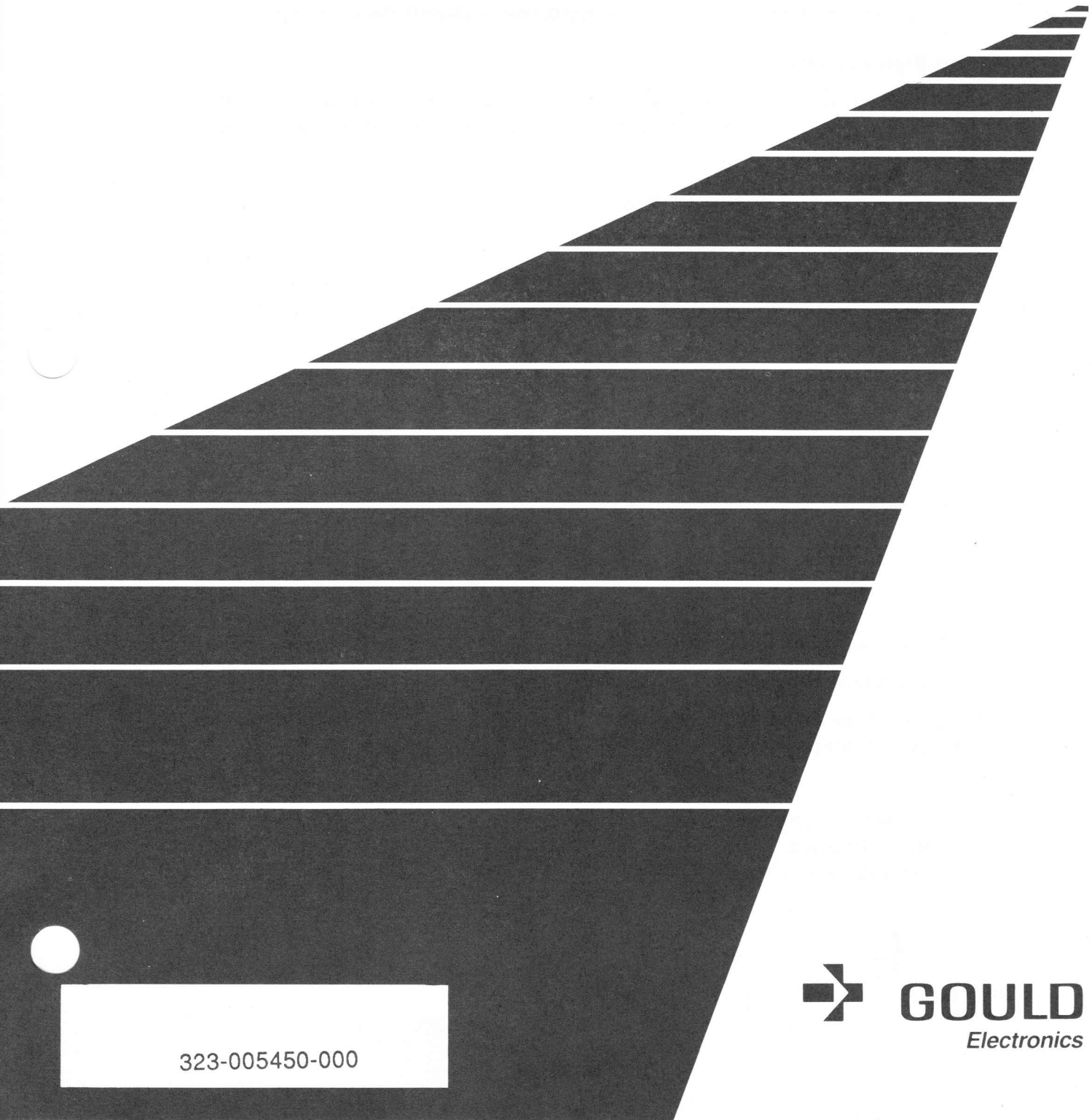
CPL

UTX/32™

Assembler Reference Manual

January 1988



323-005450-000

GOULD
Electronics

Limited Rights

This manual is supplied without representation or warranty of any kind. Gould Inc. therefore assumes no responsibility and shall have no liability of any kind arising from the supply or use of this publication or any material contained herein.

Proprietary Information

The information contained herein is proprietary to Gould CSD and/or its vendors, and its use, disclosure or duplication is subject to the restrictions stated in the Gould CSD license agreement Form No. 620-06 or the appropriate third-party sublicense agreement.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the rights in Technical Data and Computer Software Clause at 52.277.7013.

Gould Inc., Computer Systems Division
6901 West Sunrise Boulevard
Fort Lauderdale, Florida 33313

History

The *UTX/32 Assembler Reference Manual*, Release 2.0, Publication Order Number 323-005450-000, was printed in September 1986.

Change Package 1, Publication Order Number 323-005450-001, was printed in January 1988. *Note:* All references in this manual to Release 2.0 of UTX/32 apply to subsequent releases of UTX/32 for the Gould CONCEPT Product Line unless otherwise noted in future change packages.

The updated manual contains the following pages:

A zero in the Change Number column indicates an original page. A 1 in this column indicates a page to be substituted or added from this change package.

Every changed or new page in the document has the change number noted in the page footer. The changed portion of a page is marked by a vertical bar in the outer margin. A completely new or changed page will have no change bars, only the change notation in the footer. Reverse sides of new or changed pages that are not themselves changed have no change notation.

Assembler Reference Manual for Gould UTX/32™

Release 2.0

September 1986

Publication Order Number: 323-005450-000

GOULD
Electronics

Gould Inc. Computer Systems Division
6901 West Sunrise Boulevard
Fort Lauderdale, FL 33313

History

The *Assembler Reference Manual for Gould UTX/32* , Release 2.0, Publication Order Number 323-005450-000, was printed in September 1986.

The document contains the following components:

Title Page
Copyright Page
History Page, iii through iv
Table of Contents, v through vi
List of Tables, vii through viii
Chapter 1, 1-1 through 1-4
Chapter 2, 2-1 through 2-2
Chapter 3, 3-1 through 3-2
Chapter 4, 4-1 through 4-2
Chapter 5, 5-1 through 5-2
Chapter 6, 6-1 through 6-4
Chapter 7, 7-1 through 7-4
Chapter 8, 8-1 through 8-2
Chapter 9, 9-1 through 9-6
Chapter 10, 10-1 through 10-16
Chapter 11, 11-1 through 11-2
References, RF-1 through RF-2

Table of Contents

List of Tables

1 Introduction

Gould UTX/32 2.0 runs on the Gould PowerNode series of computers. UTX/32 2.0 supports an assembler written for Gould computers. This manual is based on the *Berkeley VAX/UNIX Assembler Reference Manual* and has been modified for UTX/32 2.0. It describes the usage and input syntax rules of the UTX/32 2.0 assembler, **as**.

as is designed for assembling object code produced by the C compiler. This document is intended only as a reference for system programmers who are writing or maintaining compilers or writing assembly language code. It is not intended to instruct programmers in how to write assembler source programs.

1.1 Organization

This manual is divided into eleven chapters and a reference list:

Chapter 1	Provides a brief introduction
Chapter 2	Describes the use and options of the assembler
Chapter 3	Describes the assembler identifiers, constants, and operators
Chapter 4	Describes the assembly segments and location counters
Chapter 5	Describes the sequence of statements
Chapter 6	Describes the expression operators and data types
Chapter 7	Describes the addressing modes for accessing data in memory
Chapter 8	Describes the pseudo-instructions for calling conventions
Chapter 9	Describes the keywords introducing directives or instructions
Chapter 10	Describes information about the opcodes
Chapter 11	Describes the assembler diagnostics
References	An alphabetical list of the documents cited in this manual

Interspersed throughout this manual are the following special pieces of information that serve to highlight or augment instructions:

WARNINGS	Emphasize procedures that are essential to proper assembler use

[TM] UTX/32 2.0 and PowerNode are trademarks of Gould Inc.

NOTES — Provide useful information that is not critical to the assembler's operation

1.2 Documentation Conventions

Boldface

Command and utility names, system-specific words and special characters, filenames and pathnames, and reserved words in code are boldface within text.

Example:

The **nroff** command is used to format text.

Also, expressions that must be entered exactly as shown are boldface within a command line or an example.

Example:

```
# tar xv
```

Note that the prompt # is not boldface, because it is a computer response instead of a user input.

Italics

Italics are used to refer to a manual page, to introduce new terms, for titles of documents, and occasionally for emphasis.

Example:

The first tape, called the *boot tape*, contains three initial boot programs.

< >

Angle brackets enclose descriptions of variable expressions that must be replaced with a value or character string.

Example:

```
% cd <directory>
```

Ellipses. . .

Vertical or horizontal ellipses tells you that information has been omitted, either on a syntax line or within examples.

Example:

```
.if n <nroff command . . .>
```

2 Use and Options

Most assembly language programs may be assembled using the **cc** command. **cc** assumes that any file name suffixed by **.s** contains assembly language source, which it will pass through the C preprocessor and on to the assembler. **as** arguments appearing on the **cc** command line are passed appropriately.

The format of an **as** command line is

as [**–LRQd**] [**–o** *output file*] [*file name*]

as accepts the following command arguments:

–L The **–L** flag instructs the assembler to save all labels beginning with a capital letter **L** in the symbol table portion of the output file. Such labels in text or fartext are not saved by default due to a convention of the C compiler, which generates these as local labels.

-R The **–R** flag effectively turns **.data** *n* directives into **.text** *n* directives. (The actions actually taken are considerably more complex.) It becomes unnecessary to run editor scripts on assembly code to make initialized data read-only and shared. Uninitialized data (via **.lcomm** and **.comm** directives) are still assembled into the bss segment.

–Q and **–d** The **–Q** (source trace) and the **–d** (debug) flags enable trace output during assembler execution, provided that the assembler has been compiled with the debugging code enabled. The **–Q** option will cause the line number, location pointer, and the input line to be printed for each line of the program. This may be useful when debugging macros defined by the C preprocessor, but will also produce a large amount of output. The information printed for the **–d** option is useful when debugging the assembler.

-o The **–o** flag causes the output to be placed on the named output. By default, the output of the assembler is placed in the file **a.out** in the current directory.

The input to the assembler is normally taken from the standard input. A file name may be given as an argument from which **as** should take its input. Only one file name is allowed.

3 Lexical Conventions

Assembler tokens (a distinguishable unit in a sequence of characters) include identifiers, constants, and operators.

3.1 Identifiers

An *identifier* consists of a sequence of alphanumeric characters, including the period and underscore. (The period is also referred to as "dot.") The first character may not be numeric. Identifiers may be up to 4096 characters long (4095 significant characters plus the null at the end of the string). Identifiers are also referred to as symbols or names.

3.2 Constants

The three types of constants are scalar, floating-point, and string.

3.2.1 Scalar Constants

Scalar (nonfloating-point) constants are defined as constants that can be up to 32-bits wide. **as** does not support 64-bit integers and cannot perform arithmetic on constants larger than 32 bits. Numbers with less precision than 32 bits are treated as 32-bit quantities.

The following additional conventions apply to scalar constants:

- They are interpreted as two's complement.
- **0123456789abcdefABCDEF** are the digits used to represent them. Each digit has the obvious value.

By comparison, the following conventions apply to decimal, octal, and hexadecimal constants. Base ten (decimal) is the default radix.

decimal constant
A sequence of digits with the prefix 0t (zero and lowercase letter **t**) or 0T (zero and uppercase letter **T**)

The string **0t1234** is interpreted as a decimal value.

octal constant
A sequence of digits with the prefix 0o (zero and lowercase letter **o**), 0O (zero and uppercase letter **O**), or simply 0 (zero)

The string **01234** is interpreted as an octal value.

hexadecimal constant
A sequence of digits with the prefix 0x (zero and lowercase letter **x**) or 0X (zero and uppercase letter **X**) The string **0x1234** is interpreted as a hexadecimal value.

3.2.2 Floating-point Constants

Floating-point constants are not explicitly recognized by the assembler. The **.word** directive may be used, but the value must be manually converted to the machine floating-point format defined in the processor reference manual.

3.2.3 String Constants

A string constant is defined using the C language syntax. Strings begin and end with double quotation marks. All C language backslash conventions are observed. The assembler does not implicitly end strings with a null byte.

3.3 Operators

There are several single-character operators; see Chapter 6, "Expressions."

3.4 Blanks

Blank and tab characters may be interspersed freely between tokens, but they may not be used within tokens (except string constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

3.5 Comments

The character sequence

```
--
```

(minus sign, minus sign) introduces a comment, extending through the end of the line on which it appears.

Lines beginning with a number/pound sign (#) in column one are assumed to be C preprocessor commands. These lines are ignored by the assembler with the exception of lines having the format

<number> <string>

This convention is interpreted as an indication that the assembler is now assembling file <string> at line <number>. Such a convention allows proper location reporting of errors if the assembler source has been processed by the C preprocessor for the **#include** and **#define** directives. Comments are otherwise ignored by the assembler. The assembler will not recognize C-style comments, introduced with /* and ending with */.

4 Segments and Location Counters

This chapter explains the segmentation of assembled code and data.

Assembled code and data fall into several segments: text, fartext, data, fardata, bss, and farbss. Within the data and fardata segments are two subsegments, distinguished by number (data 1, data 2). The subsegments are for programming convenience only. The UTX/32 operating system makes some assumptions about the content of some of these segments; the assembler does not.

Before writing the output file, the assembler pads each subsegment with zeroes to a multiple of eight bytes and then concatenates the subsegments in order to form the text and data segments. Requesting that the link editor define symbols and storage regions is the only action allowed by the assembler with respect to the bss segment.

Assembly begins in the text segment. Associated with each subsegment is an implicit location counter beginning at zero and incremented by 1 for each byte assembled into the subsegment. Explicit reference to the current segment's location counter is possible by use of the dot (.), but such practice is not recommended. Note that the location counter of data subsegment 2 behaves peculiarly due to the concatenation used to form the text and data segments.

5 Statements

A source program is composed of a sequence of statements, separated either by new lines or by semicolons. Statements are either null or keyword. Either type of statement may be preceded by one or more labels. These statements and labels are described in this chapter.

5.1 Labels

A label is referenced by its name. A label consists of a name followed by a colon, as in the following example:

```
_main:
```

The effect of a label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

Labels beginning with capital letter **L** are assumed to be local labels generated by the compiler and are discarded unless the **–L** option is in effect.

5.2 Null Statements

A *null statement* is empty and ignored by the assembler; however, it may be labeled.

5.3 Keyword Statements

A *keyword statement* begins with one of the many predefined **as** keywords. The syntax of the remainder of the statement depends on the keyword. All instruction opcodes are keywords. The remaining keywords are assembler pseudo-operations, also called *directives*. These are listed in Chapter 9, "Pseudo-operations," together with the syntax they require.

6 Expressions

An *expression* is a sequence of symbols representing a value. These symbols may include expression operators, identifiers, constants, and parentheses. This chapter explains these symbols.

6.1 Expression Operators

The expressions in Table 6-1 are legal as operators.

Table 6-1
Legal Expression Operators

Operator	Meaning
−	(unary) two's complement
~	(unary) bitwise one's complement
*	multiplication
/	division
%	modulo
+	addition
−	(binary) subtraction
>	logical right shift
<	logical left shift
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR

Expressions may be grouped by use of parentheses. All operators in expressions are fundamentally binary in nature. Arithmetic is fixed point, two's complement, and has 32 bits of precision. Arithmetic on floating-point numbers is not recommended, as the floating-point format is not understood by the expression handler.

There are five levels of precedence, listed in Table 6-2 from highest precedence level to lowest.

Table 6-2
Precedence Levels

Precedence Level	Operator
(highest) unary	− ~
binary	* / %
binary	+ −
binary	< >
(lowest) binary	¦ & ^

All operators of the same precedence level are evaluated strictly left to right, except for the evaluation order enforced by parentheses.

6.2 Data Types

The assembler manipulates several different types of expressions. The types have a defined meaning only according to the output format **a.out.h** and are not explicitly used by the assembler except when the output file is created. The types are:

undefined
Upon first encounter, each symbol is *undefined.* The symbol is given a default value of zero until it is defined as a label or by a **.set** pseudo operation. It will be treated as an absolute symbol until such a definition is parsed.

undefined external
A symbol which is declared **.globl** but not defined in the current assembly is an *undefined external.* If such a symbol exists, the link editor **ld** must be used to load the assembler's output with another routine that defines the undefined reference.

absolute
An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link editor to the output file.

text
The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value, since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of dot (.) is address 0 of the text segment.

fartext
This segment has the same characteristics as the text segment, but it is located in far space. Far space symbols may only be accessed through pointers that use a full 24-bit address. Fartext symbol values are measured with respect to the fartext segment origin.

data
The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run, since previously loaded programs may have data segments. After the first **.data** statement, the value of dot (.) is address 0 in the first data subsegment.

fardata
This segment has the same characteristics as the data segment, but it is located in far space. Fardata symbol values are measured with respect to the fardata segment origin. After the first fardata statement, the value of dot (.) is address 0 in the first fardata subsegment.

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments.

farbss
This segment has the same characteristics as the bss segment, but it is located in far space. Farbss symbol values are measured with respect to the farbss segment origin.

external absolute, **text**, **fartext**, **data**, **fardata**, **bss**, or **farbss**
Symbols declared **.globl**, but defined within an assembly as absolute, text, data, or bss symbols, may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

constant
These symbols are in the constant subsegment of the text segment. This subsegment is maintained only by the assembler. Constant symbol values are measured with respect to the constant subsegment origin.

register
The symbols

```
r0 r1 r2 r3 r4 r5 r6 r7
b0 b1 b2 b3 b4 b5 b6 b7
```

are predefined as register symbols.

other types
Each keyword known to the assembler has a type that is used to select the routine that processes the associated keyword statement. The behavior of such symbols when not used as keywords is dependent upon the context in which they are used (as with any other symbol).

6.3 Type Propagation in Expressions

When operands are combined by expression operators, the result has a type that depends on the types of the operands and on the operator. The rules involved are complex to state but are intended to be sensible and predictable. For purposes of expression evaluation the important types are:

absolute
text
fartext
data
fardata
bss
farbss
constant
undefined external

The following rules govern how the operands are combined:

- If both operands are absolute, the result is absolute.
- If an absolute is combined with any of the *other types* mentioned in Section 6.2, "Data Types," the result has the other type.

These additional rules apply to particular operators:

+ If one operand is relocatable in some segment or is an undefined external, the result has the postulated type, and the other operand must be absolute.

– If the first operand is a relocatable symbol, the second operand may be absolute (in which case, the result has the type of the first operand). If the first operand is relocatable text, fartext, data, or fadata, the second operand may be a relocatable symbol of the same type. If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

other operators
It is illegal to apply these operators to any but absolute symbols.

7 Addressing Modes

The assembler provides a variety of addressing modes, which are methods for accessing data in memory or the machine registers. Each operand of an assembler operation must conform to the syntax of one of the address modes. In addition, most of the operations allow use of only some of the addressing modes. This chapter explains each of the assembler operands.

In the descriptions given below, the word *target* is used when an address mode can be used in either the source or destination operand.

7.1 Register Operands

In the register mode, a general purpose register is the target. The syntax is the name of the register (the letter **r** followed immediately by the number [0-7] of the register).

In the following example, the contents of registers 0 and 1 are copied to registers 4 and 5, respectively, by using a doubleword **move** instruction:

```
movd  r0, r4
```

7.2 Base Register Operands

In the base register mode, a base register is the target. The syntax is the name of the register (the letter **b** followed immediately by the number [0-7] of the register). Note that the hardware does not use base register 0 (**b0**) in determining an effective memory address. **b0** may however be used as a transient storage register.

In the following example, the content of base register 2 is copied to base register 3:

```
movw  b2, b3
```

7.3 Memory Operands

The memory address mode allows the direct specification of a memory address. The target may only be a numeric value or a simple expression containing no register specifications or indexing. The link editor **ld** will implicitly relocate the address by adding a base register specification to the instruction.

In the following example, the byte at address **table+6** is moved to register 2. (The value of **table** would be defined elsewhere in the program.)

```
movb table+6, r2
```

This mode is also used with most of the assembler pseudo-operations to specify numeric values.

In the following example, the **.word** pseudo-operation is used to set the content of the next location in the current segment to the decimal value 10.

```
.word 10
```

7.4 Indexed Operands

The indexed mode allows the specification of the indexing capability of the hardware. The syntax of the target is an optional expression followed by a left square bracket, [, followed by a register name (either general purpose, base register, or one of each separated by a plus sign, +), followed by a right square bracket,].

The expression is taken as the value to be used in the offset field of the machine instruction. It must evaluate to a positive number that will fit in a 16-bit field.

Note that the hardware does not use **r0** or **b0** as an effective index.

If a general register is specified, it is used in the index field of the machine instruction. Note that the hardware treats 0 in this field (if **r0** were specified) as meaning no indexing is to be done.

WARNING: This usage is prone to errors and is not recommended.

If a base register is not specified, this value is adjusted according to the base register values being used to specify segments of the program.

In the following example, a byte is copied to **r2** from a table indexed by register 1.

```
movb table+5[r1], r2
```

This address mode is also used with the F class I/O operations to specify both the register and address to be used in the machine instruction.

In the following example, a start I/O instruction is executed, where **devaddr** is defined elsewhere as a table address, and **r4** contains an offset to the specific entry.

```
sio devaddr[r4]
```

7.5 Immediate Operands

The immediate address mode is used to specify constant values the assembler operations use as the source operand (never the destination). The syntax is the character #, followed by an expression.

The implementation of this mode is somewhat complicated by restrictions within the hardware. Certain machine instructions allow 16-bit values to be placed in the offset field of the instruction but are used as immediate values. If such a variant is available and the immediate value will fit in 16 bits, the assembler will use the instruction. For some operations (notably the move operation), if the immediate value is a constant 0, a simpler register operation may be used. A **movw** of 0 to a register will actually be translated to a zero register instruction.

There are three cases when the value is stored in a special segment and the emitted instruction addresses that location. The cases are:

- An immediate instruction is not available.
- The value will not fit in a 16-bit field.
- A relocatable expression is involved.

The constant segment is merged into the object module's text segment after the second pass of the assembler is completed.

In the first example, the contents of general register 6 are set to the value 1.

```
movw    #1, r6
```

In the second example, the value of general register 3 is set to zero. The assembler will emit a zero register instruction in this case.

```
movw    #0, r3
```

In the last example, the address of the label **array** is stored in base register 4. The example uses one word in the constant segment plus the instruction word.

```
movw    #array, b4
```

8 Pseudo-instructions

This chapter presents pseudo-instructions, which are expanded into appropriate code for the calling convention that is in effect.

The **enter** pseudo-instruction generates the prolog code for entering a called function. Usage is as follows:

enter frame, regmask

The **entry** pseudo-instruction generates the prolog code for entering an ENTRY of a called function (FORTRAN). Usage is as follows:

entry _ent

The **func** pseudo-instruction generates the code for calling a function. Usage is as follows:

func #n, _ftn

or

func #n, [r#]

The **retn** pseudo-instruction generates the code for returning from a called function. **retn** has no operands, but relies on the parameters of the preceding **enter.**

9 Pseudo-operations (Directives)

The following keywords introduce directives or instructions and influence the later behavior of the assembler. The metanotation

{ **stuff** }

means that zero or more instances of the given **stuff** may appear.

The following pseudo-operations are grouped into functional categories.

9.1 C Preprocessor Commands

\# <number>
\# <number> <string>

This is the only instance where a C preprocessor statement is meaningful to the assembler. The # *must* be in the first column. This metacomment causes the assembler to believe it is on the line number indicated by <number>. The second argument, if included, causes the assembler to believe it is in the file indicated by <string>; otherwise, the current file name does not change.

This is useful for identifying where errors have occurred when files are included with the C preprocessor.

9.2 Location Counter Control

.text
.data
.data <expression>
.fartext
.fardata
.fardata <expression>

These pseudo-operations cause the assembler to begin assembling into the indicated text or data subsegment. If specified, the <expression> must be defined, absolute, and evaluate to either 1 or 2. This allows selection of one of the two subsegments within the data segment. An omitted <expression> is treated as 1. Assembly starts in the **.text** segment. The effect of a **.data** or **.fardata** directive is treated as a **.text** directive if the **–R** assembly flag is set (the actual processing involved is more complex than this implies).

9.3 Filled Data

The location counter is adjusted so that the <expression> lowest bits of the location counter become zero, where <expression> is an exponential value. This is done by assembling from 0 to (2, raised to <expression>) bytes as in the following statement:

.align <expression>

For example, the statement

.align 2

pads to make the location counter evenly divisible by 4. The text and fartext segments are padded with the NOP instruction (halfword value 0x0002). The constant, data, and fardata segments are padded with null bytes. Only the sizes of the bss and farbss segments are adjusted, as the assembler never emits code to them. The <expression> must be defined, absolute, nonnegative, and less than 16.

WARNING: The subsegment concatenation convention and the current link editor conventions may not preserve attempts at aligning to more than two low-order zero bits (8 word boundary).

.space <expression>

The location counter is advanced by <expression> bytes, where <expression> must be defined and absolute. The space is filled in with zeroes.

9.4 Initialized Data

.byte <expr>{ , <expr>}
.half <expr>{ , <expr>}
.word <expr>{ , <expr>}
.long <expr>{ , <expr>}

The expressions in the comma-separated list are truncated to the size indicated by the key word, as shown in Table 9-1.

Table 9-1
Expression Truncation

KEYWORD	LENGTH (in bytes)
.byte	1
.half	2
.word	4
.long	8

These expressions are assembled in successive locations with proper alignment. In the case of **.long**, 64 bits are emitted to the object file with sign extension if necessary. Note the restrictions on expression values discussed in Chapter 6, "Expressions," when using **.long**.

.ascii <string>

Successive characters in the list are assembled into successive locations. For example, the first character in the string is placed into the first location. The C conventions for escaping are understood. The **.ascii** directive will not pad the string with zeroes. The **.ascii** directive is identical to

.byte <string0> <,string1> ...

9.5 Symbol Definitions

The directive

.comm <name>, <expression>

is used to declare an external symbol and a storage region (known as common) in the bss or farbss segment. Provided the <name> is not defined elsewhere, its type is made an undefined external, and its value is <expression>. In fact, the name given behaves in the current assembly the same as an undefined external. However, the link editor **ld** treats this as a special case. All external symbols in **ld** that are not otherwise defined and have a nonzero value are defined to lie in the bss segment. In addition, enough space is left after the symbol to hold <expression> bytes.

.lcomm <name>, <expression>
.farlcomm <name>, <expression>

A local common space of <expression> bytes will be allocated in the bss or farbss segment and <name> assigned the location of the first byte. The <name> is not declared as global and, hence, will be unknown to the link editor.

.globl <name>

This statement makes the <name> external. If it is otherwise defined (by **.set** or by appearance as a label), it acts within the assembly exactly as if the **.globl** statement were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.

Conversely, if the given symbol is not defined within the current assembly, the link editor can combine the output of this assembly with that of others that define the symbol. The assembler makes all otherwise undefined symbols external.

.using <base>, <memory>

The base register specified by <base> is flagged as having the memory address <memory> for use by the assembler in relocating local symbols. A value of 0 for <memory> will drop the effect of this directive. This directive is used by the compiler to simplify addressing of local symbols. The base register *must not* be changed during execution of the code while a **.using** is in effect.

.set <name>, <expression>

The (<name>, <expression>) pair is entered into the symbol table. Multiple **.set** statements with the same name are legal. The most recent value replaces all previous values.

WARNING: The value of a name is not reinitialized between two assembler passes. Therefore, a value defined in the first pass will be used during the second pass if it is not defined before it is used.

.stabs <string>, <expr1>, <expr2>, <expr3>, <expr4>
.stabn <expr1>, <expr2>, <expr3>, <expr4>
.stabd <expr1>, <expr2>, <expr3>

The **stab** directives place symbols in the symbol table for the symbolic debugger, **dbx**. A **stab** is a symbol table entry. The **.stabs** is a string **stab**, the **.stabn** is a **stab** not having a string, and the **.stabd** is a "dot" **stab** that implicitly references "dot," the current location counter.

The <string> in the **.stabs** directive is the name of a symbol. If the symbol is 0, the **.stabn** directive may be used instead.

The other expressions are stored in the name list structure of the symbol table **nlist** and preserved by the link editor for reference by **dbx**; the value of the expressions are peculiar to formats required by **dbx**.

The metanotation for expressions in the following list refers to those taken by **.stabs**, **.stabn**, and **.stabd**.

<expr1>	is used as a symbol table tag **nlist** field **n_type**
<expr2>	specifies the **nlist** field **n_other**
<expr3>	is used for either the source line number, or for a nesting level **nlist** field **n_desc**
<expr4>	is used as tag specific information **nlist** field **n_value**

In the case of the **.stabd** directive, <expr4> is nonexistent and is taken to be the value of the location counter at the following instruction. Since there is no associated symbol name, (<string>), for a **.stabd** directive, it can be used only in circumstances where the symbol is 0. The effect of a **.stabd** directive can be achieved by the **.stabn** directive in the following manner:

.stabn <expr1>, <expr2>, <expr3>, **LLn**

LLn:

where **LLn** is a label.

The **.stabd** directive is preferred, because it does not fill up the string table with labels used only for the **stab** symbol entries.

10 Opcodes for the Assembler

The following chapter presents information about the opcodes for the UTX/32 2.0 assembler. The assembler may generate different machine instructions for some opcodes, depending on the nature of the operands. The text indicates the range of possible instructions in these cases.

Information about the hardware mnemonics can be found in the reference manual for each processor type. The UTX/32 2.0 assembler will function on all PowerNode processors.

The assembler uses the following suffixes to identify opcodes:

b a byte operation.

h a halfword operation.

w a word operation.

l a doubleword operation.

f a floating-point word operation.

d a floating-point doubleword operation.

The subsections in this chapter use the following abbreviations to identify the types of operands:

reg	register mode
ereg	register mode, even-numbered register required
base	base mode
mem-ind	memory or indexed mode
mem-imm-ind	memory, immediate, or indexed mode
imm	immediate mode
imm=0	immediate mode, value known to be zero
imm16	immediate mode, value known to fit in 16-bit immediate
immpow2	immediate mode, value must be a power of 2
immbig	immediate mode, value unknown or will not fit in 16 bits

10.1 Move Instructions

Tables 10-1, 10-2, and 10-3 present the possible machine instructions generated from the move instructions of the UTX/32 2.0 assembler.

10.1.1 Two-operand Move Opcodes

The machine instruction generated is determined from the operand size (byte, half, word, etc.) and type. The first operand is always the source operand, and the second is always the destination. Condition codes are not set identically in all cases.

Table 10-1
Two-operand Move Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
movw	reg	base	TRBR (0x2C01)
mov[bhwf]	reg	reg	TRR (0x2C00)
mov[bhwf]	reg	mem-ind	ST[BHW] (0xD400)
mov[ld]	eregA	eregB	TRR (0x2C00). Two TRR instructions are emitted—the first moves eregA+1 to eregB+1, and the second moves eregA to eregB.
mov[ld]	ereg	mem-ind	STD (0xD400)
movw	baseA	baseB	LABR (0x5808). BaseA cannot be b0.
movw	base	reg	TBRR (0x2C02)
movw	base	mem-ind	STWBR (0x5400)
mov[ld]	base	ereg	TBRR (0x2C00). Two TBRR instructions are emitted—the first moves base+1 to reg+1, and the second moves base to ereg.
movw	imm=0	base	LABR (0x5808)
mov[bhfw]	imm=0	reg	ZR (0x0C00)
mov[bhfw]	imm=0	mem-ind	ZM[BHW] (0xF800)
mov[ld]	imm=0	ereg	ZR (0x0C00). Two ZR instructions are emitted—the first zeroes the register, the second the register + 1.
mov[ld]	imm=0	mem-ind	ZMD (0xE800)
movw	imm16	base	LABR (0x5808)
mov[bhwf]	imm16	reg	LI (0xC800)

Table 10-1 — *Continued*
Two-operand Move Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
movw	immbig	base	LWBR (0x5C00). The instruction uses the constant segment for value.
mov[bhwf]	immbig	reg	L[BHW] (0xAC00). The instruction uses the constant segment for value.
mov[ld]	imm	ereg	LD (0xAC00). The instruction uses the constant segment for value.
movw mov[bhwf]	mem-ind mem-ind	base reg	LWBR (0x5C00) L[BHW] (0xAC00)

10.1.2 Three-operand Move Opcodes

The opcodes given in Table 10-2 generate two 2-operand move instructions. The first moves the value of operand 1 to operand 2. The second moves (the just modified) value of operand 2 to operand 3. The instructions generated to perform each move are those described in the previous section, "Two-operand Move Opcodes."

Table 10-2
Three-operand Move Opcodes

Mnemonic	Operand1	Operand2	Operand3
mov[bhwf]	imm	reg	mem-ind
mov[ld]	imm	ereg	mem-ind
movw	imm	base	mem-ind
mov[bhwf]	mem-ind	reg	mem-ind
mov[ld]	mem-ind	ereg	mem-ind
movw	mem-ind	base	mem-ind

10.1.3 Load/Store Instructions

The instructions in Table 10-3 are special purpose variants of the move instructions. **trsc** and **tscr** are privileged instructions.

Table 10-3
Load/Store Instructions

Mnemonic	Operand1	Operand2	Hardware Mnemonic
cplw	reg	reg	TRC (0x2C03)
movea	mem-ind	reg	LA (0x5000)
movea	mem-ind	base	LABR (0x5808)
subea	mem-ind	base	SUABR (0x5800)
movear	mem-ind	reg	LEAR (0x8000)
lear	mem-ind	reg	LEAR (0x8000)
neg[bhw]	mem-imm-ind	reg	LN[BHW] (0xB400)
neg[bhw]	reg	reg	TRN (0x2C04)
negl	mem-imm-ind	ereg	LND (0xB400)
file	mem-ind	reg	LF (0xCC00)
file	mem-ind	base	LFBR (0xCC08)
file	reg	mem-ind	STF (0xDC00)
file	base	mem-ind	STFBR (0xDC08)
trsc	reg	reg	TRSC (0x2C0E)
tscr	reg	reg	TSCR (0x2C0F)
xchg	reg	reg	XCR (0x2C05)
xchg	base	base	XCBR (0x2802)

10.2 Branch Instructions

All of the branch opcodes take a single operand, except for the jump-after-increment instructions (2 operands), the **func** instruction (2 operands), and the **ret** instruction (no operands). The D field of a BCT or BCF defines the condition. See the Gould PowerNode reference manuals for the condition codes that are actually tested. Caution is required to assure that the correct condition codes are being used.

The **func** opcode listed at the end of Table 10-4 implements the UTX/32 function call, which generates three or four instructions.

Table 10-4
Branch Instructions

Mnemonic	Operand1	D field	Hardware Mnemonic
jmp	mem-ind	D = 0	BU (0xECOO)
jmp	reg		TRSW (0x28OO)

Table 10-4 — *Continued*
Branch Instructions

Mnemonic	Operand1	D field	Hardware Mnemonic
jbn	mem-ind	D = 1	BCT (0xEC80), jump bit not zero. Use only after a bit manipulation instruction. This instruction is a synonym for **jbt**.
jbz	mem-ind	D = 1	BCF (0xF080), jump bit zero. Use only after a bit manipulation instruction. This instruction is a synonym for **jbf**.
jbt	mem-ind	D = 1	BCT (0xEC80), jump bit true. Use only after a bit manipulation instruction. This instruction is a synonym for **jbn**.
jbf	mem-ind	D = 1	BCF (0xF080), jump bit false. Use only after a bit manipulation instruction. This instruction is a synonym for **jbz**.
jeq	mem-ind	D = 4	BCT (0xEE00)
jge	mem-ind	D = 5	BCT (0xEE80)
jgt	mem-ind	D = 2	BCT (0xED00)
jle	mem-ind	D = 6	BCT (0xEF00)
jlt	mem-ind	D = 3	BCT (0xED80)
jne	mem-ind	D = 4	BCF (0xF200)
bct7	mem-ind	D = 7	BCT (0xEF80)
bcf2	mem-ind	D = 2	BCF (0xF100)
bcf3	mem-ind	D = 3	BCF (0xF180)
bcf5	mem-ind	D = 5	BCF (0xF280)
bcf6	mem-ind	D = 6	BCF (0xF300)
bcf7	mem-ind	D = 7	BCF (0xF380)

Table 10-4 — *Continued*
Branch Instructions

Mnemonic	Operand1	D field	Hardware Mnemonic
jft	mem-ind	D = 0	BFT (0xF000). This instruction implicitly requires r4 as a mask register.
jsr	mem-ind		BL (0xF880)
bl	mem-ind		BL (0xF880)
ret			RETURN (0x280E). The C compiler does not support the stack format assumed by the hardware for this instruction.
jib	reg	mem-ind	BIB (0xF400)
jih	reg	mem-ind	BIH (0xF420)
jiw	reg	mem-ind	BIW (0xF440)
jil	reg	mem-ind	BID (0xF460)
func	imm	mem-ind	

10.3 Shift Instructions

The register-register variant of the arithmetic and logical shifts is implemented by building a shift instruction in the first operand register and then using an execute register instruction. The shift instructions are in Table 10-5. This variant uses three instruction words and two constant-space words.

The immediate-register variant uses one halfword instruction. The shift value used is the immediate value mod 32 (arithmetic modulus operation).

Table 10-5
Shift Instructions

Mnemonic	Operand1	Operand2	Hardware Mnemonic
aslw	imm	reg	SLA (0x1C40)
aslw	reg	reg	SLA (0x1C40)
asll	imm	ereg	SLAD (0x2040)
asll	reg	ereg	SLAD (0x2040)
asrw	imm	reg	SRA (0x1C00)
asrw	reg	reg	SRA (0x1C00)
asrl	imm	ereg	SRAD (0x2000)
asrl	reg	ereg	SRAD (0x2000)
lslw	imm	reg	SLL (0x1C60)
lslw	reg	reg	SLL (0x1C60)
lsll	imm	ereg	SLLD (0x2060)
lsll	reg	ereg	SLLD (0x2060)
lsrw	imm	reg	SRL (0x1C20)
lsrw	reg	reg	SRL (0x1C20)
lsrl	imm	ereg	SRLD (0x2020)
lsrl	reg	ereg	SRLD (0x2020)
rotlw	imm	regx	SLC (0x2440)
rotlw	reg	reg	SLC (0x2440)
rotrw	imm	reg	SRC (0x2400)
rotrw	reg	reg	SRC (0x2400)
sacz	reg	reg	SACZ (0x1008)

10.4 Bit Manipulation Instructions

The first operand of a bit instruction is a 32-bit mask with a single bit position set. The assembler encodes the bit position of the set bit into the emitted instruction. These instructions are detailed in Table 10-6.

The user is cautioned that the hardware does not set the condition codes in the same way for these instructions as it does for arithmetic instructions.

Table 10-6
Bit Manipulation Instructions

Mnemonic	Operand1	Operand2	Hardware Mnemonic
baddw	immpow2	mem-ind	ABM (0xA008)
baddw	immpow2	reg	ABR (0x1808)
bclr[bhw]	immpow2	mem-ind	ZBM (0x9C08)
bclrw	immpow2	reg	ZBR (0x1804)
bset[bhw]	immpow2	mem-ind	SBM (0x9808)
bsetw	immpow2	reg	SBR (0x1800)
btst[bhw]	immpow2	mem-ind	TBM (0xA408)
btstw	immpow2	reg	TBR (0x180C)

10.5 Compare and Logical Instructions

Instructions in Table 10-7 operate on arithmetic data and set condition codes similarly to the arithmetic instructions. The compare instruction should not be used in conjunction with bit instructions, since the instructions do not use the condition codes in the same fashion.

Table 10-7
Compare and Logical Instructions

Mnemonic	Operand1	Operand2	Hardware Mnemonic
and[bhw]	mem-imm-ind	reg	ANM[BHW] (0x8400)
andl	mem-imm-ind	ereg	ANMD (0x8400)
andw	reg	reg	ANR (0x0400)
cmp[bhwf]	mem-ind	reg	CAM[BHW] (0x9000)
cmp[bhwf]	imm16	reg	CI (0xC805)
cmp[bhwf]	immbig	reg	CAM[BHW] (0x9000)
cmp[wf]	reg	reg	CAR (0x1000)
cmp[ld]	mem-imm-ind	ereg	CAMD (0x9000)
or[bhw]	mem-imm-ind	reg	ORM[BHW] (0x8400)
orl	mem-imm-ind	ereg	ORMD (0x8400)
orw	reg	reg	ORR (0x0400)

Table 10-7 — *Continued*
Compare and Logical Instructions

Mnemonic	Operand1	Operand2	Hardware Mnemonic
xor[bhw]	mem-imm-ind	reg	EORM[BHW] (0x8400)
xorl	mem-imm-ind	ereg	EORMD (0x8400)
xorw	reg	reg	EORR (0x0400)

10.6 Arithmetic Instructions

The instructions for fixed-point and floating-point arithmetic are in Tables 10-8 and 10-9.

10.6.1 Fixed-point Arithmetic

Instructions in Table 10-8 are used for fixed-point (integer) arithmetic addition, division, multiplication, and subtraction.

Table 10-8
Fixed-Point Arithmetic Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
add[bhw]	mem-ind	reg	ADM[BHW] (0xB800)
add[bhw]	reg	mem-ind	ARM[BHW] (0xE800)
addl	mem-ind	ereg	ADMD (0xB800)
addl	ereg	mem-ind	ARMD (0xE800)
addw	immpow2	mem-ind	ABM (0xA008). Note that condition codes will be set differently in this variant.
addb	imm	reg	ADMB (0xB800)
addl	imm	ereg	ADML (0xB800)
add[hw]	imm16	reg	ADI (0xC801). The hardware sign extends the immediate, hence only the half and word variants can be used here.
add[hw]	immbig	reg	ADM[HW] (0xB800)

Table 10-8 — *Continued*
Fixed-Point Arithmetic Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
addw	reg	reg	ADR (0x3800)
div[bhw]	mem-ind	ereg	DVM[BHW] (0xC400)
divb	imm	ereg	DVMB (0xC400)
div[hw]	imm16	ereg	DVI (0xC804) The hardware sign extends the immediate, hence only the half and word variants can be used here.
div[hw]	immbig	ereg	DVM[HW] (0xC400)
divw	reg	ereg	DVR (0x380A)
mul[bhw]	mem-ind	ereg	MPM[BHW] (0xC000)
mulb	imm	ereg	MPMB (0xC000)
mul[hw]	imm16	ereg	MPI (0xC803) The hardware sign extends the immediate, hence only the half and word variants can be used here.
mul[hw]	immbig	ereg	MPM[HW] (0xC000)
mulw	reg	ereg	MPR (0x3802)
sub[bhw]	mem-ind	ereg	SUM[BHW] (0xBC00)
subl	mem-ind	ereg	SUMD (0xBC00)
subb	imm	reg	SUMB (0xBC00)
subl	imm	ereg	SUML (0xBC00)
sub[hw]	imm16	reg	SUI (0xC802). The hardware sign extends the immediate, hence only the half and word variants can be used here.
sub[hw]	immbig	reg	SUM[HW] (0xBC00)

Table 10-8 — *Continued*
Fixed-Point Arithmetic Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
subw	reg	reg	SUR (0x3C00)

10.6.2 Floating-point Arithmetic

Instructions in Table 10-9 are used for floating-point (real number) arithmetic addition, division, multiplication, and subtraction. Follow the restrictions on floating-point expressions in the appropriate sections of this section.

Table 10-9
Floating-point Arithmetic Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
addf	mem-imm-ind	reg	ADFW (0xE008)
addf	reg	reg	ADRFW (0x3801)
addd	mem-imm-ind	ereg	ADFD (0xE008)
addd	reg	ereg	ADRFD (0x3809)
divf	mem-imm-ind	reg	DVFD (0xE400)
divf	reg	reg	DVRFW (0x3804)
divd	mem-imm-ind	ereg	DVFD (0xE400)
divd	reg	ereg	DVRFD (0x380C)
mulf	mem-imm-ind	reg	MPFW (0xE408)
mulf	reg	reg	MPRFW (0x3806)
muld	mem-imm-ind	ereg	MPFD (0xE408)
muld	reg	ereg	MPRFD (0x380E)
subf	mem-imm-ind	reg	SUFW (0xE000)
subf	reg	reg	SUFRW (0x3803)
subd	mem-imm-ind	ereg	SUFD (0xE000)
subd	reg	ereg	SURFD (0x380B)

10.7 Type Conversion Instructions

Table 10-10 lists instructions for converting values to different types. The **cvtcw** opcode requires one operand, and the remainder require two. The user should read the processor manual before using either **rnd** or **sign**.

Table 10-10
Type Conversion Instruction Opcodes

Mnemonic	Operand1	Operand2	Hardware Mnemonic
cvtcw	reg		The convert-character-to-word opcode has several possible implementations, due to differences between processors. No guarantees are made for the setting of the condition codes.
cvtdl	reg	reg	FIXD (0x380D)
cvtfw	reg	reg	FIXW (0x3805)
cvtld	reg	reg	FLTD (0x380F)
cvtwf	reg	reg	FLTW (0x3807)
rnd	ereg		RND (0x0005). Takes only one operand.
sign	ereg		ES (0x0004). Takes only one operand.

10.8 Control Instructions

The processor reference manual should be studied before any of the instructions in this section are used.

10.8.1 Processor Control Instructions

Some of the instructions in Table 10-11 are privileged.

Table 10-11
Opcodes for Processor Control Instructions

Mnemonic	Operand1	Operand2	Hardware Mnemonic
cea			CEA (0x000F)
cmc	reg		CMC (0x040A)
dae			DAE (0x000E)
eae			EAE (0x0008)
exm	mem-ind		EXM (0xA800)
exr	reg		EXR (0xC807)
exrr	reg		EXRR (0xC807)
halt			HALT (0x0000)
lcs	reg		LCS (0x0003)
lmap	reg		LMAP (0x2C07)
lpsd	mem-ind		LPSD (0xF980)
lpsdcm	mem-ind		LPSDCM (0xFA80)
nop			NOP (0x0002)
rdsts	reg		RDSTS (0x0009)
rpswt	reg		RPSWT (0x040B)
sea			SEA (0x000D)
setcpu	reg		SETCPU (0x2C09)
sipu			SIPU (0x000A)
smc	reg		SMC (0x0407)
svc	imm	imm	SVC(0xC806). The first operand must fit in 4 bits, the second in 12.
tccr	reg		TCCR (0x2804)
tmapr	reg	reg	TMAPR (0x2C0A)
tpcbr	base		TPCBR (0x280C)
trcc	reg		TRCC (0x2805)
trsw	reg		TRSW (0x2800)
wait			WAIT (0x0001)

10.8.2 Input/Output Control Instructions

NOTE: All of the instructions in this section are privileged.

The following two opcodes each take two operands in the memory operand syntax. (**cd** stands for command device, **td** is test device.) The first operand must be the (constant valued) device address. The second operand must be a 16-bit constant value, which is used in the low-order halfword.

cd	CD (0xFC06)
td	TD (0xFC05)

The F class I/O opcodes require one operand in any of the immediate, indexed, memory, or register operand formats. The low-order halfword of the instruction is set from either an immediate value, the nonregister part of an indexed operand, or the expression value of a memory operand.

aci	ACI (0xFC77)
daci	DACI (0xFC7F)
dci	DCI (0xFC6F)
eci	ECI (0xFC67)
grio	GRIO (0xFC3F)
hio	HIO (0xFC37)
rschnl	RSCHNL (0xFC2F)
rsctl	RSCTL (0xFC47)
sio	SIO (0xFC17)
stpio	STPIO (0xFC27)
tio	TIO (0xFC1F)

The following five interrupt control instructions each take one immediate operand, which must be a 7-bit priority level value.

ai	AI (0xFC03)
dai	DAI (0xFC04)
di	DI (0xFC01)
ei	EI (0xFC00)
ri	RI (0xFC02)

The following two interrupt control instructions take no operands.

bei	BEI (0x0006)
uei	UEI (0x0007)

10.8.3 Writable Control Store Instructions

The **ecwcs** and **wcwcs** instructions use the same operand format as the class F I/O instructions: one operand in the immediate, indexed, memory, or register format. The **ecwcs** and **wcwcs** instructions must be used together. (See the appropriate processor manual [Gould].) The **rwcs** and **wwcs** instructions require two register operands. The **jwcs** instruction takes one operand in either the memory or indexed mode. Some of these instructions are privileged.

ecwcs	ECWCS (0xFC4F)
wcwcs	WCWCS (0xFC5F)
rwcs	RWCS (0x000B)
wwcs	WWCS (0x000C)
jwcs	JWCS (0xFA08)

11 Diagnostics

Diagnostics are intended to be self-explanatory and appear in the standard error file. Error diagnostics complain about lexical, syntactic, and some semantic errors, and abort the assembly at the end of the pass in which they are found. Errors found in the first pass will block reporting of errors detected only during the second assembler pass.

References

Gould CSD. *Gould V6 and V9 Central Processing Unit Reference Manual.* 301-004310-000. 1985.

Reiser, J. F. and R. R. Henry. "The Berkeley VAX/UNIX Assembler Reference Manual." In *UNIX Programmer's Manual.* Volume 2C. University of California at Berkeley. 1983.

Gould Inc., Computer Systems Division
6901 W. Sunrise Blvd.
P. O. Box 409148
Fort Lauderdale, FL 33340-9148
Telephone (305) 587-2900

GOULD
Electronics

Users Group Membership Application

USER ORGANIZATION: _____

REPRESENTATIVE(S): _____

_____

_____

ADDRESS: _____

_____

TELEX NUMBER: _____ PHONE NUMBER: _____

NUMBER AND TYPE OF GOULD CSD COMPUTERS: _____

_____

OPERATING SYSTEM AND REV. LEVEL: _____

_____

APPLICATIONS (Please Indicate)

1. EDP
   - A. Inventory Control
   - B. Engineering & Production Data Control
   - C. Large Machine Off-Load
   - D. Remote Batch Terminal
   - E. Other

2. Communications
   - A. Telephone System Monitoring
   - B. Front End Processors
   - C. Message Switching
   - D. Other

3. Design & Drafting
   - A. Electrical
   - B. Mechanical
   - C. Architectural
   - D. Cartography
   - E. Image Processing
   - F. Other

4. Industrial Automation
   - A. Continuous Process Control Op.
   - B. Production Scheduling & Control
   - C. Process Planning
   - D. Numerical Control
   - E. Other

5. Laboratory and Computational
   - A. Seismic
   - B. Scientific Calculation
   - C. Experiment Monitoring
   - D. Mathematical Modeling
   - E. Signal Processing
   - F. Other

6. Energy Monitoring & Control
   - A. Power Generation
   - B. Power Distribution
   - C. Environmental Control
   - D. Meter Monitoring
   - E. Other

7. Simulation
   - A. Flight Simulators
   - B. Power Plant Simulators
   - C. Electronic Warfare
   - D. Other

8. Other

Please return to:

Users Group Representative

Date: _____

Gould Inc., Computer Systems Division Users Group. . .

The purpose of the Gould CSD Users Group is to help create better User/User and User/Gould CSD communications.

There is no fee to join the Users Group. Simply complete the Membership Application on the reverse side and mail to the Users Group Representative. You will automatically receive Users Group Newsletters, Referral Guide and other pertinent Users Group activity information.

Fold and Staple for Mailing

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 947 FT. LAUDERDALE, FL

POSTAGE WILL BE PAID BY ADDRESSEE

GOULD INC., COMPUTER SYSTEMS DIVISION
ATTENTION: USERS GROUP REPRESENTATIVE
6901 W. SUNRISE BLVD.
P. O. BOX 409148
FT. LAUDERDALE FL 33340-9970

Fold and Staple for Mailing

(Detach Here)

GOULD
Electronics