

TMS32010 Assembly Language Programmer's Guide

Digital Signal Processor
Products



TEXAS
INSTRUMENTS

TMS32010 Assembly Language Programmer's Guide

Digital Signal Processor Products



**TEXAS
INSTRUMENTS**

IMPORTANT NOTICE

Texas Instruments reserves the right to make changes at any time in order to improve design and to supply the best product possible.

Texas Instruments assumes no responsibility for infringement of patents or rights of others based on Texas Instruments applications assistance or product specifications, since TI does not possess full access to data concerning the use or applications of customer's products. TI also assumes no responsibility for customer product designs.

Copyright © 1983 by Texas Instruments Incorporated

TABLE OF CONTENTS

SECTION		PAGE
1.	INTRODUCTION	1-1
1.1	General Description	1-1
1.2	Assembly Language Application	1-1
1.3	Linking Program Modules	1-2
1.4	Program Relocatability	1-2
1.5	Macros	1-3
2.	GENERAL PROGRAMMING INFORMATION	2-1
2.1	Introduction	2-1
2.2	Data Areas	2-1
2.3	The TMS32010 Instruction Set	2-1
2.4	Source Statement Format	2-3
2.4.1	Label Field	2-4
2.4.2	Command Field	2-4
2.4.3	Operand Field	2-5
2.4.4	Comment Field	2-5
2.5	Constants	2-5
2.5.1	Decimal Integer Constants	2-5
2.5.2	Binary Integer Constants	2-5
2.5.3	Hexadecimal Integer Constants	2-5
2.5.4	Character Constants	2-6
2.5.5	Assembly-Time Constants	2-6
2.6	Symbols	2-6
2.6.1	Predefined Symbols	2-7
2.6.2	Terms	2-7
2.6.3	Character Strings	2-7
2.7	Expressions	2-8
2.7.1	Arithmetic Operators in Expressions	2-8
2.7.2	Parentheses in Expressions	2-8
2.7.3	Well-Defined Expressions	2-9
2.7.4	Relocatable Symbols in Expressions	2-9
2.7.5	Externally Defined Symbols in Expressions	2-10
3.	ASSEMBLY INSTRUCTIONS	3-1
3.1	Introduction	3-1
3.2	Addressing Modes	3-1
3.2.1	Direct Addressing Mode	3-1
3.2.2	Indirect Addressing Mode	3-1
3.2.3	Immediate Addressing Mode	3-2
3.3	Instruction Addressing Format	3-2
3.3.1	Direct Addressing Format	3-2
3.3.2	Indirect Addressing Format	3-2
3.3.3	Immediate Addressing Format	3-2
3.3.4	Examples of Instruction Format	3-3
3.4	Instruction Set	3-3
3.4.1	Symbols and Abbreviations	3-3

TABLE OF CONTENTS (Continued)

SECTION		PAGE
	3.4.2 Instruction Set Summary	3-4
	3.4.3 Instruction Descriptions	3-5
4.	SAMPLE ROUTINES	4-1
4.1	Introduction	4-1
4.2	Initializing the TMS32010	4-1
4.3	BIOZ Instruction	4-2
4.4	BANZ Instruction	4-3
4.5	LTD Instruction	4-4
4.6	SUBC Instruction	4-5
4.7	CALA Instruction	4-7
4.8	32-Bit Arithmetic Capabilities of TMS32010	4-8
4.9	Table Read from Program Memory Instruction	4-9
4.10	Interrupt Instruction	4-10
4.11	Stack Expansion	4-11
5.	ASSEMBLER DIRECTIVES	5-1
5.1	Introduction	5-1
5.2	The TMS32010 Assembler	5-1
5.3	Assembler Directives	5-1
5.3.1	Directives that Affect the Location Counter	5-1
5.3.2	Directives that Affect Assembler Output	5-17
5.3.3	Directives that Initialize Constants	5-23
5.3.4	Directives that Provide Linkage Between Programs	5-27
5.3.5	Miscellaneous Directives	5-33
6.	PROGRAM LINKING	6-1
6.1	Introduction	6-1
6.2	Relocation Capability	6-1
6.3	Linking Program Modules	6-2
6.3.1	External Reference Directives	6-2
6.3.2	External Definition Directive	6-3
6.3.3	Program Identifier Directive	6-3
6.3.4	Linking	6-3
7.	ASSEMBLER OUTPUT	7-1
7.1	Introduction	7-1
7.2	Source Listing	7-1
7.3	Assembler Error Messages	7-2
7.4	Cross-Reference Listing	7-5
7.5	Object Code	7-6
7.5.1	Object Code Format	7-6
7.5.2	External References in Object Code	7-10
7.5.3	Changing Object Code	7-10
8.	MACRO CAPABILITY	8-1
8.1	Introduction	8-1

TABLE OF CONTENTS (Concluded)

SECTION		PAGE
8.2	Defining Macros	8-1
	8.2.1 Sample Macros	8-3
8.3	Macro Language Elements	8-4
	8.3.1 Strings	8-4
	8.3.2 Constants and Operators	8-4
	8.3.3 Variables	8-5
	8.3.3.1 Parameters	8-5
	8.3.3.2 Macro Symbol Table	8-6
	8.3.3.3 Variable Qualifiers	8-7
	8.3.4 Keywords	8-8
	8.3.4.1 Symbol Attribute Component Keywords	8-9
	8.3.4.2 Parameter Attribute Keywords	8-9
	8.3.5 Verbs	8-10
	8.3.5.1 \$MACRO Statement	8-10
	8.3.5.2 \$VAR Statement	8-13
	8.3.5.3 \$ASG Statement	8-13
	8.3.5.4 \$IF Statement	8-14
	8.3.5.5 \$ELSE Statement	8-16
	8.3.5.6 \$ENDIF Statement	8-16
	8.3.5.7 \$END Statement	8-16
	8.3.6 Model Statements	8-16
8.4	Macro Examples	8-17
	8.4.1 Macro ID	8-17
	8.4.2 Macro GENCM T	8-18
	8.4.3 Macro FACT	8-19
	8.4.4 Macro FFT	8-19
8.5	Macro Error Messages	8-21

LIST OF APPENDICES

APPENDIX		PAGE
A	TMS32010 Hardware Summary	A-1
B	Character Sets Recognized by the Assembler	B-1

LIST OF ILLUSTRATIONS

FIGURE		PAGE
1-1	Development Process	1-2
7-1	Cross-Reference Listing Format	7-5
7-2	Sample Object Code	7-6

LIST OF TABLES

TABLE		PAGE
2-1	Results of Operations on Absolute and Relocatable Items in Expressions	2-10
3-1	Instruction Symbols	3-4
3-2	Instruction Set Summary	3-5
5-1	Assembler Directives that Affect the Location Counter	5-2
5-2	Directives that Affect Assembler Output	5-17
5-3	Directives that Initialize Constants	5-23
5-4	Directives that Provide Linkage Between Programs	5-27
5-5	Miscellaneous Directives	5-32
7-1	Assembly Listing Errors	7-3
7-2	Symbol Attributes	7-5
7-3	Object Record Format and Tags	7-9
8-1	Variable Qualifiers	8-7
8-2	Variable Qualifiers for Symbol Components	8-8
8-3	Symbol Attribute Keywords	8-9
8-4	Parameter Attribute Keywords	8-10
8-5	Macro Error Messages	8-21

1. INTRODUCTION

1.1 GENERAL DESCRIPTION

An assembly language is a computer-oriented language for writing programs, consisting of symbolic instructions and assembler directives. In assembly instructions, the user assigns symbolic addresses to memory locations and specifies instructions by means of symbolic (mnemonic) operation codes. The user specifies instruction operands by means of symbolic addresses, numbers, and expressions consisting of symbolic addresses and numbers. Assembler directives control the processes of making a machine language program from the assembly language program, placing data in the program, and assigning symbols to values to be used in the program. Assembler directives that place data in memory locations allow the user to assign symbolic addresses to those locations.

Assembly language is computer-oriented in that the mnemonic operation codes correspond directly to machine instructions. The chief advantage an assembly language offers over a machine language is that the symbols of assembly language are easier to use and easier to remember than the 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 assembly language for the TMS32010 and TMS320M10 16/32-bit high-performance digital signal processors.

1.2 ASSEMBLY LANGUAGE APPLICATION

An assembly language program, called a source program, must be processed by an assembler to obtain a machine language program that can be executed by the computer. Processing of a source program is called assembling, because it consists of assembling the binary values (that correspond to the mnemonic operation code) with the binary address information to form the machine language instruction.

To illustrate the place 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 the consequent corrections indicated.
- 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.

1.3 LINKING PROGRAM MODULES

The assembler commands include two pairs of directives, DEF/REF and SREF/LOAD, that generate the information required to link program modules, thus removing the constraint of having to assemble an entire program at once. A long program may be divided into more manageable components in order to avoid a time-consuming assembly. Also, these smaller units reduce the size of the symbol table (an entry is made in the symbol table for every symbol used in the program). Components of a large program are then linked by the link editor (also called the linker) to form a complete executable program.

1.4 PROGRAM RELOCATABILITY

A major advantage of the TMS32010 Assembler is its ability to generate relocatable object code modules which can then be linked by the link editor to form an executable program. (Absolute code, on the other hand, must occupy a dedicated area of memory and cannot be moved as necessity dictates. This means that repetitive code in a program must be written into the program each time it is needed.)

The ability to relocate modules simplifies the programming task. Programs designed as a set of modules are easier to code, test, and debug, and are easier to understand and maintain. Relocatability also permits multiple programmers to work on a program's components.

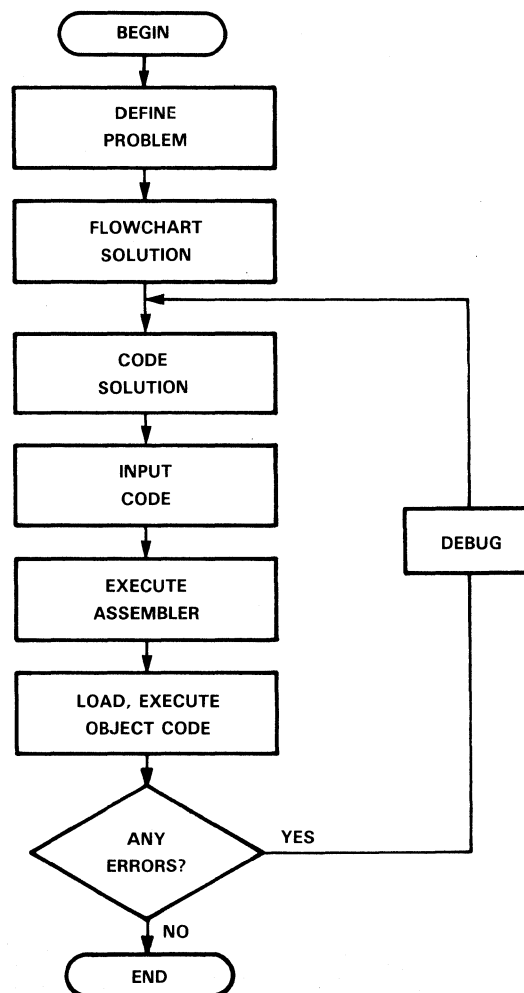


FIGURE 1-1 – DEVELOPMENT PROCESS

1.5 **MACROS**

The macro capability adds great flexibility to the assembler and provides the means to create a macro language that is capable of calling source statements from other locations within a program. A macro call statement fetches the source statements defined by the macro and substitutes them for the macro as if they had been written in that location in the program.

The obvious advantage of using macro code is that less source code must be written; this in turn means that the programs are easier to read and debug. In addition, macros usually execute faster than a comparable absolute code routine because no branching is involved. The macro capability is discussed in Section 8.

2. GENERAL PROGRAMMING INFORMATION

2.1 INTRODUCTION

The TMS32010 Assembly Language is a powerful set of instructions consisting of mnemonic operation codes (called mnemonics) that correspond directly to binary machine instructions. The assembly language program, as coded by the programmer, is called a source program. Before it can be executed by the computer, this source program must be processed by the assembler to obtain a machine language program. This processing of a source program is called assembling. This consists of assembling the binary values (which correspond to the mnemonic operation code) with the binary address information, to form the machine language instruction.

Assembler directives (see Section 5) control the process of making a machine language program from the assembly language program, placing data in the program, and assigning values to symbols to be used in the program.

2.2 DATA AREAS

The data manipulated by the TMS32010 is organized into four areas:

- Register areas: Two 16-bit auxiliary registers, a 1-bit auxiliary register pointer, a 32-bit T register; a 32-bit P register; an accumulator, and a 4 X 12 hardware stack area. In addition, The TMS32010 CPU has access to the 12-bit program counter (PC), the 16-bit status register (ST), and the 1-bit data page pointer (DP).
- 1536 X 16-bit read-only-memory (ROM) program areas containing the main program and subroutines.
- 144 X 16-bit on-chip RAM data memory areas comprising data tables.
- Eight I/O Ports.

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

2.3 THE TMS32010 INSTRUCTION SET

The TMS32010 instruction set is composed of 60 instructions that provide for the input, output, manipulation, and comparison of data. The instruction set is divided into eight functional categories. They are as follows:

- 1) ACCUMULATOR INSTRUCTIONS: Provide a variety of ways to add, subtract, load, and store the accumulator.

MNEMONIC	DESCRIPTION
ABS	ABSOLUTE VALUE OF ACCUMULATOR
ADD	ADD TO ACCUMULATOR WITH SHIFT
ADDH	ADD TO HIGH ACCUMULATOR
ADDS	ADD TO ACCUMULATOR WITH NO SIGN EXTENSION
LAC	LOAD ACCUMULATOR WITH SHIFT
LACK	LOAD ACCUMULATOR IMMEDIATE
SACH	STORE HIGH ACCUMULATOR

SACL	STORE LOW ACCUMULATOR
SUB	SUBTRACT FROM ACCUMULATOR WITH SHIFT
SUBC	CONDITIONAL SUBTRACT (FOR DIVIDE)
SUBH	SUBTRACT FROM HIGH ACCUMULATOR
SUBS	SUBTRACT FROM ACCUMULATOR WITH NO EXTENSION
ZAC	ZERO ACCUMULATOR
ZALH	ZERO ACCUMULATOR AND LOAD HIGH
ZALS	ZERO ACCUMULATOR AND LOAD LOW

- 2) AUXILIARY REGISTER AND DATA PAGE INSTRUCTION: Load, store, modify, and compare ARs and ARP.

MNEMONIC	DESCRIPTION
LAR	LOAD AUXILIARY REGISTER
LARK	LOAD AUXILIARY REGISTER IMMEDIATE
LARP	LOAD AUXILIARY REGISTER POINTER IMMEDIATE
LDP	LOAD DATA PAGE MEMORY POINTER
LDPK	LOAD DATA MEMORY PAGE POINTER IMMEDIATE
MAR	MODIFY AUXILIARY REGISTER AND POINTER
SAR	STORE AUXILIARY REGISTER

- 3) T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS: Provide for the preparation for and execution of a multiply.

MNEMONIC	DESCRIPTION
APAC	ADD P REGISTER TO ACCUMULATOR
LT	LOAD T REGISTER
LTA	LOAD T REGISTER AND ACCUMULATOR PRODUCT
LTD	LOAD T REGISTER, ACCUMULATOR PRODUCT, AND MOVE DATA IN MEMORY FORWARD ONE LOCATION
MPY	MULTIPLY T REGISTER BY DATA MEMORY VALUE AND STORE THE PRODUCT IN P REGISTER
MPYK	MULTIPLY T REGISTER BY IMMEDIATE OPERAND AND STORE VALUE IN P REGISTER
PAC	LOAD ACCUMULATOR FROM P REGISTER
SPAC	SUBTRACT P REGISTER FROM ACCUMULATOR

- 4) BRANCH INSTRUCTIONS: Permit testing of a variety of conditions, along with subroutine calls.

MNEMONIC	DESCRIPTION
B	BRANCH UNCONDITIONALLY
BANZ	BRANCH ON AUXILIARY REGISTER NOT ZERO
BGEZ	BRANCH IF ACCUMULATOR $>$ OR $=$ 0
BGZ	BRANCH IF ACCUMULATOR $>$ 0
BIOZ	BRANCH ON I/O STATUS $=$ 0
BLEZ	BRANCH IF ACCUMULATOR $<$ OR $=$ 0
BLZ	BRANCH IF ACCUMULATOR $<$ 0

BNZ	BRANCH IF ACCUMULATOR NOT = 0
BV	BRANCH ON OVERFLOW
BZ	BRANCH IF ACCUMULATOR = 0
CALA	CALL SUBROUTINE INDIRECT VIA ACCUMULATOR
CALL	CALL SUBROUTINE
RET	RETURN FROM SUBROUTINE

- 5) CONTROL INSTRUCTIONS: Affect the overflow mode, enable and disable interrupts, and store certain registers which cannot be stored by other instructions.

MNEMONIC	DESCRIPTION
DINT	DISABLE INTERRUPT
EINT	ENABLE INTERRUPT
LST	LOAD STATUS REGISTER
NOP	NO OPERATION
POP	TOP OF STACK TO ACCUMULATOR
PUSH	PUSH ACCUMULATOR TO TOP OF STACK
ROVM	RESET OVERFLOW MODE
SOVM	SET OVERFLOW MODE
SST	STORE STATUS REGISTER

- 6) BOOLEAN OPERATIONS: Perform logical operations between the accumulator and data memory.

MNEMONIC	DESCRIPTION
AND	AND WITH LOW ACCUMULATOR
OR	OR WITH LOW ACCUMULATOR
XOR	EXCLUSIVE OR WITH LOW ACCUMULATOR

- 7) I/O AND DATA MEMORY OPERATIONS: Allow input/output of data to external peripherals, provide for transfer of data within data memory or between program and data memory.

MNEMONIC	DESCRIPTION
DMOV	SHIFT CONTENTS OF DATA MEMORY ADDRESS FORWARD ONE LOCATION
IN	INPUT DATA FROM PORT
OUT	OUTPUT DATA TO PORT
TBLR	TABLE READ FROM PROGRAM MEMORY TO DATA MEMORY
TBLW	TABLE WRITE FROM DATA MEMORY TO PROGRAM MEMORY

Detailed information concerning these instructions is presented in Section 3.

2.4 SOURCE STATEMENT FORMAT

An Assembly Language source program consists of source statements that may contain assembler directives, machine 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 effect on the assembly. The source statement line may be as long as the source file format allows; however, the assembler will truncate the source line to 60 characters without warning. The user should insure that nothing other than comments extend past column 60.

The character set accepted by the TMS32010 Assembler consists of the ASCII character set as well as special characters that are undefined in ASCII. Appendix B contains tables that list the TMS32010 Assembler character set, along with associated ASCII and Hollerith codes.

The syntax for source statements is:

```
[<label>]      <mnemonic>      [<operand>]      [<comment>]  
EXAMPLE:  LOOP IN MEM,PA0      INPUT NEXT DATA SAMPLE
```

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 last source statement of a source program, usually the END directive, 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).

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 $ Where $ represents the current value of the location counter at that  
point in the assembly.
```

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:

- Assembler mnemonic of a machine instruction (e.g., IN)
- Macro directive (e.g., \$MACRO)
- Assembler directive (e.g., DATA)

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 Sections 2.5 and 2.7). 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, as 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 source 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 the following five types of constants, each internally maintained as a 16-bit quantity:

- Decimal integer constants
- Binary integer constants
- Hexadecimal integer constants
- Character constants
- Assembly-time constants

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 greater than $32,767$ are considered negative when interpreted as two's complement values.

The following are valid decimal constants:

1000	Constant equal to 1000 or $>03E8$
-32768	Constant equal to -32768 or >8000
25	Constant equal to 25 or >0019

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:

?000000000010011	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, '>'. If less than four hexadecimal digits are specified, the assembler will right-justify the bits which are specified in the resulting constant. 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 (or >0078)
>F	Constant equal to 15 (or >000F)
>37AC	Constant equal to 14252 (or >37AC)

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. If less than two characters are specified, the assembler will right-justify the given bits in the resulting constant. The characters are represented internally as 8-bit ASCII characters. A character constant consisting of only 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 >0043
'N'	Represented internally as >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 Section 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. REF and SREF directives provide access to symbols defined in other programs (see Section 5.3.4).

The following are examples of valid symbols:

START	START is assigned the value of the location where it appears in the label field.
ADD	ADD is assigned the value of the location where it appears in the label field.
OPERATION	OPERAT (assembler recognizes only the first six characters) is assigned the value of the location where it appears in the label field.

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 auxiliary register symbols are of the form "ARn," where 'n' is a constant 0 or 1.

The port addresses are of the form "PAn," where n is a constant in the range from 0 to 7.

The following are examples of valid predefined symbols:

\$	Represents the current location
ARO	Represents Auxiliary Register 0
PA0	Represents Port Address 0

2.6.2 Terms

Terms are used in the operand fields of machine instructions and assembler directives. A term may be a decimal, binary, character, 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 8-bit ASCII. Appendix B 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 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). Unary minus is the same as taking the two's complement, and unary invert is the same as taking the one's complement. The # symbol causes the value of the logical complement of the following constant, symbol, or expression to be used. An expression may not contain embedded blanks. The valid range of values for an expression is $-32,768$ to $+65,535$. Symbols that are defined as external references may be operands of arithmetic instructions within certain limits, as described in Section 2.7.4.

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

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 unary invert will be performed last. The assembler does not assign arithmetic operation precedence to any operation other than unary plus, unary minus, or unary invert. All arithmetic operations take precedence over the unary invert (#) operation. The expression following a unary invert (i.e., '#') must be resolved to an absolute value. 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/underflow conditions when arithmetic operations are performed at assembly time and gives the warning message "VALUE TRUNCATED" whenever an overflow/underflow occurs.; Examples of "VALUE TRUNCATED" messages are as follows:

$-2 * >4001$	$>FFFE + 2$	$-1 * >8001$
$>8000 * 2$	$- >8000 - 1$	$-2 * >8000$

2.7.2 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 is considered as 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; and add the resulting product to the value of LAB1.

2.7.3 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. The evaluation of a well-defined expression must also be absolute, and a well-defined expression cannot contain a character constant. An example of a well-defined expression is:

$> 1000 + X$ Where X must have been previously defined

2.7.4 Relocatable Symbols in Expressions

An expression that contains a relocatable symbol or constant immediately following a multiplication or division operator is illegal. When the result of evaluating an expression up to a multiplication or division operator is relocatable, the expression is also 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. The following are legal expressions involving relocatable symbols:

$BLUE + 1$	The sum of the value of symbol BLUE plus 1 is legal and of the same type as BLUE (BLUE can be an absolute or a relocatable symbol).
$GREEN - 4$	The result of subtracting 4 from the value of symbol GREEN is legal and of the same type as GREEN (GREEN can be an absolute or a relocatable symbol).
$2 * 16 + RED$	The sum of the value of symbol RED plus the product of 2 times 16 is legal, and of the same type as RED (RED can be an absolute or a relocatable symbol).
$440/2 - RED$	The result of dividing 440 by 2 and then subtracting the value of symbol RED from the quotient is absolute (RED must be absolute to make this a legal expression).

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

IF A IS	AND B IS	RESULT OF A + B	RESULT OF A – B	RESULT OF A*B	RESULT OF A/B
ABS	ABS	ABS	ABS	ABS	ABS (B<>0)*
ABS	RELOC	RELOC	illegal	Note 1	illegal
RELOC	ABS	RELOC	RELOC	Note 2	Note 3
RELOC	RELOC	illegal	Note 4	illegal	illegal

* <> indicates "not equal"

- NOTES:
1. Illegal unless A equals zero or one. If A equals one, the result is relocatable; if A is zero, the result is an absolute zero.
 2. Illegal unless B equals zero or one. If B equals one, the result is relocatable; if B is zero, the result is absolute zero.
 3. Illegal unless B equals one, in which case the result is relocatable.
 4. Illegal unless A and B are in the same section, in which case the result is absolute.

2.7.5 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, a comma, or a unary invert. 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 or division operations on the symbol (as for a relocatable symbol described in Section 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 externally 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.

3. ASSEMBLY INSTRUCTIONS

Assembly language instructions for the TMS32010 microcomputer are described in this section. Descriptions of the addressing modes, formats for instruction addressing, and detailed instruction descriptions are included.

3.1 INTRODUCTION

The instruction set contains a full set of branch instructions. Combined with the Boolean operations and shifters, these instructions permit the bit manipulation and bit test capability needed for high-speed control operations. Double-precision operations are also supported by the instruction set. Some examples are ADDH (add to high-order accumulator) and ADDS (add to accumulator with sign extension suppressed), which allow easy manipulation of 32-bit numbers.

The TMS32010's hardware multiplier allows the MPY instruction to be executed in a single cycle. The SUBC (conditional subtract for divide) instruction performs the shifting and conditional branching necessary to implement a divide efficiently and quickly.

Two special instructions, TBLR (table read) and TBLW (table write), allow crossover between data memory and program memory. The TBLR instruction transfers words stored in program memory to the data RAM. This eliminates the need for a coefficient ROM separate from the program ROM, thus permitting the user to make efficient trade-offs as to the amount of ROM dedicated to program or coefficient store. The accompanying instruction, TBLW, transfers words in internal data RAM to an external RAM. In conjunction with TBLR, this instruction allows the use of external RAM to expand the amount of data storage.

When a very large amount of external data must be addressed (i.e., > 4K words), TBLR and TBLW can no longer serve as a means of expanding the data RAM. Then it becomes necessary to address external data RAM as a peripheral by using the IN and OUT instructions; these instructions permit a data word to be read into the on-chip RAM in only two cycles. This procedure requires a minimal amount of external logic and permits the accessing of almost unlimited amounts of data RAM. This is very useful for pattern recognition applications, such as speech recognition or image processing.

3.2 ADDRESSING MODES

Three main addressing modes are available with the TMS32010 instruction set direct, indirect, and immediate addressing.

3.2.1 Direct Addressing Mode

In direct addressing, seven bits of the instruction word concatenated with the data page pointer form the data memory address. This implements a paging scheme in which the first page contains 128 words and the second page contains 16 words. In a typical application, infrequently accessed variables, such as those used when performing an interrupt service routine, are stored on the second page.

3.2.2 Indirect Addressing Mode

Indirect addressing forms the data memory address from the least significant eight bits of one of two auxiliary registers, ARO and AR1. The auxiliary register pointer (ARP) selects the current auxiliary register. The auxiliary registers can be automatically incremented or decremented in parallel with the execution of any indirect instruction to permit single-cycle manipulation of data tables.

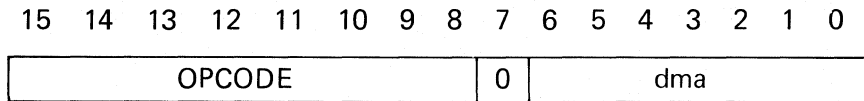
3.2.3. Immediate Addressing Mode

The TMS32010 instruction set contains special “immediate” instructions. These instructions derive data from part of the instruction word rather than from the data RAM. The constant in all immediate instructions may refer to values supplied by an external reference symbol. Some very useful immediate instructions are multiply immediate (MPYK), load accumulator immediate (LACK), and load auxiliary register immediate (LARK).

3.3 INSTRUCTION ADDRESSING FORMAT

The following sections describe the opcode format for the various addressing modes of the TMS32010.

3.3.1 Direct Addressing Format

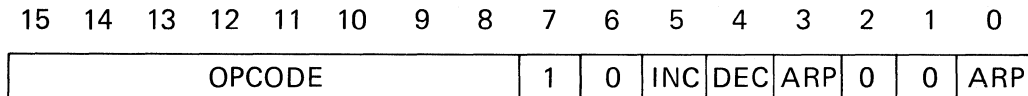


Bit 7 = 0 defines direct addressing mode. The opcode is contained in bits 15 through 8. Bits 6 through 0 contain data memory address.

The 7 bits of the data memory address (dma) field can directly address up to 128 words (1 page) of data memory. Use of the data memory page pointer is required to address the full 144 words of data memory.

Direct addressing can be used with all instructions requiring data operands except for the immediate operand instructions.

3.3.2. Indirect Addressing Format



Bit 7 = 1 defines indirect addressing mode. The opcode is contained in bits 15 through 8. Bits 6 through 0 contain indirect addressing control bits.

Bit 3 and bit 0 control the Auxiliary Register Pointer (ARP). If bit 3 = 0, then the contents of bit 0 are loaded into the ARP after execution of the current instruction. If bit 3 = 1, then the contents of the ARP remain unchanged. ARP = 0 defines the contents of ARO as a memory address. ARP = 1 defines the contents of AR1 as a memory address.

Bit 5 and bit 4 control the auxiliary registers. If bit 5 = 1, then ARP defines which auxiliary register is to be incremented by 1 after execution. If bit 4 = 1, then the ARP defines which auxiliary register is to be decremented by 1 after execution. If bit 5 and bit 4 are zero, then neither auxiliary register is incremented or decremented. Bits 6, 2, and 1 are reserved and should always be programmed to zero.

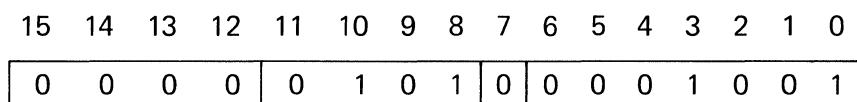
Indirect addressing can be used with all instructions requiring data operands, except for the immediate operand instructions.

3.3.3 Immediate Addressing Format

Included in the TMS32010’s instruction set are five immediate operand instructions (LDPK, LARK, MPYK, LACK, and LARP). In these instructions, the operand is contained within the instruction word.

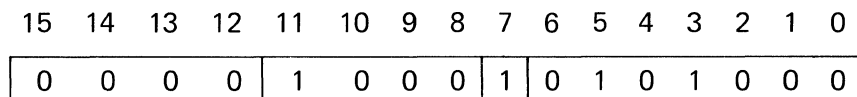
3.3.4 Examples of Opcode Format

- 1) ADD 9,5 Add to accumulator the contents of memory location 9 left-shifted 5 bits.



Note: Opcode of the ADD instruction is 0000 and appears in bits 15 through 12. Shift code of 5 appears in bits 11 through 8. Data memory address 9 appears in bits 6 through 0.

- 2) ADD *,8 Add to accumulator the contents of data memory address defined by contents of current auxiliary register. This data is left-shifted 8 bits before being added. The current auxiliary register is auto-incremented by 1.



Other variations of indirect addressing are as follows:

- 3) ADD *,8 As in example 2, but with no auto-increment; opcode would be >0888
- 4) ADD * -,8 As in example 2, except that current auxiliary register is decremented by 1; opcode would be >0898
- 5) ADD * + ,8,1 As in example 2, except that the auxiliary register pointer is loaded with the value 1 after execution; opcode would be >08A1
- 6) ADD * + ,8,0 As in example 2, except that the auxiliary register pointer is loaded with the value 0 after execution; opcode would be >08A0

3.4 INSTRUCTION SET

The following sections include the symbols and abbreviations that are used in the instruction set summary and in the instruction descriptions, the complete instruction set summary, and a description of each instruction.

All numbers are assumed to be decimal unless otherwise indicated. Hexidecimal numbers are specified by the symbol ">" before the number.

3.4.1. Symbols and Abbreviations

DATn and PRGn are assumed to have the symbolic value of n. They are used to represent any symbol with the value n.

TABLE 3-1 – INSTRUCTION SYMBOLS

SYMBOL	MEANING
ACC	Accumulator
AR	Auxiliary register (AR0 and AR1 are predefined assembler symbols equal to 0 and 1, respectively.)
ARP	Auxiliary register pointer
D	Data memory address field
DATn	Label assigned to data memory location n
dma	Data memory address
DP	Data page pointer
I	Addressing mode bit
INTM	Interrupt mode flag bit
K	Immediate operand field
>nn	Indicates nn is a hexadecimal number. All others are assumed to be decimal values.
OVM	Overflow (saturation) mode flag bit
P	Product (P) register
PA	Port address (PA0 through PA7 are predefined assembler symbols equal to 0 through 7, respectively)
PC	Program counter
pma	Program memory address
PRGn	Label assigned to program memory location n
R	1-bit operand field specifying auxiliary register
S	4-bit left-shift code
T	T register
TOS	Top of stack
X	3-bit accumulator left-shift field
→	Is assigned to .
	Indicates an absolute value
< >	Items within angle brackets are defined by user.
[]	Items within brackets are optional.
()	Indicates "contents of"
{ }	Items within braces are alternative items; one of them must be entered.
< >	Angle brackets back-to-back indicate "not equal".
	Blanks or spaces are significant.

3.4.2 Instruction Set Summary

The instruction set summary in the following table consists primarily of single-cycle single-word instructions. Only infrequently used branch and I/O instructions are multicyle.

TABLE 3-2 — INSTRUCTION SET SUMMARY

ACCUMULATOR INSTRUCTIONS				
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER
				15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ABS	Absolute value of accumulator	1	1	0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0
ADD	Add to accumulator with shift	1	1	0 0 0 0 ← S → I ← D →
ADDH	Add to high-order accumulator bits	1	1	0 1 1 0 0 0 0 0 1 ← D →
ADDS	Add to accumulator with no sign extension	1	1	0 1 1 0 0 0 0 1 1 ← D →
AND	AND with accumulator	1	1	0 1 1 1 1 0 0 1 1 ← D →
LAC	Load accumulator with shift	1	1	0 0 1 0 ← S → I ← D →
LACK	Load accumulator immediate	1	1	0 1 1 1 1 1 1 0 ← K →
OR	OR with accumulator	1	1	0 1 1 1 1 0 1 0 1 ← D →
SACH	Store high-order accumulator bits with shift	1	1	0 1 0 1 1 ← X → I ← D →
SACL	Store low-order accumulator bits	1	1	0 1 0 1 0 0 0 0 ← D →
SUB	Subtract from accumulator with shift	1	1	0 0 0 1 ← S → I ← D →
SUBC	Conditional subtract (for divide)	1	1	0 1 1 0 0 1 0 0 1 ← D →
SUBH	Subtract from high-order accumulator bits	1	1	0 1 1 0 0 0 1 0 1 ← D →
SUBS	Subtract from accumulator with no sign extension	1	1	0 1 1 0 0 0 1 1 1 ← D →
XOR	Exclusive OR with accumulator	1	1	0 1 1 1 1 0 0 0 1 ← D →
ZAC	Zero accumulator	1	1	0 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1
ZALH	Zero accumulator and load high-order bits	1	1	0 1 1 0 0 1 0 1 1 ← D →
ZALS	Zero accumulator and load low-order bits with no sign extension	1	1	0 1 1 0 0 1 1 0 1 ← D →

TABLE 3-2 - INSTRUCTION SET SUMMARY (CONTINUED)

AUXILIARY REGISTER AND DATA PAGE POINTER INSTRUCTIONS				
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER
				15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LAR	Load auxiliary register	1	1	0 0 1 1 1 0 0 R I ← D →
LARK	Load auxiliary register immediate	1	1	0 1 1 1 0 0 0 R ← K →
LARP	Load auxiliary register pointer immediate	1	1	0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 K
LDP	Load data memory page pointer	1	1	0 1 1 0 1 1 1 1 I ← D →
LDPK	Load data memory page pointer immediate	1	1	0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 K
MAR	Modify auxiliary register and pointer	1	1	0 1 1 0 1 0 0 0 I ← D →
SAR	Store auxiliary register	1	1	0 0 1 1 0 0 0 R I ← D →

BRANCH INSTRUCTIONS				
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER
				15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B	Branch unconditionally	2	2	1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BANZ	Branch on auxiliary register not zero	2	2	1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BGEZ	Branch if accumulator ≥ 0	2	2	1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BGZ	Branch if accumulator > 0	2	2	1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BIOZ	Branch on $\overline{BIO} = 0$	2	2	1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BLEZ	Branch if accumulator ≤ 0	2	2	1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BLZ	Branch if accumulator < 0	2	2	1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BNZ	Branch if accumulator ≠ 0	2	2	1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BV	Branch on overflow	2	2	1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
BZ	Branch if accumulator = 0	2	2	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
CALA	Call subroutine from accumulator	2	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0
CALL	Call subroutine immediately	2	2	1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ← BRANCH ADDRESS →
RET	Return from subroutine	2	1	0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1

TABLE 3-2 - INSTRUCTION SET SUMMARY (CONCLUDED)

T REGISTER, P REGISTER, AND MULTIPLY INSTRUCTIONS																			
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
APAC	Add P register to accumulator	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1
LT	Load T register	1	1	0	1	1	0	1	0	1	0	1	← D →						
LTA	LTA combines LT and APAC into one instruction	1	1	0	1	1	0	1	1	0	0	1	← D →						
LTD	LTD combines LT, APAC, and DMOV into one instruction	1	1	0	1	1	0	1	0	1	1	1	← D →						
MPY	Multiply with T register; store product in P register	1	1	0	1	1	0	1	1	0	1	1	← D →						
MPYK	Multiply T register with immediate operand; store product in P register	1	1	1	0	0	← K →												
PAC	Load accumulator from P register	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0
SPAC	Subtract P register from accumulator	1	1	0	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0

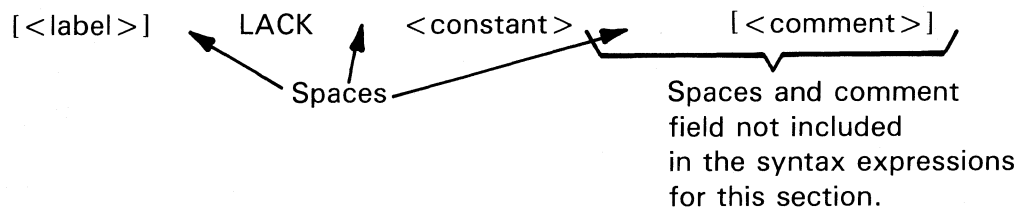
CONTROL INSTRUCTIONS																			
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DINT	Disable interrupt	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
EINT	Enable interrupt	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0
LST	Load status register	1	1	0	1	1	1	1	0	1	1	1	← D →						
NOP	No operation	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
POP	Pop stack to accumulator	2	1	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1
PUSH	Push stack from accumulator	2	1	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0
ROVM	Reset overflow mode	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	0	1	0
SOVM	Set overflow mode	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	0	1	1
SST	Store status register	1	1	0	1	1	1	1	1	0	0	1	← D →						

I/O AND DATA MEMORY OPERATIONS																			
MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE INSTRUCTION REGISTER															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMOV	Copy contents of data memory location into next location	1	1	0	1	1	0	1	0	0	1	1	← D →						
IN	Input data from port	2	1	0	1	0	0	0	← PA →		1	← D →							
OUT	Output data to port	2	1	0	1	0	0	1	← PA →		1	← D →							
TBLR	Table read from program memory to data RAM	3	1	0	1	1	0	0	1	1	1	1	← D →						
TBLW	Table write from data RAM to program memory	3	1	0	1	1	1	1	1	0	1	1	← D →						

3.4.3 Instruction Descriptions

Each instruction in the instruction set summary is described in the following pages. The instructions are listed in alphabetical order. An example is provided with each instruction.

Each instruction begins with an assembler syntax expression. Since the comment field which concludes the syntax is optional, it is not included in the syntax expression. A syntax example is given below that shows the spaces that are included and required in the syntax expression, and the optional comment field along with its preceding spaces that has been omitted.



ABS

Absolute Value of Accumulator

ABS

Assembler Syntax: [`<label>`] ABS

Operands: None

Operation: If $(ACC) < 0$
Then $-(ACC) \rightarrow ACC$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: If accumulator is greater than zero, then the accumulator is unchanged by the execution of this instruction. If the accumulator is less than zero, then the accumulator will be replaced by its two's complement value. Note that the hexadecimal number >80000000 is a special case. When the overflow mode is not set, the ABS of >80000000 is >80000000 . When in the overflow mode, the ABS of >80000000 is $>7FFFFFFF$.

Words: 1

Cycles: 1

Example: ABS

BEFORE INSTRUCTION		AFTER INSTRUCTION																			
31	0	31	0																		
ACC	<table border="1"><tr><td>></td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	>	0	0	0	0	1	2	3	4	ACC	<table border="1"><tr><td>></td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	>	0	0	0	0	1	2	3	4
>	0	0	0	0	1	2	3	4													
>	0	0	0	0	1	2	3	4													
and																					
ACC	<table border="1"><tr><td>></td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td></tr></table>	>	F	F	F	F	F	F	F	F	ACC	<table border="1"><tr><td>></td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	>	0	0	0	0	0	0	0	1
>	F	F	F	F	F	F	F	F													
>	0	0	0	0	0	0	0	1													

ADD

Add to Accumulator with Shift

ADD

Assembler Syntax:

Direct Addressing: [`<label>`] ADD `<dma>`[, `<shift>`]

Indirect Addressing: [`<label>`] ADD {`*|*+|*-`}[, `<shift>` [, `<ARP>`]]

Operands: $0 \leq \text{shift} \leq 15$
 $0 \leq \text{dma} \leq 127$
ARP = 0 or 1

Operation: $(\text{ACC}) + (\text{dma}) \times 2^{\text{shift}} \rightarrow \text{ACC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	0	0	0	SHIFT	0	DATA MEMORY ADDRESS
---------	---	---	---	---	-------	---	---------------------

Indirect:	0	0	0	0	SHIFT	1	SEE SECTION 3.3
-----------	---	---	---	---	-------	---	-----------------

Description: Contents of data memory address are left-shifted and added to accumulator. During shifting, low-order bits are zero-filled, and high-order bits are sign-extended. The result is stored in the accumulator.

Words: 1

Cycles: 1

Example: ADD DAT1,3
or
ADD *,3 If current auxiliary register contains the value 1.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
DATA MEMORY 1	2	DATA MEMORY 1	2
ACC	7	ACC	23

Note: If the contents of data memory address DAT2 is $> 8\text{BOE}$, then the following instruction sequence will leave accumulator with the value $> \text{FFF8B0E0}$.

ZAC Zero accumulator
ADD DAT2,4 ACC = $> \text{FFF8B0E0}$

ADDH

Add to High-Order Accumulator

ADDH

Assembler Syntax:

Direct Addressing: [`<label>`] ADDH `<dma>`
 Indirect Addressing: [`<label>`] ADDH `{*|*+|*-}`[`<ARP>`]

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: $(\text{ACC}) + (\text{dma}) \times 2^{16} \rightarrow \text{ACC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	0	0	0	0	0	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	0	0	0	0	0	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	---	-----------------

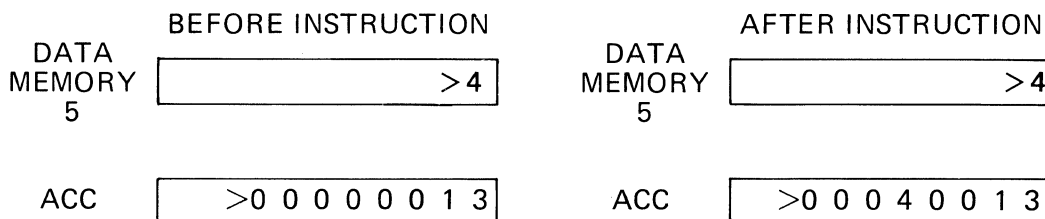
Description: Add contents of data memory address to upper half of the accumulator (bits 31 through 16).

Words: 1
Cycles: 1

Example: ADDH DAT5

or

· ADDH * If current auxiliary register contains the value 5.



Note: This instruction can be used in performing 32-bit arithmetic.

ADDS

Add to Low Accumulator with Sign-Extension Suppressed

ADDS

Assembler Syntax:

Direct Addressing: [`<label>`] `ADDS` `<dma>`
 Indirect Addressing: [`<label>`] `ADDS` `{*|*+|*-}`[,`<ARP>`]

Operands: $0 \leq dma \leq 127$
`ARP` = 0 or 1

Operation: `(ACC) + (dma) → ACC`

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	0	0	0	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	0	0	0	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: Add contents of specified data memory location with sign-extension suppressed. The data is treated as a 16-bit positive integer rather than a two's complement integer. Therefore, there is no sign-extension as there is with the `ADD` instruction.

Words: 1

Cycles: 1

Example: `ADDS DAT11`
 or
`ADDS *` If current auxiliary register contains the value 11.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
DATA MEMORY 11	>F 0 0 6	DATA MEMORY 11	>F 0 0 6
ACC	>0 0 0 0 0 0 0 3	ACC	>0 0 0 0 F 0 0 9

Notes: The following routines illustrate the difference between the `ADD` and `ADDS` instructions. Data memory location `DAT1` contains `>E007`.

```
ZAC            Zero ACC
ADDS  DAT1    ACC = >0000E007
```

```
ZAC            Zero ACC
ADD   DAT1,0   ACC = >FFFE007
```

The `ADDS` instruction can be used in implementing 32-bit arithmetic.

Assembler Syntax:

Direct Addressing: [`<label>`] AND `<dma>`
 Indirect Addressing: [`<label>`] AND {`*|*+|*-`}, [`<ARP>`]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: Zero. AND. high-order ACC bits: (`dma`). AND. low-order ACC bits → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	1	1	0	0	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	1	1	0	0	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

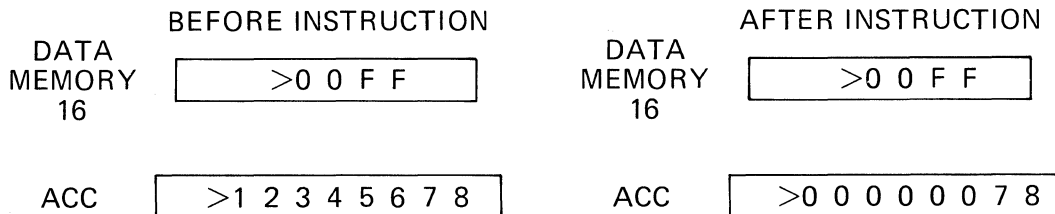
Description: The low-order bits of the accumulator are ANDed with the contents of the specified data memory address and concatenated with all zeroes ANDed with the high-order bits of the accumulator. The AND operation follows the truth table below.

DATA MEMORY BIT	ACC BIT (BEFORE)	ACC BIT (AFTER)
0	0	0
0	1	0
1	0	0
1	1	1

Words: 1
Cycles: 1

Example: AND DAT16

or
 AND * If current auxiliary register contains the value 16.



Note: This instruction is useful for examining bits of a word for high-speed control applications.

APAC

Add P Register to Accumulator

APAC

Assembler Syntax: [`<label>`] APAC

Operands: None

Operation: (ACC) + (P) → ACC

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1

Description: The contents of the P register, the result of a multiply, are added to the contents of the accumulator and the result is stored in the accumulator.

Words: 1

Cycles: 1

Example: APAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION		
P	<table border="1"><tr><td>64</td></tr></table>	64	<table border="1"><tr><td>64</td></tr></table>	64
64				
64				
ACC	<table border="1"><tr><td>32</td></tr></table>	32	<table border="1"><tr><td>96</td></tr></table>	96
32				
96				

Note: This instruction is a subset of the LTA and LTD instructions.

Assembler Syntax: [`<label>`] B `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: `pma` \rightarrow PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: Branch to location in program is specified by the program memory address (`pma`). `Pma` can be either a symbolic or a numeric address.

Words: 2

Cycles: 2

Example: B PRG191 191 is loaded into the program counter and program continues running from that location.

Assembler Syntax: [<label>] BANZ <pma>

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If (AR bits 8 through 0) $\neq 0$
 Then (AR) - 1 \rightarrow AR and pma \rightarrow PC
 Else (PC) + 2 \rightarrow PC
 (AR) - 1 \rightarrow AR

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
0				0				PROGRAM MEMORY ADDRESS							

Description: If the lower nine bits of the current auxiliary register are not equal to zero, then the auxiliary register is decremented, and the address contained in the following word is loaded into the program counter. If these bits equal zero, the current program counter is incremented and AR also is decremented. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2

Cycles: 2

Example: BANZ PRG35

BEFORE INSTRUCTION	AFTER INSTRUCTION
AR 1	AR 0
PC 46	PC 35
or	
AR 0	AR >1 F F
PC 46	PC 48

Note: This instruction can be used for loop control with the auxiliary register as loop counter. The auxiliary register is decremented after testing for zero. The auxiliary registers also behave as modulo 512 counters.

BGEZ

Branch if Accumulator Greater Than or Equal to Zero

BGEZ

Assembler Syntax: [`<label>`] BGEZ `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If (ACC) ≥ 0
Then pma \rightarrow PC
Else (PC) + 2 \rightarrow PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are greater than or equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2

Cycles: 2

Example: BGEZ PRG217 217 is loaded into the program counter if the accumulator is greater than or equal to zero.

BGZ

Branch if Accumulator Greater Than Zero

BGZ

Assembler Syntax: [`<label>`] BGZ `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If (ACC) > 0
Then pma → PC
Else (PC) + 2 → PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are greater than zero, branch to the specified program memory location. Branch to location in program specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2

Cycles: 2

Example: BGZ PRG342 342 is loaded into the program counter if the accumulator is greater than zero.

Assembler Syntax: [`<label>`] BIOZ `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If $\overline{\text{BIO}} = 0$
 Then $\text{pma} \rightarrow \text{PC}$
 Else $(\text{PC}) + 2 \rightarrow \text{PC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the $\overline{\text{BIO}}$ pin is active low, then branch to specified memory location. Otherwise, the program counter is incremented to the next instruction. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2

Cycles: 2

Example: BIOZ PRG64 If the $\overline{\text{BIO}}$ pin is active low, then a branch to location 64 occurs. Otherwise, the program counter is incremented.

Note: This instruction can be used in conjunction with the $\overline{\text{BIO}}$ pin to test if peripheral is ready to deliver an input. This type of interrupt is preferable when performing time-critical loops.

BLEZ

Branch if Accumulator Less Than or Equal to Zero

BLEZ

Assembler Syntax: [`<label>`] BLEZ `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If $(\text{ACC}) \leq 0$
Then $\text{pma} \rightarrow \text{PC}$
Else $(\text{PC}) + 2 \rightarrow \text{PC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are less than or equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2

Cycles: 2

Example: BLEZ PRG63 63 is loaded into the program counter if the accumulator is less than or equal to zero.

Assembler Syntax: [**<label>**] **BLZ** **<pma>**

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If (ACC) < 0
 Then pma → PC
 Else (PC) + 2 → PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are less than zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2

Cycles: 2

Example: BLZ PRG481 481 is loaded into the program counter if the accumulator is less than zero.

BNZ

Branch if Accumulator Not Equal to Zero

BNZ

Assembler Syntax: [`<label>`] BNZ `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If (ACC) $\neq 0$
Then `pma` \rightarrow PC
Else (PC) + 2 \rightarrow PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are not equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (`pma`). `Pma` can be either a symbolic or numeric address.

Words: 2

Cycles: 2

Example: BNZ PRG320 320 is loaded into the program counter if the accumulator does not equal zero.

Assembler Syntax: [`<label>`] BV `<pma>`

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If overflow flag = 1
 Then $\text{pma} \rightarrow \text{PC}$ and $0 \rightarrow \text{overflow flag}$
 Else $(\text{PC}) + 2 \rightarrow \text{PC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the overflow flag has been set, then a branch to the program address occurs and the overflow flag is cleared. Otherwise, the program counter is incremented to the next instruction. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or a numeric address.

Words: 2

Cycles: 2

Example: BV PRG610 If an overflow has occurred since the overflow flag was last cleared, then 610 is loaded into the program counter. Otherwise, the program counter is incremented.

Assembler Syntax: [<label>] BZ <pma>

Operands: $0 \leq \text{pma} < 2^{12}$

Operation: If (ACC) = 0
 Then pma → PC
 Else (PC) + 2 → PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: If the contents of the accumulator are equal to zero, branch to the specified program memory location. Branch to location in program is specified by the program memory address (pma). Pma can be either a symbolic or numeric address.

Words: 2

Cycles: 2

Example: BZ PRG102 102 is loaded into the program counter if accumulator is equal to zero.

Assembler Syntax: [<label>] CALA

Operands: None

Operation: (PC) + 1 → TOS
 (ACC bits 11 through 0) → PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The current program counter is incremented and pushed onto the top of the stack. Then, the contents of the 12 least significant bits of the accumulator are loaded into the PC.

Words: 1

Cycles: 2

Example: CALA

	BEFORE INSTRUCTION		AFTER INSTRUCTION
PC	25	PC	83
ACC	83	ACC	83
STACK	<div style="text-align: right; margin-right: 10px;"> 32 75 84 49 </div>	STACK	<div style="text-align: right; margin-right: 10px;"> 26 32 75 84 </div>

Note: This instruction is used to perform computed subroutine calls.

CALL

Call Subroutine Direct

CALL

Assembler Syntax: [`<label>`] CALL `<pma>`

Operands: $0 \leq \text{pma} < 212$

Operation: $(\text{PC}) + 2 \rightarrow \text{TOS}$
 $\text{pma} \rightarrow \text{PC}$

Encoding:

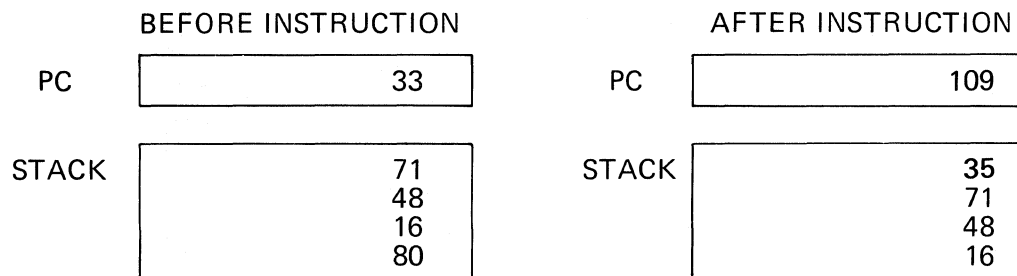
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	PROGRAM MEMORY ADDRESS											

Description: The current program counter is incremented and pushed onto the top of the stack. Then, the program memory address is loaded into the PC.

Words: 2

Cycles: 2

Example: CALL PRG109



DINT

Disable Interrupt

DINT

Assembler Syntax: [<label>] DINT

Operands: None

Operation: 1 → INTM

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The interrupt-mode flag (INTM) bit is set to logic 1. When this flag is set, any further maskable interrupts are disabled.

Words: 1

Cycles: 1

Example: DINT

Assembler Syntax:

Direct Addressing: [$\langle \text{label} \rangle$] DMOV $\langle \text{dma} \rangle$
 Indirect Addressing: [$\langle \text{label} \rangle$] DMOV $\{ * | * + | * - \} [, \langle \text{ARP} \rangle]$

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: $(\text{dma}) \rightarrow \text{dma} + 1$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	1	0	0	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	1	0	0	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: The contents of the specified data memory address are copied into the contents of the next higher address.

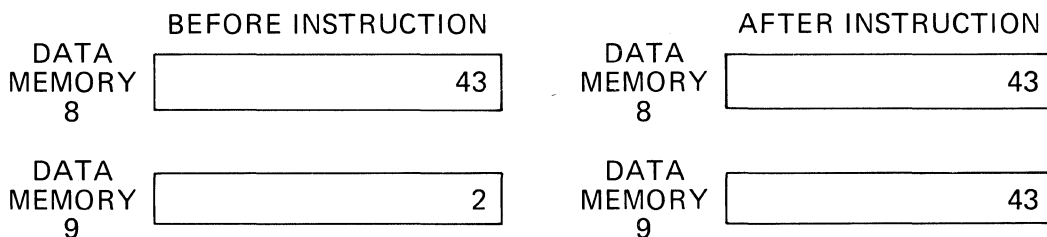
Words: 1

Cycles: 1

Example: DMOV DAT8

or

DMOV * If current auxiliary register contains the value 8.



Note: DMOV is an instruction that can be associated with Z⁻¹ in signal flow graphs. It is a subset of the LTD instruction. See LTD for more information.

EINT

Enable Interrupt

EINT

Assembler Syntax: [<label>] EINT

Operands: None

Operation: 0→INTM

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The interrupt-mode flag (INTM) in the status register is cleared to logic 0. When this flag is not set, maskable interrupts are enabled.

Words: 1

Cycles: 1

Example: EINT

Assembler Syntax:

Direct Addressing: [`<label>`] IN `<dma>`, `<PA>`
 Indirect Addressing: [`<label>`] IN {`*`|`*`+|`*`-}, `<PA>`[, `<ARP>`]

Operands: $0 \leq \text{dma} \leq 127$
 $0 \leq \text{PA} \leq 7$
 ARP=0 or 1

Operation: PA → address lines PA2-PA0
 Data bus D15-D0 → dma

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	0	0	0	PORT ADDRESS	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	--------------	---	---------------------

Indirect:	0	1	0	0	0	PORT ADDRESS	1	SEE SECTION 3.3
-----------	---	---	---	---	---	--------------	---	-----------------

Description: The IN instruction reads data from a peripheral and places it in data memory. It is a two-cycle instruction. During the first cycle, the port address is sent to address lines A2/PA2-A0/PA0. $\overline{\text{DEN}}$ goes low during the same cycle, strobing in the data which the addressed peripheral places on the data bus, D15-D0.

Words: 1

Cycles: 2

Example: IN STAT,PA5 Read in word from peripheral on port address 5.
 Store in data memory location STAT.

LARK 1,20 Load AR1 with decimal 20.

LARP 1 Load ARP with 1.

IN *,PA1,0 Read in word from peripheral on port address 1.
 Store in data memory location 20. Decrement AR1 to 19. Load the ARP with 0.

Notes: When the TMS32010 outputs address onto the three LSBs of address lines, the nine MSBs are zeroed.

Instruction causes the $\overline{\text{DEN}}$ line to go low during the first clock cycle of this instruction's execution. $\overline{\text{MEN}}$ remains high when $\overline{\text{DEN}}$ is active.

Assembler Syntax:

Direct Addressing: [`<label>`] LAC `<dma>` [, `<shift>`]
 Indirect Addressing: [`<label>`] LAC {`*|*+|*-`} [, `<shift>` [, `<ARP>`]]

Operands: $0 \leq \text{shift} \leq 15$
 $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: $(\text{dma}) \times 2^{\text{shift}} \rightarrow \text{ACC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	0	1	0	SHIFT	0	DATA MEMORY ADDRESS
---------	---	---	---	---	-------	---	---------------------

Indirect:	0	0	1	0	SHIFT	1	SEE SECTION 3.3
-----------	---	---	---	---	-------	---	-----------------

Description: Contents of data memory address are left-shifted and loaded into the accumulator. During shifting, low-order bits are zero-filled and high-order bits are sign-extended.

Words: 1
Cycles: 1

Example: LAC DAT6,4
 or
 LAC *,4 If current auxiliary register contains the value 6.

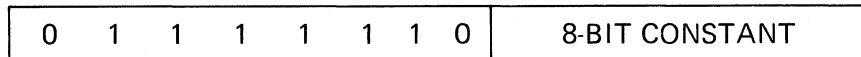
	BEFORE INSTRUCTION	AFTER INSTRUCTION		
DATA MEMORY 6	<table border="1"><tr><td>1</td></tr></table>	1	<table border="1"><tr><td>1</td></tr></table>	1
1				
1				
ACC	<table border="1"><tr><td>0</td></tr></table>	0	<table border="1"><tr><td>16</td></tr></table>	16
0				
16				

Assembler Syntax: [`<label>`] LACK `<constant>`

Operands: $0 \leq \text{constant} \leq 255$

Operation: `constant` → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



Description: The eight-bit constant is loaded into the accumulator right-justified. The upper 24 bits of the accumulator are zeros (i.e., sign extension is suppressed).

Words: 1

Cycles: 1

Example: LACK 15



Note: If a constant longer than eight bits is used, the XDS/320 assembler will truncate it to eight bits. No error message will be given.

Assembler Syntax:

Direct Addressing: [`<label>`] LAR `<AR>`,`<dma>`
 Indirect Addressing: [`<label>`] LAR `<AR>`,`{*|*+|*-}`[,`<ARP>`]

Operands: $0 \leq dma \leq 127$

AR = 0 or 1

ARP = 0 or 1

Operation: `(dma) → AR`

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

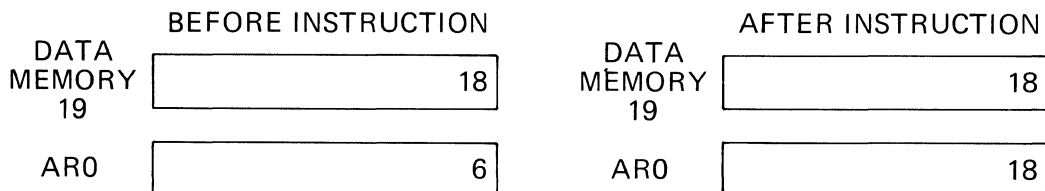


Description: The contents of the specified data memory address are loaded into the designated auxiliary register.

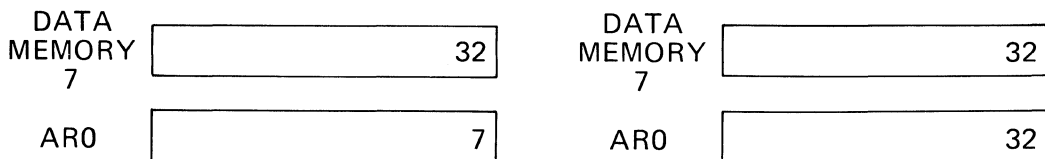
Words: 1

Cycles: 1

Example: LAR ARO,DAT19



also, LARP 0
LAR ARO,* -



Notes: ARO is not decremented after the LAR instruction. Generally as in the above case, if indirect addressing with autodecrement is used with LAR to load the current auxiliary register, the new value of the auxiliary register is not decremented as a result of instruction execution. The analogous case is true with autoincrement.

LAR and its companion instruction SAR (store auxiliary registers) should be used to store and load the auxiliary during subroutine calls and interrupts.

If an auxiliary register is not being used for indirect addressing, LAR and SAR enable it to be used as an additional storage register, especially for swapping values between data memory locations.

LARK

Load Auxiliary Register with Eight-Bit Constant

LARK

Assembler Syntax: [`<label>`] LARK `<AR>`,`<constant>`

Operands: $0 \leq \text{constant} \leq 255$
AR = 0 or 1

Operation: `constant` → AR

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	1	0	AUXILIARY REGISTER	8-BIT CONSTANT
---------	---	---	---	---	---	-----------------------	----------------

Description: The eight-bit positive constant is loaded into the designated auxiliary register right-justified and zero-filled (i.e., sign-extension suppressed).

Words: 1

Cycles: 1

Example: LARK AR0,21



Notes: This instruction is useful for loading an initial loop counter value into an auxiliary register for use with the BANZ instruction.

If a constant longer than eight bits is used, the XDS/320 assembler will truncate it to eight bits. No error message will be given.

LARP

Load Auxiliary Register Pointer Immediate

LARP

Assembler Syntax: [`<label>`] LARP `<constant>`

Operands: $0 \leq \text{constant} \leq 1$

Operation: `constant` \rightarrow ARP

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1-BIT CONSTANT
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-------------------

Description: Load a one-bit constant identifying the desired auxiliary register into the auxiliary register pointer.

Words: 1

Cycles: 1

Example: LARP 1 Any succeeding instructions will use auxiliary register 1 for indirect addressing.

Note: This instruction is a subset of MAR.

Assembler Syntax:

Direct Addressing: [`<label>`] LDP `<dma>`
 Indirect Addressing: [`<label>`] LDP `{*|*+|*-}`[`<ARP>`]

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: LSB of (dma) → DP (DP = 0 or 1)

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	1	1	1	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	1	1	1	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: The least significant bit of the contents of the specified data memory address is loaded into the data memory page pointer register (DP). All higher-order bits are ignored in the data word. DP = 0 defines page 0 which contains words 0-127. DP = 1 defines page 1 which contains words 128-143.

Words: 1

Cycles: 1

Example: LDP DAT1 LSB of location DAT1 is loaded into data page pointer.
 or
 LDP *,1 LSB of location currently addressed by auxiliary register is loaded into data page pointer. ARP is set to one.

LDPK

Load Data Page Pointer Immediate

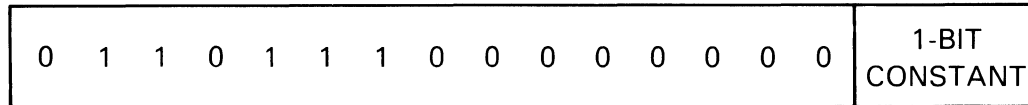
LDPK

Assembler Syntax: [`<label>`] LDPK `<constant>`

Operands: $0 \leq \text{constant} \leq 1$

Operation: `constant` → DP

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



650!

Description: The one-bit constant is loaded into the data memory page pointer register (DP). DP = 0 defines page 0 which contains words 0-127. DP = 1 defines page 1 which contains words 128-143.

Words: 1

Cycles: 1

Example: LDPK 0 Data page pointer is set to zero.

Assembler Syntax:

Direct Addressing: [$\langle \text{label} \rangle$] LST $\langle \text{dma} \rangle$
 Indirect Addressing: [$\langle \text{label} \rangle$] LST $\{ * | * + | * - \} [, \langle \text{ARP} \rangle]$

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: (dma) \rightarrow status bits

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	1	1	0	1	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	1	1	0	1	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: Restores the contents of the status register as saved by the store status (SST) instruction from a data memory word.

Words: 1

Cycles: 1

Example: LARP 0 The data memory word addressed by the contents of auxiliary register 0 replaces the status bits. The auxiliary register pointer becomes 1.
 LST *,1

Note: This instruction is used to load the TMS32010's status bits after interrupts and subroutine calls. These status bits include the Overflow Flag (OV) bit, Overflow Mode (OVM) bit, Auxiliary Register Pointer (ARP) bit, and the Data Memory Page Pointer (DP) bit. The Interrupt Mask bit cannot be changed by the LST instruction. These bits were stored (by the SST instruction) in the data memory word as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OV	OVM	INTM	1	1	1	1	ARP	1	1	1	1	1	1	1	DP

See SST.

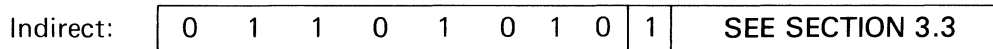
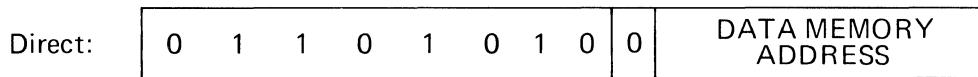
Assembler Syntax:

Direct Addressing: [$\langle \text{label} \rangle$] LT $\langle \text{dma} \rangle$
 Indirect Addressing: [$\langle \text{label} \rangle$] LT $\{ * | * + | * - \} [, \langle \text{ARP} \rangle]$

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: $(\text{dma}) \rightarrow T$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

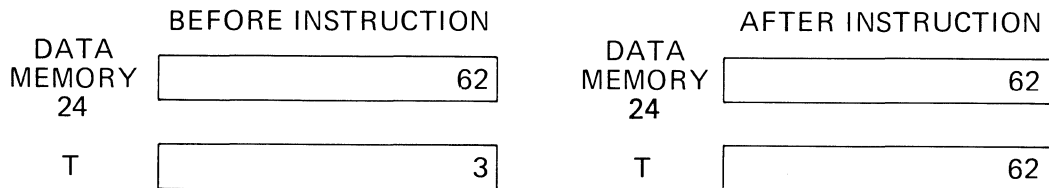


Description: LT loads the T register with the contents of the specified data memory location.

Words: 1

Cycles: 1

Example: LT DAT24
 or
 LT * If current auxiliary register contains the value 24.



Note: LT is used to load the T register in preparation for a multiplication. See MPY, LTA, and LTD.

Assembler Syntax:

Direct Addressing: [<label>] LTA <dma>
 Indirect Addressing: [<label>] LTA { * | * + | * - } [, <ARP>]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: (dma) → T
 (ACC) + (P) → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	1	1	0	0	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	1	1	0	0	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: The contents of the specified data memory address are loaded into the T register. Then, the P register, containing the previous product of the multiply operation, is added to the accumulator, and the result is stored in the accumulator.

Words: 1
Cycles: 1

Example: LTA DAT24
 or
 LTA *

If current auxiliary register contains the value 24.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
DATA MEMORY 24	62	DATA MEMORY 24	62
T	3	T	62
P	15	P	15
ACC	5	ACC	20

Note: This instruction is a subset of the LTD instruction.

Assembler Syntax:

Direct Addressing: [`<label>`] LTD `<dma>`
 Indirect Addressing: [`<label>`] LTD `{*|*+|*-}`[`,<ARP>`]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: $(dma) \rightarrow T$
 $(ACC) + (P) \rightarrow ACC$
 $(dma) \rightarrow dma + 1$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	1	0	1	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	1	0	1	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: The T register is loaded with the contents of the specified data memory address. Then, the contents of the P register are added to the accumulator. Next, the contents of the specified data memory address are transferred to the next higher data memory address.

Words: 1

Cycles: 1

Example: LTD DAT24

or

LTD *

IF current auxiliary register contains the value 24.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 24	62	62
DATA MEMORY 25	0	62
T	3	62
P	15	15
ACC	5	20

Assembler Syntax: [`<label>`] MAR {`*`|`*` + |`*` - }[, `<ARP>`]

Operands: ARP=0 or 1

Operation: Current auxiliary register is incremented, decremented, or remains the same. Auxiliary register pointer is loaded with the next ARP.

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	1	0	0	0	0	0	DATA MEMORY ADDRESS				
Indirect:	0	1	1	0	1	0	0	0	0	1	SEE SECTION 3.3				

Description: This instruction utilizes the indirect addressing mode to increment/decrement the auxiliary registers and to change the auxiliary register pointer. It has no other effect.

Words: 1

Cycles: 1

Example:

MAR *,1	Load ARP with 1.
MAR *-	Decrement current auxiliary register (in this case, AR1)
MAR *+,0	Increment current auxiliary register (AR1), load ARP with 0.

Note: In the direct addressing mode, MAR is a NOP. Also, the instruction LARP is a subset of MAR (i.e., MAR *,0 performs the same function as LARP 0).

Assembler Syntax:

Direct Addressing: [\langle label \rangle] MPY \langle dma \rangle
 Indirect Addressing: [\langle label \rangle] MPY { $|^* + |^* -$ }[, \langle ARP \rangle]

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: (T) x (dma) \rightarrow P

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	1	1	0	1	0	DATA MEMORY ADDRESS
Indirect:	0	1	1	0	1	1	0	1	1	SEE SECTION 3.3

Description: The contents of the T register are multiplied by the contents of the specified data memory address, and the result is stored in the P register.

Words: 1

Cycles: 1

Example: MPY DAT13

or
 MPY * If current auxiliary register contains the value 13.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
DATA MEMORY 13	7	DATA MEMORY 13	7
T	6	T	6
P	36	P	42

Note: During an interrupt, all registers except the P register can be saved. However, the TMS32010 has hardware protection against servicing an interrupt between an MPY or MPYK instruction and the following instruction. For this reason, it is advisable to follow MPY and MPYK with LTA, LTD, PAC, APAC, or SPAC.

No provisions are made for the condition of $> 8000 \times > 8000$. If this condition arises, the product will be $> C0000000$.

MPYK

Multiply Immediate

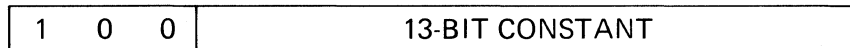
MPYK

Assembler Syntax: [`<label>`] MPYK `<constant>`

Operands: $(-2^{12}) \leq \text{constant} < 2^{12}$

Operation: $(T) \times \text{constant} \rightarrow P$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



Description: The contents of the T register are multiplied by the signed 13-bit constant and the result loaded into the P register.

Words: 1

Cycles: 1

Example: MPYK -9

	BEFORE INSTRUCTION	AFTER INSTRUCTION
T	<input type="text" value="7"/>	<input type="text" value="7"/>
P	<input type="text" value="42"/>	<input type="text" value="-63"/>

Note: No provision is made to save the contents of the P register during an interrupt. Therefore, this instruction should be followed by one of the following instructions: PAC, APAC, SPAC, LTA, or LTD. Provision is made in hardware to inhibit interrupt during MPYK until the next instruction is executed.

NOP

No Operation

NOP

Assembler Syntax: [`<label>`] NOP

7 F 80

Operands: None

Operation: None

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: No operation is performed.

Words: 1

Cycles: 1

Example: NOP

Note: NOP is useful as a "pad" or temporary instruction during program development.

Assembler Syntax:

Direct Addressing: [`<label>`] OR `<dma>`
 Indirect Addressing: [`<label>`] OR `{*|*+|*-}`[`<ARP>`]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: Zero. OR. high-order ACC bits: (dma). OR. low-order ACC bits → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	1	1	0	1	0	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	1	1	0	1	0	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description The low-order bits of the accumulator are ORed with the contents of the specified data memory address concatenated with all zeroes ORed with the high-order bits of the accumulator. The result is stored in the accumulator. The OR operation follows the truth table below.

DATA MEMORY BIT	ACC BIT (BEFORE)	ACC BIT (AFTER)
0	0	0
0	1	1
1	0	1
1	1	1

Words: 1
Cycles: 1

Example: OR DAT88

or
 OR * Where current auxiliary register contains the value 88.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 88	>F 0 0 0	>F 0 0 0
ACC	>0 0 1 0 0 0 0 2	>0 0 1 0 F 0 0 2

Note: This instruction is useful for comparing selected bits of a data word.

Assembler Syntax:

Direct Addressing: [`<label>`] OUT `<dma>`, `<PA>`
 Indirect Addressing: [`<label>`] OUT `{*|*+|*-}`, `<PA>`[`, <ARP>`]

Operands: $0 \leq dma \leq 127$
 $0 \leq PA \leq 7$
 ARP = 0 or 1

Operation: PA → address lines PA2-PA0
 (dma) → data bus D15-D0

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	0	0	1	PORT ADDRESS	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	--------------	---	---------------------

Indirect:	0	1	0	0	1	PORT ADDRESS	1	SEE SECTION 3.3
-----------	---	---	---	---	---	--------------	---	-----------------

Description: The OUT instruction transfers data from data memory to an external peripheral. The first cycle of this instruction places the port address onto address lines A2/PA2-A0/PA0. During the same cycle, \overline{WE} goes low and the data word is placed on the data bus D15-D0.

Words: 1
Cycles: 2

Example: OUT 120,7 Output data word stored in memory location 120 to peripheral on port address 7.
 OUT *,5 Output data word referenced by current auxiliary register to peripheral on port address 5.

Notes: When the TMS32010 sends the port address onto the three LSBs of the address lines, the nine MSBs are set to zero.

The OUT instruction causes the \overline{WE} line to go low during the first clock cycle of this instruction's execution. \overline{MEN} remains high during the first cycle.

Assembler Syntax: [`<label>`] PAC

Operands: None

Operation: (P) → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0

Description: The contents of the P register resulting from a multiply are loaded into the accumulator.

Words: 1

Cycles: 1

Example: PAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION
P	<input type="text" value="144"/>	<input type="text" value="144"/>
ACC	<input type="text" value="23"/>	<input type="text" value="144"/>

POP

Pop Top of Stack to Accumulator

POP

Assembler Syntax: [`<label>`] POP

Operands: None

Operation: (TOS) → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The contents of the top of stack are loaded into the accumulator. The next element on the stack becomes the top of the stack.

Words: 1

Cycles: 2

Example: POP

	BEFORE INSTRUCTION	AFTER INSTRUCTION								
ACC	<table border="1"><tr><td>82</td></tr></table>	82	<table border="1"><tr><td>45</td></tr></table>	45						
82										
45										
STACK	<table border="1"><tr><td>45</td></tr><tr><td>16</td></tr><tr><td>7</td></tr><tr><td>33</td></tr></table>	45	16	7	33	<table border="1"><tr><td>16</td></tr><tr><td>7</td></tr><tr><td>33</td></tr><tr><td>33</td></tr></table>	16	7	33	33
45										
16										
7										
33										
16										
7										
33										
33										

Note: The 12 bits of the stack are put into the accumulator in bits 11 through 0, and bits 31 through 12 are zeroed. There is no provision to check stack underflow.

PUSH

Push Accumulator onto Stack

PUSH

Assembler Syntax: [`<label>`] PUSH

Operands: None

Operation: (ACC) → TOS

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

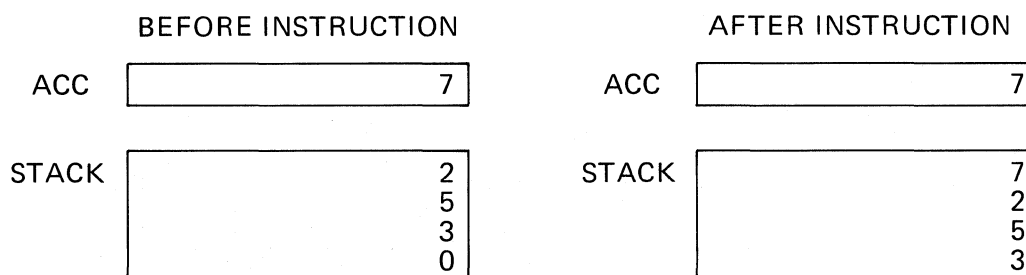
0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The contents of the lower 12 bits (11-0) of the accumulator are pushed onto the top of the hardware stack.

Words: 1

Cycles: 2

Example: PUSH



Note: There is no provision for detecting a stack overflow. Therefore, if the stack is already full, the contents of the bottom stack element will be lost upon execution of PUSH.

RET

Return from Subroutine

RET

Assembler Syntax: [`<label>`] RET

7F8D

Operands: None

Operation: (TOS) → PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The top element is popped off of the stack and loaded into the program counter.

Words: 1

Cycles: 2

Example: RET

	BEFORE INSTRUCTION		AFTER INSTRUCTION								
PC	<table border="1"><tr><td>96</td></tr></table>	96	PC	<table border="1"><tr><td>37</td></tr></table>	37						
96											
37											
STACK	<table border="1"><tr><td>37</td></tr><tr><td>45</td></tr><tr><td>75</td></tr><tr><td>75</td></tr></table>	37	45	75	75	STACK	<table border="1"><tr><td>45</td></tr><tr><td>75</td></tr><tr><td>75</td></tr><tr><td>75</td></tr></table>	45	75	75	75
37											
45											
75											
75											
45											
75											
75											
75											

Note: This instruction is used in conjunction with CALL and CALA for subroutines.

Assembler Syntax: [<label>] ROVM

Operand: None

Operation: 0→OVM

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: This instruction will reset the TMS32010 from the overflow mode it was placed in by the SOVM instruction. The overflow mode will set the accumulator and the ALU to their highest positive/negative value when an overflow occurs.

Words: 1

Cycles: 1

Example: ROVM

Note: See SOVM.

Assembler Syntax:

Direct Addressing: [] SACH <dma>[, <shift>]
 Indirect Addressing: [] SACH { * | * + | * - }[, <shift>[, <ARP>]]

Operands: $0 \leq dma \leq 127$
 shift = 0, 1, or 4
 ARP = 0 or 1

Operation: (ACC) x 2^{-(16-shift)} → dma

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	0	1	1	SHIFT	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	-------	---	---------------------

Indirect:	0	1	0	1	1	SHIFT	1	SEE SECTION 3.3
-----------	---	---	---	---	---	-------	---	-----------------

Description: Store the upper half of the accumulator in data memory with shift. The shift can only be 0, 1, or 4.

Words: 1

Cycles: 1

Example: SACH DAT70,1

or

SACH *,1 If current auxiliary register contains the value 70.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
ACC	>0 4 2 0 8 0 0 1	ACC	>0 4 2 0 8 0 0 1
DATA MEMORY 70	6	DATA MEMORY 70	>0 8 4 1

Notes: The SACH instruction copies the entire accumulator into a shifter. It then shifts this entire 32-bit number 0, 1, or 4 bits and copies the upper 16 bits of the shifted product into data memory. The accumulator itself remains unaffected.

For example, the following instruction sequence will store >8F35 in data memory location DAT1. Location DAT2 contains the number >A8F3. DAT3 contains >5000.

ZALH	DAT2	ACC =	>A8F30000
ADDS	DAT3	ACC =	>A8F35000
SACH	DAT1,4	DAT1 =	>8F35
		ACC =	>A8F35000

Assembler Syntax:

Direct Addressing: [`<label>`] SACL `<dma>` [, `<shift>`]
 Indirect Addressing: [`<label>`] SACL `{*|*+|*-}` [, `<shift>` [, `<ARP>`]]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1
 Shift = 0

Operation: (ACC bits 15 through 0) → dma

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0 1 0 1 0					0 0 0			0	DATA MEMORY ADDRESS						
Indirect:	0 1 0 1 0					0 0 0			1	SEE SECTION 3.3						

Description: Store the low-order bits of the accumulator in data memory.

Words: 1
Cycles: 1

Example: SACL DAT71
 or
 SACL * If current auxiliary register contains the value 71.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
ACC	>0 4 2 0 8 0 0 1	ACC	>0 4 2 0 8 0 0 1
DATA MEMORY 71	7	DATA MEMORY 71	>8 0 0 1

Note: There is no shift associated with this instruction, although a shift code of zero MUST be specified if the ARP is to be changed.

Assembler Syntax:

Direct Addressing: [`<label>`] SAR `<AR>`,`<dma>`
 Indirect Addressing: [`<label>`] SAR `<AR>`,`{*|*+|*-}`[`<ARP>`]

Operands: $0 \leq dma \leq 127$
 AR=0 or 1
 ARP=0 or 1

Operation: (AR) → dma

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	0	1	1	0	AUXILIARY REGISTER	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	--------------------	---	---------------------

Indirect:	0	0	1	1	0	AUXILIARY REGISTER	1	SEE SECTION 3.3
-----------	---	---	---	---	---	--------------------	---	-----------------

Description: The contents of the designated auxiliary register are stored in the specified data memory location.

Words: 1

Cycles: 1

Example: SAR AR0,DAT101

	BEFORE INSTRUCTION		AFTER INSTRUCTION		
AR0	<table border="1"><tr><td>37</td></tr></table>	37	AR0	<table border="1"><tr><td>37</td></tr></table>	37
37					
37					
DATA MEMORY 101	<table border="1"><tr><td>18</td></tr></table>	18	DATA MEMORY 101	<table border="1"><tr><td>37</td></tr></table>	37
18					
37					
also,	LARP AR0				
	SAR AR0,*+				
AR0	<table border="1"><tr><td>5</td></tr></table>	5	AR0	<table border="1"><tr><td>6</td></tr></table>	6
5					
6					
DATA MEMORY 5	<table border="1"><tr><td>0</td></tr></table>	0	DATA MEMORY 5	<table border="1"><tr><td>6</td></tr></table>	6
0					
6					

WARNING

Special problems arise when SAR is used to store the current auxiliary register with indirect addressing if autoincrement/decrement is used.

(continued)

LARP ARO
LARK ARO,10
SAR ARO,* + or SAR ARO,*-

In this case, SAR ARO, * + will cause the value 11 to be stored in location 10. SAR ARO, * - will cause the value 9 to be stored in location 10.

Note: For more information, see LAR.

Assembler Syntax: [`<label>`] SOVM

Operands: None

Operation: 1 → OVM

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	1	0	1	1

Description: When placed in the overflow mode, the TMS32010 will set the accumulator and ALU to their highest positive/negative value if an overflow/underflow occurs. The highest positive value is `>7FFFFFFF`, and the lowest negative value is `>80000000`.

Words: 1

Cycles: 1

Example: SOVM

SPAC

Subtract P Register from Accumulator

SPAC

Assembler Syntax: [`<label>`] SPAC

Operands: None

Operation: $(ACC) - (P) \rightarrow ACC$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The contents of the P register are subtracted from the contents of the accumulator, and the result is stored in the accumulator.

Words: 1

Cycles: 1

Example: SPAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION		
P	<table border="1"><tr><td>36</td></tr></table>	36	<table border="1"><tr><td>36</td></tr></table>	36
36				
36				
ACC	<table border="1"><tr><td>60</td></tr></table>	60	<table border="1"><tr><td>24</td></tr></table>	24
60				
24				

Assembler Syntax:

Direct Addressing: [<label>] SST <dma>
 Indirect Addressing: [<label>] SST { * | * + | * - } [, <ARP>]

Operands: $0 \leq \text{dma} \leq 15$
 ARP = 0 or 1

Operation: status bits → specified data memory word on page 1

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	1	1	1	0	0	0	DATA MEMORY ADDRESS					
---------	---	---	---	---	---	---	---	---	---	---------------------	--	--	--	--	--

Indirect:	0	1	1	1	1	1	0	0	1	SEE SECTION 3.3					
-----------	---	---	---	---	---	---	---	---	---	-----------------	--	--	--	--	--

Description: The status bits are saved into the specified data memory address on page 1.

Words: 1

Cycles: 1

Example: SST DAT1
 or
 SST *,1 IF current auxiliary register contains the value 1.

Note: This instruction is used to load the TMS32010's status bits after interrupts and subroutine calls. These status bits include the Overflow Flag (OV) bit, Overflow Mode (OVM) bit, Interrupt Mask (INTM) bit, Auxiliary Register Pointer (ARP) bit, and the Data Memory Page Pointer (DP) bit. These bits are stored (by the SST instruction) in the data memory word as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OV	OVM	INTM	1	1	1	1	ARP	1	1	1	1	1	1	1	DP

Note: See LST.

SUB

Subtract from Accumulator with Shift

SUB

Assembler Syntax:

Direct Addressing: [<label>] SUB <dma> [, <shift>]

Indirect Addressing: [<label>] SUB { * | * + | * - } [, <shift> [, <ARP>]]

Operands: $0 \leq \text{shift} \leq 15$
 $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: (ACC) - [(dma) × 2^{shift}] → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	0	0	1	SHIFT	0	DATA MEMORY ADDRESS
---------	---	---	---	---	-------	---	---------------------

Indirect:	0	0	0	1	SHIFT	1	SEE SECTION 3.3
-----------	---	---	---	---	-------	---	-----------------

Description: Contents of data memory address are left-shifted and subtracted from the accumulator. During shifting, the low-order bits of data are zero-filled and the high-order bit is sign-extended. The result is stored in the accumulator.

Words: 1

Cycles: 1

Example: SUB DAT59
 or
 SUB * If current auxiliary register contains the value 59.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
ACC	36	19
DATA MEMORY 59	17	17

Assembler Syntax:

Direct Addressing: [<label>] SUBC <dma>
 Indirect Addressing: [<label>] SUBC { * | * + | * - }, <ARP>]

Operands: $0 \leq \text{dma} \leq 127$,
 ARP = 0 or 1

Operation: (ACC) – [(dma) × 2¹⁵] → adder output

 If (high-order bits of adder output) ≥ 0
 Then (adder output) * 2 + 1 → ACC
 Else (ACC) × 2 → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	0	1	0	0	0	0	DATA MEMORY ADDRESS				
---------	---	---	---	---	---	---	---	---	---	---	---------------------	--	--	--	--

Indirect:	0	1	1	0	0	1	0	0	1	SEE SECTION 3.3					
-----------	---	---	---	---	---	---	---	---	---	-----------------	--	--	--	--	--

Description: This instruction performs conditional subtraction which can be used for division in algorithms.

Words: 1

Cycles: 1

Note: The next instruction after SUBC cannot use the accumulator.

SUBH

Subtract from High-Order Accumulator

SUBH

Assembler Syntax:

Direct Addressing: [<label>] SUBH <dma>
 Indirect Addressing: [<label>] SUBH { * | * + | * - } [, <ARP>]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: (ACC) – [(dma) × 2¹⁶] → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	0	0	1	0	0	0	DATA MEMORY ADDRESS				
---------	---	---	---	---	---	---	---	---	---	---	---------------------	--	--	--	--

Indirect:	0	1	1	0	0	0	1	0	1	SEE SECTION 3.3					
-----------	---	---	---	---	---	---	---	---	---	-----------------	--	--	--	--	--

Description: Subtract the contents of specified data memory location from the upper half of the accumulator. The result is stored in the accumulator.

Words: 1
Cycles: 1

Example: SUBH DAT33
 or
 SUBH * If current auxiliary register contains the value 33.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 33	5	5
	31 16 15 0	31 16 15 0
ACC	17 0	12 0

Note: The SUBH instruction can be used for performing 32-bit arithmetic.

SUBS

Subtract from Low Accumulator
with Sign-Extension Suppressed

SUBS

Assembler Syntax:

Direct Addressing: [<label>] SUBS <dma>
 Indirect Addressing: [<label>] SUBS { * | * + | * - } [, <ARP>]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: (ACC) – (dma) → ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	0	0	1	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

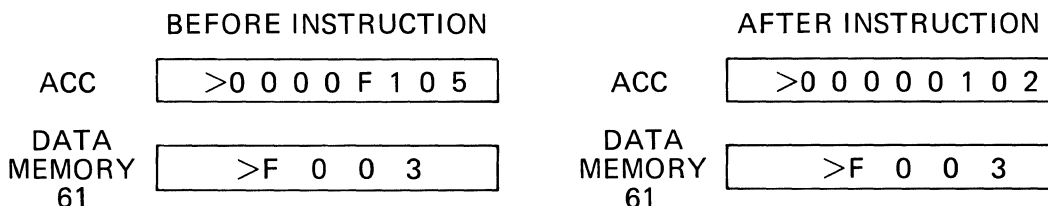
Indirect:	0	1	1	0	0	0	1	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: Subtract contents of a specified data memory location from accumulator with sign-extension suppressed. The data is treated as a 16-bit positive integer rather than a two's complement integer.

Words: 1

Cycles: 1

Example: SUBS DAT61
 or
 SUBS * If current auxiliary register contains the value 61.



Assembler Syntax:

Direct Addressing: [**<label>**] TBLR **<dma>**
 Indirect Addressing: [**<label>**] TBLR **{*|*+|*-}**[**,<ARP>**]

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: (PC) + 1 → TOS
 (ACC) → PC → address lines A11 through A0
 data bus D15 through D0 → dma
 (TOS) → PC

Encoding:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0 1 1 0 0 1 1 1 0										DATA MEMORY ADDRESS					
Indirect:	0 1 1 0 0 1 1 1 1										SEE SECTION 3.3					

Description: This instruction transfers a word from anywhere in program memory (i.e., internal ROM, external ROM, external RAM) to the specified location in data memory. The three-cycle instruction is as follows:

- Prefetch: $\overline{\text{MEN}}$ goes low and the TBLR instruction opcode is fetched. The previous instruction is executing.
- Cycle 1: $\overline{\text{MEN}}$ goes low. The address of the next instruction is placed onto address bus, but data bus is not read. Program counter is pushed onto stack. Twelve LSBs of the accumulator contents are loaded into the program counter.
- Cycle 2: $\overline{\text{MEN}}$ goes low. Contents of program counter are buffered to address lines. Address memory location is read and is copied into specified RAM location. The new program counter is popped from the stack.
- Cycle 3: $\overline{\text{MEN}}$ goes low. Next instruction opcode is prefetched.

Words: 1
Cycles: 3

Example: TBLR DAT4
 TBLR * If current auxiliary register contains the value 4.

(Continued)

	BEFORE INSTRUCTION	AFTER INSTRUCTION
ACC	17	17
PROGRAM MEMORY 17	306	306
DATA MEMORY 4	75	306

Note: This instruction is useful for reading coefficients that have been stored in program ROM, or time dependent data stored in RAM.

Assembler Syntax:

Direct Addressing: [<label>] TBLW <dma>
 Indirect Addressing: [<label>] TBLW { * | * + | * - } [, <ARP>]

Operands: $0 \leq \text{dma} \leq 127$
 ARP = 0 or 1

Operation: (PC) + 1 → TOS
 (ACC) → PC → address lines A11 through A0
 (dma) → data bus D15 through D0
 (TOS) → PC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	1	1	1	0	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	------------------------

Indirect:	0	1	1	1	1	1	0	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: This instruction transfers a word from the specified location in data memory to a location in external program RAM. The three-cycle instruction is as follows:

- Prefetch:** $\overline{\text{MEN}}$ goes low and the TBLR instruction opcode is fetched. The previous instruction is executing.
- Cycle 1:** $\overline{\text{MEN}}$ goes low. The address of the next instruction is placed onto address bus, but data bus is not read. Program counter is pushed onto stack. Twelve LSBs of the accumulator contents are loaded into the program counter.
- Cycle 2:** $\overline{\text{WE}}$ goes low. Contents of program counter are buffered to address lines. Contents of specified data memory address are placed on the data bus. The new program counter is popped off of stack.
- Cycle 3:** $\overline{\text{MEN}}$ goes low. Next instruction opcode is prefetched.

Words: 1
Cycles: 3

Example: TBLW DAT4
 TBLW * If current auxiliary register contains the value 4.

(Continued)

	BEFORE INSTRUCTION		AFTER INSTRUCTION
ACC	17	ACC	17
DATA MEMORY 4	75	DATA MEMORY 4	75
PROGRAM MEMORY 17	306	PROGRAM MEMORY 17	75

Note: The TBLW and OUT instructions use the same external signals and thus cannot be distinguished when writing to program memory addresses 0 through 7.

XOR

Exclusive-OR with Low-Order Bits of Accumulator

XOR

Assembler Syntax:

Direct Addressing: [<label>] XOR <dma>
 Indirect Addressing: [<label>] XOR { * | * + | * - } [, <ARP>]

Operands: $0 \leq dma \leq 127$
 ARP=0 or 1

Operation: Zero. XOR. high-order ACC bits: (dma). XOR. low-order ACC bits→ACC

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:

0	1	1	1	1	0	0	0	0	0	DATA MEMORY ADDRESS					
---	---	---	---	---	---	---	---	---	---	---------------------	--	--	--	--	--

Indirect:

0	1	1	1	1	0	0	0	0	1	SEE SECTION 3.3					
---	---	---	---	---	---	---	---	---	---	-----------------	--	--	--	--	--

Description: The low-order bits of the accumulator are exclusive-ORed with the specified data memory address and concatenated with the exclusive-OR of all zeroes and the high-order bits of the accumulator. The exclusive-OR operation follows the truth table below:

DATA MEMORY BIT	ACC BIT (BEFORE)	ACC BIT (AFTER)
0	0	0
0	1	1
1	0	1
1	1	0

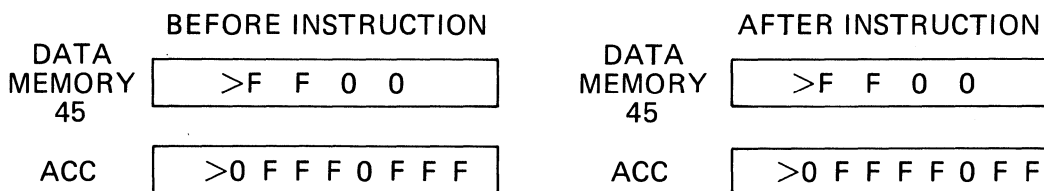
Words: 1

Cycles: 1

Example: XOR DAT45

or

XOR * If current auxiliary register contains the value 45.



Note: This instruction is useful for toggling or setting bits of a word for high-speed control. Also, the one's complement of a word can be found by exclusive-ORing it with all ones.

ZAC

Zero the Accumulator

ZAC

Assembler Syntax: [`<label>`] ZAC

Operands: None

Operation: $0 \rightarrow \text{ACC}$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Description: The accumulator is cleared (zeroed).

Words: 1

Cycles: 1

Example: ZAC

	BEFORE INSTRUCTION	AFTER INSTRUCTION																
ACC	<table border="1"><tr><td>A</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td></tr></table>	A	F	F	F	F	F	F	F	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0
A	F	F	F	F	F	F	F											
0	0	0	0	0	0	0	0											

ZALH

Zero Accumulator and Load High

ZALH

Assembler Syntax:

Direct Addressing: [`<label>`] ZALH `<dma>`
 Indirect Addressing: [`<label>`] ZALH `{*|*+|*-}`[`<ARP>`]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: $(dma) \times 2^{16} \rightarrow ACC$

Encoding: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Direct:	0	1	1	0	0	1	0	1	0	DATA MEMORY ADDRESS
---------	---	---	---	---	---	---	---	---	---	---------------------

Indirect:	0	1	1	0	0	1	0	1	1	SEE SECTION 3.3
-----------	---	---	---	---	---	---	---	---	---	-----------------

Description: ZALH clears the accumulator and loads the contents of the specified data memory location into the upper half of the accumulator. The lower half of the accumulator remains clear.

Words: 1

Cycles: 1

Example: ZALH DAT29
 or
 ZALH * If current auxiliary register contains the value 29.

	BEFORE INSTRUCTION	AFTER INSTRUCTION
DATA MEMORY 29	>3 F 0 0	>3 F 0 0
ACC	>0 0 7 7 F F F F	>3 F 0 0 0 0 0 0

Note: ZALH can be used for implementing 32-bit arithmetic.

Assembler Syntax:

Direct Addressing: [<label>] ZALS <dma>
 Indirect Addressing: [<label>] ZALS { * | * + | * - } [, <ARP>]

Operands: $0 \leq dma \leq 127$
 ARP = 0 or 1

Operation: (dma) → ACC

Encoding:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	1	0	0	DATA MEMORY ADDRESS						
Indirect:	0	1	1	0	0	1	1	0	1	SEE SECTION 3.3						

Description: Clear accumulator and load contents of specified data memory location into lower half of the accumulator. The data is treated as a 16-bit positive integer rather than a two's complement integer. Therefore, there is no sign-extension as with the LAC instruction.

Words: 1

Cycles: 1

Example: ZALS DAT22

 or
 ZALS * If current auxiliary register contains the value 22.

	BEFORE INSTRUCTION		AFTER INSTRUCTION
DATA MEMORY 22	>F 7 F F	DATA MEMORY 22	>F 7 F F
ACC	>7 F F 0 0 0 3 3	ACC	>0 0 0 0 F 7 F F

Notes: The following routine reveals the difference between the ZALS and the LAC instruction. Data memory location 1 contains the number > FA37.

ZALS	DAT1	(ACC) = > 0000FA37
ZAC		Zero ACC
LAC	DAT1	(ACC) = > FFFFA37

ZALS is useful for 32-bit arithmetic operations.

4. SAMPLE ROUTINES

4.1 INTRODUCTION

This section provides information that supplements the data presented in Section 3. The programming examples are provided to further illustrate the usage of some of the instructions.

4.2 INITIALIZING THE TMS32010

*Flags: ARP, DPP, OV

*Mode : OV INT

*

	AORG	0	
	B	INIT	Branch vector for RESET
	B	INTR	Branch vector for INTERRUPT
INIT	EINT		Initialize interrupt mode
	ROVM		Initialize overflow mode
	BV	CLROV	Clear overflow flag
CLROV	LARP	0	Initialize auxiliary register pointer
	LDPK	0	Initialize data page pointer

*

*INITIALIZATION COMPLETE

.
. .
. .

4.3 BIOZ INSTRUCTION

The TMS32010 supports an external hardware input. Testing the level of the $\overline{\text{BIO}}$ pin may be performed with the BIOZ instruction, demonstrated by the following example in which the TMS32010 fills a FIFO of unknown depth from data memory. The FIFO-full signal (full active high) is connected to the $\overline{\text{BIO}}$ pin.

```

      .
      .
      .
LDFIFO LARP 0           Select ARO
      LARK 0,0        Initialize ARO to zero
      OUT  *+,PA3     Output data to Port 3, increment ARO
      BIOZ LDFIFO     If BIO is zero, then BRANCH
      .
      .
      .
```

4.4 BANZ INSTRUCTION

This decrement and branch if not zero instruction is extremely useful in implementing loop counters.

Example 1: Shift data memory in page zero to the next higher address, $(A) \rightarrow (A + 1)$, for all addresses less than or equal to the address in ARO.

	LARK	0,9	Load ARO with 9
	LDPK	0	Initialize DP to page 0
	LARP	0	Set pointer to ARO
NXTMOV	DMOV	*	Move data pointed to by ARO
*	BANZ	NXTMOV	If ARO < or > zero, then BRANCH to NEXT MOVE. ARO is decremented.
	.		
	.		
	.		

Example 2: Input 10 data values from Port 5. Auxiliary register 0 is used as a data input counter, and auxiliary register 1 is used as the pointer to the data table. Note that AR1 is used in the auto-increment mode (*+), and ARO is decremented by the execution of BANZ.

	.		
	.		
	.		
	LARK	0,9	Load counter
	LARK	1,14	Load starting address
DATAIN	LARP	1	
*	IN	*+,PA5,ARO	Input data value from Port 5 and store in data location addressed by AR1
*			
	LARP	0	
	BANZ	DATAIN	If ARO < or > 0, then BRANCH

4.5 LTD INSTRUCTION

The Load T Register and Shift Data instruction implements three key operations in parallel. During the execution of this instruction, the P Register is added to the accumulator, the T Register is loaded with the data from the operand, and the data value is shifted to the next address.

The following example illustrates the use of this powerful instruction in implementing multipole filters. This routine assumes an I/O device on Port 1 which inputs a sample. The TMS32010 then computes a number (Y), based on this and previous samples. Y is output to the I/O device on Port 2. The following symbols are used:

Y = output to I/O device on Port 2
X1 = current sample taken at time "t"
X2 = sample taken at t-1
X3 = sample taken at t-2
X4 = sample taken at t-3

NOTE

X4 is at a higher address than X3; X3 is at a higher address than X2, etc.
The formula is: $Y = 3(X1) + 4(X2) + 5(X3) + 6(X4)$. Data is shifted down one interval by LTD.

Example:

	DSEG	0	
X1	BSS	1	
X2	BSS	1	
X3	BSS	1	
X4	BSS	1	
Y	BSS	1	
	PSEG		
START	IN	X1,PA1	Input current data sample
	LT	X4	
	ZAC		
	MPYK	6	Multiply 6 X (X4)
	LTD	X3	Move (X3) -->T and move (X3) --> (X4)
	MPYK	5	Multiply 5 X (X3)
	LTD	X2	Move (X2)--> T and move (X2) --> X3
	MPYK	4	Multiply 4 X (X2)
	LTD	X1	Move (X1) --> t and move (X1) --> X2
	MPYK	3	Multiply 3 X (X1)
	APAC		
	SACL	Y	
	OUT	Y,PA2	Output the results
	B	START	

4.6 SUBC INSTRUCTION

Division is the inverse of the multiplication process, i.e., multiplication consists of a series of add and shift operations while division consists of a series of subtract and shift operations. The TMS32010 does not have a hardware divider; however, SUBC provides an efficient means of implementing division. The only restriction for the use of this instruction is that both operands must be positive.

It is also important that the programmer understand the characteristics of the operands, whether or not the quotient can be represented as a fraction, and the degree of accuracy to which the quotient is to be computed.

FRACTIONAL DIVISION: In fractional division, the denominator is divided into the numerator: The absolute value of the numerator must be less than the absolute value of the denominator. There is no restriction on the sign of the two operands.

$$\frac{1}{10} \quad \begin{array}{l} \text{(numerator/dividend)} \\ \text{(denominator/divisor)} \end{array} \quad (1 \text{ divided by } 10 = 1/10) \quad \leftarrow \text{(quotient)}$$

The following routine can be used to divide two numbers:

```

*
DN1   LARP  0
      LT   NUMERA           Get sign of quotient
      MPY  DENOM
      PAC
      SACH TEMSGN           Save sign of quotient
      LAC  DENOM
      ABS
      SACL DENOM           Make denominator positive
      ZALH NUMERA          Align numerator
      ABS
      LARK 0,14
* (If divisor and dividend are aligned, division can start here.)
KPDVNG SUBC DENOM          15-cycle divide loop
      BANZ KPDVNG
*
      SACL QUOT
      LAC  TEMSGN
      BGEZ DONE           Done if sign is positive
*
      ZAC
      SUB  QUOT
      SACL QUOT           Negate quotient if negative
*
DONE  RET

```

The quotient is stored in the accumulator/low, the remainder in the accumulator/high.

4.7 CALA INSTRUCTION

This routine reads a 16-bit data value from an I/O device and extracts bits 0 and 1. It then calls a subroutine indirectly. Subroutines are set up at the following program locations:

```
ROUT0 at >A0
ROUT1 at >A6
ROUT2 at >AC
ROUT3 at >B2
```

The subroutine branched to depends on the value of the bits read.

```
*      LACK >A0
      SACL TEMP           Initialize TEMP with entry address
                          of ROUT0
      LACK >03
      SACL MASK           Set up mask register
      IN  VALUE,PA3
      LAC  VALUE
      AND  MASK
      SACL RESULT         Save data temporarily in RESULT
      ZALS TEMP
      LT  RESULT
      MPYK 6
      APAC
      CALA
```


4.8 32-BIT ARITHMETIC CAPABILITIES OF TMS32010

ADDH, ADDS, SUBH, SUBS, ZALH, ZALS instructions can be used to implement 32-bit arithmetic.

Example: 32-bit subtraction: Assume two 32-bit numbers are stored in four locations of data RAM. B1 and B2 are respectively the MSB and LSB parts of the number which is subtracted from numbers A1 and A2. A1 is the MSB, and A2 is the LSB part of the number.

*	ZALS	A2	Zero accumulator and load LSB into accumulator
	ADDH	A1	Add in MSB
	SUBS	B2	Subtract from accumulator low
	SUBH	B1	Subtract from accumulator high

4.9 TABLE READ FROM PROGRAM MEMORY INSTRUCTION

The TMS32010 implements separation of code and data spaces. The Table Read and Table Write from program memory instructions permit shifting constants between program memory into data memory. For example, assume the values from program memory locations 00A0, 00A1, 00A2, 00A3 are to be transferred to data memory locations 0003, 0002, 0001, 0000, respectively. Also, assume variable ONE contains value 0001.

Example:

*	LACK	>00A0	Load program memory address into accumulator
	LARP	0	ARP points to ARO
	LARK	0,03	Set up ARO as counter
LOOK	TBLR	*	Transfer value in program memory to data memory
*	ADD	ONE	Increment address (accumulator)
	BANZ	LOOK	If ARO < or > zero, decrement and branch
*			

4.10 INTERRUPT INSTRUCTION

The following example illustrates how to implement interrupt processing. Further, it demonstrates how to save and restore the original user context during entry into and exit from the interrupt program sequence.

```

STATUS  DSEG
        BSS  1
ACH     BSS  1
ACL     BSS  1
AR00    BSS  1
AR01    BSS  1
        DEND
        AORG 0
        B    INIT      Branch vector for RESET
        B    INTR     Branch vector for INTERRUPT
*
INTR    SST  STATUS    Store status on page 1
        LDPK 1        Set data page = 1
        SACH ACH      Store accumulator
        SACL ACL
        SAR  0,AR00   Store auxiliary registers
        SAR  1,AR01
        .
        .
        LDPK 1        Set data page = 1 if it was modified
        LAR  0,AR00   Load auxiliary registers with old
*                               value
        LAR  1,AR01
        ZALH ACH      Load accumulator with old value
        ADDS ACL
        LST  STATUS   Restore status
        EINT          Enable interrupts
        RET           Return to PROGRAM

```

NOTE

When an interrupt occurs, the interrupt mode is set. After the status is restored, the subroutine should enable interrupts before returning to the main program.

4.11 STACK EXPANSION

The TMS32010 allows up to four levels of subroutine call/returns when interrupts are desired and Table Read/Write is not used. This may seem to be a limitation to the implementation of large algorithms or programs, but the following routines demonstrate how the level of nesting of subroutines can be expanded. Special CALS (call subroutine) and associated RETS (return from subroutine) are listed. These utilities save the user context as well as allowing parameter passing via the accumulator. The logical stack is implemented in data memory. These utilities will assume four data locations in DP1 (ACL,ACH,ADDR,ST). AR0 = STACK POINTER (TOS + 1).

MAIN

```

.
.
.
SACL X
CALL CALS
DATA POLY
SACL QUO
.
.
.

```

POLYNOMIAL EVALUATION SUBROUTINE

```

POLY  LT    A          *Y = (A(X)**2)-(BX)+C
      MPY   X          A*X
      PAC
      SACL  POLYT
      ZAC
      LT    POLYT
      MPY   X          AX(squared)
      LTA   B
      MPY   X
      SPAC          AX(squared)-BX
      ADDS  C          AX(squared)-BX+C
      CALL  RETS

```

CALL SUBROUTINE WITH LOGICAL STACK IMPLEMENTATION IN DATA MEMORY

```

CALS  SST    ST          Save context
      SACL  ACL
      SACH  ACH          Save accumulator
      LDPK  1          Change context
      LARP  0
      POP
      SACL  *+          Move return address into accumulator
                        Put accumulator onto TOS in data
*
      TBLR  ADDR        Fetch parameter (add of routine)
      LAC   ADDR        Put parameter into accumulator
      PUSH          Put parameter onto hardware stack
      ZALS  ACL          Restore context
      ADDH  ACH
      LST   ST
      RET          Go to called routine

```

ASSOCIATE RETURN FROM SUBROUTINE WHEN CALS IS USED

RETS	SST	ST	Save context
	LDPK	1	Change context
	LARP	0	
	SACL	ACL	Save accumulator
	SACH	ACH	
	POP		Remove last address from hardware stack
*	MAR	*-	Decrement stack pointer
	LACK	1	Load accumulator with 1
	ADD	*	Add accumulator to return address(*)
	PUSH		Push onto hardware stack
	ZALS	ACL	Restore context
	ADDH	ACH	
	LST	ST	
	RET		Return from routine

5. ASSEMBLER DIRECTIVES

5.1 INTRODUCTION

The TMS32010 Assembly Language is processed by the assembler executing in a host computer. This section describes the assembler and its directives.

5.2 THE TMS32010 ASSEMBLER

The TMS32010 Assembler generates object code for the TMS32010 microcomputer. The assembler processes source code twice. On the first pass, the assembler maintains the location counter (which defines the program memory addresses assigned to the resulting words of object code), 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 assembler output
- Directives that initialize constants
- Directives that provide linkage between programs
- Miscellaneous directives.

These directive types are discussed in detail in the following paragraphs.

5.3.1 Directives that Affect the Location Counter

As the assembler reads the source statements of a program, a component of the assembler called the location counter sets the memory locations to the resulting object code. The first nine assembler directives listed below initialize the location counter and define the value as relocatable, absolute, or dummy. The last two directives set the location counter to provide a block or an area of program memory for the object code. The directives are listed in Table 5-1, and they are discussed in detail in alphabetical order on the following pages.

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

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

Description:

AORG places a value in the location counter and defines the succeeding locations as absolute. (An absolute location is not affected by relocation.) Use of the label field is optional. When a label is used, it is assigned the value that the AORG directive places in the location counter (the command field contains AORG). The operand field is optional, but when used, it must contain a well-defined expression (wd-exp). The comment field is optional and may be used only when the operand field is also used. Upon encountering an AORG statement, the assembler places the value of the well-defined expression into the location counter. 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 in the location counter.

Syntax:

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

Example:

```
AORG >1000+X
```

Symbol X must be absolute and 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.

Description:

BES advances the location counter by the value in the operand field.

Syntax:

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

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 words to be added to the location counter. The comment field is optional.

Example:

```
    BUFF2 BES > 10
```

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

Description:

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 word in the block. The operation field contains BSS. The operand field contains a well-defined expression that represents the number of words to be added to the location counter. The comment field is optional.

Syntax:

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

Example:

```
BUFF1 BSS >10
```

This directive reserves a 16-word buffer at location BUFF1. Had the location counter contained > 100 when the assembler processed this directive, BUFF1 would be assigned > 110.

Description:

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; or
- 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, and a warning message is issued. As is the case for DEND and PEND, CEND is invalid when used in absolute code.

Example:

See CSEG directive.

Description:

CSEG places a value in the location counter and defines succeeding locations as common-relocatable (i.e., relocatable with respect to a common segment).

Syntax:

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

Use of the label field is optional. When used, a label is assigned the value placed by the directive in the location counter. The command 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 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 reset to the maximum value attained during the previous assembly of any portion of that particular common segment.

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. 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 addresses 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.

Example:

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

```

COM1A CSEG      'ONE'
*   COMMON RELOCATABLE SECTION, NAMED 'ONE'
    .
    .
    .
    .
    CEND
*
COM2A CSEG      'TWO'
    .
    .
    .
*   COMMON-RELOCATABLE SECTION, NAMED 'TWO'
    .
    .
    .
COM2B CEND
COM1C CSEG      '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 counter; 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 words of this segment. The symbol COM2A is the symbolic address of the first word of common segment 'TWO'; COM2B is the common-relocatable (type 'TWO') word 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 words of common segment 'TWO'.

Description:

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.

Syntax:

```
[<label>]      DEND      [<comment>]
```

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; or
- 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.

Description:

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.

Syntax:

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

The label is assigned the value that the directive places in the location counter. 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.

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 1:

```
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 below 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 2:

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

The above 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.

Example 3:

```
CSEG 'COM1'
DORG $ "$" HAS A COMMON-RELOCATABLE VALUE
.
.
.
.
LAB 1 DATA $
MASK DATA >F000
.
.
.
.
CEND
```

In this 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.

Description:

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. Either of the following values are placed in the location counter:

- The maximum value the location counter can attain as the result of assembling any block of data-relocatable code; or
- Zero, if no data-relocatable code has been previously assembled.

Syntax:

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

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 word address of the location following the code block. The value of the symbol LRAM is the length in words of the block.

Description:

The PEND directive is provided as the program-segment counterpart to the PENDING and CENDING 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.

Syntax:

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

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 DENDING or CENDING. Like DENDING and CENDING, it is invalid when used in absolute code.

Description:

PSEG places a value in the location counter and defines succeeding locations as program-relocatable.

Syntax:

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

When used, a label is assigned the value that the directive places in the location counter. The command field contains PSEG. The operand field is not used, 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; or
- Zero, if no program-relocatable code had been previously assembled.

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.

Example:

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

Description:

RORG places a value in the location counter and 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. (Symbols are defined by the EQU directive; see Section 5.3.3.) 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, previous data segments, and specific common segments of a program replace the value of the location counter. The directive initializes the location counter to the value following the previous relocatable code of the program, or to zero if no 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. For example, if a segment begins at >0 and ends at >E, then the length will be >F. The next available address will be >F.

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 1:

```
RORG $ - 10 OVERLAY TEN WORDS
```

The \$ symbol refers to the present location counter value. This has the effect of backing up the location counter by ten words. The instructions and directives following the ROR 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.

Example 2:

```
SEG2 RORG
```

The location counter contents depend upon preceding source statements. Assume that after defining data for a program that occupied >44 words, 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.

5.3.2 Directives that Affect Assembler Output

This category contains the directive which supplies a program identifier in the object code and five directives that format the source listing. Table 5-2 lists those directives. The following paragraphs discuss the directives in detail in alphabetical order.

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

Description:

IDT assigns a name to the object module produced.

Syntax:

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

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 warning message and retains the first eight characters as the program name. The comment field is optional.

Example:

```
0001          IDT      'EXAMPLE'  
0002      0001      ONE      EQU 1  
0003      0002      TWO      EQU 2  
.  
.  
.
```

The above example directive assigns the name EXAMPLE to the module being assembled. The module name is then 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 to determine the entry point for the module. The entry point must also appear as a symbol in a REF directive (see Section 5.3.4).

NOTE

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

Description:

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.

Syntax:

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

Use of the label field is optional. When used, the 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.

Example:

```
LIST
```


Description:

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.

Syntax:

```
[<label>]    OPTION    <option-list>    [<comment>]
```

Use of the label field is optional. When used, the label assumes the current value of the location counter. The available <option-list> features are:

- DUNLST: Limit the listing of DATA directives to one line
- FUNLST: Turn off DUNLST, TUNLST options
- NOLIST: Inhibit all listing output. (This overrides the LIST directive)
- SYMLST: Produce a symbol table list in the object file
- TUNLST: Limit the listing of TEXT directives to one line
- XREF: Produce a symbol cross-reference listing

Example:

```
OPTION XREF
```

Description:

This directive 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.

Syntax:

```
[<label>]    PAGE    [<comment>]
```

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.

Example:

```
PAGE
```

This example 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. Using the PAGE directive to divide the source listing into logical divisions improves program documentation.

Description:

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 page, a TITL directive must be the first source statement submitted to the assembler. Unlike the IDT directive, the TITL directive is not printed in the source listing. The assembler does not print the comment because the TITL directive is not printed, but the line counter does increment.

Syntax:

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

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 syntax error message. The comment field is optional.

Example:

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

This above example 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.

Description:

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 expression.

Syntax:

```
[<label>]      UNL      [<comment>]
```

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.

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

5.3.3 Directives that Initialize Constants

This category consists of directives that assign hexadecimal values in successive words of the object code, 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

DIRECTIVES	MNEMONICS
Initialize word	DATA
Initialize text	TEXT
Define assembly-time constant	EQU

Description:

DATA places one or more values in one or more successive words in program memory.

Syntax:

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

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 twos complement number. The command field is optional.

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. The DATA directive should be used to place coefficients or other data words in program memory. During TMS32010 execution, TBLR can then be used to transfer the data words from ROM to RAM. The user may have as many operands as desired; however the total line length may not exceed 60 characters.

Description:

EQU assigns a value to a symbol.

Syntax:

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

<exp> may not contain a symbol that appears in a REF directive nor contain forward references. 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.

Example 1:

```
SUM    EQU    AR1
```

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:

Example 2:

```
TIME    EQU    HOURS
```

The above example 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 the program, the value is a relocatable value. After execution of the above directive, the two symbols may be used interchangeably. Symbols in the operand field must be previously defined. Certain symbols, such as ARO and PA0, have predefined values (see Section 2.6.1).

Description:

TEXT places one or more characters of a string of characters in successive words of program memory, two characters per word. The assembler negates the last character of the string when the string is preceded by a minus (–) sign (unary minus).

Syntax:

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

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.

Example 1:

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

In the example above, the directive places the eight-bit ASCII representations of each character in memory and fills the unused byte of the last word with a blank. This blank is considered the last character if the negate option is specified. The result is >4558, >414D, >504C, and >4520. The label MSG1 is assigned the first word's address, which contains the value >4558. Had the operand been "example", the last word would have been contained in >45E0, the negation being applied to the blank filler.

Example 2:

```
0001                                IDT      'EXAMPLE'
0002                                TEXT      'NUMBER'
                                0000      4E
                                0000      55
                                .
                                .
                                .
                                0002      52
0003                                END
NO ERRORS, NO WARNINGS
```

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. The DEF directive places one or more symbols defined in the module into the object code of the assembled module, thus making them available for linking. The REF directive places symbols used in the module but defined in another module into the object code of the assembled module, allowing them to be linked. Table 5-4 lists these directives. The following paragraphs discuss each in detail in alphabetical order. For further information, see Section 6.

TABLE 5-4 – DIRECTIVES THAT PROVIDE LINKAGE BETWEEN PROGRAMS

DIRECTIVES	MNEMONICS
External definition	DEF
External reference	REF
Secondary external reference	SREF
Force load	LOAD

Description:

DEF makes one or more symbols available to other programs. All symbols used in the DEF statement must be defined in the same module.

Syntax:

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

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 comment field is optional.

Example 1:

```
DEF ENTER,ANS
```

This example causes the assembler to include symbols ENTER and ANS in the object code; these symbols are available to other programs. For details on how the assembler places symbols in the object code for the DEF directive, see Section 7.5.1. For further information on the DEF directive, see Section 6.3.2.

Example 2:

```
0001      0000  ABC      EQU      0
0002      0001  DEF      EQU      1
0003 0000
0004      DEF      ABC,DEF
0005
NO ERRORS, NO WARNINGS
```

The object code for the above example is:

```
K0000NO$IDT  60000ABC  60001DEF  7F89AF      NO$IDT 1
```

The symbol name follows the four-digit hex numbers assigned to the symbol by the EQU directive (see Section 5.3.3). The number 6 preceding the four-digit hex number is an object code tag (see Section 7.5.1).

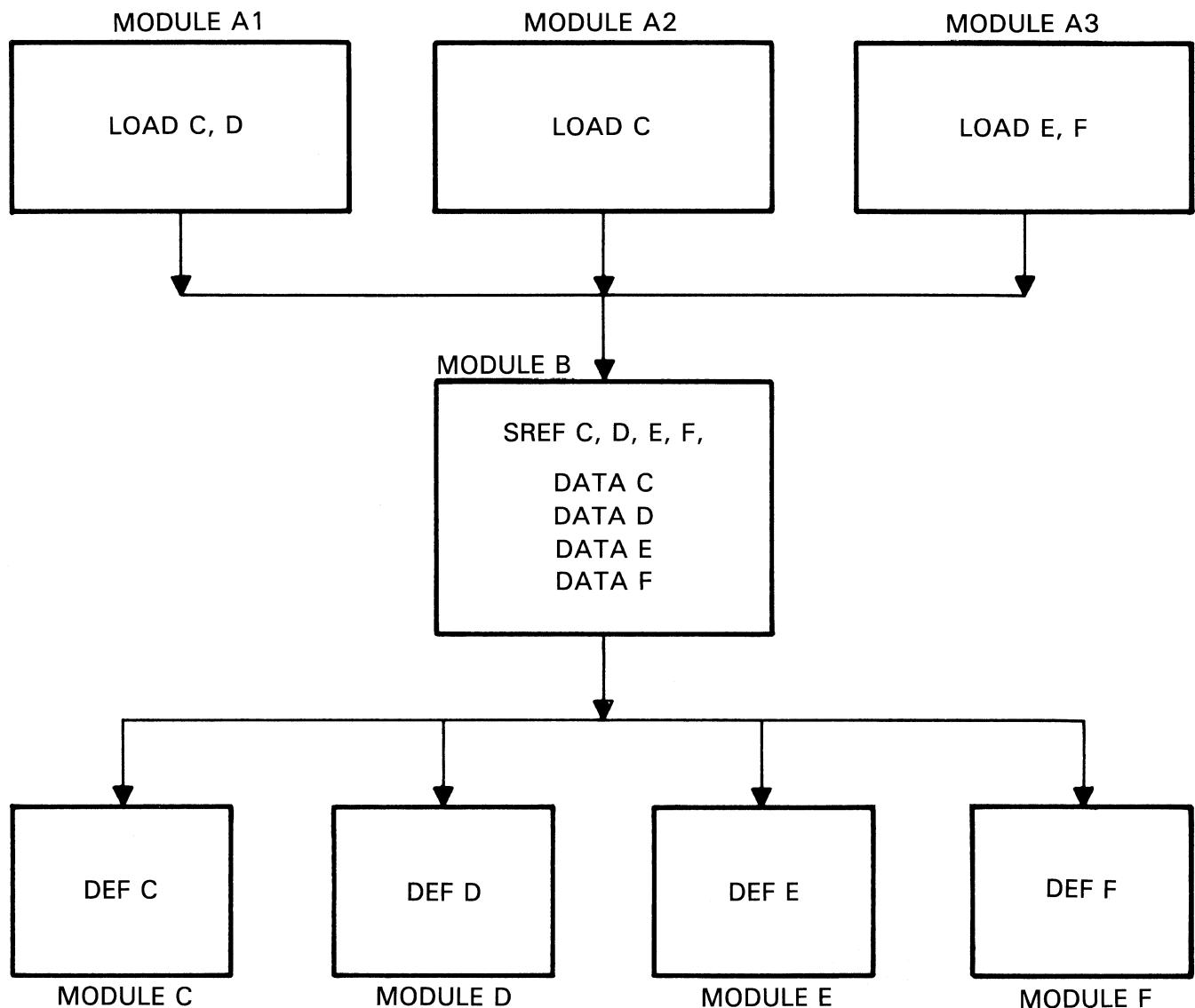
Description:

The LOAD directive is like a REF directive, except that the symbols do not need to be used in the module containing the LOAD directive. All symbols used in the LOAD directive must be defined in some other module during link editing time. LOADs are used with SREFs. If a one-to-one matching of LOAD and DEF pairs does not occur, then unresolved references will occur during link editing.

Syntax:

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

Example 1:



Module A(n) uses a branch table in module B to obtain one module C, D, E, or F. Module A1 knows which of modules C, D, E, and F it requires. Module B has an SREF for C, D, E, and F. Since SREF does not require symbols C, D, E, and F to have a corresponding symbol defined in another module, module C, D, E, and F do not have to be included in one link editing time. 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 in order to handle a highly involved case and still be linked together without unnecessary modules. A(n) only has LOAD directives for the modules it needs. This is especially useful when developing large codes which may have about a hundred modules. To test a particular function, not all modules are required. Including only the required module saves memory space and time.

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

Example 2 (using a TI 990 host computer):

```
TASK TSTLOAD
FORMAT ASCII
DATA 0
PROGRAM 0
LIBRARY <PATHNAME>.LIB
INCLUDE <PATHNAME>.OBJ.A1
INCLUDE <PATHNAME>.OBJ.B
END
```

In the above example, the <PATHNAME>.LIB is a directory that contains 990-tagged object modules for modules C, D, E, and F. In this case only modules C and D are to be linked into the LOAD object module, while modules E and F are not.

Description:

REF provides access to one or more symbols defined in other programs.

Syntax:

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

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.

Example:

```
REF      ARG1,ARG2
```

This example causes the assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs (see Section 6.3.1).

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 link editor will generate a summary list of all "unresolved references".

Description:

SREF provides access to one or more symbols defined in 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 not included in the summary list of the unresolved references.

Syntax:

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

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.

Example:

```
SREF      ARG1,ARG2
```

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

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 in alphabetical order.

TABLE 5-5 – MISCELLANEOUS DIRECTIVES

DIRECTIVES	MNEMONICS
Program end	END
Copy source file	COPY
Define MACRO library	MLIB

Description:

COPY causes the assembler to read source statements from a different file.

Syntax:

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

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 one of the following:

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

The comment field is optional.

Example:

```
COPY    .SFILE
```

The directive in the example causes the assembler to take its source statements from a file called 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 set by the host operating system.

Description:

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.

Syntax:

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

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 to the link editor the entry point of the program. The entry point is the program address at which execution of the assembled module will begin. When the operand field is not used, no entry point is placed in the object code. If the entry point symbol is specified in the link control file, it must be REFed; otherwise, the linker cannot find the entry symbol. The comment field may be used only with an operand field.

Example 1:

```

*           AORG      0           The symbol ENTRY is assigned
*           NOP                               the value
*           NOP                               1 by the assembler. Since ENTRY
*           ENTRY  NOP                    appears
*           END      ENTRY                as the operand of the END
*                                           directive,
*                                           the value of the symbol will ap-
*                                           pear as a four-digit hex character
*                                           following the object code tag
*                                           character 1, as seen in the sample
*                                           printout below:

```

(SAMPLE PRINTOUT)

```

                                Value of the symbol
                                ↓ ↓
K0000NO$IDT  90000B7F80B7F80100017F8A3F           NO$IDT 1
                                NO$IDT 11/12/82 13:47:31 ASM320 2.1 83.074           NO$IDT 2

```


END

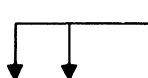
END

Example 2:

```
      AORG    >20      The symbol ENTRY is assigned
*                               the value
ENTRY  NOP           >20. As in the example above,
*                               the value
      END      ENTRY  appears in the object code
*                               following
*                               the tag character 1.
```

(SAMPLE PRINTOUT)

Value of the symbol.



```
K0000NO$IDT 90020B7F80100207F9C7F      NO$IDT 1
:          NO$IDT 10/06/82 12:42:48 ASM320 2.1 83.074      NO$IDT 2
```

Description:

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) 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 of 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)
```

NOTE

On VAX systems, the user must make certain that the filename of all files in the macro library have an extension name of `".ASM"`. For example, if the statement: `MLIB 'DRC1:[MACROS]'` has been used, the VAX version of the macro library processor would expect to find files such as `MYMACRO.ASM`, `NEWMAC.ASM`, etc., within the macro library `'MACROS'`.

When the program finds a macro call `SUBMAC` (not previously defined), the above example would cause the macro function to search first for a file named `USER32.BIGPROJ.MYTASK.MACROS.SUBMAC`, and then, if that file is not found, to search for a file named `MYVOLUME.MACDIR.CMPXMACS.NEWMACS.SUBMAC`, in that order.

6. PROGRAM LINKING

6.1 INTRODUCTION

The TMS32010 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 guide lines to assist the user in taking full advantage of these capabilities.

6.2 RELOCATION CAPABILITY

Relocatable code includes information that allows a link editor to place the code in any available area of memory, thus providing the most efficient use of available memory. Absolute code must be loaded into a specified area of memory.

Object code generated by an assembler comprises the assembled program and consists of machine language instructions, addresses, and data. 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 (the load address defines the beginning of the module). 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 (see Section 7.5.1).

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 Section 2.7.4). 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.

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. Modules common to several programs may also 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' which 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 other program modules must be placed in the operand field of a DEF directive. The example below shows a program named 'ROUTINES' which DEFs a routine named 'SUBR1'. The label 'SUBR1' must be defined in the program.

```
(FILE B)
          IDT          'ROUTINES'
          DEF          SUBR1,SUBR2
          .
          .
SUBR1     EQU          $
          .
          .
          RET
SUBR2     EQU          $
          .
          .
          RET
          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 must 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.

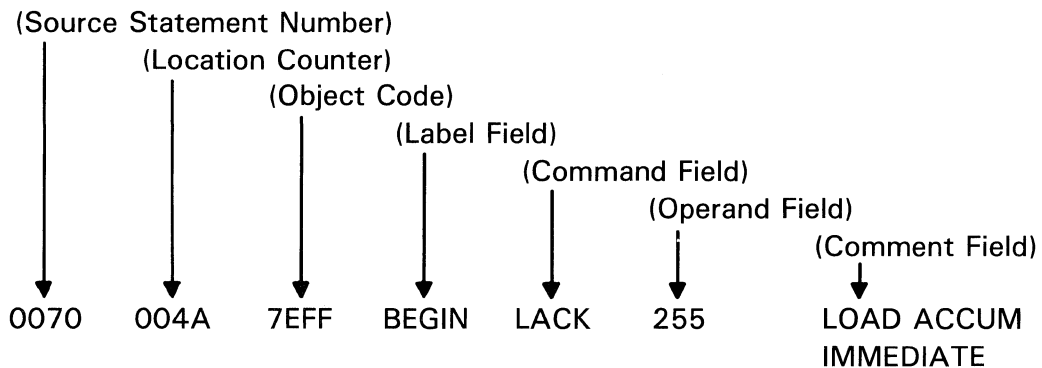
7. ASSEMBLER OUTPUT

7.1 INTRODUCTION

This section presents information concerning the various data output by the assembler, including 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. If the TITL directive is not used, the title line is left blank. 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 word of object code. The assembler prints the location counter value and object code on a separate line for each additional word. Each added line is printed immediately following the source statement line. The following is an example of a source statement line:



The source statement number, 0070 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 the number of source records entered and not listed. Source records generated by a MACRO call, however, are re-numbered starting at 0001. The original sequence continues after the MACRO expansion is complete.

The next field in the listing contains the location counter value, a hexadecimal value. In the example, 004A 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, END directives leave the location counter field blank.

The third field contains the hexadecimal representation of the object code, 7EFF in the above example. All machine instructions and the DATA and TEXT directives use this field to list object code. The EQU directive places the value corresponding to the label in the object code field.

The fourth field contains the characters of the source statement as they were scanned by the assembler. The maximum line length which the assembler will accept is 60 characters. 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.

NOTE

Each of the four fields must be separated by at least one blank space.

7.3 ASSEMBLER ERROR MESSAGES

The assembler issues three types of error messages: informative, nonfatal, and fatal. 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. The errors are referenced by number. At the end of a module (as delineated by an IDT and 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	<p>This is a warning that the following two statements have the same result:</p> <pre style="margin-left: 40px;">CSEG 'DATA' DSEG</pre>
WARNING – SYMBOL TRUNCATED	<p>The maximum length for a symbol is 6 characters. The assembler ignores the extra characters.</p>
WARNING – STRING TRUNCATED	<p>Check the syntax for the directive question to determine the maximum length for the string.</p>
WARNING – TRAILING OPERAND(S)	<p>The assembler found fewer or more operands than expected in the flagged instruction.</p>
WARNING – NULL STRING DEFINED	<p>A null string (i.e., ' ') is defined for directives that required a string operand.</p>
FATAL ERRORS	
ABSOLUTE VALUE REQUIRED	<p>A relocatable symbol was used where an absolute symbol was expected.</p>
DISPLACEMENT TOO BIG	<p>The maximum value of the operand was exceeded.</p>
INVALID EXPRESSION	<p>This may indicate invalid use of a relocatable symbol in arithmetic.</p>
EXPRESSION OUT OF BOUNDS	<p>Range limit for the value of the operand was exceeded.</p>
DUPLICATE DEFINITION	<p>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.</p>
INVALID RELOCATION TYPE	<p>An absolute variable cannot be made relocatable.</p>
INVALID OPCODE	<p>The command field of the source record has an entry that is not a defined instruction, directive, psuedo-op, DXOP, DFOP, or macroname.</p>
INVALID OPTION	<p>The option given in the OPTION directive is invalid. An option is often misspelled.</p>
INVALID REGISTER VALUE	<p>The register specified is too large or too small. Only values of 0 or 1 are allowed for AR0 and AR1, respectively.</p>
INVALID SYMBOL	<p>The symbol has invalid characters in it (see Section 2).</p>
VALUE TRUNCATED	<p>The value is too big for the field and has been truncated. This message also appears when a label string exceeds its maximum length.</p>
SYMBOL USED IN BOTH REF AND DEF	
COPY FIELD OPEN ERROR	<p>File does not exist or is already being used.</p>

TABLE 7-1 – ASSEMBLY LISTING ERRORS

MESSAGE	EXPLANATION/RESPONSE
EXPRESSION SYNTAX ERROR	Unbalanced parentheses or invalid operations on relocatable symbols.
INVALID ABSOLUTE CODE DIRECTIVE	The directives PEND, DEND and CEND have no meaning in absolute code.
LABEL REQUIRED	The flagged directive must have a label.
BLANK MISSING	A blank or blanks must separate each field of the source statement.
COMMA MISSING	Expected a comma but did not find one. Usually means that more operands were expected.
COPY FILENAME MISSING	Filename specified cannot be found.
SYMBOL REQUIRED	OPTION, DEF, REF, SREF, and LOAD directives require symbols as operands.
OPERAND MISSING	An operand must be supplied.
CLOSE (") MISSING	All strings must be enclosed in quotes.
CLOSE (') MISSING	Mismatched parentheses.
STRING REQUIRED	TEXT directive used with no text following.
PASS1/PASS2 OPERAND CONFLICT	A-symbol in the symbol table did not have the same value in PASS1 and PASS2.
SYNTAX ERROR	
UNDEFINED SYMBOL	The symbol has not been REF'ed or it has been DEF'ed but not used.
DIVIDE BY ZERO	An expression or well-defined expression contains invalid division.
ILLEGAL SHIFT COUNT	The shift count requested is not valid.
INFORMATION MESSAGES	
OPCODES REDEFINED	As a result of an MLIB directive, one or more assembler opcodes has been redefined 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) within a MACRO DIRECTORY. The user should take action if this is not intended.

7.4 CROSS-REFERENCE LISTING

The assembler prints an optional cross-reference listing following the source listing. (The cross-reference listing is created by using the OPTION directive.) The format of the listing is shown in Figure 7-1.

```

LABEL  VALUE  DEFN  REFERENCES                                     PAGE 0004
BASE2  029B   0095
BC      0236   0009 0003 0025 0030 0035 0060 0061 0064 0067 0069
BCDONE REF  0004 0082 0084 0086 0088 0090 0092 0094
CTXT0  023B   0014 0020 0079
CTXT1  023C   0015 0021 0077
CTXT2  023D   0016 0022 0078
IORT   SREF   0005
IORT1B UNDF           0039 0043
IORT1F 0256   0040 0028
IORT2F 025B   0044 0036
IORT3F 0281   0076 0072
IORTB1 0298   0093 0058
IORTB2 0295   0091 0055
IORTB3 0292   0089 0052
IORTB4 028F   0087 0049

```

FIGURE 7-1 – CROSS-REFERENCE LISTING FORMAT

As shown in Figure 7-1, in the LABEL column the assembler prints each symbol defined or referenced in the assembly. The VALUE column contains a four-digit hexadecimal number and is possibly followed by either a character or a name which represents the attributes of the symbol. A four-digit hexadecimal number represents the value assigned to the symbol. The characters that could possibly follow the four-digit number or the names which could be in the value column have their meanings listed in Table 7-2. The number of the statement in which the symbol is defined appears in the Definition column. For undefined symbols, this column is left blank. The Reference column lists the numbers of statements that reference the symbol. A blank in this column indicates the symbol was never used.

TABLE 7-2 – SYMBOL ATTRIBUTES

CHARACTER OR NAME	MEANING
REF	External reference (REF)
UNDF	Undefined
SREF	Secondary reference (SREF)
'	Symbol defined in a program segment
''	Symbol defined in a data segment
+	Symbol defined in a common segment

7.5 OBJECT CODE

The assembler produces object code, which may be linked to other object code modules or programs, and is 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 is an example of object code.

```
K0095SAMPROG A0000B0000B0000B0000B0020BA0A0B0D0AB0000B0001B00007F2EDF SAMPROG1
BF900C0019BF600C0013B5000B5801B4801B4900B7F82B7F8DB5000B5801B4A017F1AEFSAMPROG2
B4B00B7F80B7F8DB6E00B4002B4003B4004B4005B7F89B6880B7002B7104B7F827F182FSAMPROG3
B4800B7FB0B7D02B7F81B7F80B6A04B6D05B8002B7F8EB7F8CB4006B7F89B7F817F0F9FSAMPROG4
B6506B7F81B6805B6806B6D05B7F82B7F8EB6880B7006B6A88B6D04B8010B80027F14DFSAMPROG5
B7D03B7F81B6806B7F89B7F82B7F8EBF800A0089B4007B7F89B7F81B6607B7F807F101FSAMPROG6
B5000B4B00B7F80B0105B5000B4C00B4D00B4E00B4F00B4800B4900B6604B7F827F1B2FSAMPROG7
B6106B7F80B7F8AB7F8EB5000B4800B7F8DB6881BF900C002E7F478F SAMPROG8
: SAMPROG 9/20/83 9:43:19 ASM320 2.1 83.074 SAMPROG9
```

FIGURE 7-2 - SAMPLE OBJECT CODE

7.5.1 Object Code Format

Object code is formatted to contain records made up of fields sandwiched between tag characters. Table 7-3 lists field and character information.

A tag character occupies the first position on each line of object code and identifies the fields it precedes. The specific tag character used depends on the function of the field 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 words of program relocatable code; field two contains the program identifier assigned to the program by an IDT directive. When no IDT directive is entered, NO\$IDT is put into field two. The linker uses the program identifier to identify the program, and the number of words 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 words, 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 at which execution starts when linking is complete. Tag character 1 is used when the entry address is absolute; tag character 2 is used when the address is relocatable. The field lists the address in hexadecimal.

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 character B is used when the data is absolute (i.e., an instruction word or a word that contains text characters or absolute constants). 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 #, %, and & are also used where a 7-bit field of an instruction refers to a data element in either a DSEG, PSEG, or CSEG. Tag character # identifies an instruction containing a reference to a seven-bit data-relative item. The second field following the tag contains a mask which indicates to the link editor the width of the field (mask = >007F indicates the least seven bits). The link editor generates the final version of this instruction by adding to the masked data word, the beginning location of the data segment and reinserting the sum in the seven-bit field within the data word. Note that field overflow may occur in this operation, and error messages may be generated by the link edit operation which were not evident at assembly time. The description of the % tag is the same as above, except that it represents the use of a program-relative item as the operand. The fields used with the & tag are identical to the # and % tags, except that the second field is the common number and the mask becomes the third field.

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 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 identify 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 to references identified by 3, 4, V, X, and Y 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 where a seven-bit field of an instruction refers to an external reference. The determining factor here is that the field can only be seven bits in width, rather than the usual 16-bit word. The format of the ! sequence is:

! (external symbol number) (opcode/offset) (mask)

The processing of this tag and its associated fields is the same as that of the # tag, above.

Tag characters G, H, and J are used when the symbol table option (SYMLST) is specified under the OPTION directive. 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 loader. Field one contains zeros. Field two contains the symbol for which the loader will search for a definition. Refer to the LOAD directive for further details.

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 identify 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 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 characters' eight-bit ASCII values from the first tag of the record through the checksum tag, tag character 7.

Tag character 8 is also associated with the checksum field but is used when the checksum field is to be ignored (as when changing the object code).

Tag character D is used to specify a load bias. Its lone associated field contains the absolute address that will be used by the loader to relocate symbols. The link editor does not accept the D tag. Refer to Section 7.5.3 for additional information.

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 tag character 7, followed by the checksum field and tag character F. 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 the module name, the date of the assembly, and the time of the assembly. Table 7-3 defines the object record format and tags.

TABLE 7-3 – 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 #	
(DATA WORD)			
B	ABSOLUTE 16-BIT VALUE		
C	P-R ADDRESS		
T	D-R ADDRESS		
N	C-R ADDRESS	COMMON #	
#	OPCODE/DR ADDRESS	MASK	
%	OPCODE/PR ADDRESS	MASK	
&	OPCODE/CR ADDRESS	COMMON #	MASK
(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	OPCODE/OFFSET	MASK
(SYMBOL DEFINITIONS)			
G	P-R ADDRESS	SYMBOL(6)	
H	ABSOLUTE VALUE	SYMBOL(6)	
J	D-R/C-R ADDRESS	SYMBOL(6)	COMMON #
(FORCE LOAD)			
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 #
(CHECKSUM)			
7	VALUE		

- NOTES: 1. All field widths are four characters unless otherwise specified.
 2. If the first tag is >01, 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)

TABLE 7-3 – OBJECT RECORD FORMAT AND TAGS (Concluded)

TAG	1ST FIELD	2ND FIELD	3RD FIELD
(IGNORE CHECKSUM)			
8	ANY VALUE		
(LOAD BIAS)			
D	ABSOLUTE ADDRESS		
(END OF RECORD)			
F			
(END OF MODULE)			
:		(LANGUAGE PROCESSOR DEPENDENT)	

- NOTES:
1. All field widths are four characters unless otherwise specified.
 2. If the first tag is >01, 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 (see Section 5).

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 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 only require 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 seven tag characters. 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 tag character 7 to 8.

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 (load address tag characters), 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 (data word tag characters) 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 words of relocatable code. For example, if the object field written by the assembler contained 1000 hexadecimal words of relocatable code and the change has added eight words 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. However, the link editor will not accept tag character D in changed or added object records.

8. MACRO CAPABILITY

8.1 INTRODUCTION

The assembler recognizes a macro definition language that may be used to simplify programming. A macro definition is a set of source statements (machine instructions, macro language statements, and assembler directives) which constitute a template for generating other statements within a source program.

When the assembler processes a macro call, it substitutes the predefined statements of the macro definition for the macro call statement in the source program, 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. 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 opcode field, and possibly a list of formal parameters separated by commas in the opcode 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. This 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 host operating system.

Directions for the use of a macro library are 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 opcode/directive table is made. If found there, the opcode 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 opcode 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, that library should include a text file named 'MLIST', which contains the names of the opcodes and currently defined macros (one name per line, starting with column one) which are to be redefined.

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 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 (not floppy) disks.

CAUTION

The use of the assembler opcode table override feature can result in unpredictable behavior of the assembler. Care should be exercised in electing to use this option.

8.2.1 Sample Macros

The following is a simple example of a macro definition:

```
INCX    $MACRO
        LACK 1
        ADD  X
        SACL X
        $END
```

The above code defines a macro named INCX. \$MACRO identifies the beginning of the macro definition, and \$END identifies the end of the macro definition. LACK 1, ADD X, and SACL X are model statements which will be placed into the source program upon a macro call. The macro INCX 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.:

```
LACK 1
ADD  X
SACL 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
        LACK 1
        ADD  :M.S:
        SACL :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:

```
LACK 1
ADD  Y
SACL Y
```

but

```
INC Z
```

will be replaced by:

```
LACK 1
ADD  Z
SACL 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 Section
                        8.3.3.1)
        LACK      :NUM.V:
        ADD       :X.S:
        SACL      :X.S:
        $END
```

NUM.V is the value component of the parameter NUM. The call:

```
ADDK Y,3
```

will result in:

```
LACK 3
ADD Y
SACL Y
```

These and other macro commands will be explained 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. Statements containing macro language verbs are used to define the macro and macro variables, and to determine which model statements are assembled. All macro statements which do not contain verbs are processed as model statements.

8.3.1 Strings

The literal strings of the macro language consist of one or more characters which do not need enclosing quotes except in the \$ASG statement (see Section 8.3.5.3). They are identical to the character string used in the assembly language. Examples of a string are:

```
'ONE'
' ' (a blank space)
```

8.3.2 Constants And Operators

Constants for the macro language are defined in the same manner as constants for the 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 is an example 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 a variable and a constant, 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 block if value component of variable A is > 5.
\$IF	B.L# = A.L	Process succeeding block if length component of variable B is not equal to length component of variable A (see Section 8.3.5.4).

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) Process succeeding block if both expressions in parentheses are true.

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 in juxtaposition 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 six characters in length. 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 (see Section 8.3.4.1).

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.

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. Each entry consists of four components: the string, value, length, and attributes of a variable or parameter. The macro assembler places parameters in the MST as it processes a macro call, and places variables in the MST as it processes the macro language \$VAR statements that declare variables.

The string component of an entry in the MST contains a character string assigned to the macro variable or parameter by the macro expander. The value component contains the numerical equivalent of the string component, if the string component is an integer. The value component can also contain the numerical value of the symbol, if the string component is a symbol in the Assembler Symbol Table (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.

Example of a macro definition:

```
ADDK $MACRO X,NUM
```

identifies a macro, ADDK, having parameters X and NUM.

A macro call to activate that macro definition is coded as follows:

```
ADDK VAR1,3
```

The MST now contains parameters X and NUM. The string component of parameter X is the character string VAR1. The attribute component (\$PCALL) indicates that the parameter is supplied in a macro call. The length component is four. The string component of parameter NUM is the ASCII character 3. The value component is three, (expressed as a binary number), and the length component is one. The attribute component indicates that the parameter is supplied in the macro call.

Each component of a macro variable may be accessed individually. Reference to a variable component is made in either binary mode or 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 writing 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 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 X. In the example of the macro call for the macro ADDK in Section 8.3.3.2, X.S equals the binary equivalent for VA or >5641. A string mode indicated as :X.S: is equal to VAR1.
- X.A Attribute component of variable X. This component may be accessed by the use of logical operators (&, ++, and --) and attribute keywords (described in Table 8-4).
- X.V Value component of variable X. :X.V: is equal to the value of VAR1.
- X.L Length component of variable X. In the same example of the macro call for the macro ADDK, :X.L: = 4.

TABLE 8-1 – VARIABLE QUALIFIERS

QUALIFIER	MEANING
S	String component of the variable
A	Attribute component of the variable
V	Value component of the variable
L	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:

- (1) :CT.S: WAY Variable CT qualified
- (2) :CT: WAY Variable CT unqualified

Note that 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 symbols.

The following are examples of qualified variables that specify symbol components of string components of variables. Assume that V1.S have been defined as MASK, and the statement

MASK EQU >FF

has been previously encountered in the assembly language source program.

- V1.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 string mode, :V1.SV: equals the characters "255".

- V1.SA Attribute component of the symbol MASK. This component may be accessed by logical operators and keywords, as described later.
- 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'

and provides 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 defines the variable qualifiers for symbol components.

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 symbol that 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 attribute component of assembler symbols and macro parameters contains information on various attributes of those symbols and parameters. The macro language recognizes certain keywords that are used to access that information. A keyword is used with a logical operator and the attribute component to test or to 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: (Assume V1.S has been defined as MASK.)

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 an DEF directive
\$STR	Symbol has been assigned a component string
\$MAC	Symbol is defined as a macro name
\$UNDF	Symbol is not defined

* 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
\$PSVM	Parameter is a symbolic memory address. (Note that a symbolic memory address is recognized when the variable is preceded by an "@" character.)

8.3.5 Verbs

The macro language supports seven 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 macro (see Section 8.3.3.1). 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-list>    [<comment>]]
```

where <parm-list> is a sequence of parameters separated by commas. The macro definition is used in the expansion of macro calls where that macro name appears in the instruction field.

The syntax for a call is:

```
<macro name>    [<operand-list>    [<comment>]]
```

where <operand-list> is a sequence of operands, separated by commas. 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 the list) may be used. A list is processed as a set after removal of the outer parentheses during macro expansion.

Operands (or lists) may be nested in parentheses in the macro call for use within macro definitions. For example:

```
ONE $MACRO P1,P2
```

specifies two 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 (see Table 8-4) 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 has 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 AMAC AB1,AB2 sets \$PCALL parameters P1 and P2 but not P3.

Similarly, AMAC XY,,XY3 causes \$PCALL to be set for P1 and P3 but not P2.

When the macro call has more operands than the number of parameters in the \$MACRO statement, the excess operands are combined with the operand or list corresponding to the last parameter to form a list (or a longer list). For example, in the macro statements shown below, the operands of the two macro calls would be assigned to the parameters in the same way:

(1) ONE	EQU	9	
TWO	EQU	43	
THREE	EQU	86	
FIX	\$MACRO	P1,P2	MACRO FIX
	.		
	.		
	.		
	FIX	ONE,TWO,THREE	MACRO CALL
	FIX	ONE,(TWO,THREE)	MACRO CALL

(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
	PARAM	\$MACRO	P1,P2,P3,P4,P5,P6,P7,P8,P9
	.	.	.
	.	.	.
	.	.	.
	PARAM		A,,B,(I),C,(D),E,(G,H,I))

Parameter assignments:

P1.S = A	P2.S =	(no string)
P1.A = \$PCALL	P2.A =	(all false)
P1.L = 1	P2.L = 0	
P1.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.V = 95	P8.V = 2	
P9.S =	(no string)	
P9.A =	(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, they 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.

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, which must not have been declared as parameters.

The \$VAR statement does not assign values to any components of the variables; that is the function of the \$ASG statement (see below). \$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 <express/string> TO <var> [<comment>]
```

The expression operand may be any expression which is valid to the assembler and may contain binary mode variable references and the keywords shown in Tables 8-3 and 8-4.

A string may be one or more characters enclosed in single quotes or the concatenation of such a literal string with the string mode value of a qualified variable. The <var> may be either an unqualified or 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. This adds 2 to the length component of P3.

\$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 symbol component qualifiers to modify the components of symbol TEN.

\$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 six.

\$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 string component of the symbol in the 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. When quoted strings are assigned to the string component of some variable, that string may later appear in the list of undefined symbols. In most cases, the programmer will not be concerned with their appearance in that list since their definition as labels was never intended.

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, it 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 is evaluated as TRUE), the block of statements following the \$IF statement is processed. When the expression in the \$IF state-

ment has a zero value (or is 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 8.3.2). In addition, \$IF blocks may be nested at most 44 levels.

The following example shows conditional processing in macro definition:

<pre> . . . \$IF KY.SV . . BLOCK A . \$ELSE . . BLOCK B . \$ENDIF . . \$IF -- (T.A&\$PCALL) . . BLOCK A . . \$ENDIF . . \$IF T.L = 5 BLOCK A \$ENDIF </pre>	<p>Process the statement of BLOCK A when the value component of the symbol in the string component of variable KY contains a non-zero value. Process the statements of BLOCK B when the component contains zero. After processing either block of statement, continue processing at the statement following the \$ENDIF statement.</p> <p>Process the statements of BLOCK A when the attribute component of parameter T indicates that parameter T was not supplied in the macro instruction. If parameter T was supplied, do not process the statements of BLOCK A. Continue processing at the statement following the \$ENDIF statements in either case. Process the statements of BLOCK A when the length component of variable T is equal to 5; otherwise, do not process the statements of BLOCK A. Continue processing at the statement following the \$ENDIF statement.</p>
---	---

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 8.3.5.4).

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 Section 8.3.5.4.

Syntax:

```
$ENDIF    [<comment>]
```

8.3.5.7 *\$END Statement*

The `$END` statement marks the end of the group of statements of the macro definition named in the operand. When executed, the `$END` statement terminates the processing of the macro definition. The macro name may appear as a comment to enhance readability.

Syntax:

```
$END    [<macro name>]    [<comment>]
```

The following is an example of an `$END` statement:

```
$END FIX           Terminates the definition of a macro
```

8.3.6 **Model Statements**

As stated previously, a macro definition consists of model statements and statements that contain macro language verbs. Processing a model statement 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 8.3.3.3). In any case, the resulting source statement must be a legal assembler language statement. The following examples show model statements:

(1) `IN *+,PA7,1`

This model statement is itself an assembly language source statement that contains a machine instruction.

(2) `:P7.S: LAR :P2.S:,R8 :V4.S:`

This model statement begins with the string component of variable P7. Three blanks, LAR, and three more blanks are concatenated 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.

(3) :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 them. 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. The macro 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		Start of the alternate portion of the 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 label ST 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. FACT accomplishes this by calling FACT1, which calls itself recursively.

```
FACT  $MACRO N,LOC
      $IF N.V < 2
      LACK 1                                * 1! = 0! = 1
      SACL :LOC:
      $ELSE
      LACK :N.V:                            * N GREATER THAN/EQUAL 2 SO,
      SACL :LOC:                            * STORE 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
      LT :AREA:                             * MULTIPLY FACTORIAL SO FAR
      MPYK :M.V:                             * BY CURRENT POSITION
      PAC
      SACL :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 FFT

Macro FFT supplies a macro for a generalized FFT "butterfly". To use this macro, data should be scaled so that it is totally fractional.

In general, the macro will perform additions and subtractions in the first fifteen bits of the upper accumulator, leaving the sixteenth bit open for a one-bit overflow if it occurs; results are shifted out so that they contain the overflow bit. Only the absolute values of the components of W should be stored in data memory.

```
IDT      'FFT'

*
*      THIS MACRO IS FOR FFT BUTTERFLIES.
*
*      THE AIM OF THE MACRO IS TO TRANSFORM THE COMPLEX
*      NUMBERS (XR,XI) AND (YR,YI) AS FOLLOWS:
*
*      (XR,XI) → (XR + RE[Y*W],XI + IM[YW])
*      (YR,YI) → (XR - RE[Y*W],XI - IM[Y*W])
*
*      THE RESULT OF THE TRANSFORMATION IS STORED IN
```

```

*      THE SAME MEMORY LOCATIONS AS THE ORIGINAL
*
*      THE FIRST STEP IS TO COMPUTE (W) (Y). THE REAL
*      PARTS OF THIS PRODUCT WILL BE STORED IN TEMP,
*      THE IMAGINARY PARTS IN YI, ACCORDING TO:
*      (WR-iWI) (YR+iYI)=(YR*WR + YI*WI) + i(YI*WR-YR*WI)
*****
FFT $MACRO  XR, XI, YR, YI, WR, WI
*  THE REAL PART IS:
*
      LT      :YR:
      MPY     :WR:      :YR*:WR:
      PAC
      LT      :YI:
      MPY     :WI:      :YI*:WI:
      APAC    ACC=REAL PART. STORE IN TEMP.
      SACH    TEMP,1    SHIFTING TO ELIMINATE SECOND SIGN BIT.
*
*  THE IMAGINARY PART IS:
*
      MPY     :WR:      :YI*:WR:
      PAC
      LT      :YR:
      MPY     :WI:      :YR*:WI:
      SPAC    ACC=IMAG PART. STORE IN YI.
      SACH    :YI:,1    SHIFT TO ELIMINATE SECOND SIGN BIT.
*  NOW TEMP HAS REAL PART AND YI HAS IMAGINARY PART OF
*  PRODUCT. NOW DO ADDITIONS AND SUBTRACTIONS. DO REAL
*  PARTS FIRST. TEMP CONTAINS RE[Y*W].
*
      LAC     :XR:,15
      SUB     TEMP,15
      SACH    :YR:      :XR: - RE[ ] -> :YR:
      ADDH    TEMP      :XR: - RE[ ] + 2*RE[ ]
      SACH    :XR:      :XR: + RE[ ] -> :XR:
*
*  CALCULATE IMAGINARY POINTS:
*
      LAC     :XI:,15
      ADD     :YI:,15    :XI: + IM[ ]
      SACH    :XI:      :XI: + IM[ ] -> :XI:
      SUBH    :YI:      :XI: + IM[ ] - 2*IM[ ]
      SACH    :YI:      :XI: - IM[ ] -> :YI:
      $END
*  NOW THAT THE MACRO IS COMPLETED, SHOW HOW TO USE IT.
      AORG
XOR EQU 0
XOI EQU 1

```

```

X1R    EQU    2
X1I    EQU    3
WOR    EQU    4
WOI    EQU    5
TEMP   EQU    127
*
      FFT    XOR, XOI, X1R, X1I, WOR, WOI
*

```

END

NO ERRORS, NO WARNINGS

8.5 MACRO ERROR MESSAGES

Table 8-5 lists and defines the macro error messages, and gives correction information.

TABLE 8-5 – MACRO ERROR MESSAGES

MEANING	DESCRIPTION
MACRO LINE TOO LONG	In a macro definition, macro directive lines may only be 58 characters long. 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 MACRO VARIABLES	The total number of macro parameter variables and labels in a single 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

TMS32010 HARDWARE SUMMARY

A.1 COMPONENTS

NAME	SYMBOL	SIZE (bits)	DESCRIPTION
Program ROM	—	1536 X 16	On-chip masked ROM containing program code.
Program Counter	PC	12	Register containing current address of program memory.
Stack	—	4 X 12	Four 12-bit registers for saving program counter contents during subroutine and interrupt calls.
Data RAM	—	144 X 16	On-chip RAM containing data. It can be addressed both directly and indirectly. The instruction DMOVE enables the user to move the contents of a given location in RAM to the next higher location in one machine cycle. This is a very useful function in many applications, such as signal processing.
Data Memory Page Pointer	DP	1	A single-bit register containing the page address of data RAM. 1 page = 128 words. Note that the second page utilizes only the first 16 words.
Auxiliary Registers 0 and 1	AR	2 X 16	The eight least significant bits are used for indirect addressing of data memory. The nine least significant bits can also be configured as bidirectional counters for loop control, with options for autoincrement/decrement.
Auxiliary Register Pointer	ARP	1	A single-bit register which points to current auxiliary register.
Shifter	—	—	Two shifters are present. One left-shifts data from 0 to 15 bits on its way to the ALU; the other left-shifts the result of the accumulator either 0, 1, or 4 bits. The shifter is controlled by four bits in the opcode of arithmetic operations, and its output is always a 32-bit word. To handle two's complement arithmetic, shifted data is zero-filled, and the high-order bit is sign-extended. In addition, there are instructions that suppress sign extension.

NAME	SYMBOL	SIZE (bits)	DESCRIPTION
T Register	T	16	Contains the multiplicand in multiply operations.
Multiplier	—	—	Multiplies two 16-bit numbers. The result is 32 bits. The multiplier is a word from the data RAM or a 13-bit immediate value in the instruction word. The immediate value is loaded right-justified and sign extended.
P Register	P	32	Contains the 32-bit product of multiply operations.
Arithmetic Logic Unit	ALU	32	Performs all arithmetic and logical functions except multiply. Logical operations are between the 16 least significant bits of the accumulator and the data memory value.
Accumulator	ACC	32	Accumulates results of ALU. Holds branch address of program memory during branch operations. Contains an overflow mode (see below).
Interrupt Flag Register	INTF	1	Used to indicate an interrupt. Automatically cleared upon grant of an interrupt.
Interrupt Mode Register	INTM	1	Used to mask the Interrupt Flag. Upon grant of an interrupt, this bit is set to one by the DINT instruction. This disables further interrupts. This register is reset by the EINT instruction.
Overflow Flag	OV	1	A one indicates an overflow in arithmetic operations. The BV (branch on overflow) instruction tests if this flag is clear and clears it. This feature allows the flexibility of overflow examination outside time-critical loops.
Overflow Mode	OVM	1	Defines whether the TMS32010 operates in the saturated or unsaturated mode during arithmetic operations. In the saturated mode, an overflow/underflow causes the accumulator to be set to its largest/smallest representative value. A logic one enables the overflow mode, and a logic zero disables it.

A.2 ADDRESSING MODES AND INSTRUCTION FORMAT (SEE SECTION 3.3)

A.3 INTERFACE AND CONTROL

A.3.1 Program Control

In the microcomputer mode, the TMS32010 can access 2.5K words of program off-chip, in addition to the 1.5K words on-chip. To facilitate this ability, the program counter outputs are buffered to the address pins A11-A0. A strobe output (\overline{MEN}) is generated every machine cycle to enable external memory, except when an IN, OUT, or TBLW instruction is being executed. Data from external memory is transferred to the TMS32010 via the data bus (D15-D0).

The TMS32010 suffers no performance degradation in fetching program words from off-chip memory, as long as the memory access time is approximately 100 ns.

A.3.2 Interrupts

The TMS32010 supports a single-level vectored interrupt with provisions for a full context save. A negative going edge on the \overline{INT} pin generates an interrupt and sets the interrupt flag. When servicing an interrupt, the TMS32010 pushes PC + 1 onto the stack and branches to location 2. Locations 0 and 1 are reserved for RESET.

TMS32010 has an interrupt mode bit which is set by the Disable Interrupt (DINT) instruction, and cleared by the Enable Interrupt (EINT) instruction. When set, this bit inhibits the TMS32010 from responding to an interrupt. Upon grant of an interrupt by the processor, the INT flag is automatically cleared, and the INT mode bit is set. This disables servicing future interrupts until EINT is executed. This configuration allows the TMS32010 to complete time-critical loops before servicing an interrupt.

This instruction set allows for the storage and recovery of all registers and status bits, except the P register. The TMS32010 also has hardware protection that prevents response to an interrupt between an MPY or MPYK instruction and the next instruction. Thus, the contents of the P register will be accumulated before the interrupt is serviced. In addition, the TMS32010 has hardware that prevents the servicing of an interrupt until the end of multicycle instructions.

A.3.3 Branch Instructions

There are a variety of branch instructions that allow testing for the following conditions:

- Auxiliary register counter portion not zero
- Overflow
- Low-level on the I/O Branch Control pin (\overline{BIO})
- Accumulator less than zero, less than or equal to zero, greater than zero, greater than or equal to zero, not zero, equal to zero.

A.3.4 Clock

The TMS32010 can use either its internal oscillator or an external frequency source for a clock. The internal oscillator is enabled by connecting a crystal across X1 and X2/CLKIN. The frequency of CLKOUT is $\frac{1}{4}$ the frequency of the input.

A.3.5 I/O

The TMS32010 has an external parallel 12-bit address bus and an external parallel 16-bit data bus. The I/O space is separate from the general memory space and allows direct addressing of up to eight peripherals.

Two instructions, IN and OUT, cause input and output of data to and from the TMS32010 over the data bus. These instructions contain a 3-bit port address which is multiplexed over the three least significant address lines (A2/PA2 – A0/PA0) while the remaining address lines are held at a logic zero.

Inputs or outputs are distinguished by the $\overline{\text{DEN}}$ and $\overline{\text{WE}}$ strobe. Execution of an IN instruction generates a $\overline{\text{DEN}}$ strobe which enters the data on the data bus into the data RAM. Execution of an OUT instruction outputs data from the data RAM onto the data bus and generates a $\overline{\text{WE}}$ strobe.

In addition, two instructions, TBLR (table read) and TBLW (table write), allow a transfer between data and program spaces. TBLR reads a word from program memory and transfers it to the data RAM. TBLW copies a word from the data RAM into external program memory (presumably a RAM). In both instructions, the data memory address is the instruction operand, and the program memory address is contained in the accumulator.

INSTRUCTION	OPERATION PERFORMED
IN PA, A	(D15 through D0) → (A) (PA) → (ports A2/PA2 through A0/PA0)
TBLW A	(PC) + 1 → top of stack (ACC) → (PC) → (A11 through A0) (A) → (D15 through D0) Top of stack → (PC)

NOTE: () = contents of the named location.

APPENDIX B

CHARACTER SETS RECOGNIZED BY THE ASSEMBLER

The TMS32010 Assembler recognizes the ASCII characters listed in Table B-1. It also accepts the characters listed in Table B-2, if they occur within quoted strings or in comment fields. The special characters in Table B-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 programs buffer) all the characters listed in Tables B-1, B-2, and B-3.

All of the tables include the ASCII code for each character represented, a hexadecimal value, and a decimal value. The tables also include the keypunch (Hollerith) code for each character.

TABLE B-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
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

TABLE B-1 – ASCII CHARACTER SET (Concluded)

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) HOLLERITH CODE
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 B-2 – SPECIAL CHARACTERS RECOGNIZED IN QUOTED STRINGS AND COMMENT FIELDS

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) 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

TABLE B-2 – SPECIAL CHARACTERS RECOGNIZED IN QUOTED STRINGS AND COMMENT FIELDS (Continued)

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) HOLLERITH CODE
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
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

TABLE B-2 – SPECIAL CHARACTERS RECOGNIZED IN QUOTED STRINGS AND COMMENT FIELDS (Concluded)

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) HOLLERITH CODE
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
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 B-3 – ADDITIONAL CHARACTERS RECOGNIZED BY THE OPERATIVE SYSTEM DEVICE SERVICE ROUTINE

HEXADECIMAL VALUE	DECIMAL VALUE	CHARACTER	(KEYPUNCH) 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

CROSS-ASSEMBLER DIRECTIVES (CONCLUDED)

PAGE TITLE TITL
 TITL supplies title to be printed in the heading of each page of the source listing.
 Syntax: [**label**] TITL '<string>' [**comment**]

RESTART SOURCE LISTING LIST
 LIST restores printing of the source listing.
 Syntax: [**label**] LIST [**comment**]

STOP SOURCE LISTING UNL
 UNL halts the source listing output until the occurrence of a LIST directive.
 Syntax: [**label**] UNL [**comment**]

EJECT PAGE 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.
 Syntax: [**label**] PAGE [**comment**]

INITIALIZE WORD DATA
 DATA places one or more values in one or more successive words of memory.
 Syntax: [**label**] DATA <exp>[,<exp>] [**comment**]

INITIALIZE TEXT TEXT
 TEXT places one or more characters in successive words of memory.
 Syntax: [**label**] TEXT [-]'<string>' [**comment**]

DEFINE ASSEMBLY-TIME CONSTANT EQU
 EQU assigns a value to a symbol.
 Syntax: <label> EQU <exp> [**comment**]

EXTERNAL DEFINITION DEF
 DEF makes one or more symbols available to other programs for reference.
 Syntax: [**label**] DEF <symbol>[,<symbol>] [**comment**]

EXTERNAL REFERENCE REF
 REF provides access to one or more symbols defined in other programs.
 Syntax: [**label**] REF <symbol>[,<symbol>] [**comment**]

SECONDARY EXTERNAL REFERENCE SREF
 SREF provides access to one or more symbols defined in other programs.
 Syntax: [**label**] SREF <symbol>[,<symbol>] [**comment**]

FORCE LOAD LOAD
 LOAD is similar to REF, but the symbol does not need to be used in the module containing the LOAD. The symbol used in LOAD must be defined in some other module. LOADs are used with SREFs.
 Syntax: [**label**] LOAD <symbol>[,<symbol>] [**comment**]

PROGRAM END END
 END terminates the assembly. The last source statement of a program is the END directive.
 Syntax: [**label**] END [**comment**]

COPY SOURCE FILE COPY
 COPY changes the source input for the assembler.
 Syntax: [**label**] COPY <file name> [**comment**]

DEFINE MACRO LIBRARY MLIB
 MLIB provides the name of a library containing macro definitions.
 Syntax: [**label**] MLIB '<pathname>' [**comment**]

TMS32010 DIGITAL SIGNAL PROCESSOR Programmer's Reference Card

ASCII REFERENCE TABLE

	00	10	20	30	40	50	60	70
00	NUL	DLE	SP	0	@	P	\	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EQT	DC4	\$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	'	7	G	W	g	w
08	BS	CAN	(8	H	X	h	x
09	HT	EM)	9	I	Y	i	y
0A	LF	SUB	*	:	J	Z	j	z
0B	VT	ESC	+	;	K	[k	{
0C	FF	FS	,	<	L	\	l	
0D	CR	GS	-	=	M]	m	}
0E	SO	RS	.	>	N	^	n	~
0F	SI	US	/	?	O	_	o	DEL

HEX-DECIMAL TABLE

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0000	0	000	0	00	0	0	0
1000	4,096	100	256	10	16	1	1
2000	8,192	200	512	20	32	2	2
3000	12,288	300	768	30	48	3	3
4000	16,384	400	1,024	40	64	4	4
5000	20,480	500	1,280	50	80	5	5
6000	24,576	600	1,536	60	96	6	6
7000	28,672	700	1,792	70	112	7	7
8000	32,768	800	2,048	80	128	8	8
9000	36,864	900	2,304	90	144	9	9
A000	40,960	A00	2,560	A0	160	A	10
B000	45,056	B00	2,816	B0	176	B	11
C000	49,152	C00	3,072	C0	192	C	12
D000	53,248	D00	3,328	D0	208	D	13
E000	57,344	E00	3,584	E0	224	E	14
F000	61,440	F00	3,840	F0	240	F	15

RTC HOTLINE NUMBERS

For help with the TMS32010, call the TI Regional Technology Center nearest you. The centers are staffed with application engineers ready to answer all your questions.

Atlanta 404/452-4686
 Boston 617/890-4271
 Chicago 312/228-6008
 Dallas 214/680-5096
 Northern California 408/980-0305
 Southern California 714/660-8164



CROSS-ASSEMBLER DIRECTIVES (CONCLUDED)

TMS32010 DIGITAL SIGNAL PROCESSOR Programmer's Reference Card

PAGE TITLE	TITL
TITL supplies title to be printed in the heading of each page of the source listing.	
Syntax: <label> TITL '<string>' <comment>	
RESTART SOURCE LISTING	LIST
LIST restores printing of the source listing.	
Syntax: <label> LIST <comment>	
STOP SOURCE LISTING	UNL
UNL halts the source listing output until the occurrence of a LIST directive.	
Syntax: <label> UNL <comment>	
EJECT PAGE	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.	
Syntax: <label> PAGE <comment>	
INITIALIZE WORD	DATA
DATA places one or more values in one or more successive words of memory.	
Syntax: <label> DATA <exp> ,<exp> <comment>	
INITIALIZE TEXT	TEXT
TEXT places one or more characters in successive words of memory.	
Syntax: <label> TEXT ' <string>' <comment>	
DEFINE ASSEMBLY-TIME CONSTANT	EQU
EQU assigns a value to a symbol.	
Syntax: <label> EQU <exp> <comment>	
EXTERNAL DEFINITION	DEF
DEF makes one or more symbols available to other programs for reference.	
Syntax: <label> DEF <symbol> ,<symbol> <comment>	
EXTERNAL REFERENCE	REF
REF provides access to one or more symbols defined in other programs.	
Syntax: <label> REF <symbol> ,<symbol> <comment>	
SECONDARY EXTERNAL REFERENCE	SREF
SREF provides access to one or more symbols defined in other programs.	
Syntax: <label> SREF <symbol> ,<symbol> <comment>	
FORCE LOAD	LOAD
LOAD is similar to REF, but the symbol does not need to be used in the module containing the LOAD. The symbol used in LOAD must be defined in some other module. LOADs are used with SREFs.	
Syntax: <label> LOAD <symbol> ,<symbol> <comment>	
PROGRAM END	END
END terminates the assembly. The last source statement of a program is the END directive.	
Syntax: <label> END <symbol> <comment>	
COPY SOURCE FILE	COPY
COPY changes the source input for the assembler.	
Syntax: <label> COPY <file name> <comment>	
DEFINE MACRO LIBRARY	MLIB
MLIB provides the name of a library containing macro definitions.	
Syntax: <label> MLIB '<pathname>' <comment>	

ASCII REFERENCE TABLE

	00	10	20	30	40	50	60	70
00	NUL	DLE	SP	0	@	P	\	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EQT	DC4	\$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	'	7	G	W	g	w
08	BS	CAN	(8	H	X	h	x
09	HT	EM)	9	I	Y	i	y
0A	LF	SUB	*	:	J	Z	j	z
0B	VT	ESC	+	;	K	[k	{
0C	FF	FS	,	<	L	\	l	
0D	CR	GS	-	=	M]	m	}
0E	SO	RS	.	>	N	^	n	~
0F	SI	US	/	?	O	_	o	DEL

HEX-DECIMAL TABLE

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0000	0	000	0	00	0	0	0
1000	4,096	100	256	10	16	1	1
2000	8,192	200	512	20	32	2	2
3000	12,288	300	768	30	48	3	3
4000	16,384	400	1,024	40	64	4	4
5000	20,480	500	1,280	50	80	5	5
6000	24,576	600	1,536	60	96	6	6
7000	28,672	700	1,792	70	112	7	7
8000	32,768	800	2,048	80	128	8	8
9000	36,864	900	2,304	90	144	9	9
A000	40,960	A00	2,560	A0	160	A	10
B000	45,056	B00	2,816	B0	176	B	11
C000	49,152	C00	3,072	C0	192	C	12
D000	53,248	D00	3,328	D0	208	D	13
E000	57,344	E00	3,584	E0	224	E	14
F000	61,440	F00	3,840	F0	240	F	15

RTC HOTLINE NUMBERS

For help with the TMS32010, call the TI Regional Technology Center nearest you. The centers are staffed with applications engineers ready to answer all your questions.

Atlanta	404/452-4686
Boston	617/890-4271
Chicago	312/228-6008
Dallas	214/680-5096
Northern California	408/980-0305
Southern California	714/660-8164

TEXAS
INSTRUMENTS



SYMBOLS FOR INSTRUCTION SET SUMMARY

SYMBOL	MEANING
D	Data memory address field
I	Addressing mode bit
K	Immediate operand field
PA	3-bit port address field (PA0 through PA7 are predefined assembler symbols equal to 0 through 7, respectively)
R	1-bit operand field specifying auxiliary register
S	4-bit left-shift code
X	3-bit accumulator left-shift field

INSTRUCTION SET SUMMARY

MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE REGISTER															
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ABS	Absolute value of accumulator	1	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0	
ADD	Add to accumulator with shift	1	1	0	0	0	0	←S→		I	←D→								
ADDH	Add to high-order accumulator bits	1	1	0	1	1	0	0	0	0	0	1	←D→						
ADDS	Add to accumulator with no sign extension	1	1	0	1	1	0	0	0	0	1	←D→							
AND	AND with accumulator	1	1	0	1	1	1	1	0	0	1	←D→							
APAC	Add P Register to accumulator	1	1	0	1	1	1	1	1	1	0	0	0	1	1	1	1		
B	Branch unconditionally	2	2	1	1	1	1	0	0	1	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BANZ	Branch on auxiliary register not zero	2	2	1	1	1	1	0	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BGEZ	Branch if accumulator >= 0	2	2	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BGZ	Branch if accumulator > 0	2	2	1	1	1	1	1	0	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BIOZ	Branch on BIO = 0	2	2	1	1	1	0	1	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BLEZ	Branch if accumulator <= 0	2	2	1	1	1	1	0	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BLZ	Branch if accumulator < 0	2	2	1	1	1	1	0	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BNZ	Branch if accumulator ≠ 0	2	2	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BV	Branch on overflow	2	2	1	1	1	0	1	0	1	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
BZ	Branch if accumulator = 0	2	2	1	1	1	1	1	1	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
CALA	Call subroutine from accumulator	2	1	0	1	1	1	1	1	1	0	0	0	1	1	0	0		
CALL	Call subroutine immediately	2	2	1	1	1	1	0	0	0	0	0	0	0	0	0	0		
				0	0	0	0	←BRANCH ADDRESS→											
DINT	Disable interrupt	1	1	0	1	1	1	1	1	1	0	0	0	0	0	1	1		
DMOV	Copy contents of data memory location into next location	1	1	0	1	1	0	1	0	0	1	1	←D→						
EINT	Enable interrupt	1	1	0	1	1	1	1	1	1	0	0	0	0	0	1	0		
IN	Input data from port	2	1	0	1	0	0	←PA→		I	←D→								
LAC	Load accumulator with shift	1	1	0	0	1	0	←S→		I	←D→								
LACK	Load accumulator immediate	1	1	0	1	1	1	1	1	0	←K→								
LAR	Load auxiliary register	1	1	0	0	1	1	1	0	0	R	←D→							
LARK	Load auxiliary register immediate	1	1	0	1	1	1	0	0	R	←K→								
LARP	Load auxiliary register pointer immediate	1	1	0	1	1	0	1	0	0	1	0	0	0	0	K	K		
LDP	Load data memory page pointer	1	1	0	1	1	0	1	1	1	←D→								
LDPK	Load data memory page pointer immediate	1	1	0	1	1	0	1	1	1	0	0	0	0	0	K	K		
LST	Load status register	1	1	0	1	1	1	0	1	1	←D→								
LT	Load T Register	1	1	0	1	1	0	1	0	1	←D→								
LTA	LTA combines LT and APAC into one instruction	1	1	0	1	1	0	1	1	0	1	←D→							
LTD	LTD combines LT, APAC, and DMOV into one instruction	1	1	0	1	1	0	1	1	←D→									
MAR	Modify auxiliary register and pointer	1	1	0	1	1	0	1	0	0	1	←D→							
MPY	Multiply with T Register, store product in P Register	1	1	0	1	1	0	1	1	0	1	←D→							
MPYK	Multiply T Register with immediate operand; store product in P Register	1	1	1	0	0	←K→												
NOP	No operation	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0		
OR	OR with accumulator	1	1	0	1	1	1	0	1	0	←D→								
OUT	Output data to port	2	1	0	1	0	0	←PA→		I	←D→								
PAC	Load accumulator from P Register	1	1	0	1	1	1	1	1	1	0	0	0	1	1	1	0		
POP	Pop stack to accumulator	2	1	0	1	1	1	1	1	1	0	0	1	1	1	0	1		
PUSH	Push stack from accumulator	2	1	0	1	1	1	1	1	1	0	0	1	1	1	0	0		
RET	Return from subroutine	2	1	0	1	1	1	1	1	1	0	0	0	1	1	0	1		
ROVM	Reset overflow mode	1	1	0	1	1	1	1	1	1	0	0	0	1	0	1	0		
SACH	Store high-order accumulator bits with shift	1	1	0	1	0	1	1	←X→		I	←D→							
SACL	Store low-order accumulator bits	1	1	0	1	0	1	0	0	0	1	←D→							
SAR	Store auxiliary register	1	1	0	0	1	1	0	0	R	←D→								
SOVM	Set overflow mode	1	1	0	1	1	1	1	1	1	0	0	0	1	0	1	1		
SPAC	Subtract P Register from accumulator	1	1	0	1	1	1	1	1	1	0	0	1	0	0	0	0		
SST	Store status register	1	1	0	1	1	1	1	0	0	1	←D→							
SUB	Subtract from accumulator with shift	1	1	0	0	0	1	←S→		I	←D→								

(Continued)

INSTRUCTION SET SUMMARY (CONCLUDED)

MNEMONIC	DESCRIPTION	NO. CYCLES	NO. WORDS	OPCODE REGISTER																
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
SUBC	Conditional subtract (for divide)	1	1	0	1	1	0	0	1	0	0	1	←D→							
SUBH	Subtract from high-order accumulator bits	1	1	0	1	1	0	0	0	1	0	1	←D→							
SUBS	Subtract from accumulator with no sign extension	1	1	0	1	1	0	0	0	1	1	1	←D→							
TBLR	Table read from program memory to data RAM	3	1	0	1	1	0	0	1	1	1	1	←D→							
TBLW	Table write from data RAM to program memory	3	1	0	1	1	1	1	1	0	1	1	←D→							
XOR	Exclusive OR with accumulator	1	1	0	1	1	1	1	0	0	0	1	←D→							
ZAC	Zero accumulator	1	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	1		
ZALH	Zero accumulator and load high-order bits	1	1	0	1	1	0	0	1	0	1	1	←D→							
ZALS	Zero accumulator and load low-order bits with no sign extension	1	1	0	1	1	0	0	1	0	1	0	1	←D→						

CROSS-ASSEMBLER DIRECTIVES

ABSOLUTE ORIGIN **AORG**

AORG places a value in the location counter and defines the succeeding locations as absolute.

Syntax: [**<label>**] AORG [**<wd-exp>**] [**<comment>**]

RELOCATABLE ORIGIN **RORG**

RORG places a value in the location counter and defines the succeeding locations as program relocatable.

Syntax: [**<label>**] RORG [**<exp>**] [**<comment>**]

DUMMY ORIGIN **DORG**

DORG places a value in the location counter and defines the succeeding locations as a dummy section. No object code is generated in a dummy section.

Syntax: [**<label>**] DORG **<exp>** [**<comment>**]

BLOCK STARTING WITH SYMBOL **BSS**

BSS first assigns the label, if present, and then advances the location counter by the value of the expression.

Syntax: [**<label>**] BSS **<wd-exp>** [**<comment>**]

BLOCK ENDING WITH SYMBOL **BES**

BES first advances the location counter by the value of the expression and then assigns the label, if present.

Syntax: [**<label>**] BES **<wd-exp>** [**<comment>**]

DATA SEGMENT **DSEG**

DSEG places a value in the location counter and defines succeeding locations as data relocatable.

Syntax: [**<label>**] DSEG [**<comment>**]

DATA SEGMENT END **DEND**

DEND terminates a block of data-relocatable code by placing a value in the location counter and defining succeeding locations as program-relocatable.

Syntax: [**<label>**] DEND [**<comment>**]

COMMON SEGMENT **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).

Syntax: [**<label>**] CSEG [**'<string>'**] [**<comment>**]

COMMON SEGMENT END **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.

Syntax: [**<label>**] CEND [**<comment>**]

PROGRAM SEGMENT **PSEG**

PSEG places a value in the location counter and defines succeeding locations as program-relocatable.

Syntax: [**<label>**] PSEG [**<comment>**]

PROGRAM SEGMENT END **PEND**

PEND places a value in the location counter and defines succeeding locations as program-relocatable. (Since PEND properly appears only in program-relocatable code, the relocation type of succeeding locations remains unchanged.)

Syntax: [**<label>**] PEND [**<comment>**]

OUTPUT OPTIONS **OPTION**

OPTION selects several options for the assembler listing output.

Syntax: [**<label>**] OPTION **<option-list>** [**<comment>**]

PROGRAM IDENTIFIER **IDT**

IDT assigns a name to the object module produced.

Syntax: [**<label>**] IDT [**'<string>'**] [**<comment>**]

(Continued)



November 1983
Revision B
1603770-9701
Printed in the U.S.A.

Creating useful products
and services for you.

SPRU002B