MDEX 9900 ASSEMBLER

USER GUIDE

by John Walker

Marinchip Systems

Mill Valley, CA 94941

Marinchip 9900 Assembler
User Guide
by John Walker

Revised July 1978

Marinchip Systems 16 St. Jude Road Mill Valley, CA 94941 ■ (415) 383-1545

Marinchip 9900 Assembler User Guide

Table of contents

Table of contents

Table of contents

Marinchip 9900 Assembler User Guide

1. Introduction

The Marinchip 9900 Assembler is a relocatable assembler for the Marinchip 9900 computer. It runs on the 9900 computer, accepts source files in a format essentially compatible with the Texas Instruments assembler, and outputs relocatable code completely compatible with T.I. format. The assembler optionally produces an assembly listing, which may be sent to a file or to a printer or console. Lines containing errors will be printed on the user console.

The assembler allows "address expressions", which permit a symbol to be equated to a fully general 9900 address. Such a symbol can be used on instruction fields, and permits the later redefinition of storage use simply by changing the value the symbol was equated to. For example, static storage may easily be changed to storage based on an index register in this manner.

The assembler allows conditional assembly of code depending on assembly-time variables. This allows programs to be easily configured for various code options at assembly time. Conditional code sequences are written with a simple IF - ELSE - ENDF construction, and may be nested to arbitrary depth.

The assembler runs under any Marinchip operating system, and uses the operating system for all its I/O. In addition, since the operating system performs all memory allocation, the assembler automatically uses all configured memory without modification.

1.1. Conventions

In this manual, items which are elements of the language are written in UPPER CASE TYPE. All examples of assembly programs will be in upper case. The assembler itself is insensitive to the case of text, so the programmer need not follow this convention. Items supplied by the programmer will be indicated by <corner brackets>, with text inside the brackets describing the item to be coded. Optional items will be enclosed in [square brackets]. The description of the item will make clear what action is taken when specifications are omitted.

2. Using the assembler

2.1. Calling the assembler

The assembler is invoked from operating system command mode by a command of the form:

ASM <reloc>=<source>[,<listing>]

where <reloc> is the name of the file where the relocatable output of the assembly is to be written, <source> is the name of the assembly language program to be assembled, and the optional <listing> is the file where the assembly listing (described below) is written. If the <listing> specification and the comma that precedes it are omitted, the assembler will make no listing, but will print lines on which errors are detected.

For example, to assemble the file MYFILE, place the relocatable object code in file MYOBJ, and send the listing to the file LISTNG, one would use:

ASM MYOBJ=MYFILE,LISTNG

To produce no listing, one would use:

ASM MYOBJ=MYFILE

The files used with the assembler may be either device files or disc files. The listing file, in particular, is frequently sent to the console

or a line printer by specifying the name of that device file.

2.2. Assembly listing format

The assembler optionally generates a listing which includes the source code being assembled and a hexadecimal representation of the object code generated. The format of the listing is explained below.

2.2.1. Error flags

If any errors have been detected on this line of the assembly, one or more single-character error flags will be printed which indicate the nature of the error. In some rare cases, the error flag may refer to an error on the line preceding the line on which the flag was printed. The meanings of the flags are as follows:

D — Duplicate. The item in the label field is being redefined. This normally indicates a duplicate label in the program.

E — Expression error. This error indicates an improper expression, or a general syntax error on an instruction or directive.

I — Instruction error. The item in the operation field was not a known instruction or directive.

L — Level overflow. An expression was too complicated. It should be rewritten without so many nested parentheses.

R — Relocation error. This generally results from use of a relocatable value when a nonrelocatable value is required, or from an improper mix of relocatable and nonrelocatable values in an expression.

T — Truncation. A value was too large for the field in which it had to be placed. For example, using a value greater than 15 when a register number is required will cause this error.

U — Undefined. This is not an error, but simply flags a reference to an external symbol. It is issued simply to make external references easier to spot when reading code.

V — Value error. A string or numeric value is too big or badly formed.

$ — Internal error. This error indicates an internal assembler error. Please submit the source program to Marinchip Systems so that the error may be corrected. PLEASE ALSO SUBMIT ANY FILES INCLUDED BY COPY STATEMENTS IN THE PROGRAM WHICH CAUSED THE ERROR.

The occurrence of any error flag (other than the "U" flag) on a line will cause that line to be printed on the system console, even if the listing has not been requested or is being sent to another file. If the listing is being sent to another file, the error line will be printed both in the listing file and on the system console.

2.2.2. Location counter

If this line generates code, the value of the location counter at the start of the line will be printed. The location counter will be edited as four hexadecimal digits.

2.2.3. Object code

If code is generated by the line, it will be printed in hexadecimal format. Words will be separated by spaces, and a maximum of three words of code (6 bytes) will be printed on a line. If a line generates more

than three words of code, the code will be continued onto as many additional lines as are required. If a line generates an odd number of bytes, only the actual number of bytes generated will be printed.

2.2.4. Line number

The line number of each line in the source program will be printed, followed by a period. If the line being listed is from a COPY file (where listing was specified on the COPY directive), the line number printed will be the line number of the COPY statement in the original program. In this case, the line number will be followed by an asterisk (*) to identify the code as having come from a COPY file.

2.2.5. Source code

The input source line will be listed exactly as read by the assembler.

3. The assembly language

The source language accepted by the assembler consists of two major types of statements: assembler directives and machine instructions. Assembler directives are statements that control the assembler iteslf, and also perform such functions as generation of constant numeric data. Machine instructions are mnemonics for the hardware instructions of the M9900 CPU itself, and are translated into the appropriate binary codes by the assembler and placed in the output file.

3.1. Source language syntax

Input to the assembler is written in three major fields, the LABEL field, the OPERATION field, and the OPERAND field. Any information following the operand field is ignored, and may be used for comments. All input is totally free format: information need not be aligned into specific columns. In order to make programs more readable, it is recommended that the operation, operand, and comment fields be aligned. The standard columns used in all Marinchip software are:

Field	Column
Label	1
Operation	11
Operand	21
Comment	41

If a line has a label, it must start in column one. If column one of a line is blank, it is considered to be an unlabeled line, and the first word on the line is interpreted as an operation.

3.1.1. Labels

All labels defined by the programmer must start with an alphabetic ASCII character, and consist only of alphabetic and numeric characters, or the special character dollar sign ($). Examples of proper labels are:

RUNKA BLEEP1 C$53F1$ A$BOOGIE

Examples of improper labels are:

6GOBBLE $ZAP 23SKIDOO

Labels may be up to 80 characters in length, and all characters are significant. The case of alphabetic characters is ignored in comparing labels, so the two labels:

TESTING and testing

are considered identical by the assembler.

3.1.2. Numbers

Numbers accepted by the assembler may be either decimal or hexadecimal. Any number that starts with a leading zero is considered to be hexadecimal, and the characters "A" through "F" are accepted as part of it. Numbers must be less than 65535 decimal, or OFFFF hexadecimal. A sign m generate its two's complement negative ay representation. The sign is not actually part of the number, as a signed number is considered an expression and evaluated as any other expression: see the section "Expressions" below.

3.1.3. Strings

A string is an arbitrary group of characters enclosed in quote marks. Either single quotes (') or double quotes (") may be used. Strings must be less than 80 characters in length. Two consecutive quotes will cause the second quote to be inserted in the string as a normal character. Hence the string:

'It just ain''t fair.'

will be interpreted by the assembler as the characters:

It just ain't fair.

Strings are used in various contexts by the assembler. A one or two character string may be used wherever a number appears, and has the value of a one or two byte ASCII representation of the quoted characters.

3.1.4. Expressions

The element on which the assembler operates is the expression. An expression is composed of numbers, labels, and operators which act on the operands (numbers, labels, and other expressions). An expression may be as simple as a single number, or as complex as imaginable.

Expressions are divided into two major categories: numeric expressions and address expressions. A numeric expression is composed only of nonrelocatable values and numeric operators, while an address expression may contain values with relocation attached, and may be composed through the use of addressing operators. These terms and the distinctions involved should become more clear as the following sections are read.

3.1.4.1. Relocatability

The assembler is capable of generating either relocatable or absolute code for output. Absolute code is able to be loaded for execution only at the address for which it was assembled, but relocatable code is able to be processed for loading at any address by the Linker. In order to allow this, the assembler must, in relocatable code, distinguish between absolute binary values which are not associated with program addresses and hence are invariant, and values which represent program locations and which must be adjusted to reflect the actual address at which the program is loaded. A pure numeric value is referred to as a nonrelocatable quantity, and a program address is referred to as a relocatable quantity. In a typical program, numbers and labels set equal to numbers would be nonrelocatable, while all program labels would be relocatable. To further complicate things, the assembler can also generate absolute code: when in this mode program labels are also absolute.

Because the complete value of a relocatable quantity is not known at assembly time, restrictions are imposed on the operations in which a relocatable quantity may participate. It is legal to:

1. Add a constant to a relocatable quantity.

2. Subtract a constant from a relocatable quantity.
3. Subtract two relocatable quantities yielding an absolute quantity.
4. Compare two relocatable quantities with any of the relational operators.

All other operations will cause an "R" flag (Relocation error) if performed with relocatable quantities.

3.1.4.2. Numeric operators

The numeric operators are as follows:

( )	Expression brackets
+ - ~~	Unary Plus, Minus, NOT
<< >>	Shift left, Shift right
**	Logical AND
++ --	Logical OR, XOR
* /	Multiply, Divide
+ -	Add, Subtract
= > <	Equal, Greater, Less

The operators are executed in the order given above. For example, in the expression:

```
A+B*C
```

B and C will be multiplied, and the product added to A. This occurs because the "*" operator has a higher priority than the "+" operator (according to the above table), and hence is executed first. Operators listed in the same line above will be evaluated left to right. Since the parentheses are first in the table, subexpressions within parentheses will always be evaluated first. For example:

```
(A+B)*C
```

will cause A and B to be added, and the sum multiplied by C.

The normal arithmetic operators (+, -, *, and /) operate on 16 bit signed numbers. The division operator (/) discards the remainder from the division. The unary minus operator takes the two's complement negative of its operand (e.g., -A).

The logical operators (++, --, and **) perform bit-by-bit logical operations on their 16 bit operands. The functions performed are OR, XOR, and AND respectively. If the "--" operator is used as a unary operator (e.g., --F), the one's complement negative (inversion of all bits) of the operand will be the result.

The shift operators (<<, >>) logically shift their left operand left or right, respectively, the number of bits in the right operand, modulo 16. For example:

```
1<<2 is 4
16>>3 is 2
```

The relational operators (=, >, and <) return 1 if the relation between the two operands is true and zero otherwise. Note that the relational operators can be used with the logical operators to form complex logical expressions.

3.1.4.3. Address operators

The address operators are used to build an address value from one or more operands. The form of the address operators are as follows:

*<reg>	Indirect through <reg>
*<reg>+	Indirect through <reg>, increment
<val>(<reg>)	Index <val> by <reg>
@<val>	Direct address <val>

The first construction, *<reg>, causes the expression <reg>, which must be nonrelocatable and have a value between 0 and 15, to be treated as a register which contains the address of an operand.

The second construction, *<reg>+, is identical to the first, except that following the reference to the register for the address of the operand it will be incremented by the length of the operand (1 if used in a byte instruction, 2 if used in a word instruction).

The third construction, <val>(<reg>), is used to address an operand whose address is the location <val> with the contents of register <reg> added to it. <val> may be any value, relocatable or not, but <reg> must be nonrelocatable and between 0 and 15.

The fourth construction, @<val>, simply indicates that <val> is to be used as the address of the operand. This is seldom necessary because the assembler automatically generates a direct address for any relocatable operand or absolute operand greater than 15. It is always permissible, though, and required to directly address an absolute address between 0 and 15

The address operators may be used in any instruction where a "general address" operand is required, and identify the addressing mode to be used with the operand. If none of these operators appear, the mode generated will be the contents of the register if the operand is absolute and between 0 and 15, and direct otherwise. The address operators may also be used with the EQU directive to equate a simple name to a complex address expression. This can be used to simplify coding and to make programs easier to modify.

3.1.4.4. Predefined symbols

At the start of an assembly, the names for the workspace registers are automatically defined by the assembler. The labels R0 through R15 are equated to 0 through 15.

3.1.4.5. Programmer-defined symbols

Normally, the appearance of a label in column 1 of a line will cause the label to be equated to the current location counter. This is the mechanism by which labels are given to program and data addresses. The EQU (equate) directive allows the programmer to define a label equal to any numeric or address expression. The directive:

<label> EQU <expression>

will set <label> equal to the <expression>. Henceforth in the assembly, the appearance of the <label> will be equivalent to the value of the <expression> at the time the EQU statement was evaluated. Examples of the EQU directive are:

```
TPORT   EQU     020
ALTENT  EQU     ENTRY+4
STKTOP  EQU     STACK(R1)
```

3.1.5. Comments

Comments may be placed following the last field evaluated by an instruction or directive. The last field is terminated by a space following the last item, and after that space any sequence of characters may be included as a comment. The assembler will always treat the sequence "period space" (". ") as the end of line, except when it appears inside a quoted string. This convention permits comments to be included on lines where a parameter is omitted (for example, END or RORG), and also allows lines which are all comment. Lines which are to be all comment must have a period space as their first two nonblank characters. The period space may begin in column 1, or some later column. Examples of comments are as follows:

```
MOV       R0,HEADER          THIS IS A COMMENT
This line is all comment
RORG                         RELOCATABLE CODE
          . MORE COMMENT
```

3.2. Object code linking

The assembler produces relocatable object code which is turned into an executable program by the Linker. The relocatable code allows programs to define labels accessable to other programs, and to reference code and data defined in other assemblies, then combined by the linker into an executable program.

3.2.1. External symbol definition

To define a symbol as an external reference, an asterisk (*) must follow the appearance of the symbol in the label field. For example, to externally define the entry point to a subroutine, one might code:

```
ZONK*     MOV       R11,SAVEIT          Save return point
```

The label ZONK will be externally defined, and may be referenced by other programs when the Linker is used to produce an executable program. The label ZONK may be referenced within the assembly just like any other non-externally defined label. The format of relocatable code restricts the significance of external symbols to 6 characters. As a result, even though within the assembly labels may be up to 80 characters long and are significant to their full length, externally defined labels must be unique in their first 6 characters.

3.2.2. External symbol reference

An external symbol is referenced simply by using it in the assembly. For example, another program might call the subroutine ZONK defined above with the statement:

```
          BL        ZONK                Zonk the data
```

An external symbol may appear wherever a 16 bit relocatable quantity may be used. However, an expression such as:

```
ZONK+2
```

is not permitted. An external symbol must be simply used as its defined value (this is a restriction of the relocatable code format). Note that since external symbols are unique only to the first 6 characters, the following symbols:

```
BOGGLE BOGGLEKLUNK BOGGLEBARGLE
```

will all reference the same external symbol, "BOGGLE". Any line which references an external symbol will be flagged with a "U" in the first column of the assembly listing. This is not an error, only an indication that this line references an external symbol.

3.3. Conditional assembly

The assembler allows selection of the code to be assembled based upon the value of assembly-time expressions. This is accomplished through use of the IF, ELSE, and ENDF directives (described in more detail in the "Assembly directives" chapter below). The IF directive takes an expression as an operand. That expression is evaluated, and if nonzero the code following the IF is assembled. If zero, the code between the IF and the matching ENDF will be skipped by the assembler. While skipping code, the assembler will scan for IF and ENDF lines, so that only the ENDF that matches the IF that turned off the assembly will restore the generation of code. This allows IF - ENDF pairs to be nested to any

desired depth. The ELSE directive turns code off if encountered while assembling code, and turns code back on if encountered while skipping code at the outermost IF level. Note that since the IF directive turns code on if its operand is nonzero and off otherwise, and the relational and logical operators follow the convention that 1 means TRUE and 0 means FALSE, IF directives may be coded using relational and logical operators with their normal mathematical meaning. Examples of conditional assembly follow:

```
MAXMEM   EQU     04000           Maximum memory size
SIGNON   EQU     1               Print Sign-on if nonzero
.        .       .
.        .       .
         IF      SIGNON
         JSYS    PRSIGN          Print sign-on message
         ENDF
.        .       .
.        .       .
         IF      SIGNON

PRSIGN   BYTE    WRITE$,0        Packet to print signon
         DATA    0,SIGNM,SIGNL,0

SIGNM    TEXT    "APL\9900 "
         IF      MAXMEM>02000
         TEXT    "(Large version)"
         ELSE
         TEXT    "(Small version)"
         ENDF
         BYTE    0D              Carriage return
SIGNL    EQU     $-SIGNM         Length of message
         EVEN
         ENDF
```

4. Assembly directives

Assembly directives are statements in the assembly language that do not correspond to machine instructions. Some of these statements generate data, others simply specify information used by the assembler. A <label> may be specified on any assembly directive. Normally, the <label> will simply be set equal to the value of the location counter prior to processing the directive. In cases where some other action is taken regarding the <label> this will be noted in the description of the directive. The directives are discussed in alphabetical order.

4.1. AORG - Absolute origin

```
<label>  AORG    <expression>
```

The location counter is set to the value of <expression> and the assembler begins generating absolute code. Absolute code is not relocated by the Linker, so data generated following the AORG directive will be placed starting at the address specified by <expression>, regardless of where the program is loaded. If a <label> is specified, it will be set equal to the new location counter following the processing of the AORG directive.

4.2. BES - Block ending with symbol

```
<label>  BES     <expression>
```

The number of bytes indicated by <expression> will be reserved by adding the <expression> to the location counter. The <label> will be set equal to the first address following the reserved area. This directive is used for reserving tables which are addressed in order of descending address. For example, if the location counter were at 0200 and the directive:

```
STACK    BES     040
```

were processed, the label STACK would be set equal to 0240.

4.3. BSS - Block starting with symbol

<label> BSS <expression>

A block of storage with the length in bytes specified by <expression> is reserved by adding <expression> to the location counter. If a <label> is specified, it is set to the address of the first byte of the block reserved. This is the normal means by which blocks of memory are reserved.

4.4. BYTE - Byte generation

<label> BYTE <expression>,<expression>,...

Each <expression> specified is output in a single byte of data. If the <expression> is relocatable or has a value greater than 255 a truncation error will be flagged. Any number of bytes may be generated by specifying multiple expressions.

4.5. COPY - Copy source file

<label> COPY <string>,<expression>

The <string> specifies the name of a file. That file is read and included in the assembly. Text from the file is processed as if it came from the file being assembled, except that it is not listed unless the <expression> is specified and has a nonzero value. The COPY directive is normally used to include common definitions or frequently used pieces of code in multiple assemblies without the need for physically including the code in each assembly.

4.6. DATA - Define 16 bit data

<label> DATA <expression>,<expression>,...

The DATA directive generates one 16 bit word for each <expression> specified. The <expressions> may be either relocatable or nonrelocatable.

4.7. DORG - Dummy origin

<label> DORG <expression>

The DORG directive sets the location counter to the absolute value specified by <expression>, places the assembler in absolute mode, and turns off the actual generation of code. DORG is most often used when defining data structures. A "DORG 0" directive can be used to turn on dummy assembly and set the location counter to zero. The components of a data structure can then be reserved by BSS directives, and then normal assembly can be resumed with an AORG or RORG directive. This allows use of the assembler to allocate storage within a data structure without explicit definition by the programmer. If a <label> is specified, it will be set equal to the new location counter value, in other words, to <expression>.

4.8. ELSE - Conditional assembly alternative

<label> ELSE

When the ELSE directive is encountered, and the IF nesting level is one (indicating that the outermost IF is in effect), if code was turned off by the IF it is turned back on. If the code was turned on by the IF, the ELSE turns it off. This allows alternate code to be generated by the sequence: IF - ELSE - ENDF.

4.9. END - End of program

<label> END [<expression>]

The END directive identifies the end of an assembly and must be the final line in any file to be assembled. If no <expression> is specified as an operand, the program will be generated with no starting address specified. If an <expression> is specified, a starting address will be generated, making this a main program. The <expression> will normally simply be the label on the line containing the first instruction to be executed in the program.

4.10. ENDF - End conditional assembly

<label> ENDF

If encountered while skipping code because of an IF or ELSE directive, the IF nesting is decremented. If zero, indicating that the ENDF matches the IF or ELSE that turned off the processing of code, the assembly resumes with the next line. If nonzero, this ENDF matches an IF within a region of code turned off by an outer IF, and the nesting level is simply decremented. If the ENDF is encountered during normal assembly, it is simply ignored, since the block of code it terminates was turned on.

4.11. EQU - Define assembly-time variable

<label> EQU <expression>

The <label> is set equal to the value of the <expression> and may be used henceforth in the assembly to represent the value of the <expression>. The <expression> may be either a numeric or address expression, and may be either relocatable or nonrelocatable (subsequent use of the label, of course, may occur only in a context where the <expression> itself would be permissible). A <label> may be redefined at any time by an EQU statement.

4.12. EVEN - Force word alignment

<label> EVEN

If the location counter is odd, one byte will be reserved to force it to be even. This directive is normally used following a block of BYTE or TEXT data, or byte-aligned blocks reserved by BSS or BES, to insure that the location counter is at a word boundary before generating instructions or data which require word alignment.

4.13. IDT - Identify program

<label> IDT <string>

The program identification is set equal to the <string> specified. The program identification may be up to 8 characters in length, and is printed by the Linker when a memory map is requested. If no IDT statement is supplied in the program, the value "NO IDT !" will be used.

4.14. IF - Conditional assembly

<label> IF <expression>

The <expression>, which must be a nonrelocatable quantity, is evaluated. If nonzero, the IF directive is ignored and the assembly continues normally. If zero, the assembler skips all subsequent lines until a matching ELSE or ENDF directive is found. While skipping code, the assembler will scan for IF and ENDF directives so that nested IF - ENDF sequences will be correctly processed. Lines skipped by an IF directive

will be listed in the output listing (if one is being generated), but no code will be assembled for them.

4.15. LIST - Enable assembly listing

<label> LIST

If the assembly listing has been suppressed by a preceding UNL directive, it will be resumed.

4.16. PAGE - Eject page in assembly listing

<label> PAGE

If an assembly listing is being generated, a page eject will occur following the line containing the PAGE directive. This may be used to cause separate pieces of a program to be listed on separate pages.

4.17. RORG - Relocatable origin

<label> RORG <expression>

The RORG directive sets the assembler producing relocatable code, and sets the relocatable location counter to the value of <expression>. If <expression> is omitted, the location counter will be set to the length of all relocatable code previously generated in the program. This is useful to return to relocatable code generation following an absolute block (AORG) or dummy block (DORG). When an assembly starts, the assembler assumes a:

```
RORG    0
```

directive is in effect. If a <label> is specified, it will be set equal to the value of the location counter following processing of the RORG directive.

4.18. TEXT - Generate character data

<label> TEXT [-]<string>

The TEXT directive generates bytes containing the ASCII character codes for each character in a <string>. If the <string> is preceded by a minus sign, the two's complement of the final character will be generated (to serve as an end of string indication). Note that TEXT generates only as many bytes as there are characters in the string, and that as a result an odd number of bytes may be generated. If the data following the TEXT directive must be aligned on word boundaries, an EVEN directive should follow the TEXT line.

4.19. TITL - Title for assembly

<label> TITL <string>

The TITL directive specifies a title to be printed on each page of the assembly listing. The <string> will be printed at the head of each page of the assembly listing until the end of the assembly or until superceded by another TITL directive.

4.20. UNL - Turn off assembly listing

<label> UNL

The assembly listing will be suppressed. The listing may be later

resumed, if desired, by a LIST directive.

5. Instruction descriptions

The following sections describe the machine instructions of the M9900 CPU, giving the symbolic assembler syntax used to generate each instruction. The instructions are grouped by general type: each instruction within a type shares a common syntax with all others of the same type.

Within the instruction descriptions, the following nomenclature will be used for items supplied by the programmer when coding an instruction:

<gas>	General address of source
<gad>	General address of destination
<wa>	Workspace register address
<iop>	Immediate operand
<destination>	Program address
<count>	Bit count from 0 to 16
<index>	XOP index from 0 to 15

In coding these fields, any address expression may be used for a general address (<gas> or <gad>). A workspace register address must be a nonrelocatable expression between 0 and 15. An immediate operand may be a relocatable or nonrelocatable value, but may not be an address expression. A <destination> address is a value whose relocation must be the same as the code currently being generated. A <count> must be nonrelocatable and between 0 and 16, and an <index> must be nonrelocatable and between 0 and 15.

5.1. Two general address instructions

The two general address instructions take two general address operands. The first is referred to as the "source", and the second is referred to as the "destination". The instruction normally operates upon the two operands and stores its result, if any, into the destination.

5.1.1. A - Add words

A <gas>,<gad>

The source operand word is added to the destination operand and the result is stored into the destination operand. The result is compared against zero and the condition bits are set accordingly. If there is a carry out of the add, the carry bit is set, and if the result cannot be represented as a 16 bit integer, the overflow flag is set.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: A000

Format: 11010ttddddttssss
------ Destination
------ Source

5.1.2. AB - Add bytes

AB <gas>,<gad>

The source byte operand is added to the destination byte operand and the result is stored into the destination byte operand. When the source or destination operand is a workspace register, the byte addressed will be the upper byte in the register (bits 15 - 8). The other byte in the destination address will be unchanged. The result of the addition is compared to zero and the status bits are set accordingly. If there is a carry out of the bytes added, the carry bit will be set, and if the result of the addition is not representable as an 8 bit two's complement integer,

the overflow bit will be set. The parity status bit is set when the number of bits in the result is odd, and is cleared otherwise.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow, parity.

Op code: B000

Format: 1011ttddddttssss
------ Destination
------ Source

5.1.3. S - Subtract words

S <gas>,<gad>

The source operand is subtracted from the destination operand and the result is stored in the destination operand. The result is compared with zero and the status bits are set resulting from the operation. When there is a carry out of the most significant bit, the carry bit is set, and when the difference cannot be represented as a 16 bit integer, the overflow bit is set.

Status bits affected; Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 6000

Format: 0110ttddddttssss
------ Destination
------ Source

5.1.4. SB - Subtract bytes

SB <gas>,<gad>

The source byte operand is subtracted from the destination byte operand. The result is stored in the destination byte operand. The result is compared with zero and the status bits are set accordingly. If there is a carry out of the most significant bit of the byte, the carry bit will be set, and if the difference cannot be represented as an 8 bit integer, the overflow bit will be set. If there are an odd number of one bits in the result byte, the parity status bit will be set, otherwise the odd parity status bit will be cleared. If either the source or destination operand is a workspace register, the high byte (bits 15-8) will be used as the operand. The other byte in the word containing the destination operand will not be changed by this instruction.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow, parity.

Op code: 7000

Format: 0111ttddddttssss
------ Destination
------ Source

5.1.5. C - Compare words

C <gas>,<gad>

The source operand is compared to the destination operand and the status bits are set as a result of the comparison. The arithmetic greater bit is set if the source operand is greater than the destination when both are considered as 16 bit two's complement integers. The logical greater bit is set if the source operand is greater than the destination operand when both are considered as 16 bit unsigned numbers. The equal bit is set if the two operands are bit-for-bit equal.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 8000

Format: 1000ttddddttssss
------ Destination
------ Source

5.1.6. CB - Compare bytes

CB <gas>,<gad>

The source byte operand is compared to the destination byte operand. The arithmetic greater than, logical greater than, and equal bits are set as described above for the C instruction, except that the operands are considered as 8 bit signed and unsigned quantities, respectively. In addition, the parity bit is set if the source operand contains an odd number of one bits.

Status bits affected: Logical greater than, arithmetic greater than, equal, parity.

Op code: 9000

Format: 1001ttddddttssss
------ Destination
------ Source

5.1.7. MOV - Move word

MOV <gas>,<gad>

The source operand is moved to the destination. The result is compared with zero and the status bits are set as a result of the comparison.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: C000

Format: 1100ttddddttssss
------ Destination
------ Source

5.1.8. MOVB - Move byte

MOVB <gas>,<gad>

The source byte operand is moved to the destination byte. The other byte in the destination is unaffected. If the source or destination is a workspace register, the upper byte (bits 15-8) will be addressed. The byte moved is compared against zero and the condition bits are set as a result of the comparison. If the byte moved contains an odd number of one bits, the parity bit will be set, otherwise it will be cleared.

Status bits affected: Logical greater than, arithmetic greater than, equal, parity.

Op code: D000

Format: 1101ttddddttssss
------ Destination
------ Source

5.1.9. SOC - Set ones corresponding

SOC <gas>,<gad>

For each one bit in the source operand, the corresponding bit in the destination operand is set to one. Bits in the destination that correspond to zero bits in the source will be unchanged. The result is compared to zero and the status bits are set accordingly. This instruction is equivalent to a logical OR of the source with the destination.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: E000

Format: 1110ttddddttssss
------ Destination
------ Source

5.1.10. SOCB - Set ones corresponding byte

SOCB <gas>,<gad>

For each one bit in the byte source operand, the corresponding bit in the destination byte operand is set to one. Bits in the destination that correspond to zero bits in the source will be unchanged, as will the other byte in the destination word. If the source or destination operand are workspace registers, the upper byte (bits 15-8) will be used as the operand. The result byte is compared to zero and the status bits are set accordingly. If the result byte has an odd number of one bits, the parity bit will be set, and cleared otherwise. This instruction is equivalent to a logical OR of the source and destination bytes.

Status bits affected: Logical greater than, arithmetic greater than, equal, parity.

Op code: F000

Format: 1111ttddddttssss
------ Destination
------ Source

5.1.11. SZC - Set zeroes corresponding

SZC <gas>,<gad>

For each one bit in the source operand, the corresponding bit in the destination operand is cleared to zero. Bits corresponding to bits in the source operand will be unchanged. The result of the operation will be compared to zero and the status bits will be set accordingly. This instruction is equivalent to an AND with the one's complement of the source operand.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 4000

Format: 0100ttddddttssss
------ Destination
------ Source

5.1.12. SZCB - Set zeroes corresponding byte

SZCB <gas>,<gad>

For each one bit in the source byte operand, the corresponding bit in the

destination byte operand will be cleared to zero. Bits in the destination that correspond to zero bits in the source will be unchanged, as will be the other byte in the destination word. The result byte will be compared to zero and the status bits will be set according to the result of the comparison. If the result byte contains an odd number of one bits, the parity bit will be set, otherwise it will be cleared. If the source or destination operands are workspace registers, the byte addressed will be the upper byte (bits 15-8).

Status bits affected: Logical greater than, arithmetic greater than, equal, parity.

Op code: 5000

Format: 0101ttddddttssss
------ Destination
------ Source

5.2. One general address instructions

The one general address instructions contain a single general address specification. The source operand is obtained from this address and the result, if any, is stored back at the same address.

5.2.1. B - Branch

B <ga>

The program counter is set equal to the address to which the general address evaluates. Execution will continue at that address.

Status bits affected: none.

Op code: 0440

Format: 0000010001ttdddd
------ Destination

5.2.2. BL - Branch and link

BL <ga>

Workspace register 11 (R11) is set equal to the address following the BL instruction, and the program counter is set equal to the address to which the general address <ga> evaluates. This is the normal subroutine call instruction.

Status bits affected: none.

Op code: 0680

Format: 0000011010ttdddd
------ Destination

5.2.3. BLWP - Branch and load workspace pointer

BLWP <ga>

The workspace pointer is set equal to the word at the address designated by <ga>, and the program counter is loaded with the contents of the following word. Within the new workspace, register R13 is set equal to the previous workspace pointer, R14 is set equal to the address following the BLWP instruction, and R15 is set equal to the status register. This instruction enters a subroutine which operates with a different register set. The loading of the calling context into the registers of the new workspace permits the subroutine to return simply by executing a RTWP instruction.

Status bits affected: none.

Op code: 0400

Format: 0000010000ttdddd
------ Destination

5.2.4. CLR - Clear

CLR <ga>

The word at address <ga> is cleared to zero.

Status bits affected: none.

Op code: 04C0

Format: 0000010011ttdddd
------ Destination

5.2.5. SETO - Set to all ones

SETO <ga>

The word at address <ga> is set to all ones (0FFFF).

Status bits affected: none.

Op code: 0700

Format: 0000011100ttdddd
------ Destination

5.2.6. INV - Invert

INV <ga>

The word at address <ga> is one's complemented. That is, all zero bits are set to one and all one bits are set to zero. The result is compared to zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 0540

Format: 0000010101ttdddd
------ Destination

5.2.7. NEG - Negate

NEG <ga>

The word at address <ga> is negated (two's complement negative). This is accomplished by inverting all bits of the operand and then adding one. The result of the negation is compared to zero and the status bits are set accordingly. In two's complement representation there is one more negative number than there are positive numbers. The number -32768, hexadecimal 08000, has no positive counterpart. An attempt to negate this number will set the overflow status bit.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 0500

Format: 0000010100ttdddd

------ Destination

5.2.8. ABS - Absolute value

ABS <ga>

The absolute value of the operand at <ga> is taken and stored back at <ga>. The absolute value is taken by examining the sign bit of the operand. If zero, the operand is left unchanged. If one, the operand is two's complemented. The original operand is compared against zero and the status bits are set accordingly. If the original operand is -32768 (08000 hexadecimal), a negative number which has no positive counterpart, the overflow status bit will be set.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 0740

Format: 0000011101ttdddd
------ Destination

5.2.9. SWPB - Swap bytes

SWPB <ga>

The most significant and least significant bytes of the operand at <ga> are exchanged. The result is equivalent to shifting the source operand circularly 8 bits.

Status bits affected: none.

Op code: 06C0

Format: 0000011011ttdddd
------ Destination

5.2.10. INC - Increment

INC <ga>

The value at <ga> is incremented by one. The result is compared to zero and the status bits are set accordingly. If a carry out or overflow occurs, the status bits will be set to reflect it.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 0580

Format: 0000010110ttdddd
------ Destination

5.2.11. INCT - Increment by two

INCT <ga>

The value at <ga> is incremented by two. The result is compared to zero and the status bits are set accordingly. If a carry out or overflow occurs, the status bits will be set to reflect it.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 05C0

Format: 0000010111ttdddd

------ Destination

5.2.12. DEC - Decrement

DEC <ga>

The value at <ga> is decremented by one. The result is compared to zero and the status bits are set accordingly. If a carry out or overflow occurs the status bits will be set to reflect it.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 0600

Format: 0000011000ttdddd
------ Destination

5.2.13. DECT - Decrement by two

DECT <ga>

The value at <ga> is decremented by two. The result is compared to zero and the status bits are set accordingly. If a carry out or overflow occurs the status bits will be set to reflect it.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 0640

Format: 0000011001ttdddd
------ Destination

5.2.14. X - Execute remote

X <ga>

The word at <ga> will be used as the operation code of the current instruction. In other words, the X instruction causes the instruction at the specified address to be executed. The program counter is not changed, however, so execution will continue in line following the X instruction (assuming that the instruction executed does not itself change the program counter). There are several wrinkles associated with this instruction. First, if the instruction executed requires data words following it for immediate data or direct addresses, these words will be loaded following the X instruction, not following the address executed. Second, if the instruction executed is a relative jump instruction, the displacement will be added to the address of the X instruction, not the address of the instruction; this is seldom what the programmer had in mind. Third, the hardware instruction fetch signal is not turned on when the operand of the X instruction is fetched. This may confuse some user implemented special purpose hardware that counts on this signal. Fourth, while the X instruction itself changes no status bits, the instruction executed will set the status bits as it normally would.

Status bits affected: none, however instruction executed will set status bits normally.

Op code: 0480

Format: 0000010010ttdddd
------ Destination

5.3. Register / general address instructions

These instructions take their source operand from the general address specified in the instruction, and use a workspace register specified in the instruction as the destination operand.

5.3.1. COC - Compare ones corresponding

COC <gas>,<wa>

The contents of the workspace register <wa> are tested against the contents of the general address <gas>. If every bit which is a one in <gas> is also a one in <wa>, the equal status bit is set. Otherwise (if one or more bits in <gas> which are ones correspond to zero bits in <wa>) the equal status bit is cleared. This instruction is most often used where the workspace register contains a set of bits indicating logical values. and the general address operand is a word containing a single bit denoting one of those values. The COC instruction permits a simple test for whether that bit is set in the workspace register. COC can, of course, also be used to test multiple bits for being set.

Status bits affected: Equal.

Op code: 2000

Format: 001000wwwwttssss

Workspace register
Source

5.3.2. CZC - Compare zeroes corresponding

CZC <gas>,<wa>

The contents of the workspace register <wa> are tested against the contents of the general address <gas>. If for every position in <gas> which is a one bit, the corresponding bit in <wa> is a zero, the equal status bit is set. Otherwise (one or more bits which are ones in <gas> correspond to zero bits in <wa>), the equal status bit is cleared. This instruction can be used similarly to COC to test whether a single bit is off, and is particularly useful to test whether a field of bits are all zero.

Status bits affected: Equal.

Op code: 2400

Format: 001001wwwwttssss

Workspace register
Source

5.3.3. XOR - Exclusive or

XOR <gas>,<wa>

The contents of the general address operand <gas> and the workspace register <wa> are bit-by-bit exclusive ored, and the result is stored in the workspace register <wa>. The result is compared against zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 2800

Format: 001010wwwwttssss

Workspace register destination
Source

5.3.4. MPY - Multiply

MPY <gas>,<wa>

The contents of the general address operand <gas> and the workspace register <wa> are multiplied as two unsigned 16 bit quantities. The 32 bit product is stored with the high-order 16 bits in the workspace register <wa> and the low order 16 bits in the next higher register (if <wa> is R15, the low order word will be stored in the next word in memory following the current workspace register set). Note that this instruction performs UNSIGNED multiplication, and hence if used with two's complement numbers will produce incorrect results. If used with signed quantities, user code must handle the sign. This instruction changes no status bits.

Status bits affected: none.

Op code: 3800

Format: 001110wwwwttssss
---- Workspace register destination
------ Source

5.3.5. DIV - Divide

DIV <gas>,<wa>

The 32 bit quantity with the high order 16 bits in workspace register <wa> and the low order 16 bits in the next consecutive memory location is divided by the 16 bit contents of the general address operand <gas>. The division is done treating both operands as unsigned numbers. The quotient from the division is stored in workspace register <wa>, and the remainder is stored in the next consecutive memory location (which will be the next higher register unless <wa> is R15). If the quotient from the division would exceed 65535 (OFFFF hexadecimal), the operation will be aborted and the overflow status bit will be set.

Status bits affected: Overflow.

Op code: 3C00

Format: 001111wwwwttssss
---- Workspace register destination
------ Source

5.4. Shift instructions

The shift instructions shift the contents of a workspace register in various manners. The number of bits to shift is determined in the following manner: each shift instruction contains a four bit "shift count" field. If that field is nonzero, the shift instruction shifts that number of bits. If the shift count field is zero, the low-order 4 bits of workspace register 0 (R0) are used for shift count. If the low-order 4 bits of R0 are zero, the shift will be 16 bits, otherwise the shift will be the number represented by the low 4 bits of R0.

5.4.1. SLA - Shift left arithmetic

SLA <wa>,<count>

The contents of the workspace register <wa> are shifted left by the specified number of bits, and the result is stored back in <wa>. Vacated bit positions on the right are filled with zeroes. If the sign bit of the operand changes during the operation, the overflow status bit will be set. The value of the last bit shifted out will be placed in the carry status bit. The result is compared to zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than,

equal, carry, overflow.

Op code: 0A00

Format: 00001010ccccwwww
---- Shift count
---- Workspace register

5.4.2. SRA - Shift right arithmetic

SRA <wa>,<count>

The contents of the workspace register <wa> are shifted right the specified number of bits, and the result is stored back in <wa>. Vacated bits on the left are filled with the original sign bit of the operand. The value of the last bit shifted out is placed in the carry status bit. The result is compared to zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry.

Op code: 0800

Format: 00001000ccccwwww
---- Shift count
---- Workspace register

5.4.3. SRC - Shift right circular

SRC <wa>,<count>

The contents of the workspace register <wa> are shifted right circularly the specified number of bits and the result is stored back in <wa>. In a circular shift, bits shifted out of the least significant bit shift back into the most significant bit. The value of the last bit shifted from the least significant position to the most significant position will be placed in the carry status bit. The result is compared with zero and the status bits are set accordingly. Note that although there is no left circular shift, a right circular shift of 16-X bits is equivalent to a left circular shift of X bits.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry.

Op code: 0B00

Format: 00001011ccccwwww
---- Shift count
---- Workspace register

5.4.4. SRL - Shift right logical

SRL <wa>,<count>

The contents of the workspace register <wa> are shifted right the specified number of bits and the result is stored back in <wa>. Vacated bits on the left are filled with zeroes. The value of the last bit shifted out is placed in the carry status bit. The result is compared with zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry.

Op code: 0900

Format: 00001001ccccwwww
---- Shift count
---- Workspace register

5.5. Immediate operand instructions

The immediate instructions are distinguished by the fact that they take an operand that follows the instruction in memory. The immediate instructions are primarily used when one of the operands is a constant known at assembly time. The immediate instructions are further divided into the "Workspace register immediate instructions", where the second operand is a workspace register, and the "Internal register immediate instructions", where the second operand is an internal CPU register.

5.5.1. Workspace register immediate instructions

5.5.1.1. AI - Add immediate

AI <wa>,<iop>

The immediate operand word following the instruction is added to the contents of the specified workspace register. The status bits are set as for the add (A) instruction described above.

Status bits affected: Logical greater than, arithmetic greater than, equal, carry, overflow.

Op code: 0240

Format: 0000000100010wwww wwww = workspace register
iiiiiiiiiiiiiiii Immediate operand

5.5.1.2. ANDI - And immediate

ANDI <wa>,<iop>

The immediate operand word is logically ANDed with the contents of the specified workspace register, and the result is stored in the workspace register. The result is compared with zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 0240

Format: 0000000100100wwww wwww = workspace register
iiiiiiiiiiiiiiii Immediate operand

5.5.1.3. CI - Compare immediate

CI <wa>,<iop>

The operand in the workspace register is compared with the immediate operand following the instruction and the status bits are set to reflect the comparison. The arithmetic greater than bit is set if the register operand is greater than the immediate operand when both are considered as 16 bit two's complement signed numbers. The logical greater than bit is set if the register operand is greater than the immediate operand when both are considered as 16 bit unsigned numbers. The equal bit is set if the two operands are equal.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 0280

Format: 0000000101000wwww wwww = workspace register
iiiiiiiiiiiiiiii Immediate operand

5.5.1.4. LI - Load immediate

LI <wa>,<iop>

The immediate operand is loaded into the designated workspace register. The value loaded is compared against zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 0200

Format: 0000001000000wwww wwww = workspace register
iiiiiiiiiiiiiiii Immediate operand

5.5.1.5. ORI - Or immediate

ORI <wa>,<iop>

The contents of the workspace register are logically ORed with the immediate operand and the result is stored back in the workspace register. The result is compared against zero and the status bits are set accordingly.

Status bits affected: Logical greater than, arithmetic greater than, equal.

Op code: 0260

Format: 0000001001100wwww wwww = workspace register
iiiiiiiiiiiiiiii Immediate operand

5.5.2. Internal register immediate instructions

5.5.2.1. LIMI - Load interrupt mask immediate

LIMI <iop>

The least significant 4 bits of the immediate operand are placed in the interrupt mask portion of the status register. This causes the CPU to immediately start to mask interrupts according to the new mask value. This instruction is most often used as the first instruction of a system subroutine called with BLWP and which is called from various interrupt levels. If the first instruction of the routine performs a LIMI to lock out the level of the highest interrupt which may call the routine, the subroutine is guaranteed that it can run to completion without being interrupted (this because the BLWP instruction delays interrupt service for one instruction time to give the LIMI a chance to be executed). When the subroutine returns via a RTWP instruction, the RTWP will reload the status register with the interrupt mask at the time the subroutine was called and the environment will be restored.

Status bits affected: none.

Op code: 0300

Format: 0000001100000000
0000000000000iiii iiii = new interrupt mask

5.5.2.2. LWPI - Load workspace pointer immediate

LWPI <iop>

The immediate operand is loaded into the workspace pointer register in the

CPU. References to workspace registers made after the execution of the LWPI instruction will refer to the 16 word (32 byte) area of memory starting at the address specified by the immediate operand.

Status bits affected: none.

Op code: 02E0

Format: 0000001011100000
iiiiiiiiiiiiiiii New workspace address

5.6. Jump instructions

The jump instructions are conditional and unconditional instructions that allow transfer within the range from -128 to +127 words from the instruction following the jump instruction itself. Each jump instruction contains an 8 bit displacement field. If the jump is taken, the displacement field is extracted, sign extended to 16 bits, shifted left one bit, and added to the contents of the program counter which will point to the word following the jump instruction. The result of this is that if the displacement field is 0, no transfer occurs, if -1, the jump instruction itself is the destination, and otherwise the displacement is the signed number of words to jump.

The assembler automatically computes the displacement field for jump instructions, so the programmer need not be concerned with the details explained above. When a jump instruction is coded, its operand is an expression (normally just a program label) for the destination of the jump. The assembler will insert the correct displacement in the jump instruction, and give a truncation error flag if the displacement is too large to fit in the instruction. For example, in the sequence of code:

```
        JMP     TAG1
        MOV     R0,R5
TAG1    BL      SUBR
```

The assembler would insert a displacement of 1 in the JMP instruction.

5.6.1. JMP - Jump unconditional

JMP <destination>

The destination address, indicated by the displacement in the instruction, will be the next instruction executed.

Status bits affected: none.

Op code: 1000

Format: 00010000dddddddd

Displacement

5.6.2. JEQ - Jump equal

JEQ <destination>

The destination address is jumped to if the equal status bit is set.

Status bits affected: none.

Op code: 1300

Format: 00010011dddddddd

Displacement

5.6.3. JNE - Jump not equal

JNE <destination>

The destination address is jumped to if the equal status bit is clear.

Status bits affected: none.

Op code: 1600

Format: 00010110dddddddd

Displacement

5.6.4. JGT - Jump greater than

JGT <destination>

The destination is jumped to if the arithmetic greater than bit is set.

Status bits affected: none.

Op code: 1500

Format: 00010101dddddddd

Displacement

5.6.5. JLT - Jump less than

JLT <destination>

The destination is jumped to if both the arithmetic greater than bit and the equal bits are clear in the status register.

Status bits affected: none.

Op code: 1100

Format: 00010001dddddddd

Displacement

5.6.6. JH - Jump high

JH <destination>

The destination address is jumped to if the logical greater than bit is set and equal bit is clear.

Status bits affected: none.

Op code: 1B00

Format: 00011011dddddddd

Displacement

5.6.7. JHE - Jump high or equal

JHE <destination>

The destination address is jumped to if either the logical greater than or the equal status bit are set in the status register.

Status bits affected: none.

Op code: 1400

Format: 00010100dddddddd

-------- Displacement

5.6.8. JL - Jump low

JL <destination>

The destination address is jumped to if both the logical greater than and the equal status bits are clear.

Status bits affected: none.

Op code: 1A00

Format: 00011010dddddddd
-------- Displacement

5.6.9. JLE - Jump low or equal

JLE <destination>

The destination address is jumped to if either the logical greater than bit is clear, or the equal bit is set in the status register.

Status bits affected: none.

Op.code: 1200

Format: 00010010dddddddd
-------- Displacement

5.6.10. JOC - Jump on carry

JOC <destination>

The destination address is jumped to if the carry bit is set in the status register.

Status bits affected: none.

Op code: 1800

Format: 00011000dddddddd
-------- Displacement

5.6.11. JNC - Jump no carry

JNC <destination>

The destination address is jumped to if the carry bit is clear in the status register.

Status bits affected: none.

Op code: 1700

Format: 00010111dddddddd
-------- Displacement

5.6.12. JNO - Jump no overflow

JNO <destination>

The destination address is jumped to if the overflow bit is clear in the status register.

Status bits affected: none.

Op code: 1900

Format: 00011001dddddddd

Displacement

5.6.13. JOP - Jump odd parity

JOP <destination>

The destination is jumped to if the parity bit in the status register is set. (The parity bit is set by the byte instructions if the number of bits in the result byte is odd.)

Status bits affected: none.

Op code: 1C00

Format: 00011100dddddddd

Displacement

5.7. Internal register store instructions

The internal register store instructions store the contents of internal CPU registers into registers in the workspace. None of these instructions affect any status bits.

5.7.1. STST - Store status

STST <wa>

The contents of the CPU status register are stored into the designated workspace register.

Status bits affected: none.

Op code: 02C0

Format: 0000001011000wwww

Workspace register

5.7.2. STWP - Store workspace pointer

STWP <wa>

The contents of the CPU workspace pointer are stored into the designated workspace register. This instruction stores the memory address at which the current workspace starts into one of the workspace registers. This is useful in code which needs to access its workspace as memory but which does not explicitly know its workspace location or may be called with several different workspaces.

Status bits affected: none.

Op code: 02A0

Format: 0000001010100wwww

Workspace register

5.8 Control instructions

These instructions perform control functions on the CPU. These instructions actually have little effect in the TMS9900, but are executed by auxiliary hardware in the M9900 CPU, so the action of these

instructions in other TMS9900 systems cannot be guaranteed.

5.8.1. IDLE - Idle processor

IDLE

The execution of instructions is suspended until the occurrence of an interrupt, or an external RESET or LOAD signal. Note that the program counter is incremented before execution is suspended, so that the program counter captured by an interrupt that terminates the IDLE state will point to the instruction following the IDLE instruction.

Status bits affected: none.

Op code: 0340

Format: 0000001101000000

5.8.2. LREX - Load and restart execution

LREX

The processor performs a context switch through a vector located at absolute memory address OFFFC, and prevents all interrupts except the non-maskable interrupt (level 0). In a normally configured M9900 system this will return control to the debug monitor or disc boot PROM in high memory.

Status bits affected: none.

Op code: 03E0

Format: 0000001111100000

5.8.3. RSET - Reset

RSET

The processor halts, sends a signal which resets all memory and I/O devices, locks out all interrupts (except level 0), and resumes execution by performing a context switch through a vector at memory address 0000. The action of this instruction is identical to pressing the RESET switch on the computer front panel.

Status bits affected: none.

Op code: 0360

Format: 0000001101100000

5.8.4. CKON - Clock on

CKON

This instruction generates a signal on the M9900 CPU board which is available for user application, but causes no other action.

Status bits affected: none.

Op code: 03A0

Format: 0000001110100000

5.8.5. CKOF - Clock off

```
CKOF
```

This instruction generates a signal on the M9900 CPU board which is available for user application, but causes no other action.

Status bits affected: none.

Op code: 03C0

Format: 0000001111000000

5.9. Extended operation instruction

The extended operation instructions allow user extension of the hardware instruction set by providing instructions that invoke subroutines via a vector in low memory. Marinchip software uses some of the XOP codes for various functions and pseudo instructions are defined for those special codes. Refer to the section in this manual on pseudo instructions for more information on those codes.

5.9.1. XOP - Extended operation

```
XOP     <gas>,<index>
```

A context switch will be performed with a vector whose address is computed by adding 040 hexadecimal to four times the <index> value coded on the instruction. The first word of that vector will be loaded as the workspace pointer, and the second word will be loaded into the program counter, performing a branch to that location. Within the new workspace register set, register R13 will be set to the address of the workspace in effect when the XOP was executed, R14 will be set to the address of the next instruction following the XOP, and R15 will be loaded with the contents of the status register. The effective address of the operand will be evaluated and stored in register R11 of the new workspace. Note that what is stored is the ADDRESS of the operand, NOT the operand value. The processor will not honour interrupt requests until the execution of the next instruction following the XOP. This permits the XOP routine to alter the interrupt mask to protect its common data. The XOP instruction so sets the workspace switched to that the instruction implementing routine may return after the XOP instruction simply by executing a RTWP instruction.

Status bits affected: Extended operation.

Op code: 2C00

Format: 001011iiiittssss

```
            ----          Index
                ------    Source
```

5.10. Communication register unit (CRU) instructions

The CRU instructions transfer data to and from the Communication Register Unit, which is the bit-oriented I/O facility of the TMS9900. The instructions are divided into two groups, those which reference single bits, and those which address groups of from 1 to 16 bits.

5.10.1. CRU single bit instructions

The CRU single bit instructions address bits in the CRU by adding a signed displacement in the instruction to a CRU base address in workspace register R12. The CRU address space provides 4096 individually addressable bits, so only the low order 12 bits of the sum of the displacement and the contents of R12 will be used as the CRU bit address.

The displacement field in the instruction will be sign-extended before adding it to the contents of R12, so the displacement should be interpreted as a two's complement integer.

5.10.1.1. SBO - Set bit to one

SBO <displacement>

The addressed CRU bit is set to one.

Status bits affected: none.

Op code: 1D00

Format: 00011101dddddddd

Displacement

5.10.1.2. SBZ - Set bit to zero

SBZ <displacement>

The addressed CRU bit is set to zero.

Status bits affected: none.

Op code: 1E00

Format: 00011110dddddddd

Displacement

5.10.1.3. TB - Test bit

TB <displacement>

The addressed CRU bit is read in and the equal status bit is set to the value of the bit. In other words, if the CRU bit was a one, a JEQ instruction will jump following the TB; if the bit was a zero, a JNE instruction will jump.

Status bits affected: Equal.

Op code: 1F00

Format: 00011111dddddddd

Displacement

5.10.2. CRU multiple bit instructions

The CRU multiple bit instructions, LDCR and STCR, allow groups of from 1 to 16 bits to be transferred to and from the CRU. These instructions contain a "count" field which specifies the number of bits to be transferred. If the count field is a number from 1 to 15, that number of bits will be transferred. If the count field is zero, 16 bits will be transferred. Workspace register R12 must be loaded with the address of the starting bit prior to the execution of these instructions. Bit addresses will begin with the bit number in R12, and increment with each bit sent. These instructions behave as byte instructions if the count field specifies 1 to 8 bits, and as word instructions if the count field specifies 9 to 16 bits. This means that if the transfer specifies 1 to 8 bits, the general address will be taken as a byte address, an auto-increment specification will cause the register to be incremented by one, and the parity status bit will be set if the number of one bits transferred is odd and cleared otherwise. If the transfer specifies 9 to 16 bits, the general address will be a word address, an auto-increment specification will increment the register by two, and the parity bit will not be affected by the instruction. In addition, on 9 to 16 bit transfers, if the general address is odd, the data will be byte-reversed

before being sent to the CRU (LDCR) or stored in memory (STCR).

5.10.2.1. LDCR - Load communication register

```
LDCR    <gas>,<count>
```

The number of bits specified by <count> are serially transferred to the CRU starting with the bit number in workspace register R12. <gas> specifies the address from which data is to be taken, and is a byte address if <count> is 1 to 8, and a word address if count is 9 to 16. The bits transferred will be compared to zero and the status will be set accordingly, and if the number of bits transferred is 8 or less, the parity bit will be set if the number of one bits transferred is odd and cleared otherwise.

Status bits affected: Logical greater than, arithmetic greater than, equal, parity (if count is from 1 to 8).

Op code: 3000

Format: 001100ccccttssss

```
              ----      Count
                ------  Source
```

5.10.2.2. STCR - Store communication register

```
STCR    <gad>,<count>
```

The number of bits specified by <count> will be serially read from the CRU starting at the bit number in workspace register R12. The bits read are right justified in a byte if the count is 8 or less, and a word if 9 or more, and unfilled bits are set to zero. The resulting byte or word is stored at the general address <gad>. The data stored will be compared to zero and the status bits set accordingly. If the length of transfer is 8 or fewer bits, the parity bit will be set if the number of 1 bits in the data transferred is odd and cleared otherwise.

Status bits affected: Logical greater than, arithmetic greater than, equal, parity (if transfer is from 1 to 8 bits).

Op code: 3400

Format: 001101ccccttdddd

```
              ----      Count
                ------  Destination
```

6. Pseudo-instructions

The assembler provides pseudo instructions for several commonly used special instructions. A pseudo instruction is simply an alternate name for an instruction or group of instructions that would be more cumbersome to write out explicitly.

6.1. FLOP - Floating operation

```
FLOP    <ga>
```

This pseudo instruction generates the same code as the sequence:

```
XOP     <ga>,2
```

The FLOP instruction is used to invoke the floating point emulation software in the Marinchip operating system.

6.2. JSYS - Jump to system

```
JSYS    <ga>
```

This pseudo instruction generates the same code as:

```
XOP     <ga>,1
```

JSYS is used as the standard Marinchip operating system call.

6.3. NOP - No operation

```
NOP
```

The TMS9900 does not have an explicit no operation instruction. The NOP pseudo instruction generates a:

```
JMP     $+2
```

which by jumping to the next instruction in line achieves the same effect.

6.4. RT - Return

```
RT
```

The RT pseudo instruction generates an indirect branch through register 11, e.g.,

```
B       *R11
```

This is the method used most often for returning from a subroutine called with the BL instruction, since the BL instruction loads the return address into R11.

6.5. Stack pseudo instructions

The TMS9900 does not have special "stack instructions", but its powerful auto-increment and indirect addressing modes permit the programmer to define and manipulate stacks using the normal instructions. The Marinchip assembler contains pseudo instructions which automatically generate the instructions used in stack manipulation.

In order to effectively use a stack, one of the workspace registers must be dedicated as a "stack pointer" and an area of memory must be reserved as a stack area. The stack pointer register must be initially set equal to the start of the stack area. Then, any register may be saved by pushing it onto the stack and restored by popping it from the stack. It is important to remember that a pop restores that last thing pushed on the stack, so saves and restores must be done in reverse order. The stack facility is most often used to save return points to subroutines. If each subroutine begins by pushing the return point register (R11) on the stack and returns by jumping to the address on the stack top, subroutines may be nested to a depth limited only by the storage assigned to the stack area. The user need not be concerned with saving and restoring the return point, and coding of recursive subroutines becomes almost automatic.

In the following discussions, <sp> will be assumed to be one of the workspace registers. It is common practice to use register R10 as the stack pointer, but the user need not follow this convention.

6.5.1. DSTK - Define stack pointer

```
DSTK    <sp>
```

The DSTK directive generates no code, but defines <sp> as the stack pointer register. All stack pseudo instructions which follow the DSTK

directive will use that register to reference the stack. The stack pointer register may be changed for another block of code by a subsequent DSTK directive. It is critical to remember: DSTK only defines which register is being used for the stack pointer; an ISTK must be used to initialise the register itself.

6.5.2. ISTK - Initialise stack pointer

```
        ISTK    <expression>
```

The ISTK directive generates a Load Immediate of the stack pointer register (previously defined by a DSTK directive) with the value of the <expression>, which will normally be the label at the start of the storage block which is being used as the stack area. The ISTK directive is normally used at the start of a program to generate the load that initially sets up the stack pointer register.

6.5.3. PSHR - Push value on stack

```
        PSHR    <ga>
```

The operand identified by <ga> is pushed onto the top of the stack and the stack pointer is incremented.

6.5.4. POPR - Pop value from stack

```
        POPR    <ga>
```

The value on the top of the stack is popped into the location indicated by <ga> and the stack pointer is decremented.

6.5.5. POPJ - Jump to stack top

```
        POPJ
```

The value at the top of the stack is loaded into register R11, the stack pointer is decremented, and a branch to the address in R11 is performed. This instruction is used to return from a subroutine which pushed its return point using a PSHR instruction.

6.5.6. Example of stack use

The following program fragments contain definition of a stack and its use by a sample subroutine which saves its return point and registers R6 and R7 using the stack, then restores them from the stack and returns to the saved call address.

```
.       Sample program
.
        DSTK    R10
BEGIN   ISTK    STACK8          Initialise stack pointer
.
        BL      XSUB            Call subroutine
.
.
.       XSUB - A typical subroutine
.
XSUB    PSHR    R11             Save return point
        PSHR    R6              Save R6
        PSHR    R7              Save R7
.
```

```
        POPR    R7              Restore R7
        POPR    R6              Restore R6
        POPJ                    Return to caller
```

7. Machine reference information

This section of the manual contains general reference information pertaining to the CPU and its instruction set.

7.1. Instruction summary

Mnemonic	Op code	Instruction
A	A000	Add words
AB	B000	Add bytes
ABS	0740	Absolute value
AI	0220	Add immediate
ANDI	0240	And immediate
B	0440	Branch
BL	0680	Branch and link
BLWP	0400	Branch and load workspace pointer
C	8000	Compare words
CB	9000	Compare bytes
CI	0280	Compare immediate
CKOF	03C0	Clock off
CKON	03A0	Clock on
CLR	04C0	Clear
COC	2000	Compare ones corresponding
CZC	2400	Compare zeroes corresponding
DEC	0600	Decrement
DECT	0640	Decrement by two
DIV	3C00	Divide
IDLE	0340	Idle CPU
INC	0580	Increment
INCT	05C0	Increment by two
INV	0540	Invert (one's complement)
JEQ	1300	Jump equal
JGT	1500	Jump greater than
JH	1B00	Jump high
JHE	1400	Jump high or equal
JL	1A00	Jump low
JLE	1200	Jump low or equal
JLT	1100	Jump less than
JMP	1000	Jump
JNC	1700	Jump no carry
JNE	1600	Jump not equal
JNO	1900	Jump no overflow
JOC	1800	Jump on carry
JOP	1C00	Jump odd parity
LDCR	3000	Load communication register
LI	0200	Load immediate
LIMI	0300	Load interrupt mask immediate
LREX	03E0	Load and restart execution
LWPI	02E0	Load workspace pointer immediate
MOV	C000	Move words
MOVB	D000	Move bytes
MPY	3800	Multiply
NEG	0500	Negate
ORI	0260	Or immediate

RSET	0360	Reset
RTWP	0380	Return to workspace pointer
S	6000	Subtract words
SB	7000	Subtract bytes
SBO	1D00	Set CRU bit to one
SBZ	1E00	Set CRU but to zero
SETO	0700	Set to all ones
SLA	0A00	Shift left arithmetic
SOC	E000	Set ones corresponding word
SOCB	F000	Set ones corresponding byte
SRA	0800	Shift right arithmetic
SRC	0B00	Shift right circular
SRL	0900	Shift right logical
STCR	3400	Store communication register
STST	02C0	Store status register
STWP	02A0	Store workspace pointer
SWPB	06C0	Swap bytes
SZC	4000	Set zeroes corresponding word
SZCB	5000	Set zeroes corresponding byte
TB	1F00	Test CRU bit
X	0480	Execute
XOP	2C00	Extended operation
XOR	2800	Exclusive or

7.2. Status register bits

```
L A = C O P X 0 0 0 0 0 I I I I
              -------
              |                 Interrupt mask
              +-----------------Extended operation
              +-----------------Parity odd
              +-----------------Overflow
              +-----------------Carry
              +-----------------Equal
              +-----------------Arithmetic greater than
              +-----------------Logical greater than
```

7.3. General address types

The tt field in a general address specification indicates the addressing mode of the operand. The values, their meaning, and sample assembly language coding is given below:

tt	Coding	Meaning
00	R7	Contents of workspace register
01	*R9	Register contains address of operand
10	@TAG	Direct: address follows instruction
10	@TAG(R2)	Indexed: add register to word following
11	*R3+	Register contains address of operand, auto-increment register by operand length

Both direct and indexed operands have a code of 10 in the type (tt) field, and are distinguished by the contents of the register field. Direct operands specify 0, while indexed operands specify the register to be used to index the operand. Note therefore that register 0 may not be used as an index register, but it may be used with indirect (tt=01) or auto-increment (tt=11) addressing.

If the address mode is direct or indexed (tt=10), a word will follow the instruction containing the direct address. If the instruction contains two general addresses (e.g., MOV), and both are direct or indexed, the source operand address will precede the destination address. For example, the instruction:

```
        MOV     01200(R7),@8000
```

will assemble into:

```
C817
1200
8000
```

8. Sample assembly language program

The following is an example of an assembly language program written according to the specifications of this manual.

```
*
*       Copy text subroutine
*
*       This subroutine will copy a string of bytes of
*       arbitrary length from one location to another.
*       The two areas must not overlap.
*
*  To call:
*
*       LI      R0,<length in bytes>
*       LI      R1,<source address>
*       LI      R2,<destination address>
*       BL      COPYTX
*       <return>                        R0, R1, R2, R11 destroyed
*
        IDT     "COPYTEXT"              Program id
*
COPYTX  DEC     R0                      More to copy ?
        JLT     COPYTD                  No.  Return
        MOVB    *R1+,*R2+               Yes.  Copy a byte
        JMP     COPYTX                  Keep on going
*
COPYTD  RT                              Return to caller
*
        END
```