B	SBB B,A	SBB %n,R	STA @n	STA •Rn	STA @n(B)	DECD A	DECD B	DECD Rn	TRAP 20	TRAP 4
1100 C		MPY Rn,A	MPY %n,A	MPY Rn,B	MPY Rn,Rn	MPY %n,B	MPY B,A	MPY %n,R	BR @n	BR •Rn	BR @n(B)	RR A	RR B	RR Rn	TRAP 19	TRAP 3
1101 D	LDSP	CMP Rn,A	CMP %n,A	CMP Rn,B	CMP Rn,Rn	CMP %n,B	CMP B,A	CMP %n,R	CMPA @n	CMPA •Rn	CMPA @n(B)	RRC A	RRC B	RRC Rn	TRAP 18	TRAP 2
1110 E	PUSH ST	DAC Rn,A	DAC %n,A	DAC Rn,B	DAC Rn,Rn	DAC %n,B	DAC B,A	DAC %n,R	CALL @n	CALL •Rn	CALL @n(B)	RL A	RL B	RL Rn	TRAP 17	TRAP 1
1111 F		DSB Rn,A	DSB %n,A	DSB Rn,B	DSB Rn,Rn	DSB %n,B	DSB B,A	DSB %n,R				RLC A	RLC B	RLC Rn	TRAP 16	TRAP 0

A	–	A Register
B	–	B Register
Rn	–	Register File
Pn	–	Peripheral File
%n	–	Immediate
@n	–	Direct
•Rn	–	Indirect

APPENDIX E

TMS7000 INSTRUCTION OPCODE SET

	SINGLE OPERAND			DUAL OPERAND										PERIPHERAL					EXTENDED			OTHER	STATUS WORD	
	A	B	Rn	A,B	B,A	Rn,A	%n,A	Rn,B	%n,B	Rn,Rn	%n,Rn	A,Rn	B,Rn	A,Pn	Pn,A	B,Pn	Pn,B	%n,Pn	DIRECT	INDIRECT	INDEXED	OTHER	COND BITS	INT. EN
ADC					69	19	29	39	59	49	79												X	
ADD					68	18	28	38	58	48	78												X	
AND					63	13	23	33	53	43	73												X	
ANDP														83		93		A3					X	
BTJO					66	16	26	36	56	46	76												X	
BTJOP														86		96		A6					X	
BTJZ					67	17	27	37	57	47	77												X	
BTJZP														87		97		A7					X	
BR																			8C	9C	AC			
CALL																			8E	9E	AE			
CLR	B5	C5	D5																				X	
CLRC																						B0	X	
CMP					6D	1D	2D	3D	5D	4D	7D												X	
CMPA																			8D	9D	AD		X	
DAC					6E	1E	2E	3E	5E	4E	7E												X	
DEC	B2	C2	D2																				X	
DECD	BB	CB	DB																				X	
DINT																						06	X	X
DJNZ	BA	CA	DA																					
DSB					6F	1F	2F	3F	5F	4F	7F												X	
EINT																						05	X	X
IDLE																						01		
INC	B3	C3	D3																				X	
INV	B4	C4	D4																				X	
JMP																						E0		
JC/JHS																						E3		
JN/JLT																						E1		
JNC/JL																						E7		
JNZ/JNE																						E6		
JP/JGT																						E4		
JPZ/JGE																						E5		
JZ/JEQ																						E2		
LDA																			8A	9A	AA		X	
LDSP																						0D		
MOV				C0	62	12	22	32	52	42	72	D0	D1										X	
MOVD																			88	98	A8		X	
MOVP														82	80	92	91	A2					X	
MPY					6C	1C	2C	3C	5C	4C	7C												X	
NOP																						00		
OR					64	14	24	34	54	44	74												X	
ORP														84		94		A4					X	
POP	B9	C9	D9																			08	X	
PUSH	B8	C8	D8																			0E	X	
RETI																						0B		
RETS																						0A		
RL	BE	CE	DE																				X	
RLC	BF	CF	DF																				X	
RR	BC	CC	DC																				X	
RRC	BD	CD	DD																				X	
SBB					6B	1B	2B	3B	5B	4B	7B												X	
SETC																						07	X	
STA																			8B	9B	AB		X	
STSP																						09		
SUB					6A	1A	2A	3A	5A	4A	7A												X	
SWAP	B7	C7	D7																				X	
TSTA																						B0	X	
TSTB																						C1	X	
TRAP																						E8-FF		
XCHB	B6		D6																				X	
XOR					65	15	25	35	55	45	75												X	
XORP														85		95		A5					X	

INDEX