i960® Processor Assembler User's Guide

Revision	Revision History	Date
-001	Original Issue.	12/92
-002	Minor corrections.	09/93
-003	Revised for CTOOLS960 R4.5 and GNU/960 Tools R2.4.	05/94
-004	Revised for Release 5.0.	02/96
-005	Revised for Release 5.1	01/97

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office.

printed on
recycled paper

Contents

Chapter 4   Directives

Chapter 5   Messages

Chapter 6   Assembly Language

Examples

Figures

Tables

Overview 1

This chapter of the *i960® Processor Assembler User's Guide* introduces you to the i960 processor assembler and to this manual.

This chapter describes:

- using the assembler with other i960 processor software tools
- the standards and conventions used by the assembler and in this manual
- the trademarks and copyrights pertaining to this manual

i960® Processor Assembler and Related Tools

The i960 processor assembler is part of a complete set of software and hardware tools for developing embedded applications for the i960 processors. Use ic960 or gcc960, the i960 processor assembler, and the i960 processor software utilities to translate, link, and format source text into executable or PROM-programmable code. You can write assembly source text directly in a text editor or compile a C program to produce assembly output. To create object files, you can assemble your source text or the assembly output from the C compiler. Disassembled text from the dumper is for debugging only and cannot be reassembled. For more information on how the software tools work together, see the Getting Started manual.

Compatibility and Standards

The assembler described in this manual supports the i960 Sx, Kx, Cx, Jx, Hx, and RP processors.

The assembler accepts output from Release 3.0 and later of the CTOOLS960 compiler and from Release 1.2 and later of the GNU/960 compiler.

You can specify the assembler object file output format as either common object file format (COFF), b.out or ELF format. The output format depends on the assembler invocation command, as shown:

- For b.out format, invoke the assembler with the `gas960` command.
- For COFF format, invoke the assembler with the `gas960c` or `asm960` command.
- For ELF format, invoke the assembler with the `gas960e` command.

For backwards compatibility with your existing script or batch files, the directory structures and search paths used by the assembler depend on the invocation name, as shown:

- For behavior similar to the GNU/960 (Release 1.2 or later) assembler, invoke the assembler with `gas960`, `gas960c`, or `gas960e`.
- For behavior similar to the CTOOLS960 (Release 3.5) assembler, invoke the assembler with `asm960`.

Note that when you invoke the assembler as `asm960` you can generate the COFF output format only.

About This Manual

This manual, the *i960 Processor Assembler User's Guide*, is part of the i960 processor software development tools manual set. See *Getting Started with the i960 Processor Software Development Tools* for a list of all manuals in the i960 processor development tools library.

The *i960 Processor Assembler User's Guide* provides operating instructions for the assembler. This manual does not teach development techniques.

Target Audience

To use the assembler effectively, you must be familiar with the i960 architecture and the development process.

This manual does not provide detailed information about the target processor. The processor manuals listed in *Getting Started with the i960 Processor Software Development Tools* contain information such as:

- a description of the i960 architecture
- the processor theory of operation and descriptions of the on-chip devices
- information about low-level programming for particular processors

For additional information about these topics, order the relevant publications listed in *Getting Started with the i960 Processor Software Development Tools*.

Conventions

In addition to the standard typographical conventions listed on the front inside cover, this manual uses the following notation and format conventions.

Case is significant for directives, functions, options, and option arguments. On UNIX*, case is also significant for invocation names and filenames.

Arguments and operands are in italics. The operand names indicate the function of the operands (for example, *`filename`*, *`expr`*).

Directive and pseudo-instruction operands use the following notation:

`addr`   represents an address.

`align`   represents an exponent of 2, used as an alignment factor.

data	represents ordinal, integer, or floating-point data; the format of the data depends on the instruction or directive.
int	represents a positive integer.
name	represents a symbol or label.
size	represents an integer, used as a size factor.
string	represents a sequence of ASCII characters.
expr	indicates an expression.

Special characters, delimiters, and other punctuation used with the operands, such as quotation marks and commas, are explicitly shown.

Notation for registers is one or more letters indicating the kind of register and a number between `0` and `15`, as follows:

global register	a register `g0` through `g14`, and `fp`.
local register	a register `pfp`, `sp`, `rip`, and `r3` through `r15`.
special function register	a register available only on the i960 Cx and Hx processors: `sf0-sf2 (Cx) and sf0-sf4 (Hx)`.
floating-point register	a register available only with on-chip floating-point support: `fp0`, `fp1`, `fp2`, and `fp3`.

For more information on the registers, see the processor manuals listed in *Getting Started with the i960 Processor Software Development Tools*.

Target expressions (*targ*) representing a memory address are assembled as a signed displacement value representing an IP-relative address:

Format	Displacement	Target (targ)
COBR	$-2^{10} : 2^{10}-1$	$-2^{12} : 2^{12}-4$ from IP
CTRL	$-2^{21} : 2^{21}-1$	$-2^{23} : 2^{23}-4$ from IP

For convenience in cross-referencing material, the notation used in the reference sections follows that of the processor manuals listed in *Getting Started with the i960 Processor Software Development Tools*.

Customer Service

If you need service or assistance, see *Getting Started with the i960 Processor Software Development Tools*.

Invoking the Assembler

2

This chapter discusses the assembler invocation syntax, options, input, and output and explains how to automate assembly. You can invoke the assembler from the operating system prompt or from a script or batch file.

Invocation Command

Invoke the assembler as follows:

```
asm960 | gas960 [ c | e ] [-option]... [source]... [...]
```

`asm960` or `gas960c`	invokes the assembler to generate COFF output. The dual syntax provides backwards compatibility with previous versions of the iC-960 and gcc960 C compilers.
`gas960`	invokes the assembler to generate b.out format output.
`gas960e`	invokes the assembler to generate ELF format output.
`option`	is an invocation option (described in Chapter 3) affecting assembler input, operation, and output. Arguments can follow some options. Case is significant.
	Precede the options with a hyphen (`-`). In Windows, you can use a slash (`/`) instead of the hyphen.
`source`	is an assembly source filename. You can provide a complete path name for each source file. The default search path is the current directory.

You can interleave options and source filenames.

NOTES. *To use options relevant for COFF output, invoke the assembler with the* `asm960` *or* `gas960c` *command. To use options relevant for b.out-format output, invoke the assembler with the* `gas960` *command. To use options relevant for ELF output, invoke the assembler with the* `gas960e` *command.*

On UNIX, case is significant for all parts of the assembler invocation syntax. In Windows, case is significant only for the options and option arguments.

Examples throughout this manual use a UNIX host system and the `gas960e` *invocation command and directory structures, unless otherwise noted.*

The b.out assembler does not support the i960 RP Processor.

Specifying Option Arguments

Some options require arguments. The assembler interprets any string following such an option as the option argument. Omitting an option argument at the end of the command line causes an error. For example:

```
gas960e myprog.as -o
gas960: Expected a filename after -o.
```

You can put a space between an option and its argument. The following are both correct:

```
gas960e myprog.as -omyprog.o
gas960e myprog.as -o myprog.o
```

An incorrect argument causes an error message appropriate to the option. See Chapter 3, Option Reference, for information on the valid arguments for each option.

Specifying Single and Multiple Options

Precede options with a hyphen (-):

```
gas960e myprog.as -o myprog.obj -W -V
```

On Windows* 95/Windows NT*-based machines, you can use a slash (/) instead of the hyphen.

Any string that does not begin with a hyphen and is not positioned as an option argument is interpreted as a source filename. The following example shows the message caused when the V option is specified without a hyphen and no file named V is in the search path:

```
gas960e -W V myprog.as
Can't open V for reading.
No such file or directory.
```

Some options consist of a single character with no arguments. You can specify two or more such options as an option group with a single hyphen:

```
gas960e myprog.as -o myprog.obj -WV
```

Using Uppercase and Lowercase

Depending on your host system, case can be significant in the assembler invocation name. For example, on Windows, entering ASM960 is the same as entering asm960. On UNIX, you can invoke the assembler with asm960 but not with ASM960.

Regardless of your host system, case is significant in the options and arguments. For example, an uppercase W is valid, but a lowercase w causes the following message:

```
Unrecognized option: w
```

2

Naming the Object File

After a successful assembly, the assembler produces an object file in common object file format (COFF), b.out or ELF format. To generate a COFF object file, invoke the assembler with `asm960` or `gas960c`. To generate a b.out format object file, invoke the assembler with `gas960`. To generate an ELF file, use `gas960e`. For a description of the COFF file format, see your utilities user's guide. For a description of ELF, see the Intel 80960 EABI specification (Intel Literature order number 631999) listed in *Getting Started*.

When you specify a source file with the `.s` or `.as` extension, the assembler creates an object file with the extension `.o`. When you specify a file with any other extension (or none) the assembler creates an object file with full source filename (including its original extension) with `.o` appended.

When you provide the first block of input interactively, the object filename is `a.out` for COFF output, `b.out` for b.out format output, and `e.out` for ELF output. For example, the following produces a single object file named `ex1.o`:

gas960e ex1.s ex2.s ex3.s

To specify the object filename, use the `o` option. For example, the following creates or replaces an object file named `ex1.o`:

gas960e example.src -o ex1.o

The assembler can overwrite an existing file unless the filename ends in `.s`, `.as`, or `.asm`. To ensure your source files are not accidentally overwritten, use the protected filename extensions. For example, if `ex1.s` exists, the following stops assembly with an error:

gas960e example.s -o ex1.s
FATAL: Output file will overwrite existing protected file.

You can use assembler directives, as described in Chapter 4, Directives, to specify the contents of the object file. Additional software utilities are available to read and reformat the object file, as described in the *i960 Processor Software Utilities User's Guide*.

2

Providing Source Input

You must provide source text from at least one of:

- a file named in the assembler invocation command
- `stdin`, such as the keyboard or the redirected output of another command

For information on `stdin`, see your host operating system documentation.

An assembly source file is an ASCII file of assembly language instructions and assembler directives. You can write the assembly source using a text editor or generate an assembly file with the C compiler.

For interactive input, specify the `i` option and provide lines of assembly source from `stdin` (for example, lines entered from the keyboard or piped from another application). The following example pipes the output of a script named `mybuild` (invoked with the UNIX C shell primitive `source` command) into the assembler:

```
source mybuild | gas960e -i
```

For information on piping, see your host operating system documentation.

To end keyboard input, type the `Ctrl-d` key combination on a new line. The following keyboard-entry example assembles five lines, naming the output object file `e.out`:

```
gas960e -i
roundr g0, fp0
subr fp0, g0, g0
expr g0, g0
addr 1.0, g0, g0
scaler g1, g0, g0
^d
```

In the invocation command, list sources in the order in which you want them assembled. The assembler concatenates all source files and interactive input, then assembles instructions and data into sections by order of appearance in the source text.

The following example assembles source from `ex1.s`, then from interactive input (the `i` option), then from `ex2.s`. Program elements from any one block of the input (for example, `ex1.s`) are available to any other block of the input (for example, `ex2.s`) as if all the input were in a single, sequential file.

```
gas960e ex1.s -i ex2.s
```

You can use other assembler options and source files with interactive input. The following example displays the assembler version and begins interactive input from the keyboard:

```
asm960 -V -i
```

To ensure your source files are not accidentally overwritten, use the `.s`, `.as`, or `.asm` protected filename extensions, as described in Naming the Object File on page 2-4.

Environment Variables

Environment variables set default operating parameters, such as search paths and the target architecture. For a list of environment variables and their uses, see your *Getting Started* manual. Define the environment variables before invoking the assembler.

The assembler supports all `I960` and `G960` environment variables, preferring those that match the invocation style. For example, when you invoke the assembler as `asm960`, the assembler looks first for `I960` environment variables, and for those settings not found, looks for `G960` environment variables. The environment variables used by the assembler are listed in Table 2-1 below.

Table 2-1.   Assembler Environment Variables

gnu Tools Name	CTOOLS Name	Purpose
G960ARCH	I960ARCH	Specifies target architecture.
G960IDENT	I960IDENT	Allows use of the COFF .ident directive.
G960INC	I960INC	Specifies include directory path.
G960BASE	I960BASE	Specifies base environment directory.

For more information on environment variables, see your host operating system documentation.

Selecting the Instruction Set and Libraries

The assembler reports an error for any instruction in your source text that is not valid for your target processor instruction set. To assemble for a specific i960 processor, you can define the `I960ARCH` or `G960ARCH` architecture environment variable. Then, you need use the `A` option (described in Chapter 3) only to override the environment variable. Leaving the environment variable undefined and omitting the `A` option assembles for the i960 KB architecture.

To specify the default instruction set, define the architecture environment variable as `SA`, `SB`, `KA`, `KB`, `CA`, `CF`, `JA`, `JD`, `JF`, `RP`, `HA`, `HD` or `HT`. For example, the following specify SA instructions unless a different processor is specified with the `A` option:

```
csh        setenv I960ARCH SA

sh or ksh    I960ARCH=SA;export I960ARCH
```

Other i960 processor software tools also use the architecture environment variable, as described in *Getting Started.*

Defining a Base Directory Path

You can set an environment variable to the assembler and utilities base directory. Such a value can be useful for setting other search-path environment variables. The following defines a base-directory environment variable named `G960BASE`:

```
csh        setenv G960BASE /usr/local/intel960

sh or ksh    G960BASE=/usr/local/intel960;export G960BASE
```

Defining an Identification String

To put assembler identification and information from the `.ident` directive into a COFF object file, define the `I960IDENT` or `G960IDENT` environment variable to any non-null value, as shown in the following example:

```
csh        setenv I960IDENT 1

sh or ksh    I960IDENT=1;export I960IDENT
```

Redirecting Error and Warning Message Output

The `I960ERR` variable lets you specify whether messages are directed to `stdout` or `stderr`. When `I960ERR` is not set, messages go to `stdout`. When `I960ERR` is set to a non-null string, the output goes to `stderr`. This variable functions under Windows only.

Building a Search Path for Include Files

You can extend the search path as follows for files included with `.include`:

- The assembler always searches the current directory first.
- You can specify additional directories with the I option, described in Chapter 3, Option Reference.
- You can specify a default list of directories, separated with colons (:), with I960INC or G960INC. When you do not use the I option, the assembler searches the directories specified by I960INC or G960INC.

Note that when you use both the `I` option and the `I960INC` or `G960INC` variables, the environment variable setting takes precedence.

The following commands set `G960INC` to `/usr/local/intel960/include`:

```
csh        setenv G960INC /usr/local/intel960/include

sh or ksh    G960INC=/usr/local/intel960/include;export G960INC
```

Building the Search Path for the Assembler Executable

To invoke the assembler from any directory, add the assembler directory to your `PATH` environment variable. Once the directory is in your `PATH`, you need not use the directory path name to invoke the assembler.

For example, with `I960BASE` set to your assembler base directory, you can augment your `PATH` as follows:

```
csh        setenv PATH $I960BASE/bin:$PATH

sh or ksh    PATH=$I960BASE/bin:$PATH;export PATH
```

Option Reference

This chapter describes the assembler options alphabetically. Table 3-1 summarizes the option names, arguments, effects, and defaults.

The following notation is used in this chapter:

{item|item}   Select one of the items listed between braces. A vertical bar (|) separates the items.

[items]   Items enclosed in brackets are optional.

Table 3-1   Assembler Options

Option	Effect of the Option	Default Action of the Assembler
A { SA | SB | KA | KB | CA | CF | JA | JD | JF | RP | HA | HD | HT | CORE0 | CORE1 | CORE2 | CORE3 | ANY }	selects the instruction set.	uses the instruction set specified by the I960ARCH or G960ARCH environment variable, if defined; otherwise, uses KB.
D *sym*[*=value*]	defines an absolute symbol. Symbols defined in this way can be used in .if and .ifdef expressions.	symbols must be defined in the source text.
d	retains debug information for local symbols beginning with L or a dot (.).	discards symbolic information for local symbols beginning with L or a dot (.).
G	generates big-endian COFF or ELF code.	generates little-endian code.
h	Help: prints a brief description of each option.	no help text is printed.

3

Table 3-1   Assembler Options (continued)

Option	Effect of the Option	Default Action of the Assembler
I *directory path*	adds directories to the search path for include files.	searches in the current directory and uses the I960INC or G960INC environment variable.
i	reads source from `stdin`.	reads source from files.
L *list_options*	generates a listing. Listing sub-options modify the listing behavior.	no listing is generated.
n	do not replace compare-and-branch instructions.	replaces compare-and-branch instructions.
o *objfile*	specifies an object filename.	uses a.out, b.out, e.out, or a filename derived from the first source filename.
p {c | d | b}	generates position-independent instructions and/or data.	generates position-dependent code and data.
V	displays a version message and continues the assembly.	displays no version message.
v960	displays a version message and stops the assembly.	displays no version message; the assembly proceeds.
W	suppresses the warning messages.	displays the warning messages.
x	generates warnings about architecture mismatches.	generates error message when it encounters architecture mismatch.
z	suppresses the object file header time-and-date stamp for COFF assembler.	writes the assembly time and date in the object file header.

A: Architecture

Select the architecture (instruction set)

A *arch*

arch   is SA, SB, KA, KB, CA, CF, JA, JD, JF, RP, HA, HD, HT, CORE0, CORE1, CORE2, CORE3, or ANY.

Discussion

To select your i960 processor instruction set, specify the A option. The assembler displays an error message for each instruction in the source text that is invalid for the selected architecture, or a warning when you use the x option.

Some pairs of A arguments have identical effects:
- SA is the same as KA.
- SB is the same as KB.
- CA is the same as CF.

Without the A option, the assembler uses the instruction set specified by the I960ARCH or G960ARCH environment variable. If the architecture environment variable is undefined, the assembler uses the KB instruction set.

New CORE Architecture Options

With CTOOLS release 5.1, the assembler supports architecture settings to allow the generation of code that is compatible with multiple i960 processor types. These settings are referred to as *core* architectures. Table 3-2 shows the types of i960 processors that are supported by each core architecture.

3

Table 3-2. **CORE0-3 Architecture Compatibilities**

-A Switch Used	Compatible Architectures
CORE0	Jx, Hx, RP
CORE1	Kx, Sx, Cx, Jx, Hx
CORE2	Jx, Hx
CORE3	Cx, Jx, Hx

i960 RP Architecture Available Instructions

With CTOOLS 5.1, the -ARP architecture option provides a subset of the 80960Jx instruction set. These restrictions are enforced by the assembler and other tools when the -ARP switch is used or when the RP architecture is specified using the I960ARCH or G960ARCH environment variables.

The following 80960Jx instructions are not supported with the -ARP architecture:

addi	halt	remo
addi<cc>	intctl	shli
atadd	ldt	shrdi
atmod	mark	spanbit
cmpdeci	modac	stib
cmpdeco	modi	stis
cmpinci	modify	stt
cmpinco	modtc	subi
concmpi	movl	subi<cc>
concmpo	movq	sysctl
eshro	movt	test<cc>
extract	notor	xnor
fault<cc>	remi	

i960 RP Architecture Address Mode Restrictions

The following addressing mode restrictions exist for MEM format instructions when specifying an 80960RP target:

- Indexed addressing modes are not available.
- IP-relative addressing is not available.
- Two-word MEM-format is not available for the following instructions:
  — ldl
  — stl
  — ldq
  — stq
  — bx
  — callx
- The `balx` instruction may only use register-indirect addressing (no offsets or displacements allowed).

Other consequences of using the RP output architecture are:

- The `calls` instruction may use register g13 or a literal as its target only.
- For the `modpc` instruction, the mask cannot specify the same register as the src/dst register.
- The `calljx` pseudo-instruction requires a second argument – a temporary register into which the address of the first argument can be loaded. See Chapter 7 for information on the use of `calljx` with the RP architecture.

Examples

1. Floating-point instructions are invalid for the CA. In the following, line 5 of the `fp.s` source file contains a `roundr` floating-point instruction:

   ```
   gas960e fp.s -ACA
   fp.s:5: Opcode is not in target architecture: "roundr".
   ```

2. The following sets `I960ARCH` to `SA`. Invoking the assembler without `A` uses the SA instruction set, which is identical to the KB.

   ```
   setenv I960ARCH SB
   gas960e fp.s
   ```

3

D: Define symbol

*Define an absolute
symbol from the
command line*

```
D symbol[=value]
```

symbol   is the name of the symbol you want to create.

value   is any valid non-relocatable expression.

Discussion

This option is intended to be used with the `.if` and `.ifdef` directives for conditional assembly. It resembles the similar compiler preprocessor option. If *=value* is left blank, then the value of *name* is set to 1. If you want to include spaces anywhere with *symbol=value*, then the entire *symbol=value* must be quoted.

Examples

The following creates a symbol called `foo` and sets its value to 1:

gas960 -D foo file.s

Within `file.s`, both of the following would evaluate to true:

```
.if foo
.ifdef foo
```

gas960 -D "foo = bar * 12" file.s

Within `file.s`, the symbol `bar` must be defined and be non-relocatable.

gas960 -D foo=0 file.s

Within `file.s`, the expression

`.ifdef foo`

is true, but the expression

`.if foo`

is false. (See the discussion of `.if` and `.ifdef` in Chapter 4.)

d: Debug symbols

*Keep debugging
information about
assembler temporary
symbols*

`d`

Discussion

The assembly output from the compiler contains local symbols beginning
with an `L`, as generated by a `gcc960` invocation of the compiler, or a dot
(`.`), as generated by an `ic960` invocation of the compiler. To retain such
symbols in the object-file symbol table, specify the `d` option. Without `d`,
the assembler removes all such local symbols.

Examples

The following shows the original C source text and the corresponding
assembly output with the local symbols generated by a `gcc960` invocation
of the compiler:

```
if (a==b)
    hi=b;
else
    hi=c;
```

The compiler assembly output (in the file `cmset.s`) is:

```
    cmpi  g0,g1
    be    L1
    b     L2
L1: st    g1,hi
    b     A1
L2: st    r6,hi
A1:
```

The following puts L1 and L2 in the object-file symbol table:

gas960c -d cmset.s

G: Big-endian target

*Produce a COFF or
ELF file for a big-
endian target*

G

Discussion

You can configure COFF or ELF program text-type and data-type sections
in either big-endian or little-endian byte order. For big-endian instructions
and data, specify the G option when:

- assembling for the C-series, J-series, or H-series architecture
- invoking the assembler with asm960 or gas960c (COFF only) or
  gas960e (ELF only)

Note that the i960 RP processor does not support big-endian byte order,
even though its core processor is an 80960Jx.

For byte-order information, see *C: A Reference Manual*.

Example

The following produces a COFF file for a big-endian target. The .text-
and .data-style sections of the COFF file is in the host byte order,
regardless of the `G` option.

```
gas960c -G -ACA big.a
```

I: Include-file search path

*Augment the search
path for include files*

I *path*

path   is a directory pathname.

Discussion

The assembler always searches the current directory for `.include`
filenames. You can augment the search path by:

- defining the I960INC or G960INC environment variable (described in
  Chapter 2) before invoking the assembler
- using the I option once or more when invoking the assembler

The search path sequence is:

1. the current working directory
2. any directories specified by `I960INC` or `G960INC`, in the order defined
3. any directories specified with `I`, in the order on the command line

3

Example

The following line in the `mathr.s` source file includes the `/mylib/fp.s` source file:

```
.include "fp.s"
```

when invoked as:

```
asm960 mathr.s -I/mylib
```

i: Input from stdin

*Include keyboard or
redirected input*

```
i
```

Discussion

You must provide source text from at least one of:
- a filename specified on the command line
- the keyboard, the redirected output of another command, or any other device designated as stdin

For `stdin` input, use the `i` option once in the assembler invocation. To assemble keyboard input, after entering the command line, enter lines of source text from the keyboard. To end the keyboard input, enter the `Ctrl-d` key sequence on a new line. To assemble redirected output from another application, pipe the application output into the assembler invocation. You need not enter `Ctrl-d` to end the redirected input.

You can use both the `i` option and zero or more source filenames. The assembler processes the `stdin` input in sequence with any source files.

When `stdin` is the first or only source specified on the command line, the default object filename is `a.out` for a COFF object file, `b.out` for a b.out-format object file, and `e.out` for an ELF format objet file. Use the `o` option to specify a different object filename.

For information on piping and on `stdin`, see your host system manual.

Examples

1. The following assembles several lines of code from the keyboard after the source text from the `predef.s` file:

    ```
    gas960e predef.s -i
        roundr   g0, fp0
        subr     fp0, g0, g0
        expr     g0, g0
        addr     1.0, g0, g0
        scaler   g1, g0, g0
    ^d
    ```

2. The following assembles the output from the `getpatch` script (invoked with the csh primitive `source` command) between `src1.s` and `src2.s`:

    ```
    source getpatch | gas960e src1.s -i src2.s
    ```

L: Generate a listing

Print an assembly listing on the screen or into a file

```
L [option [option-arg]]
```

option   is one of the following:

    `a`   list all lines, ignoring .nolist directives.

    `e`   list text and data in target-endian byte order.

f	print the listing into a file. option-arg is the name of the file.
n	do not list files included with .include.
t	use option-arg as the listing title. If the title contains spaces, then it must be quoted. This option overrides .title directives in the source.
z	do not print the listing header.

Discussion

With no options, the listing is printed on the standard output and all listing defaults are in effect. Options that do not take arguments can be catenated together (with no spaces) after a single `L` option. Space is optional between an option that takes an argument and the argument.

The byte order of the listing is always target-endian when listing data sections. For text sections, instructions are printed big-endian ("left-to-right") unless you specify `Le`, and then they are printed in target-endian byte order.

Examples

Several example listings follow. Where appropriate, the contents of the assembly language file is also shown. The first example shows the simplest listing invocation.

The file `listex1.s` contains:

```
    .title "Listing Example 1"
.text
    mov g0,g1
.data
    .short 0x1234
```

The assembler invocation command is:

```
$ gas960e -L listex1.s
```

```
Listing Example 1
ASSEMBLER VERSION: Intel 80960 ELF Assembler, 5.1.5002, Thu Sep 26
23:25:43 MST 1996
TIME OF ASSEMBLY: Mon Oct 21 23:48:16 1996 COMMAND LINE: gas960e -L
listex1.s

Number of errors:     0
Number of warnings:   0
Source File: listex1.s
1  000000                                    .title     "Listing Example
1"
2  000000                            .text
3  000000   5c881610                     mov    g0,g1
4  000004                            .data
5  000004   3412                              .short     0x1234
```

The next example shows `-Lz`, (don't print the listing header), and `-Lf`,
(print listing in a file).

$ gas960 -Lz -Lf listex1.L listex1.s

The file `listex1.L` contains:

```
Source File: listex1.s
    1  000000                              .title "Listing Example 1"
    2  000000                     .text
    3  000000   5c881610              mov g0,g1
    4  000004                     .data
    5  000004   3412                      .short 0x1234
```

The next example shows the effect of the `.nolist` directive on the listing.
The file `listex2.s` contains:

```
        .title "Listing Example 2"
    .text
        mov g0,g1
    .data
        .short 0x1234
    .nolist
        .asciz "Skip strings in the listing"
        .asciz "Skip this one too"
    .list
        .word 0x12345678
```

The assembler command is:

```
$ gas960e -L listex2.s
```

```
Listing Example 2
ASSEMBLER VERSION: Intel 80960 ELF Assembler, 5.1.5002, Thu Sep 26
23:25:43 MST 1996
TIME OF ASSEMBLY: Mon Oct 21 23:51:23 1996 COMMAND LINE: gas960e -L
listex2.s
Number of errors:    0
Number of warnings:  0
Source File: listex2.s
1  000000                             .title     "Listing Example 2"
2  000000                       .text
3  000000  5c881610                   mov     g0,g1
4  000004                       .data
5  000004  3412                       .short     0x1234
6  000006                       .nolist
7  000006                       .list
8  000006      7856 3412             .word     0x12345678
```

The .nolist directive can be defeated from the command line with -La:

```
$ gas960c -Lza listex2.s
```

```
Source File: listex2.s
    1  000000                             .title "Listing Example 2"
    2  000000                       .text
    3  000000  5c881610                   mov g0,g1
    4  000004                       .data
    5  000004  3412                       .short 0x1234
    6  000006                       .nolist
    7  000006      536b 69702073     .asciz "Skip strings in the
listing"
    7  00000c  7472696e 67732069
    7  000014  6e207468 65206c69
    7  00001c  7374696e 6700
    8  000022              536b     .asciz "Skip this one too"
    8  000024  69702074 68697320
    8  00002c  6f6e6520 746f6f00
    9  000034                       .list
   10  000034  78563412                   .word 0x12345678
```

Normally, text sections are listed in big-endian byte order. This matches left-to-right ordering of instructions in manuals. You can override this behavior on the command line with `-Le`. Note in the next example that the listing show the exact ordering of bytes in the object file:

```
$ gas960c -Lze listex2.s
```

```
Source File: listex2.s
    1  000000                          .title "Listing Example 2"
    2  000000                      .text
    3  000000  1016885c                mov g0,g1
    4  000004                      .data
    5  000004  3412                    .short 0x1234
    6  000006                      .nolist
    9  000034                      .list
   10  000034  78563412                .word 0x12345678
```

Here is another example that shows big-endian byte order in both the text and data sections:

```
$ gas960c -ACA -G -Lze listex2.s
```

```
Source File: listex2.s
    1  000000                          .title "Listing Example 2"
    2  000000                      .text
    3  000000  5c881610                mov g0,g1
    4  000004                      .data
    5  000004  1234                    .short 0x1234
    6  000006                      .nolist
    9  000034                      .list
   10  000034  12345678                .word 0x12345678
```

The next example shows the effect of the `.include` directive on the listing. The file `listex3.s` contains:

```
    .title "Listing Example 3"
.text
    mov g0,g1
.ifdef INCLUDE4
.include "listex4.s"
.endif
.data
    .short 0x1234
```

The file `listex4.s` contains:

```
foo:
    ldconst -1, g6
```

The assembler command is:

$ gas960 -Lz -D INCLUDE4 listex3.s

```
Source File: listex3.s
    1  000000                            .title "Listing Example 3"
    2  000000                        .text
    3  000000  5c881610                   mov g0,g1
    4  000004                        .ifdef INCLUDE4
    5  000004                        .include "listex4.s"

Source File: ./listex4.s
    1  000004                        foo:
    2  000004  59b01901                   ldconst -1, g6

Source File: listex3.s
    6  000008                        .endif
    7  000008                        .data
    8  000008  3412                       .short 0x1234
```

You can tell the assembler to not list include files with `-Ln`:

$ gas960 -Lzn -D INCLUDE4 listex3.s

```
Source File: listex3.s
    1  000000                            .title "Listing Example 3"
    2  000000                        .text
    3  000000  5c881610                   mov g0,g1
    4  000004                        .ifdef INCLUDE4
    5  000004                        .include "listex4.s"
    6  000008                        .endif
    7  000008                        .data
    8  000008  3412                       .short 0x1234
```

The last example shows how to override the `.title` directive from the command line with `-Lt`:

$ gas960e -Lt "LISTING EXAMPLE 247" listex3.s

```
LISTING EXAMPLE 247
ASSEMBLER VERSION: Intel 80960 ELF Assembler, 5.1.5002, Thu Sep 26
23:25:43 MST 1996
TIME OF ASSEMBLY: Mon Oct 21 23:54:54 1996
COMMAND LINE: gas960e -Lt LISTING EXAMPLE 247 listex3.s

Number of errors:    0

Number of warnings:  0

Source File: listex3.s
1  000000                              .title     "Listing Example 3"
2  000000                      .text
3  000000  5c881610                mov      g0,g1
4  000004                      .ifdef INCLUDE4
5  000004                      .include "listex4.s"
6  000004                      .endif
7  000004                      .data
8  000004  3412                    .short     0x1234
```

n: No compare-and-branch replacement

*Do not replace
compare-and-branch
instructions*

n

Discussion

For short conditional branches and jumps, you can save execution time
and space by using a single compare-and-branch (COBR) instruction. The
branch address can be any expression that evaluates to a 13-bit value.

To stop the assembler with an error when the branch address is out of
range, specify the n option. Without n, the assembler replaces the short-
range compare-and-branch instruction with two instructions.

3

Examples

1. In the following, `n` prevents the assembler from expanding the `cmpibe` instruction for the undefined external `m1`. The assembler displays an error message.

    ```
    $ gas960e -i -n
    cmpibe g0,g1,ml
    ^D
    can't use COBR format with external label
    ```

2. Without `n`, and with the `s` option, the following replaces `cmpibe`:

    ```
    0:  5a046090            cmpi    g0,g1
    4:  12fffffc            be      ml
    ```

o: Object filename

Name the object file

```
o objfile
```

`objfile`   is a valid filename.

Discussion

To specify the object filename, use the `o` option with a filename or a complete pathname. The default object filename is in the current directory:

*   a filename based on the first source filename on the command line, replacing any .s or .as source-filename extension with .o or appending .o to any other source filename after the extension.
*   a.out, when you invoke the assembler with asm960 or gas960c (for COFF output) with interactive input as the first or only source.
*   b.out, when you invoke the assembler with gas960 (for b.out-format output) with interactive input as the first or only source.
*   e.out, when you invoke the assembler with gas960e (for ELF output) with interactive input as the first or only source.

To avoid accidentally overwriting your source files, use a protected `.s`, `.as`, or `.asm` source-filename extension (the assembler does not overwrite existing files with one of these extensions).

Example

The following names the output file `prog1.o`:

asm960 myprog.asm -o prog1.o

p: Position independence

Mark the COFF or ELF object file as containing position-independent code or data

p *type*

type   is one of the following:
- `c`   indicates position-independent code.
- `d`   indicates position-independent data.
- `b`   indicates both position-independent code and data.

Discussion

To indicate position-independent code or data in the COFF or ELF file, use the `p` option. You can also use the `.pic`, `.pid`, and `.link_pix` directives, described in Chapter 4.

Example

The following marks the object file as position-independent:

asm960 -pb mypi23.s

3

V, v960: Version

Display the assembler version number and creation date

$$\begin{Bmatrix} \text{V} \\ \text{v960} \end{Bmatrix}$$

Discussion

To display a version message on `stdout` during assembly, use `V`. After displaying the message, the assembler continues. For information on `stdout`, see your host system manual.

To display the message without assembling, use `v960`. After displaying the message, the assembler stops.

The message includes the assembler version number and the assembler creation date and time.

Example

The following shows a sample version message:

```
$ gas960e myprog.asm -v960
```

```
Intel 80960 ELF Assembler, 5.1.5002, Thu Sep 26 23:25:43
MST 1996
```

W: Warnings

Suppress the warning messages

`W`

Discussion

To suppress the warning messages, use `W`. The error messages continue to appear on `stderr`. For information about the message formats, see Chapter 5. For information on `stderr`, see your host system manual.

Example

The following suppresses the warning messages during assembly:

```
asm960 myprog.asm -W
```

x: Allow mixed architectures

Allow architecture mismatches

`x`

Discussion

Using the `x` option causes the assembler to generate warnings (not errors) when it encounters mixed architectures (e.g., opcode not in target architecture).

3

z: Time stamp

*Suppress the time stamp
in the COFF output file*

```
z
```

Discussion

The assembler puts the current time and date in the file header of the
COFF output file. On most UNIX systems, to put Time Zero in place of
the current time stamp, specify `z`. Time Zero is 4:00, 31 December, 1969.

The `z` option has no effect on b.out or ELF format output.

Example

The following command specifies Time Zero for the time stamp:

```
gas960e -z file1.s
```

Directives

This chapter describes how to use the assembler directives in your source text. The Directives Reference section, which begins on page 4-10 provides an encyclopedia of the directives.

Table 4-1   Functions Performed by Directives

Category	Function	Directives
The input-specification directives	specify how the assembler finds and reads input and controls conditional assembly.	.if, .else, .endif, .ifdef, .ifndef, .ifnotdef, .include
The location-counter control directives	change the location counter and specify program sectioning.	.align, .bss, .data, .org, .section, .text
The data and memory initialization directives	assemble data in integer, ordinal, and real formats and initialize strings and memory blocks.	.ascii, .asciz, .byte, .double, .extended, .fill, .float, .single, .int, .long, .word, .short, .hword, .space
The symbol and debugger-support directives	define symbols and provide source and symbolic information for debugging.	.comm, .def, .endef, .desc, .elf_size, .elf_type, .equ, .global, .globl, .set, .lsym, .file, .lcomm, .line, .ln, .stabd, .stabn, .stabs, .scl, .size, .tag, .type, .val
The optimization directives	optimize memory addressing and procedure calls.	.leafproc, .lomem, .sysproc

i960 Processor Assembler User's Guide

Table 4-1   Functions Performed by Directives (continued)

Category	Function	Directives
The identification directive	identifies the assembly.	.ident
The abort directive	stops the assembly.	.ABORT
The position-independence directives	mark object files as position-independent.	.pic, .pid, .link_pix
The listing directives	control listing behavior.	.list, .nolist, .title, .eject

NOTE. *To assemble directives relevant for COFF development, invoke the assembler as* `asm960` *or* `gas960c`. *For directives relevant for b.out-format development, use* `gas960`. *For directives relevant to ELF development, use* `gas960e`.

Syntax

For the directives in your source text, use the following syntax:

`.name arg_string`

`name`   is the directive keyword. The leading dot (`.`) is required.

`arg_string`   is zero or more arguments, according to the requirements of the directive.

4

Specifying the Input

When invoking the assembler, you must specify a source file on the command line, as described in Chapter 2. For additional source text, you can include the contents of other files with the `.include` directive. The assembler inserts the included source text in place of the `.include` line.

You can specify blocks of source text to be assembled or ignored based on conditions determined during assembly. To delimit text for conditional assembly based on expression evaluations, use the `.if`, `.else`, and `.endif` directives. To delimit text for conditional assembly based on symbol definitions, use the `.ifdef`, `.ifndef`, and `.ifnotdef` directives. These directives are especially useful when used in combination with the `D` option (described in Chapter 3).

Controlling the Location Counter

The assembler uses the location counter to determine the address of each instruction or data item. The location counter begins at zero and increases by one for each byte assembled. A dot (`.`) symbolically represents the location counter.

Setting the Location Counter to a Specific Value

To manipulate the location counter directly:

`.align`	increments the location counter to the next address boundary fitting the alignment factor. Also stores the largest alignment found per section into the output file for later use by the linker.
`.org`	sets the location counter to the address you specify.
`.`(dot)	is the location-counter symbol for expressions and assignments.

To align data and instructions, use `.align`. The assembler starts the next instruction or data item on an address that fits the specified alignment, padding the intervening bytes with zeros or with a value you provide.

To set the location counter to a specific address, use `.org` or an assignment statement. The assembler gives the location counter the value you provide. You can express the new address in terms of the current location counter, represented by the dot (`.`). For example, the following advances the location counter by four bytes:

```
.org . + 4
```

The following example uses the location counter (`.`) as an operand behaving just like a local label:

```
        lda ., g5
        lda . - 4, g5
        lda . + 6, g7

alab:   b blab
blab:   cmpojne 0, 0, alab

        lda . - alab, g6
.set    symname, . - alab

.data
.word .
.word . + 4
.word . - 16
```

Moving the Location Counter to a Section

In COFF and ELF programs, you can define multiple sections of executable code (text-type sections), initialized data (data-type sections), or uninitialized data (bss-type sections). In b.out-format programs, you can define one `.bss` section, one `.text` text-type section, and one `.data` data-type section. For more information on section types and object-file formats, see the utilities user's guide.

You can start a new section or continue a previous section at any point in your source text with the section directives:

`.text`	puts executable code into a section named `.text`.
`.data`	puts initialized data into a section named `.data`.
`.bss`	puts uninitialized data into a section named `.bss`.
`.section`	for COFF and ELF programs only, puts executable code, initialized data, or uninitialized data into a section that you name.

COFF and ELF programs contain three or more sections; b.out-format programs contain exactly three sections. All object files contain at least the standard `.text`, `.data`, and `.bss` sections.

The order of sections in any program and the names and number of sections in a COFF or ELF program depend on the section definitions in your source text. Omitting the `.text`, `.data`, or `.bss` section directives creates the standard sections with zero size.

The first section directive creates the section and points the location counter to the beginning of the section. Later in the program, you can append text or data to existing sections with additional `.text`, `.data`, or `.bss` section directives or (for COFF and ELF programs) with additional `.section` directives specifying the same section names.

Initializing Data

To define data in memory, use the data-initialization directives according to the size of the memory block to be initialized and the data format:
- a single memory location with byte, ordinal or integer data
- a single memory location with real data types
- a memory block with string data
- a memory block with specified values or zeros

Initializing Byte, Ordinal, and Integer Data

To initialize data in byte, ordinal, and integer formats, use:

`.byte`   for byte-aligned data (8 bits or shorter).

`.short`, `.hword`   for half-word-aligned data (16 bits or shorter).

`.int`, `.long`, `.word`   for word-aligned data (32 bits or shorter).

You can specify a bit field of up to 32 bits with arguments to the byte-initialization, half-word initialization, and word-initialization directives. For more information, see the Directives Reference on page 4-10.

Initializing Floating-point Data

To initialize data in real or floating-point formats, use:

`.float`, `.single`   for 32-bit real data.

`.double`   for 64-bit real data.

`.extended`   for 80-bit real data (stored in 12 bytes).

How the processor treats real data depends on whether floating-point instructions are supported. The KB and SB include on-chip floating-point support and can use the accelerated floating-point (AFP-960) library. The other i960 processors emulate floating-point arithmetic in software.

For more information on floating-point support, see the AFP-960 library supplement and the processor handbooks.

Initializing String Data

To define character strings, use:

`.ascii`   for a string.

`.asciz`   for a null-terminated string.

For information on characters and escape sequences allowed in character strings, see Chapter 6.

To terminate the string with a null character (ASCII 0), for C language compatibility, use the `.asciz` directive. Using `.ascii` does not append a null character.

You can use `.byte` with a set of character constants in place of `.ascii`. For example, the following assemble the same data:

```
.ascii "cat"        # assemble an ascii string
.byte 'c', 'a', 't' # assemble 3 ascii bytes
```

Initializing Blocks of Memory

To put a repeated value into a block of memory, use:

`.fill`	fills the block with a value you specify.
`.space`	fills the block with zeros.

Defining Symbols

To define symbols, use:

`.globl`, `.global`	for global symbols in the object-file symbol table.
`.comm`	for common symbols in the object-file symbol table.
`.lcomm`	for local common symbols.
`.set`, `.equ`, `.lsym`	for non-relocatable symbols.

You can make a symbol external implicitly. Using a symbol without defining it adds it to the symbol table as an undefined external symbol. The symbol type and other symbolic information are derived from the context in which you use the symbol.

The assembler uses an internal symbol table that is not retained in the object file. To define and initialize non-relocatable symbols for the internal symbol table, use the `.set`, `.equ`, or `.lsym` directives.

With the optimization and debugging directives, you need use no additional symbol-definition directives. For more information on debugging and optimizing, see the Providing Debugger Information (page 4-8) section and the Optimizing section (page 4-9).

Providing Debugger Information

For COFF debugging, the compiler puts the following directives in the assembly output:

.def	begins a symbol definition.
.dim	specifies the array dimensions.
.endef	ends a symbol definition.
.line	sets a line number.
.ln	specifies a line number and the associated address.
.scl	declares a storage class.
.size	specifies the symbol size.
.tag	specifies an associated tag.
.type	declares a symbol type.
.val	declares the symbol value.

For b.out-format debugging, the compiler puts the following directives in the assembly output:

.desc	sets the symbol descriptor.
.lsym	creates and initializes a debugging symbol with no additional symbolic information.
.stabd	creates a debugging symbol for the location counter.

`.stabn`	creates and initializes a debugging symbol named the empty string (`""`).
`.stabs`	creates and initializes a debugging symbol with all possible symbolic information.

For ELF-format symbol table embellishment, the compiler puts the following directives in the assembly output:

`.elf_size`	sets the size of the symbol to the given quantity.
`.elf_type`	sets the ELF type of the symbol to the given type.

NOTE. *For ELF-format object files, the compiler provides debugging information in DWARF format in separate sections. See the 80960 Embedded ABI (Intel order #631999) for more information on DWARF format.*

For more information on the symbol table, see the *i960 Software Utilities User's Guide*.

Optimizing

To optimize leaf and system procedures, use:

`.leafproc`	identifies a procedure for branch-and-link optimization.
`.sysproc`	identifies a procedure for system-call optimization.

Marking Position Independence

To mark object files as position-independent, use:

`.pic`	indicates position-independent code.
`.pid`	indicates position-independent data.
`.link_pix`	indicates a position-dependent file intended for linking with position-independent code or data.

Controlling the Listing

When you request a listing, with the `L` command-line option, you can use these directives in the source text:

`.nolist`	turn off listing until the next `.list` directive.
`.list`	turn listing on again after a `.nolist`.
`.title`	specify the listing title.
`.eject`	add a form feed to the listing.

Directives Reference

This section describes the assembly directives alphabetically.

.ABORT

Abort the assembly

`.ABORT`

Discussion

Use `.ABORT` to stop assembly immediately, suppressing the object file.

Example

If `MAX_ERRS` is defined, assembly stops at the `.ABORT` line:

```
.ifdef MAX_ERRS
.ABORT
.endif
```

.align

*Align the location
counter*

```
.align align_expr [, data_expr]
```

`align_expr`	specifies the location-counter alignment. This expression is non-relocatable, non-negative, and evaluates to 31 or less.
`data_expr`	optionally specifies a byte value for filling bytes between the old and new location-counter addresses.

Discussion

To align the location counter on byte, word, double-word, or quad-word boundaries, use `.align`. The assembler does the following:

- Increments the location counter to the next value evenly divisible by 2align_expr.
- Puts data_expr in any unused bytes between the previous and newly aligned location-counter values. Omitting data_expr fills the intervening bytes with zeros.
- The align directive also updates the output section's alignment field in the section header to be the largest alignment per section. This field is used by the linker to enforce alignments of input sections.

- When not specified, the default alignments for the following OMFs are as follows:

	b.out	COFF	ELF
.text	2	$0_{(1)}$	2
.data	0	$0_{(1)}$	2
.bss	0	$4_{(1)(2)}$	4
.section text	NA	$0_{(1)}$	2
.section data	NA	$0_{(1)}$	2
.section bss	NA	$4_{(1)(2)}$	4

(1) The COFF assembler emits sections that are multiples of at least 32-bit words. Therefore the smallest default alignment is 2.

(2) The smallest alignment for bss sections in COFF is 4. Anything less is ignored.

Example

The following sets the location counter to 14 hexadecimal and increments it to 18 hexadecimal, the next address evenly divisible by 8 (2^3). The bytes between `0x14` and `0x18` are filled with zeros.

```
.org 0x14
.align 3
```

.ascii, .asciz

Assemble ASCII string data

```
.ascii "string"
.asciz "string"
```

string   is the character string to assemble. The quotation marks are required.

4

Discussion

To define a character string, use `.ascii` or `.asciz`. The first character occupies the address indicated by the location counter. Successive characters occupy sequential byte locations.

The `.asciz` directive ends the string with a null character; `ascii` does not.

Use a backslash (\) for special characters, as described in Chapter 6.

Examples

1. The following example assembles a string without a null end (13 bytes of information are assembled).

    `.ascii "Name\tAddress\n"`

2. The following example assembles the same string with a null end (14 bytes of information are assembled).

    `.asciz "Name\tAddress\n"`

.bss

Identify a symbol for uninitialized data storage

`.bss name, size_expr, align_expr`

name	is the symbol name.
size_expr	specifies a non-negative symbol size, in bytes.
align_expr	aligns the symbol. This expression is non-relocatable, non-negative, and evaluates to 31 or less. The assembler assumes a zero alignment if you specify a negative alignment.

Discussion

To create uninitialized symbols, use `.bss`. The *name* appears in the symbol table. The assembler extends the `.bss` section by reserving *size_expr* bytes, aligned on the next address evenly divisible by 2^{align_expr}. You can create any number of sections of uninitialized data in a COFF or ELF program. (Use `.section name,bss` to create another one). You can use any number of `.bss` directives to extend the `.bss` section.

For programs with no uninitialized data, the assembler inserts a `.bss` section of zero size.

Example

The following example, with the location counter starting at `0x14`, defines an uninitialized-data symbol named `buffer` at `0x18`, which is the next boundary evenly divisible by 8 (2^3). The assembler reserves 256 words (4 bytes each) in the `.bss` section.

```
.org 0x14
.bss buffer, 256 * 4, 3
```

Related Topic

`.section`

.byte

Assemble byte data

```
.byte [int_expr:]data_expr [, ...]
```

int_expr	is the number of bits (up to 8) to reserve for the data.
data_expr	is the byte value to assemble.

Discussion

To define byte or bit-field data, use `.byte`. The first byte or bit field is byte-aligned on the address indicated by the location counter. Successive bytes and bit fields occupy sequential locations and do not cross byte boundaries.

Each *data_expr* must evaluate to an eight-bit (one-byte) or shorter value. For a bit field shorter than eight bits, specify *int_expr*. The assembler truncates *data_expr* to *int_expr* number of bits. When the bit field cannot fit into the current byte, the assembler pads the current byte with zeros and aligns the bit field on the next eight-bit boundary.

Examples

1. The following allocates three bit fields from the least-significant to the most-significant bit within the byte. No bit field is allocated to the bits marked z, which contain zeros. The first byte is allocated at the address contained in the program counter (pc); the second byte is at the subsequent address (pc + 1).

    `.byte 3:1,2:1,5:1`

bit number:	7	6	5	4	3	2	1	0
pc	z	z	z	0	1	0	0	1
pc + 1	z	z	z	0	0	0	0	1

    Assembling for a big-endian target with the `G` option (see Chapter 3) allocates the bit fields from the most-significant to the least-significant bit within the byte:

bit number:	7	6	5	4	3	2	1	0
pc	0	0	1	0	1	z	z	z
pc + 1	0	0	0	0	1	z	z	z

2. The following assembles three characters:

    `.byte  'a','b','c'`

4

.comm

*Declare a common
symbol*

```
.comm name, data_expr [, elf_comm_alignment]
```

name   is the symbol name.

data_expr   specifies a non-negative symbol size, in bytes.

elf_comm_alignment   In ELF, you can optionally specify the alignment
                       of common symbols.

Discussion

To use a common symbol in more than one module, add the symbol to the object-file symbol table with `.comm`. Specify the size of the symbol in bytes with the `data_expr` argument. The assembler creates the symbol as an undefined external type. The linker resolves any references to the symbol from other modules.

The default alignment of a common symbol is determined by the log (base 2) of the size of the symbol:

Size	Default Alignment
0,1	0
2	1
3,4	2
5,6,7,8	3
>= 9	4

When you include a alignment expression, you override the default behavior. The alignment expression is useable only in the ELF assembler.

Examples

The following directives define three common symbols: `_a` occupies four bytes, `_b` occupies two bytes, and `_c` occupies one byte.

```
.comm _a,4
.comm _b,2
.comm _c,1
```

Another example: you have a table of 100 characters, and 100 shorts, and 100 words. You are using the ELF assembler and RAM space is critical so you align them manually:

```
.comm chars,100,0
.comm shorts,200,1
.comm words,400,2
```

.data

Create or extend a data-type section

```
.data
```

Discussion

To initialize variables, use `.data`. When a `.data` section already exists, this directive sets the location counter to the end of that section.

Omitting `.data` inserts a `.data` section of zero size.

Example

The following lines resume or begin the data section of a program:

```
.data
.word 0
.double 0d2.5e10
```

Related Topics

```
.bss
.section
.text
```

.def, .endef

*Provide symbolic
information for COFF
debugging*

```
.def name
```

`name`   is the symbol to be described.

Discussion

When you compile a high-level language program for COFF symbolic debugging, the compiler puts symbol descriptions in the assembly output. Such descriptions start with `.def` and end with `.endef`.

Example

The following is C language source text:

```
main() {
  int a;
}
```

The compiler produces the following symbol description for the debugger. The `_a` automatic variable appears on the stack `0x40` bytes from the integer frame pointer.

```
.def _a; .val 0x40; .scl 1; .type 0x4; .endef
```

Related Topics

```
.dim          .size          .val
.line         .tag
.scl          .type
```

.desc

Set the symbol descriptor for b.out-format debugging

```
.desc name, abs_expr
```

name   is the symbol name.

abs_expr   evaluates to a non-relocatable value.

Discussion

Compiling a high-level program for b.out-format symbolic debugging adds `.desc` symbol descriptors as the low-order 16 bits of *abs_expr*.

.dim

Declare the dimensions of an array for COFF debugging

```
.dim size_expr [,size_expr [,size_expr [,size_expr]]]
```

size_expr   evaluates to a positive integer for an array dimension.

Discussion

Compiling for COFF symbolic debugging puts symbol descriptions (`.def`, `.endef` pairs) for any arrays in the assembly output. The `.dim` directives specify up to four dimensions for each array.

Related Topics

`.def, .endef`	`.size`	`.val`
`.line`	`.tag`	
`.scl`	`.type`	

.double

Assemble double-
precision (64-bit)
floating-point values

`.double` *double_const* `[,` *double_const*`] ...`

double_const	is a non-relocatable 64-bit floating-point constant, or one of the following:
`nan` or `qnan`	generates a quiet nan value
`snan`	generates a signalling nan value
`+inf`	generates positive infinity
`-inf`	generates negative infinity

Discussion

To define double-precision floating-point data, use `.double`. The first value occupies the address indicated by the location counter. Successive values appear in sequential locations. To align the data on particular address boundaries, use the `.align` directive.

To ensure correct double-precision floating-point evaluation, precede each literal value in the expression with `0d`.

Example

The following line assembles the 64-bit value `3.14159`:

`.double 0d3.14159`

Related Topic

`.float  .extended`

.eject

Put a page break into the listing

`.eject`

Discussion

Use this directive in the source text to insert a page break (formfeed character) in the listing.

Related Topics

`.list            .nolist`
`.title`

4

.elf_size

*Adds the given size to
the named ELF symbol
table entry*

```
.elf_size name,size_expr
```

Discussion

The `.elf_size` directive applies only to the ELF assembler. This
directive adds the given size to the ELF symbol table. You can view the
ELF symbol table with the dumper/ disassembler (use `[g]dmp960 -t`).
This information is not used in DWARF.

Example

```
.text
foo:
lda 0,g0
ret
Lendfoo:
.elf_size foo,Lendfoo - foo
```

Related Topic

`.elf_type`

.elf_type

Adds the given type to the named ELF symbol table entry

```
.elf_type name,{ function | object }
```

Discussion

The `.elf_type` directive applies only to the ELF assembler. This directive adds the given type to the ELF symbol table entry. You can view the ELF symbol table with the dumper/ disassembler (use `[g]dmp960 -t`). This information is not used in DWARF.

Example

```
.text
foo:
lda 0,g0
ret
Lendfoo:
.elf_size foo,Lendfoo - foo
.elf_type foo,function
```

Related Topic

```
.elf_size
```

.else

See .if

.endef

See .def

.endif

See .if

.equ, .lsym, .set

*Set the value of a
symbol*

$$\begin{Bmatrix} \texttt{.equ} \\ \texttt{.lsym} \\ \texttt{.set} \end{Bmatrix} \textit{name, data\_expr}$$

name   is the symbol name.

data_expr   evaluates to a constant during assembly and is
                                     assigned to the symbol. The expression must be
                                     non-relocatable.

Discussion

To assign a new value to a symbol, use `.equ`, `.lsym,` or `.set`. The value
you specify defines or redefines the symbol type.

You may use the same *name* in more than one `.set` set per assembly.

A symbol defined with `.equ`, `.lsym`, or `.set` does not appear in the
symbol table unless the assembler finds a `.global` for the symbol name.

Examples

1. The following defines an integer symbol named `useful` with an initial value of `3`:

    ```
    .equ useful, 3
    ```

2. The following defines a global symbol named `x`, then specifies `x` as an integer with an initial value of `1`:

    ```
    .global x
    .set x, 1
    ```

3. The following sets the temporary symbol `xbase` to `10` and then to `24`:

    ```
    .lsym xbase, 10
    .lsym y, xbase
    .lsym xbase, (2*y)+4
    ```

.extended

*Assemble extended-
precision (80-bit)
floating-point data*

```
.extended float_expr [, float_expr]...
```

`float_expr`	is the 80-bit floating-point value to assemble, or one of the following:
`nan` or `qnan`	generates a quiet nan value
`snan`	generates a signalling nan value
`+inf`	generates positive infinity
`-inf`	generates negative infinity

Discussion

To define extended-precision floating-point data, use `.extended`. The first value occupies the address indicated by the location counter. Successive values appear in sequential locations. To align the data on particular address boundaries, use the `.align` directive.

To simplify addressing, the 80-bit floating-point data items defined with `.extended` occupy 12 bytes (96 bits) instead of 10 bytes. The additional two bytes are padded with zeros.

Example

The following line assembles the 80-bit value `3.14159`:

```
.extended   3.14159
```

Related Topics

```
.double
.float
```

.file

Identify the source file

```
.file "string"
```

string   is a source filename, without a pathname. The quotation marks are required.

Discussion

When you compile a high-level language program, the compiler puts a `.file` directive in the assembly source output to identify the primary high-level language source filename. Source debuggers use the `.file` information to identify the original C source file in b.out and COFF.

Source debuggers using ELF/DWARF obtain source file information from DWARF. However, in ELF, the `.file` directive modifies the ELF symbol table. You can view the ELF symbol table with the `[g]dmp960 -t` command.

Example

The following line identifies the source filename as `example.c`:

```
.file "example.c"
```

.fill

Initialize a memory block

```
.fill int_exp, size_expr, data_expr
```

`int_expr`	is a non-relocatable expression specifying how many times to repeat the fill data.
`size_expr`	is a non-relocatable expression specifying the size, in bytes, of the fill data (up to eight bytes).
`data_expr`	is a non-relocatable expression specifying the fill data. This expression must evaluate to a byte value.

Discussion

To initialize a memory block with a repeated value, use `.fill`. The assembler puts `data_expr` into memory `int_expr` times, beginning at the current location counter. The memory block occupies (`int_expr *` `size_expr`) bytes.

To align the memory block on a particular address boundary, use the `.align` directive.

Specify the size of *data_expr* with *size_expr*, up to eight bytes. When *size_expr* is larger than needed by *data_expr*, the excess high-order bytes contain zeros.

Examples

1. The following example initializes a memory block of 16 words, filling each word with `0x0f` (decimal 15).

    ```
    .fill  16, 4, 2*8-1
    ```

2. The `.fill` and `.space` directives are similar. The following lines have identical effects, initializing 4 bytes with the value 1 in each byte:

    ```
    .fill 4, 1, 1
    .space 4, 1
    ```

Related Topic

`.space`

.float, .single

*Assemble
single-precision (32-bit)
floating-point data*

$\left\{ \begin{matrix} \text{.float} \\ \text{.single} \end{matrix} \right\}$ *float_const* [,*float_const*] ...

float_const   is a 32-bit floating-point value to be assembled, or one of the following:

`nan` or `qnan`   generates a quiet nan value

snan	generates a signalling nan value
+inf	generates positive infinity
-inf	generates negative infinity

Discussion

To define single-precision floating-point data, use `.float` or `.single`. The first value occupies the address indicated by the location counter. Successive values appear in sequential locations. To align the data on particular address boundaries, use the `.align` directive.

Examples

1. The following line assembles the 32-bit value `3.14159`:

   ```
   .float 3.14159
   ```

2. The `.float` and `.single` directives have identical effects. The following lines assemble the 32-bit value `3.14159` twice:

   ```
   .float 3.14159
   .single 3.14159
   ```

Related Topics

```
.double
.extended
```

.global, .globl

Declare a global symbol

```
.global name
.globl name
```

name	is the name of the external symbol.

4

Discussion

To make the defined symbol *name* an external symbol, use `.globl` or `.global`. The linker resolves any references to the symbol from other modules.

Example

The following example makes the label `_exit` a global symbol:

`.globl _exit`

.hword

See .short

.ident

Include identification, date, and time in the object file

`.ident ident_str[, time_value]`

`ident_str`   identifies the compiler.

`time_value`   is the time value returned by the `time` function.

Discussion

To put compiler information in the symbol table, use the `.ident` directive
and the `I960IDENT` environment variable. Add an identification string
with *ident_str*. Put a specific time and date in the symbol table with
time_value. Omitting *time_value* puts the assembly time and date in
the symbol table.

Assembly language output from the compiler includes a `.ident` line.

Example

The following identifies the compiler at 10:20, 13 November, 1991:

```
.ident "iC960 V4.0X, 0x29216cde"
```

.if, .ifdef, .ifndef, .ifnotdef, .else, .endif

*Identify blocks of source
text for conditional
assembly*

$$
\left\{ \begin{array}{l} \text{.if } cond\_expr \\ \left\{ \begin{array}{l} \text{.ifdef} \\ \text{.ifndef} \\ \text{.ifnotdef} \end{array} \right\} symbol \end{array} \right\} \; stmt\_block \; [\text{.else } stmt\_block] \; \text{.endif}
$$

cond_expr	evaluates to a non-relocatable constant during assembly. The condition is false when *cond_expr* is zero and true otherwise.
symbol	is a symbol name.
stmt_block	is a block of one or more assembly statements.

4

Discussion

To conditionally assemble a block of source text, begin the block with `.if`, `.ifdef`, `.ifndef`, or `.ifnotdef` and end the block with `.endif`. The assembler selects the block to assemble as follows:

with `.if`   when *cond_expr* is non-zero

with `.ifdef`   when *symbol* is defined

with `.ifndef`   when *symbol* is not defined
or `.ifnotdef`

with `.else`   when the preceding `.if`, `.ifdef`, `.ifndef`, or `.ifnotdef` block is not selected

The `.else` directive ends an `.if`, `.ifdef`, `.ifndef`, or `.ifnotdef` block and the `.endif` directive ends any conditional-assembly block. You can nest conditional-assembly blocks.

These directives are best used in combination with the `D` option (described in Chapter 3).

Example

The following code assembles a double-precision floating-point value when `UseDouble` is nonzero and a single-precision floating-point value otherwise:

```
.if UseDouble
.double 3.14159
.else
.float 3.14159
.endif
```

.include

Insert source text from a file

```
.include "filename"
```

filename   is the include filename. The quotation marks are required.

Discussion

To insert source text from a file, use the `.include` directive. The contents of the included file are assembled in place of the `.include` statement.

To include a file from elsewhere than the current directory, you can:
• provide the complete pathname for the file
• use the `I` option (described in Chapter 3)
• define the `I960INC` or `G960INC` environment variable (described in Chapter 2).

Example

The following includes the source files `gen_d.asm` and `gen_e.asm` in the `stdin` input:

```
asm960 -i
.equ UseDouble, 1
.include "gen_d.asm"
.ifdef D_ERR
.ABORT
.endif
.include "gen_e.asm"
^d
```

4

.int

See .word

.lcomm

*Declare a local common
symbol*

.lcomm *symbol*, *size_expr*

symbol   names the symbol.

size_expr   evaluates to the length, in bytes, of the symbol.

Discussion

To declare a local common symbol, use the .lcomm directive. The
assembler allocates space in the .bss (uninitialized-data) section for the
symbol. The symbol appears in the symbol table as static.

Example

The following declares a 4-byte (1-word) local common symbol named
mycom:

.lcomm mycom, 4

.leafproc

Declare a leaf procedure

.leafproc *name*[, *bal_entr*]

name   is the leaf procedure name, as used in the high-level-language procedure reference.

bal_entry   is the branch-and-link entry-point label.

Discussion

You can optimize some procedure calls by substituting branch-and-link (`bal` or `balx`) instructions for call (`call` or `callx`) instructions. Identify such procedures with `.leafproc`. Specify the call entry point with *name* and the branch-and-link entry point with *bal_entry*.

A leaf procedure must meet the following requirements:

- The procedure must use registers minimally. Available registers are g0 through g7 for the first eight words of an argument list, g8 through g11 for an additional four words, and g13.
- The procedure can call no other procedures.
- The procedure can have no stack requirements, because no stack frame is allocated for leaf procedures.
- The procedure can have no argument block because register g14 contains the calling-procedure return address.
- The procedure cannot accept a variable argument list.

A leaf procedure has two entry points. The entry point for call instructions must provide a return sequence (prolog and epilog). The entry point for branch-and-link instructions must skip the return sequence.

4

When you compile a high-level language program for leaf-procedure optimization, the compiler identifies the leaf procedures, inserts the `.leafproc` directives, and generates the calling-convention blocks. For the call entry point, the compiler adds a single underscore (_) to the beginning of the procedure name. For the branch-and-link entry point, the compiler appends the suffix `.lf`.

If you don't specify a branch-and-link entry point, the assembler assumes that the branch-and-link entry point and the name entry point are the same.

Example

Compiling the following C source code produces the `_add` entry point:

```
int add(a, b)
int a,b;
{
    return(a+b);
}
```

The resulting assembly code is:

```
        .align  4
        .def _add; .val _add; .scl 108; .type 0x44; .endef
        .globl  _add
        .leafproc _add,  add.lf
_add:
        lda  LR2, g14
add.lf:
        mov  g14, g7
        addi g0, g1, g0
        mov  0, g14
        bx   (g7)
LR2     ret
        .ln  3
        .def _add; .val .; .scl -1; .endef
```

The `.scl 108` storage-class indicates an external leaf procedure. The `_add` is the call entry point. The `add.lf` is the branch-and-link entry point.

Since this example is compiled for source debugging, the compiler adds the `.def` directives.

.line

*Identify the line number
of a COFF debugging
symbol*

.line *int_expr*

int_expr   evaluates to a positive integer to be used as a line
number.

Discussion

Compiling for COFF source debugging puts .line directives in the
symbol definitions (.def, .endef pairs). The *int_expr* is the line
number for the line defining the symbol declared in the .def block.

Related Topics

.def   .ln
.endef

.link_pix

See .pic

4

.list

*Re-enable listing after a
.nolist*

```
.list
```

Discussion

Listing resumes on the instruction or directive immediately following this
directive. This option is useful in combination with `.nolist` when you
want to list only part of the source text.

Related Topics

```
.nolist          .title
.eject
```

.ln

*Specify a line number
within a function*

```
.ln int_expr [, addr_expr]
```

int_expr	evaluates to a positive integer to be used as a line number.
addr_expr	is the address of a line.

Discussion

Compiling for source debugging puts `.ln` directives in the source text to reset the source line numbers relative to the beginning of functions.

The assembler numbers the line containing the `.ln` directive as *int_expr* and the subsequent line as (*int_expr* + 1). Omitting *addr_expr* uses the location counter (`.`).

The `.ln` directive appears outside of any debugging symbol definition (`.def`, `.endef` pair).

Example

The following specifies line number `10` for the current position of the location counter:

`.ln  10, .`

Related Topic

`.line`

.lomem

Generate short memory-access instructions

`.lomem `*name*

name   identifies a symbol.

Discussion

This directive identifies a symbol's address as falling within the range
0 - 0xfff. This is the range of addresses that can be reached with the
absolute offset of a MEMA format instruction. The assembler uses
MEMA format for all MEM format instructions that reference the symbol.
This yields a space savings of 4 bytes per instruction over MEMB format.

The symbol's `.lomem` declaration must appear before the first use of the
symbol in a MEM format instruction. Otherwise the assembler defaults to
MEMB format.

To declare an entire section's symbols "lomem" use the `lomem` attribute to
the `.section` directive.

Example

The following example declares the symbol `foo` to be "lomem" and then
loads its contents into a register. The `ld` instruction that follows is 4 bytes
long.

```
foo:
.lomem foo
ld foo, r4
```

Related Topic

`.section`

.long

See .word

.lsym

See .equ

.nolist

Turn listing off

.nolist

Discussion

Listing stops immediately, and does not resume again until a .list
directive is seen. This option is useful in combination with .list when
you want to list only part of the source text. The assembler ignores this
directive when you use the La option on the command line.

Related Topics

.list   .title
.eject

.org

Set the location counter

.org *addr_expr*[, *abs_expr*]

addr_expr   is an integer expression.

abs_expr   is a non-relocatable byte value to be used as a fill
                        value.

Discussion

To point the location counter to a specific address, relative to the start of the current segment, use `.org`. Specify the new address with *addr_expr*. The assembler puts zeros in the bytes between the old and new addresses. You can specify a value other than zero with *abs_expr*.

The assembler does not issue a warning for large *addr_expr* values. Note that such use can fill up a hard disk quickly.

Example

The following example advances the location counter (`.`) by four bytes:

```
.org . + 4
```

Related Topics

```
.align          .section
.bss            .text
.data
```

.pic, .pid, .link_pix

Mark the object file as compatible with position-independent modules

$$\left\{ \begin{array}{l} \texttt{.link\_pix} \\ \texttt{.pic} \\ \texttt{.pid} \end{array} \right\}$$

Discussion

For position-independent programs, you must ensure consistent position independence of the object code and data across the object files. The linker examines each object file header and issues warning messages for mismatches. To suppress the warning messages, put one of the following directives at the beginning of your source text:

.pic   indicates a file containing position-independent code.

.pid   indicates a file containing position-independent data.

.link_pix   indicates compatibility with position-independent code, data, or both.

For more information on position independence, see your compiler manual.

.scl

Declare the storage class for COFF symbolic debugging

.scl *int_expr*

int_expr   evaluates to a positive integer indicating the storage class.

Discussion

Compiling for COFF symbolic debugging puts symbol descriptions (.def, .endef pairs) in the assembly output. The .scl directives specify the storage class for each symbol so described.

4

Example

The following example specifies the `113` storage class, indicating a static leaf procedure, for the `add` procedure:

```
.def _add; .val _add; .scl 113; .type 0x44; .endef;
```

Related Topics

```
.def            .line           .type
.dim            .size           .val
.endef          .tag
```

.section

*Creates or extends a
COFF or ELF program
section*

```
.section name, [, attribute_list]
```

`name`   identifies the section.

`attribute_list`   identifies the attribute(s) associated with this section.

Discussion

To create or extend a program section that you name, use `.section`. When the named section already exists, `.section` sets the location counter to the end of the named section. You can create multiple sections of instructions (text-type) or initialized data (data-type) or uninitialized data (bss-type) in a COFF or ELF program but not in a b.out-format program.

You can have any number of attributes for any given section. Attributes can be duplicated. An empty attribute list is allowed and means the section does not have any of the attributes. The attributes apply to both COFF and ELF unless otherwise indicated. If a COFF-only attribute is given to the ELF assembler, it is silently ignored and vice versa.

The attributes and their effects are:

`alloc`	The section should cause the linker to allocate memory (e.g., DWARF sections are not allocated).
`bss`	The named section takes on the same attributes as the .bss section.
`data`	The named section takes on the same attributes as the .data section.
`exec`	The named section contains executable code.
`info`	The section contains information only. (COFF only)
`lomem`	The named section is intended to be located in low memory. References to labels in this section will be via MEMA format instructions. (See `.lomem` for more information about MEMA format instructions.)
`msb`	The section is generated in big-endian byte order. (ELF only)
`pi`	The named section is position independent. (ELF only)
`read`	The section contains readable memory. (ELF only)

`super_read`
`super_write`
`super_exec`
	The memory space where the section resides corresponds to memory that is readable, writeable, or executable when the processor is in supervisor mode only. (ELF only)

`text`	The named section takes on the same attributes as the .text section.
`write`	The section contains writeable memory. (ELF only)

Note that for the `super_read`, `super_write`, and `super_exec` attributes, the assembler ORs the following bits into the corresponding section header flag word: `SHF_960_SUPER_READ`, `SHF_960_SUPER_WRITE`, `SHF_960_SUPER_EXECINSTR`. See the 80960 ABI specification (Intel order #631999) for more information. The linker passes these bits on from input files to the output file, ORing all of the flagwords together. The runtime does not ensure that these semantics are enforced. These bits are here for convenience, and to let you specify code bound for supervisor mode.

Example

The following lines begin a data section named `sram` that is bound for low memory, create another data section named `mydata` that is position-independent, and then continue `sram`:

```
.section sram, data, lomem
.globl _a
_a: .space 4

.section mydata, data, pi
.globl _b
_b: .word 444

.section sram, data
.globl _d
_d: .word 44
```

Related Topics

```
.bss              .text
.data             .lomem
```

.set

See .equ

.short, .hword

Assembles 16-bit data

$\left\{\begin{array}{l}\texttt{.short}\\\texttt{.hword}\end{array}\right\}$ [*int_expr*] *data_expr* [, ...]

int_expr   is the bit-field length, up to 16 bits.

data_expr   is a 16-bit value to be assembled.

Discussion

To define half-word or short-integer data, use the .short or .hword directive. The first value occupies the address specified by the location counter. Successive values occupy sequential two-byte locations. To align the data on particular address boundaries, use the .align directive.

For a bit field, specify the number of bits with *int_expr*. The assembler truncates the *data_expr* value to *int_expr* number of bits. When the bit field cannot fit into the current half-word, the assembler fills the remaining bits of the current half-word with zeros and begins the bit field on the next 16-bit boundary.

Examples

1. The .short and .hword directives have identical effects. The following assembles two half-words of data:

    ```
    .hword 0xFEFE
    .short 0xEFEF
    ```

2. The following allocates three bit fields from the least-significant to the most-significant bit within the half-word. No bit field is allocated to the bits marked z, which contain zeros. The first half-word is allocated at the address contained in the program counter (pc); the second word is at the subsequent address (pc + 2).

    ```
    .hword 3:3, 6:62, 9:21
    ```

bit number:	7	6	5	4	3	2	1	0
pc	1	1	1	1	0	0	1	1
pc + 1	z	z	z	z	z	z	z	1
pc + 2	0	0	0	1	0	1	0	1
pc + 3	z	z	z	z	z	z	z	0

Assembling for a big-endian target (with the G option) allocates the bit fields from the most-significant to the least-significant bit within the byte:

bit number:	7	6	5	4	3	2	1	0
pc	0	1	1	1	1	1	1	1
pc + 1	0	z	z	z	z	z	z	z
pc + 2	0	0	0	0	1	0	1	0
pc + 3	1	z	z	z	z	z	z	z

Related Topics

.ascii	.extended	.octa
.asciiz	.float	.quad
.byte	.int	.single
.double	.long	.word

.single

See .float

.size

Declare the size of a symbol for COFF debugging

.size *size_expr*

size_expr   is the size of a symbol, up to 64 kilobytes (65535 in decimal or 0xFFFF in hexadecimal). The expression must evaluate to a positive integer.

Discussion

Compiling for COFF symbolic debugging puts symbol descriptions (.def, .endef pairs) in the assembly output. The .size directive defines the size of a symbol so described. For structures and arrays, .size specifies the total extent of the symbol.

Due to COFF limitations, specifying too large a symbol size generates invalid debug information.

Related Topics

.def	.line	.type
.dim	.scl	.val
.endef	.tag	

.space

*Initialize a memory
block with byte values*

```
.space size_expr[, data_expr]
```

size_expr	is the number of bytes to be initialized. The expression must evaluate to a positive integer.
data_expr	is a byte value to be put repeatedly into the memory block.

Discussion

To increment the location counter and initialize the intervening bytes, use
`.space`. This directive advances the location counter by *size_expr*
bytes and fills the bytes between the old and new locations with the
data_expr value. Omitting *data_expr* fills the intervening bytes with
zeros.

Examples

1. The following example initializes 64 bytes with zeros:

    ```
    .space  16 * 4
    ```

2. The `.fill` and `.space` directives are similar. Using `.space` has the
    same effect as using `.fill` with a data size of 1 byte. The following
    lines have identical effects, initializing 4 bytes with the value `1` in
    each byte:

    ```
    .fill 4, 1, 1
    .space 4, 1
    ```

For more examples, see Chapter 8.

Related Topic

```
.fill
```

.stabd, .stabn, .stabs

Create b.out-format
debugging symbols

```
.stabd type, other, desc
.stabn type, other, desc, value
.stabs name, type, other, desc, value
```

name	is the new symbol name, with any characters except `\000`.
value	is a non-relocatable expression initializing the symbol.
type	is a non-relocatable expression for the symbol type.
other	is a non-relocatable expression.
desc	is a non-relocatable expression for the symbol descriptor.

Discussion

For symbolic debugging, you can create symbols that cannot be referenced by name during assembly. Such symbols can have the following attributes:

value	To record the location counter during assembly, define a symbol with `.stabd`. For any other initial value, use `.stabn`, or `.stabs`.
type	Provide the symbol type as the low-order eight bits of a non-relocatable expression.
name	Since the symbol name can contain almost any character, a debugger can use this field for additional information.

4

other	The debugger can use this attribute for any purpose. For `.stabd`, `.stabn`, and `.stabs`, provide the initial *other* value as the low-order eight bits of a non-relocatable expression.
desc	Provide the symbol descriptor as the low-order 16 bits of a non-relocatable expression.

.sysproc

Declare a system procedure

```
.sysproc name, int_expr
```

name	is the procedure name.
int_expr	is the system-procedure table index. The expression must evaluate to an integer between zero and 259, inclusive.

Discussion

To use the i960 processor system-call feature, identify functions as system procedures with the `.sysproc` directive. You need specify any function as a system procedure only once in your program.

Assign each system procedure an *int_expr* index number in the system procedure table, as follows:

- For b.out-format programs, the index must be between 1 and 253, inclusive.
- For COFF and ELF programs, the index must be between zero and 259, inclusive.

If you don't provide an index number, the assembler assigns the special index number -1. This number tells the linker to look for the real index number in another module. You must supply the real index number in at least one assembly source file or your application will not link.

For more information on system calls and the system procedure table, see your processor manual.

Example

The following example specifies `_add` as a system procedure with an index of `29`:

```
        .align  4
        .globl  _add
        .sysproc _add, 29
_add:
        addi g0, g1, g0
        ret
        .ln  3
```

Related Topic

`.leafproc`

.tag

*Declare a tag for a
COFF debugging
symbol*

`.tag` *string*

string   is the symbol name.

Discussion

Compiling for COFF symbolic debugging puts symbol descriptions (`.def`, `.endef` pairs) in the assembly output. References from within a symbol-description block to a previous block use the `.tag` directive. The *string* is the name of the symbol defined in the previous block.

In a structure or union symbol-description block, the `.tag` identifies a structure or union defined in a previous block.

Related Topics

`.def`	`.line`	`.type`
`.dim`	`.scl`	`.val`
`.endef`	`.size`	

.text

Create or extend a text-type section

```
.text
```

Discussion

To create a program section for instructions, use the `.text` directive. If a `.text` section already exists, this directive sets the location counter to the end of that section. Omitting `.text` inserts a `.text` section of zero size.

Example

The following lines resume or begin the `.text` section of a program:

```
.text
mov r3, r4
ldconst 0xff, g5
```

Related Topics

`.bss`	`.section`
`.data`	

.title

Specify the listing title

```
.title "string"
```

string   is the title you want to appear in the listing. The
                                    quotation marks are required.

Discussion

Use this directive anywhere in the source text to specify the listing title.
Only the first such directive has meaning. This directive is ignored when
you also give the Lt command-line option.

Related Topics

```
.list           .nolist
.eject
```

.type

*Declare the COFF
debugging-symbol type*

```
.type int_expr
```

int_expr   evaluates to a positive integer specifying a COFF
                                    type.

Discussion

Compiling for COFF symbolic debugging puts symbol descriptions (.def,
.endef pairs) in the assembly output. The .type directive adds type
information to the symbol description.

Related Topics

```
.def            .line           .tag
.dim            .scl            .val
.endef          .size
```

.val

Declare a debugger-
symbol value

.val *data_expr*

data_expr   is the value of the symbol.

Discussion

Compiling for COFF symbolic debugging puts symbol descriptions (.def, .endef pairs) in the assembly output. The .val directive initializes the symbol.

Example

The following example shows .val and other debugging directives in a symbol-description block describing the myfcn function:

```
myfcn:
.def myfcn; .val myfcn; .scl 2; .type 0x44; .endef
```

Related Topics

```
.def            .line           .tag
.dim            .scl            .type
.endef          .size
```

.word, .int, .long

Assemble word data

```
⎧ .int  ⎫
⎨ .long ⎬ [int_expr:]data_expr[, ...]
⎩ .word ⎭
```

int_expr   is the length of the data field, up to 32 bits.

data_expr   is the 32-bit integer value to be assembled.

Discussion

To define word-aligned integer, word, or bit-field data, use `.int`, `.long`, or `.word`. The first value occupies the address specified by the location counter. Successive values occupy sequential locations. To align the first value on a particular address boundary, use the `.align` directive.

For a bit field, specify the number of bits with *int_expr*. The assembler truncates the *data_expr* value to *int_expr* number of bits. When the bit field cannot fit into the current word, the assembler fills the remaining bits of the current word with zeros and begins the bit field on the next 32-bit boundary.

Examples

1. The `.int`, `.long`, and `.word` directives have identical effects:

    ```
    .int 5
    .long 5
    .word 5
    ```

2. The following allocates three bit fields from the least-significant to the most-significant bit within the word. No bit field is allocated to the bits marked z, which contain zeros. The first word is allocated at the address contained in the program counter (pc); the second word is at the subsequent address (pc + 4).

    ```
    .int 16:1,10:1,8:1
    ```

bit number:	7	6	5	4	3	2	1	0
pc	0	0	0	0	0	0	0	1
pc + 1	0	0	0	0	0	0	0	0
pc + 2	0	0	0	0	0	0	0	1
pc + 3	z	z	z	z	z	z	0	0
pc + 4	0	0	0	0	0	0	0	1
pc + 5	z	z	z	z	z	z	z	z
pc + 6	z	z	z	z	z	z	z	z
pc + 7	z	z	z	z	z	z	z	z

Assembling for a big-endian target (with the G option) allocates the bit fields from the most-significant to the least-significant bit within the byte:

bit number:	7	6	5	4	3	2	1	0
pc	0	0	0	0	0	0	0	0
pc + 1	0	0	0	0	0	0	0	1
pc + 2	0	0	0	0	0	0	0	0
pc + 3	0	1	z	z	z	z	z	z
pc + 4	0	0	0	0	0	0	0	1
pc + 5	z	z	z	z	z	z	z	z
pc + 6	z	z	z	z	z	z	z	z
pc + 7	z	z	z	z	z	z	z	z

Related Topics

```
.ascii        .double        .hword
.asciiz       .extended      .short
.byte         .float         .single
```

Messages   5

Assembler error and warning messages appear on `stderr` as:

source, [n]: message

source	is the source filename.
n	is the line number of the error, appearing only for source-assembly errors. File, I/O, and command-line errors do not have source line numbers.
message	is the text of the message.

Error messages report file-specification or syntax errors during assembly. In addition to producing a message, the assembler acts on the severity of the error as follows:

- For fatal errors, assembly stops. No object file is produced.
- For non-fatal errors, assembly continues to the end of the input, but no object file is produced.
- For warnings, assembly continues and an object file is produced.

Assembly Language

6

This chapter provides:

- an overview of assembly language directive and instruction syntax
- a description of the assembly language elements
- a description of the assembly language statement syntax

Assembly Language Statement Format

Assembly language source is a sequence of statements separated with newline characters or semicolons. A valid assembly language statement follows this syntax:

`[`*label*`]` `[`*keyword*`]` `[`*operands*`]`

A *keyword* can be any of the following:

Directives	affect the assembly, as explained in Chapter 4.
Instructions	specify processor operations.
Pseudo-instructions	(also called pseudo operations) are replaced with machine instructions by the assembler or linker.

You can write null statements, including empty lines and lines with only a semicolon. For null statements, the assembler generates no machine code, allocates no storage, and does not change the location counter.

A statement can contain one or more labels. Place labels before instruction keywords, as described in the Labels section of this chapter. One or more operands can follow the keyword, as needed.

6

Lexical elements are the building blocks of assembly language statements, used to construct labels, keywords, and operands. The lexical elements supported by the assembler are:

- the character set
- tokens and separators
- identifiers
- constants
- labels
- operators
- expressions
- comments

Character Set

The character set used in assembly language programming is a subset of the ASCII character set. Table 6-1 shows the valid character set.

Table 6-1   Assembly Language Character Set

Characters	Comment
ABCDEFGHIJKLMNOPQRSTUVWXYZ	alphabetic, UPPERCASE
abcdefghijklmnopqrstuvwxyz	alphabetic, lowercase
0123456789	numbers
	characters
+ - * / ( ) [ ] < > ; ' . " _ : ? @ $ & # \ | % ! ~ ^	special characters
space tab newline[1]	delimiters

1   In Windows, a newline is a carriage return-linefeed combination while on UNIX it is a linefeed only.

The assembler is case-sensitive. You can write labels and comments in uppercase or lowercase, but references to a label must match the case in the label definition. For example, the label zz is different from the label zz. instruction mnemonics and most directives use only lowercase characters.

Tokens and Separators

The assembler processes statements constructed of tokens and separators. Assembly language tokens include identifiers (symbols or names), constants, operators, and keywords.

The keywords are directives, instruction mnemonics, and pseudo-instructions. Statement syntax depends on the keyword. Directives are described in Chapter 4. Machine instructions are described in this chapter briefly, and in greater detail in the processor user's manuals. Pseudo-instructions are described in Chapter 7.

Separate identifiers or constants with at least one blank space or tab character. You can also use a blank or tab to separate other tokens such as operators or keywords. Put no blanks or tabs within tokens.

Identifiers

An identifier is a sequence of alphanumeric characters, including the underscore (_), dollar sign ($), and period (.). The first character in an identifier must not be numeric. Identifiers can have up to 255 significant characters.

Constants

There are three kinds of constants: simple, character, and string.

Simple Constants

Simple constants are either numeric or single-character. The digits in numeric constants are:

```
0123456789
abcdef
ABCDEF
```

Digits 0 through 9 represent corresponding numeric values, depending on the current number base (octal, decimal, or hexadecimal). The digits a, b, c, d, e, and f are identical to A, B, C, D, E, and F, representing hexadecimal values corresponding to the decimal values 10 through 15. Integer and ordinal constants are 32-bit-wide, two's-complement numbers.

The following types of constants are formed:

octal   An octal constant is a sequence of the digits 0 through 7 with a leading 0. For example, 012 represents decimal 10.

decimal   A decimal constant is a sequence of the digits 0 through 9 without a leading 0. For example, 10 represents decimal 10.

hexadecimal   A hexadecimal constant is a sequence of the digits 0 through 9, a, b, c, d, e, f, or A, B, C, D, E, F with a prefix of 0x or 0X. For example, 0x1a represents the decimal value 26.

floating-point   A floating-point constant consists of one or more characters that the C library function atof recognizes as a floating-point number, preceded by an optional prefix listed in Table 6-2.

Representing Floating Point Numbers

All floating-point constants are represented according to the *IEEE Standard for Binary Floating-point Arithmetic*.

Table 6-2   Prefixes for Floating-point Constants

Prefix	Used for
0f or 0F	Single-precision value, 32 bits
0d or 0D	Double-precision value, 64 bits
0e or 0E	Extended-precision value, 80 bits

The characters e, E, d or D designate the exponent field. You can use only
.0 and 0.0 as floating-point literals with numerics instructions, as shown
in Table 6-3.

Table 6-3   Floating-point Literals

Representation	Value Assembled
0.0	0f+0.0
1.0	0f+1.0

Example 6-1 uses numeric constants and literal values in assembly
language instructions.

Example 6-1   Example of Constants and Literal Values

```
                            /* example of numeric constants */
mov 31,g5                                      /* decimal */
mov 037,g5                              /* same in octal */
mov 0x1f,g5                               /* same in hex */
movr 0.0,g5                             /* float literal */
movrl 0f1.0,g4                          /* float literal */
addr 0.0,1.0,g0                             /* together */
```

Character Constants

A single-character constant is an ASCII character enclosed within
apostrophes ('). (The apostrophe is ASCII decimal character 39.)

The value of an ASCII character constant is either the ASCII code for the
character or the C language interpretation of an escape sequence,
beginning with a backslash, as shown in Table 6-4.

Table 6-4   **Character Constants**

Escape Sequence	Interpretation
\b	backspace
\f	form feed
\n	new-line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\'	apostrophe
\"	quotation mark
\octal constant	ASCII value of constant

String Constants

A string constant has the same syntax and semantics used in the C
language. Each string begins and ends with a quotation mark (`"`). All C
language conventions for the backslash are observed. See Table 6-4 for a
summary.

Strings are identified by value and length. However, the assembler does
not implicitly end strings with a null byte, unlike the C compiler. For
information on adding ASCII string data to your assembly files, see the
`.ascii` and `.asciz` directive entries in Chapter 4.

Labels

A label is a symbol with a location counter value and type. The assembler
recognizes the following kinds of labels:

global   is an alphanumeric identifier, also called a name.

local   is a single decimal digit (`0` to `9`), also called a
                                   numeric label.

Global labels are uniquely defined and remain in the output symbol table unless the label name begins with a period (`.`) or an `L`. Labels beginning with a period (`.`) or an `L` can be included in the symbol table by using the assembler `-d` option. See Chapter 3 for more information.

Name (Global) Labels

A global label consists of an identifier followed by a colon (`:`). In effect, a name label assigns the current value and section (e.g., .text or .data) of the location counter to the name. A global label is referenced by its name. Global labels beginning with a dot (`.`) or an `L` are discarded from the output symbol table, unless you use the `-d` option.

The assembler generates an error if a symbol is multiply defined.

Numeric (Local) Labels

A numeric label consists of a digit `0` to `9` followed by a colon (`:`). Numeric labels define temporary symbols of the form *nb* and *nf*, where *n* is the numeric digit of the label. References to symbols of the form *nb* refer to the first numeric label *n*`:` backward from the reference; *nf* symbols refer to the first numeric label *n*`:` forward from the reference.

As with global labels, a numeric label assigns the current value and section (e.g., .text or .data) of the location counter to a symbol. Unlike global symbols, which you can define only once within an assembly, numeric labels are local symbols. Therefore, programs can define several identical numeric labels (the same digit) within an assembly.

Expressions

An expression is a sequence of symbols representing a calculated value. An expression can consist of identifiers, constants, operators, and other expressions. Each expression has a type. Expressions can be grouped by enclosing them within parentheses.

Integer quantities appearing in arithmetic expressions are represented internally as two's-complement numbers with 32-bit precision. You can add only one forward-referenced external symbol to an expression. Further, you can subtract only one forward-referenced external symbol from an expression. The exception to these rules is that the difference expression of backwards-reference external symbols in the same section is treated as a constant value (see Example 6-2).

Example 6-2   Forward-reference External Symbol in Expressions

```
/* LEGAL: Forward (+) Reference to a symbol */

.word _label

/* LEGAL: A single (+) and single (-) forward reference */

_label4:
_label5:

_label6:
_label7:

/* LEGAL: The difference expression of two labels in the
 * same section is treated as a constant, allowing for
 * other (+) or (-) references, up to 1 each maximum.
 */
.word (_label5 - _label4) + (_label7 - _label6) - _label8 + _label9
```

Operators

The assembler recognizes certain operators that you can use to form valid expressions. These operators and the operations they represent appear in Table 6-5.

Table 6-5   Expression Operators

Symbol	Operation
+	addition
-	subtraction
*	multiplication
/	division
%	modulo
&	bitwise and
|	bitwise or
~	one's complement
^	bitwise exclusive or
>>	logical right shift
<<	logical left shift
< <= > >=	less than, less than or equal to, greater than, greater than or equal to
== !=	equals, not equals
&&	logical and, does not short circuit
||	logical or, does not short circuit
!	!a == if (a) then 0 else 1; (logical negation)

In Table 6-6, operators are listed in order of precedence from highest to lowest.

Table 6-6   Operator Precedence

Type	Operators
unary	-, +, !, ~
binary	*, /, %
binary	+, -
binary	<<, >>

Table 6-6 **Operator Precedence** (continued)

Type	Operators
binary	<, <=, >, >=
binary	==, !=
binary	&
binary	^
binary	|
binary	&&
binary	||

All binary operators with the same precedence are evaluated from left to right in the expression, except for any evaluation order enforced by parentheses.

Expression Types

The assembler deals with several types of symbols and expressions. The assembler recognizes the following expression types:

absolute
: An absolute symbol is defined ultimately from a constant. Applying the linker to the output file does not affect the value of absolute symbols or expressions.

bss
: The value of a `.bss` symbol is measured as the number of bytes from the beginning of the `.bss` section of a program. Like `.text` and `.data` symbols, the value of a `.bss` symbol can change on different linker runs.

data
: The value of a `.data` symbol is measured as the number of bytes from the origin of the `.data` section. Like `.text` symbols, the value of `.data` symbols can change on different linker runs. After the first `.data` statement, the value of the location counter 0 of the `.data` section.

external absolute text, data, or bss	Symbols can be declared as `.globl` but defined within an assembly as absolute `.text`, `.data`, or `.bss` symbols. These symbols are used exactly as if they were not declared as globals. However, their value and type are available to the linker so that the program can be combined with others that reference these symbols.
register	The assembler recognizes the predefined register symbols shown in Table 6-6.
text	The value of a `.text` symbol is measured as a number of bytes from the beginning of the `.text` section of the program. When assembler output is linked, `.text` symbols can change in value. Most `.text` symbols are labels in the assembly that define data or instruction locations. At the start of an assembly, the value of the location counter 0 of the `.text` section.
undefined	When the assembler identifies a new symbol during assembly, the symbol is considered undefined. It becomes defined when it is assigned a value or location. A symbol can subsequently become undefined again if assigned an undefined expression. Undefined operands are not permitted with certain operators. A symbol that remains undefined after assembly is considered an undefined external.
undefined external	A symbol declared `.globl` but not defined in the current assembly is an undefined external. If you declare such a symbol, use the linker to combine the assembler's output with another routine that defines the undefined reference.

Table 6-7   **Predefined Register Symbols**

Registers	Symbol	Alias	Purpose
local	r0*	pfp	previous frame pointer
	r1*	sp	stack pointer
	r2*	rip	return instruction pointer
	r3 through r15		general-purpose
global	g0 through g13		general-purpose
	g14		linkage for bal instruction
	g15*	fp	frame pointer
floating-point	fp0 through fp3		general-purpose
special function	sf0 through sf4		registers for architecture-specific functions such as DMA or cache control. See your processor user's manual.
processor state	ip		instruction pointer
	ac		arithmetic controls
	pc		process controls
	tc		trace controls

*   You must use the aliases, not the symbols, for registers r0, r1, r2, and g15.

Example 6-2 uses local, global, floating-point registers, and the instruction pointer and is valid only for processors with the numerics architecture (i960 SB or KB processors). Users targeting the KA, SA, CA, CF, JA, JF, JD, HA, HD, HT, or RP processor can use the AFP-960 library for emulated floating-point operation, which is described in the *i960 Processor Library Supplement*. In the assembly source, the register names must not be capitalized.

Example 6-3   Example of Register Usage

```
                        /* example of fp register usage */
    movr  1.0, fp0      # set fp0 = 0f+1.0
    movr fp1, r6        # convert real formats
    ld 0(g14), r0       # load based on g14
    addrl 1.0, fp1, g8  # g8:g9 = fp1 + 0f+1.0
    st g5, 4(ip)        # store based on ip
    lda (ip), g14       # g14 = value of ip
```

As shown in the example, the instruction pointer register can be used only to indicate indirection in instructions that allow an IP indirect addressing mode. You cannot use `ip` as an operand specifier by itself; it is not a general-purpose register. See the Memory-addressing Modes section in this chapter for additional information on memory addressing modes.

The special function registers `sf0` through `sf2` are defined in the i960 processor architecture but implemented only on the CA and CF processors. The i960 Hx processor supports special function registers `sf0` through `sf4`. For more information about these registers, see your processor manual.

Type Propagation in Expressions

When operands are combined using operators, the resulting expression is assigned a type that depends on the types of the operands and on the operator. For purposes of expression evaluation, the assembler recognizes these types:

- undefined
- absolute
- text
- data
- bss
- undefined external

When the assembler evaluates an expression with operands of different types, the type of the resulting expression is determined by the following rules:

- When one of the operands is undefined, the result is undefined.
- When both operands are absolute, the result is absolute.
- When an absolute is combined with a type that is not absolute (relocatable), the result is the same type as the non-absolute operand.

These rules apply to the following binary operators. At least one operand must be absolute; any other combination is illegal:

+ When one operand is a relocatable `.text`, `.data`, or `.bss` symbol or an undefined external symbol, the result has the postulated type: the other operand must be absolute.

− When the first operand is a relocatable, the result is relocatable.

When both operands are absolute, the result is absolute.

Comments

The assembler recognizes the following as comments:

- Standard comments introduced by the # character.
- C-style comments placed between /* and */ characters.

The `#` character introduces a comment that extends through the end of the line on which it appears.

The assembler also recognizes C-style comments, introduced with `/*` and ending with `*/`. C-style comments cannot be nested. The first `*/` token terminates the comment, regardless of the number of `/*` tokens preceding it. C-style comments can extend across multiple lines.

Summary of Core Instructions

The core instruction set implements ordinal and integer arithmetic operations along with program and processor control functions that support the architecture. In this manual, the core instruction set is divided into these categories:

Data manipulation and processing	These instructions move data, convert between different data types, and perform basic arithmetic and boolean operations.
Program control	These instructions alter the normal execution sequence based upon specified conditions. These instructions include ordinal and integer comparisons, branching, and procedure call and return.
Processor support	These instructions explicitly or implicitly make use of features of the i960 processor: including fault, trace, and process controls words, IAC messages, and multiprocessor design support.

Data Movement

The data movement instructions transfer integer and ordinal data between memory and the global and local registers (load and store instructions) and between registers (move instructions). The mnemonic opcodes indicate the size and type of data.

Besides moving data, the data movement instructions implicitly convert between different data types. For example, the load integer short instruction (`ldis`) copies a half-word (16 bits) from memory into a register. The `ldis` instruction implicitly converts the half word to a full word in the register, and the processor automatically sign-extends the high-order 16 bits.

6

Load

These instructions copy data from memory to selected registers or register groups:

`ld`	load
`ldob`	load byte ordinal
`ldos`	load short ordinal
`ldib`	load byte integer
`ldis`	load short integer
`ldl`	load long
`ldt`	load triple
`ldq`	load quad
`lda`	load address

All the load instructions use the MEM instruction format. Except for load address, which stores the memory location address itself in the designated register, the load instructions copy data from the addressed location to a specified register or successive registers.

Byte and short ordinal operands are zero-extended when loaded; byte and short integers are automatically sign-extended. Multi-register operations require appropriate register alignment. Besides moving data, these instructions are used for implicit data type conversions.

Store

These instructions copy data from selected registers or register groups to memory:

`st`	store
`stob`	store byte ordinal
`stos`	store short ordinal
`stib`	store byte integer
`stis`	store short integer

`stl`	store long
`stt`	store triple
`stq`	store quad

All the store instructions use the MEM instruction format. The store instructions copy data from the specified register or successive registers to the addressed location. The processor reformats short and byte ordinal and integer operands for the smaller memory location. Multi-register operations require appropriate register alignment.

Move

The move instructions copy data from a selected register or register group to another register or register group:

`mov`	move word
`movl`	move long
`movt`	move triple
`movq`	move quad

To move data in real format between the global or local registers and the floating-point registers, the numerics architecture of the KB and SB processors provides a set of move real instructions. Multi-register operations require appropriate register alignment.

Select

These data movement instructions are available on the i960 Jx, Hx, and RP processors. They select one of two source registers to copy into a destination register, based on the status of the condition code. All are REG format instructions.

`selno`	select based on unordered
`selg`	select based on greater
`sele`	select based on equal
`selge`	select based on greater or equal

`sell`	select based on less
`selne`	select based on not equal
`selle`	select based on less or equal
`selo`	select based on ordered

Ordinal and Integer Arithmetic

Core instructions that perform ordinal, integer, and decimal arithmetic belong to the following categories:

- basic arithmetic
- extended arithmetic
- conditional arithmetic
- remainder and modulo
- shift and rotate

All the instructions in this category use the REG instruction format.

Basic Arithmetic

These instructions perform the basic arithmetic operations: add, subtract, multiply, and divide:

`addo`	add ordinal
`addi`	add integer
`subo`	subtract ordinal
`subi`	subtract integer
`mulo`	multiply ordinal
`muli`	multiply integer
`divo`	divide ordinal
`divi`	divide integer

The basic arithmetic operations are carried out on ordinal and integer word operands contained in global or local registers. Use the load and store instructions to move data between memory and the registers.

Extended Arithmetic

The extended arithmetic instructions support operations on single- or dual-word operands:

addc   add ordinal with carry

subc   subtract ordinal with carry

emul   extended multiply

ediv   extended divide

In the add and subtract with carry instructions, the carry bit in the condition code (CC) of the arithmetic controls word (AC) participates in the operation. The integer overflow flag in the AC (used with the integer overflow mask) is set to indicate whether or not an overflow condition resulted from the operation. These two instructions facilitate multiple-precision addition and subtraction in assembly language programs.

The extended multiply (emul) instruction multiplies two ordinals in registers and copies the result into an aligned dual-register group. The extended divide (ediv) instruction performs the inverse operation, dividing a long ordinal (double-word) by an ordinal (word) resulting in a quotient and remainder (both ordinals) in a dual-register group.

Conditional Arithmetic

The conditional arithmetic instructions are available on the i960 Jx and Hx processors. They combine addition or subtraction with checking the condition code. They add or subtract the two source registers and copy the result into the destination, but only if the status of the condition code is correct for the given instruction. All are REG format instructions.

addono   add ordinal if ordered

addog   add ordinal if greater

addoe   add ordinal if equal

addoge   add ordinal if greater or equal

addol   add ordinal if less

addone	add ordinal if not equal
addole	add ordinal if less or equal
addoo	add ordinal if ordered
addino	add integer if ordered
addig	add integer if greater
addie	add integer if equal
addige	add integer if greater or equal
addil	add integer if less
addine	add integer if not equal
addile	add integer if less or equal
addio	add integer if ordered
subono	subtract ordinal if ordered
subog	subtract ordinal if greater
suboe	subtract ordinal if equal
suboge	subtract ordinal if greater or equal
subol	subtract ordinal if less
subone	subtract ordinal if not equal
subole	subtract ordinal if less or equal
suboo	subtract ordinal if ordered
subino	subtract integer if ordered
subig	subtract integer if greater
subie	subtract integer if equal
subige	subtract integer if greater or equal
subil	subtract integer if less

subine	subtract integer if not equal
subile	subtract integer if less or equal
subio	subtract integer if ordered

Remainder and Modulo

These arithmetic instructions divide the operands and retain the remainder of the operation, discarding the quotient:

remi	remainder integer
remo	remainder ordinal
modi	modulo integer

In the remainder instructions, the result has the same sign as the dividend. The result of the modulo instruction has the same sign as the divisor.

Shift and Rotate

The shift and rotate instructions implicitly perform arithmetic functions by shifting the bits in a register operand:

eshro	extended shift right ordinal (i960 Cx, Jx, and Hx processors only)
shlo	shift left ordinal
shro	shift right ordinal
shli	shift left integer
shri	shift right integer
shrdi	shift right dividing integer
rotate	rotate bits

The shift instructions discard bits shifted out of the high-order or low-order bits of the register. The rotate instruction replaces bits shifted out of the high-order bits of the operand in the vacated low-order bit positions.

The shift right integer instruction does not correctly divide negative operands by powers of two arithmetic, although it does perform a conventional shift operation. To divide negative integer operands correctly, use the shift right dividing integer (`shrdi`) instruction instead of the shift right integer (`shri`) instruction.

The extended shift right ordinal instruction (`eshro`) is the equivalent of an extended divide by a power of 2, which produces no remainder.

Logical

These instructions perform the bitwise boolean (logical) functions on word operands in the specified registers. The only unary operation is carried out by the `not` instruction, which negates the bits in the *src* operand, represented by A in the list below.

In describing the remaining logical instructions, the letter A represents a bit in the *src2* operand and B represents the corresponding bit in the *src1* operand.

NOTE. *The binary logic functions process the source operands in reverse order.*

not	not A
and	A and B
notand	(not A) and B
andnot	A and (not B)
nand	not (A and B)
or	A or B
notor	(not A) or B
ornot	A or (not B)

nor	not (A or B)
xor	not (A = B)
xnor	A = B

Tables 6-8 through 6-10 show the operands and results of the binary logical operations. The unary `not` instruction simply complements bits (clears bits that are set and sets bits cleared to 0) in a bitwise fashion for each of the 32 bits of the *src* operand.

Table 6-8   Unary Operation

A	not
0	1
1	0

Table 6-9   Binary Operations

A	B	and	notand	andnot	nand	or
0	0	0	0	0	1	0
0	1	0	1	0	1	1
1	0	0	0	1	1	1
1	1	1	0	0	0	1

Table 6-10   Binary Operations Continued

A	B	notor	ornot	nor	xor	xnor
0	0	1	1	1	0	1
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	1	1	1	0	0	1

6

Bit, Bit Field, Byte

The bit and bit field instructions perform operations on a contiguous series of bits within an ordinal word. As with the arithmetic instructions, the bit and bit field instructions operate only on data placed in global or local registers. Use the data movement instructions to transfer data between memory and the registers. The processor also provides two byte operations, `scanbyte` and `bswap`.

Bit Operations

These instructions operate on a single specified bit in a global or local register.

`setbit`	set bit
`clrbit`	clear bit
`notbit`	not bit
`chkbit`	check bit
`alterbit`	alter bit
`scanbit`	scan for bit
`spanbit`	span over bit

The `setbit`, `clrbit`, and `notbit` instructions set, clear, or complement the specified bit in an ordinal word. The `chkbit` instruction sets the condition code (CC) in the arithmetic controls word (AC) according to the state of the specified bit. The `alterbit` instruction changes the state of the bit based on the condition code setting.

The `scanbit` and `spanbit` instructions return the bit number of the most-significant set and clear bit in the source operand, respectively.

Bit Field Operations

Two instructions operate on a bit field, specified by the bit position of the least-significant bit in the field and the length of the field:

extract   extract bit field

modify   modify under mask

The extract instruction shifts the specified bit field to the right and fills the vacated high-order positions with zeros. The modify instruction copies the specified bit field in one register to another, under control of a mask. This instruction preserves bits corresponding to masked bit positions.

Byte Operations

The scanbyte instruction compares two ordinals on a byte-by-byte basis, testing whether or not any two corresponding bytes in the ordinals are equal. The scanbyte instruction then sets the condition code (CC) according the outcome: successful (TRUE) or unsuccessful (FALSE).

The bswap instruction, available only on i960 Jx, Hx, and RP processors, reverses the byte order within a 4-byte word. Bytes 0 and 3 are swapped, and bytes 1 and 2 are swapped. This is a REG format instruction.

Comparison

Several types of instructions facilitate the comparison of instruction operands. These instructions often are used for program decision-making and can result in a subsequent call or branch. Compare and conditional-compare instructions, as well as compare-and-increment or compare-and-decrement instructions, are included in the core architecture.

The comparison instructions use REG format and operate on the following types of data:
- ordinal
- integer
- real

This chapter discusses comparison of ordinal and integer data types; the real data types and related operations are discussed in your processor manual.

Compare and Conditional Compare

The following instructions compare the specified operands, in global or local registers, and set the condition code (`CC`) in the arithmetic controls word (`AC`) according to the results of the test:

`cmpi`	compare integer
`cmpo`	compare ordinal
`concmpi`	conditional-compare integer
`concmpo`	conditional-compare ordinal
`cmpob`	compare ordinal byte
`cmpib`	compare integer byte
`cmpos`	compare ordinal short
`cmpis`	compare integer short

The byte and short versions of this instruction are available only on the i960 Jx and Hx processors.

The `cmpi` and `cmpo` instructions simply test the operands and set the condition code. The `concmpi` and `concmpo` instructions first examine the status bit (bit 2) of the condition code and compare the operands only if the status bit is not set. If the status bit is set, no further action occurs.

These instructions optimize two-sided range comparisons, to test whether a given value lies between two others. A compare instruction (`cmpi` or `compo`) checks one side of the range and a conditional-compare instruction (`concmpi` or `concmpo`) checks the other, based upon the result of the first comparison.

Compare and Increment or Decrement

The following compare-and-increment or compare-and-decrement instructions compare two specified source operands and set the condition code based on the result:

cmpinci	compare and increment integer
cmpinco	compare and increment ordinal
cmpdeci	compare and decrement integer
cmpdeco	compare and decrement ordinal

These instructions either increment or decrement the destination register by 1. The compare-and-increment or compare-and-decrement instructions provide a convenient way to control iterative program loops.

Branch

The branch instructions direct the processor to continue executing a program's instructions at another memory address, sometimes conditionally. To accomplish this end, these instructions modify the current instruction pointer (IP). The new value of the IP can be specified as a displacement applied to the instruction pointer (COBR and CTRL instruction formats), or defined using several memory addressing modes (MEM instruction formats).

The branch instructions provide the following program control functions:
- unconditional branch
- conditional branch
- compare and branch

In addition to these machine instructions, Chapter 7 describes several sets of pseudo-instructions to simplify coding branch instructions.

6

Unconditional Branch

The following instructions direct the processor to continue executing
instructions from a supplied address under any condition:

b	branch
bx	branch extended
bal	branch and link
balx	branch and link extended

The branch (b) instruction uses the CTRL format, with a limited
addressing range, while the branch extended (bx) instruction uses MEM
format with a full addressing range and corresponding memory address
modes.

Like the branch instructions, the bal and balx instructions use CTRL and
MEM formats, respectively. These instructions save the address of the
next sequential instruction and branch unconditionally to the specified
address.

Typically, the branch-and-link instructions are used to pass control to local
program procedures. (Local procedures are procedures that do not require
the processor's call-and-return mechanism.)

Conditional Branch

The following instructions direct the processor to continue executing
instructions from a supplied address depending on the status of the
condition code (CC) bits in the arithmetic controls (AC) word:

be	branch if equal
bne	branch if not equal
bl	branch if less
ble	branch if less or equal
bg	branch if greater

`bge`	branch if greater or equal
`bo`	branch if ordered
`bno`	branch if unordered

These instructions also use the CTRL format and specify the target memory address as a displacement from the current instruction pointer (IP). Use the branch if ordered (`bo`) and branch if unordered (`bno`) to compare real number operands.

A set of branch real pseudo-instructions supplement the `bo` and `bno` instructions to include comparisons of real numbers. In addition, the branch if true (`bt`) and branch if false (`bf`) directives provide convenient mnemonics for branching on specific conditions. See Chapter 7 for more information on the branch pseudo-instructions.

Compare and Branch

The ordinal and integer compare-and-branch instructions compare the two source operands, set the condition code (CC), and branch to the specified address depending on the result. These instructions are:

`cmpobe`	compare ordinal and branch if equal
`cmpobne`	compare ordinal and branch if not equal
`cmpobl`	compare ordinal and branch if less
`cmpoble`	compare ordinal and branch if less or equal
`cmpobg`	compare ordinal and branch if greater
`cmpobge`	compare ordinal and branch if greater or equal
`cmpibo`	compare integer and branch if ordered
`cmpibe`	compare integer and branch if equal
`cmpibne`	compare integer and branch if not equal
`cmpibl`	compare integer and branch if less
`cmpible`	compare integer and branch if less or equal

cmpibg	compare integer and branch if greater
cmpibge	compare integer and branch if greater or equal
cmpibno	compare integer and branch if not ordered

Two other compare and branch instructions operate on a single-bit operand in an ordinal word in a global or local register:

bbc	branch on bit clear
bbs	branch on bit set

All compare-and-branch instructions use the COBR instruction format, implying a limited address range. See also the compare-and-jump pseudo-instructions, described in Chapter 7.

Call and Return

For programming convenience, i960 processors provide various mechanisms for making procedure calls. The following instructions support the processor's call-and-return mechanism:

call	call to local procedure using 24-bit addressing
callx	call to procedure using full 32-bit addressing
calls	call to a system procedure
ret	return

Like the branch instructions, the call instruction uses the CTRL format, with a limited addressing range, while the callx instruction uses MEM format with a full addressing range and corresponding memory address modes.

The calls instruction provides a supervisor call capability, deriving the procedure address from the system procedure table, using a specified index number to determine the correct table entry to reference. The table entry determines whether procedures in the table can execute in supervisor mode. Upon return from the called procedure, the processor resumes its previous execution mode.

The assembler provides two pseudo-instructions which are optimized by the linker:

callj   stands for a `call`, `bal`, or `calls` instruction

calljx   stands for a `callx`, `balx`, or `calls` instruction

With the `callj` and `calljx` pseudo-instructions, you can make symbolic references to a variety of function types without using an explicit call or branch-and-link instruction. The linker chooses the appropriate instruction or instruction sequence for the symbol type and performs call optimization, if possible. For additional information on call optimization, see the *i960 Processor Software Utilities User's Guide*.

Fault

Normally, the processor implicitly generates faults when exceptions occur and handles them automatically through the programmer-defined fault table. The address of the fault table is supplied to the processor at initialization time. You can inhibit certain faults by using the fault controls, or masks.

The following fault-if instructions allow a running program to raise a fault condition explicitly:

faulte   fault if equal

faultne   fault if not equal

faultl   fault if less

faultle   fault if less or equal

faultg   fault if greater

faultge   fault if greater or equal

faulto   fault if ordered

faultno   fault if unordered

The processor services a fault generated by one of these instructions as if it were generated implicitly, as a result of an exception. See your processor-manual for information on enabling and masking faults.

Chapter 7 explains the fault if true (`faultt`) and fault if false (`faultf`) assembler pseudo-instructions that provide a mnemonic method for generating faults based on logic conditions.

Debug

Several processor instructions support the processor's on-chip debugging facilities. These facilities include a trace controls word and associated masks, allowing the program to enable or disable specific types of trace functions. The debug instructions are:

`modtc`	modify trace controls
`mark`	mark a breakpoint
`fmark`	force mark a breakpoint

The `modtc` instruction allows a running program to change the bits in the processor's trace controls word. The `mark` and `fmark` instructions generate a breakpoint trace event: the `mark` instruction generates the event if the breakpoint trace mode is enabled by the trace controls word, while the `fmark` instruction generates an unconditional breakpoint event.

See your processor manual for information on the trace mechanism and associated controls.

Processor Management

The following instructions read or modify bits in the arithmetic and processor controls words:

`modac`	modify arithmetic controls
`modpc`	modify process controls
`sysctl`	perform system control function on the i960 Cx, Jx or Hx processors

Note that with there are special rules for using a `modpc` instruction with the i960 RP architecture . The syntax for using the `modpc` instruction with any i960 architecture other than RP is:

```
modpc src, mask, src/dst
```

With the i960 RP architecture, the first and third arguments must be the same. If these arguments are not the same, the assembler generates a warning.

Another instruction that is useful for processor management is the `flushreg` instruction. `flushreg` saves all but the current local register set ensuring that the local register save areas contain the same data as the processor's local register sets.

The following processor management instructions are specific to the i960 Jx, RP and Hx processors:

Table 6-11   Supported Processor Management Instructions

Instruction	Description	80960Jx	80960RP	80960Hx
intdis	global interrupt disable	Yes	Yes	Yes
inten	global interrupt enable	Yes	Yes	Yes
intctl	global enable and disable of interrupts	Yes	No	Yes
icctl	icache control	Yes	Yes	Yes
dcctl	dcache control	Yes	Yes	Yes
halt	halt CPU	Yes	No	No
dcinva	data cache invalidate by address	No	No	Yes

The following test-if instructions allow programs to examine the bits of the condition code, which can then be used to redirect program flow:

teste	test if equal
testne	test if not equal
testl	test if less

`testle`	test if less or equal
`testg`	test if greater
`testge`	test if greater or equal
`testo`	test if ordered
`testno`	test if unordered

Synchronous (K-series only)

On K-series processors, the synchronous instructions move data from a register to memory or from one memory location to another.

`synld`	synchronous load
`synmov`	synchronous move
`synmovl`	synchronous move long
`synmovq`	synchronous move quad

The `synld` instruction copies a word from a register into memory. The synchronous move instructions transfer data from one location in memory to another.

Normally the processor executes store instruction asynchronously with respect to the memory controller. That is, after placing information on the data bus for storage in memory, the processor assumes that bus control logic carries out the operation and continues with the next instruction. In contrast, the synchronous instructions perform store and move operations synchronously with memory.

When executing any of the synchronous instructions, the processor must wait until that instruction and any other pending memory access instructions are completed before executing the next instruction.

The processor indicates that a synchronous instruction is complete by setting the condition code bit (CC) in the arithmetic controls word (AC). Use these instructions when you must be sure that memory operations are completed before further processing takes place, as in multiprocessor designs. See also the section on atomic instructions below.

Also, the synchronous instructions can be used as a mechanism to avoid interrupts when sending interagent communication (IAC) messages.

Atomic

An atomic access is a processor read-modify-write operation on a 32-bit word of memory. In multiple-processor designs, while one processor performs an atomic access, other processors in the system cannot access the same memory block until the original operation is complete. The atomic instructions are:

atadd   atomic add

atmod   atomic modify

The atadd and atmod instructions add or modify the data in a specified memory location and guarantee the integrity of the operation.

Summary of On-chip Numerics Instructions

Floating-point instructions have at least one operand that is a real data type. They include the following functional categories of instructions:

* data movement instructions
* sign copying instructions
* data type conversion instructions
* comparison and classification instructions
* basic arithmetic instructions
* trigonometric functions
* logarithmic, exponential, and scale instructions
* decimal data manipulation instructions

The following sections summarize the instructions in each group.

Data Movment

Several ordinal and integer load and store instructions (ld/st, ldl/stl, ldt/stt, ldq/stq) move 4, 8, 12, or 16 bytes of data between memory and local or global registers without regard to data type. The core architecture move instructions (mov, movl, movt, movq) can then transfer

the contents of 1 to 4 local or global registers to another non-overlapping group of 1 - 4 local or global registers without changing formats: real values remain real, integer values remain integer, and so on.

Three move real instructions are provided in the numerics architecture:

movr   move real

movrl   move long-real

movre   move extended-real

The movr and movrl instructions are most often used to transfer real-valued data between global and local registers and floating-point registers when a format change is desired. This technique implicitly converts 32-bit, 64-bit, or 96-bit real data to 80-bit extended-real format and vice versa.

The following procedure converts 32-bit real data to a 64-bit real representation:

1. Move a 32-bit real data word into a floating-point register using the movr instruction. This step implicitly converts the real value into an extended-real value.

2. Move the extended-real value from the floating-point register to two global or local registers using the movrl instruction. The processor explicitly converts the extended-real number into a 64-bit long-real value in two global or local registers.

To convert implicitly from real and long-real to extended-real data format, use the floating-point registers as operands in arithmetic, trigonometric, logarithmic, and exponential operations.

The movre instruction copies extended-real values between a 80-bit floating-point register and a triple global or local register group (96 bits). The instruction does not alter the data type. However, when moving data from a floating-point register to a register group, the movre instruction inserts 16 zeros in the high-order bit positions to pad the third data word. When moving the contents of the register group to a floating-point register, this instruction deletes the most significant 16 bits of the word in the third register.

Sign Copying

The numerics architecture provides two sign-copying instructions:

cpysre   copy sign extended-real

cpyrsre   copy reverse sign extended-real

These instructions enable you to copy the sign of one extended-real value, or its reverse, to another. Both operate exclusively on extended-real data types, and at least one of the values must be in a floating-point register. To copy the signs of real or long-real values, use the chkbit and alterbit instructions.

Data Type Conversion

To convert between floating-point formats, for example between real and extended-real formats, use the move real instructions described in the Data Movement section. To convert between integer and real number formats, the numerics architecture provides these explicit instructions:

cvtir   convert integer to real

cvtilr   convert long integer to real

cvtri   convert real to integer

cvtril   convert real to long integer

cvtzri   convert truncated real to integer

cvtzril   convert truncated real to long integer

The cvtir and cvtilr instructions can change their 32-bit and 64-bit data types to 80-bit extended-real values or 32-bit real values, respectively. The move real instructions can then convert the result to 64-bit long-real format if desired.

The cvtri and cvtril instructions change 32-bit real or 80-bit extended-real numbers to integers. Hence, to convert a 64-bit long-real value to an integer, first convert it to an extended-real format using the appropriate move real instruction. Then use one of the convert real instructions to transform the extended-real value to the desired integer format.

The `cvtzri` and `cvtril` instructions allow efficient implementation of FORTRAN or C-style truncation semantics. They ignore the rounding mode bits in the arithmetic controls word, and round toward zero always.

Basic Arithmetic

The following instructions perform the basic arithmetic operations specified in the IEEE standard:

`addr`	add real
`addrl`	add long-real
`subr`	subtract real
`subrl`	subtract long-real
`mulr`	multiply real
`mulrl`	multiply long-real
`divr`	divide real
`divrl`	divide long-real
`remr`	remainder real
`remrl`	remainder long-real
`roundr`	round real
`roundrl`	round long-real
`sqrtr`	square root real
`sqrtrl`	square root long-real

These instructions correspond to many of the core architecture instructions in the same functional category. However, in the numerics architecture all arithmetic operations require real or long-real data types as operands and result in real numbers.

The results and operands of instructions such as `addr`, `subr`, `mulr`, etc., can be 32-bit real, or 80-bit extended-real values. Similarly, results and operands of the arithmetic long-real instructions, such as `addrl`, `subrl`, and `mulrl`, can be 64-bit long-real, or 80-bit extended-real values.

The add, subtract, multiply, divide, and square root instructions represent relatively standard, straight-forward mathematical functions performing the operations their names imply.

The `remr` and `remrl` instructions divide the contents of a register or dual-register group by the value in another register (or pair) and produce the remainder of the quotient; the quotient itself is ignored. For example, if the real number 987.34 is divided by 185.769, the quotient is 5.31488... and the remainder is the fractional portion of the quotient, .31488... These instructions differ from the IEEE standard by the way in which the integer portion of the quotient is determined.

The `roundr` and `roundrl` instructions convert a real or long-real operand to an integer value based on the current rounding mode. The integer result remains in floating-point format. The current rounding mode is determined by the setting of the rounding mode bits in the arithmetic controls word (AC).

For example, the real-valued result 137.85 is rounded to 137.0 if the rounding controls are set to round toward zero. The same number is rounded to 138.0 if the rounding controls are set to round to infinity.

Decimal

The decimal instructions operate on 32-bit operands that contain an ASCII-coded decimal digit in the least-significant 8 bits of the data word.

`dmovt`	decimal move and test
`daddc`	decimal add with carry
`dsubc`	decimal subtract with carry

The `dmovt` instruction moves a 32-bit word from one register to another and tests the least-significant byte of the operand to determine if it is a valid ASCII-coded decimal digit (00110000_2 through 00111001_2, corresponding to the decimal digits 0 through 9). For valid digits, the condition code (CC) is set to 000_2; otherwise the condition code is set to 010_2.

6

i960 Processor Assembler User's Guide

The `daddc` and `dsubc` instructions operate on two decimal digits. Bit 1 of the condition code indicates a decimal carry-in or carry-out condition. For example, you can use the decimal instructions iteratively to validate ASCII digit strings and to add or subtract ASCII-coded decimal values.

Note that with CTOOLS release 5.1, the assembler no longer accepts decimal instructions when assembling for a KA or an SA target, since decimal instructions are not supported by those processors.

Comparison and Classification

To compare and classify floating-point values, use the numerics instructions:

`cmpr`	compare real
`cmprl`	compare long-real
`cmpor`	compare ordered real
`cmporl`	compare ordered long-real
`classr`	classify real
`classrl`	classify long-real

The `cmpr` and `cmprl` instructions compare the contents of two registers and set the condition code bits (CC) in the arithmetic controls word (AC) to indicate the results of the comparison. For floating-point operands, when at least one comparand is a NaNs, the condition code indicates unordered.

The `cmpor` and `cmporl` instructions set the invalid-operation flag for an unordered condition.

Use the core-architecture branch-ordered (`bo`) and branch-unordered (`bno`) instructions to test the floating-point comparison results, with conditional branching if an ordered or unordered condition is detected.

The `classr` and `classrl` instructions determine the class of a real or long-real operand as zero, denormalized finite, normalized finite, infinite, SNaN, or QNaN. The AC arithmetic status bits indicate the result.

6-40

6

Trigonometric Functions

For the common trigonometric functions, use the numerics instructions:

`sinr`   sine real

`sinrl`   sine long-real

`cosr`   cosine real

`cosrl`   cosine long-real

`tanr`   tangent real

`tanrl`   tangent long-real

`atanr`   arctangent real

`atanrl`   arctangent long-real

All the trigonometric functions require real or long-real operands and yield floating-point results. The values of angles must be given in radians.

The results and operands of instructions such as `sinr`, `cosr`, and `tanr` can be 32-bit real or 80-bit extended-real values. Similarly, results and operands of the trigonometric long-real instructions, such as `sinrl`, `cosrl`, and `tanrl`, can be in 64-bit long-real or 80-bit extended-real format.

The `atanr` and `atanrl` instructions return a result in radians. As well as supplying the inverse tangent of the argument, these instructions facilitate conversion from rectangular to polar coordinates.

If the operands of trigonometric functions are computed using pi, then the full 66-bit representation for pi given in your processor-specific manual must be used. Truncated values are permissible when accuracy is not crucial.

6

Logarithmic, Exponential, and Scale

For logarithmic, exponential, and scale functions, use the numerics instructions:

`logbnr`	log binary real
`logbnrl`	log binary long-real
`logr`	log real
`logrl`	log long-real
`logepr`	log epsilon real
`logeprl`	log epsilon long-real
`expr`	exponent real
`exprl`	exponent long-real
`scaler`	scale real
`scalerl`	scale long-real

All these functions require real or long-real operands and yield floating-point results.

The results and operands of instructions such as `logr`, `expr`, and `scaler` can be 32-bit real or 80-bit extended-real format. Similarly, results and operands of the trigonometric long-real instructions, such as `logrl`, `exprl`, and `scalerl` can be in 64-bit long-real or 80-bit extended-real format.

The `logbnr` and `logbnrl` instructions compute the logarithm to the base 2 of the source operand and retain only the integer component. The result is an integer that is the binary log of the given number. For instance, $\log_2 3249 = 11.65532...$, but the binary log function returns the value 11 (decimal) in floating-point format. The `logbnr` and `logbnrl` instructions determine the order of magnitude of a specified number.

The `logr` and `logrl` instructions compute the logarithm to the base 2 of one source operand (`src1`) and scale the result by a second operand (`src2`), to obtain the result (`dst`):

$$dst = src1 * \log_2 src2$$

By carefully specifying the `src2` operand, logarithms to any base can be computed using these instructions. For instance, by specifying a scale factor of `src1` = `.30102...`, the logarithm base 10 (common log) is obtained.

The `logepr` and `logeprl` instructions compute the logarithm to the base 2 of `1.0` plus the `src1` operand and scale (`src2` operand) the result to obtain the result (`dst`):

$$dst = src2 * \log_2 (1.0 + src1)$$

The `src1` operand is restricted to values near zero which yields maximum accuracy for `1.0` + `src1` near unity (i.e., the `src1` operand is close to zero). This condition, for instance, is commonly encountered when computing compound interest. By carefully choosing the `src2` operand, logarithms to any base can be computed.

The `expr` and `exprl` instructions compute the value:

$$dst = 2^{src} - 1$$

The `src` must be in the range `-.5` to `+.5`. The `scale` and `scalerl` instructions multiply the `src2` operand by 2 to an integer power, denoted by the `src1` operand, for the result (`dst`):

$$dst = src2 * 2^{src1}$$

The exponent and scale instructions can be used together to create an algorithm for computing the value of 2 to any power by noting that:

$$2^Y = 2^{(X + I)} = 2^I * [(2^X - 1) + 1]$$

The `Y` is an arbitrary exponent: `I` and `X` represent the integer and fractional portions of the exponent, respectively.

Pseudo-instructions   **7**

This chapter describes the pseudo-instructions (pseudo-ops) recognized by the assembler.

Pseudo-instructions appear in the assembly file like valid machine instructions. In actuality, the assembler substitutes one or more machine-level instructions for them. For example, when you enter the optimized load constant or `ldconst`, the assembler selects the fastest instruction available to place the specified value in the designated register. This instruction can be a move, add, subtract, shift, or load-address, depending on the given value.

For convenience, the assembler provides pseudo-instructions that are synonyms for certain branch, fault, load, and compare-and-branch instructions, as described in the following sections. These pseudo-instructions are functions of the assembler and not of any particular processor implementation. In general, you can use them in any assembly language source file. Any implementation-dependent differences are noted.

Syntax

Pseudo-instructions use the same syntax for operands as machine instructions.

The operand names describe the function of the operands (for example, *src*, *dst*, *targ*).

Branch Pseudo-instructions

The assembler recognizes the pseudo-instructions bt (branch if true) and bf (branch if false) as synonyms for the instructions bo (branch if ordered) and bno (branch if not ordered), respectively.

For convenience in checking the results of real number (floating-point) comparisons, several branch pseudo-instruction are available. Table 7-1 lists these pseudo-instructions with the equivalent instructions.

Table 7-1   Branch Real Pseudo-instructions

Directive	Operation	Instruction
bre	branch real if equal	be
brg	branch real if greater	bg
brge	branch real if greater or equal	bge
brl	branch real if less	bl
brle	branch real if less or equal	ble
brlg	branch real if less or greater	bne
bro	branch real if ordered	bo
bru	branch real if unordered	bno
brue	branch real if unordered equal	be,bno
brug	branch real if unordered greater	bg,bno
bruge	branch real if unordered greater or equal	bge,bno
brul	branch real if unordered less	bl,bno
brule	branch real if unordered less or equal	ble,bno
brulg	branch real if unordered less or greater	bne,bno

Conditional Faults Pseudo-instructions

The assembler also has equivalent pseudo-instructions that help with conditional faults. The assembler recognizes `faultt` (fault true) and `faultf` (fault false) as synonyms for the instructions `faulto` and `faultno`. These pseudo-instructions have the same syntax as the machine instructions

Load Pseudo-instructions

The `ldconst` pseudo-instruction automatically optimizes loading of integer and ordinal immediate constant values. Immediate values that cannot be expressed as literals must be explicitly loaded into a register before they can be used as operands for machine instructions. For integer and ordinal operands, loading can be done using the `ldconst` directive. The `ldconst` directive generates different instructions for several different immediate values, based on architecture performance concern. For a list of `ldconst` substitutions, see the Example section of the alphabetical reference entry for `ldconst` later in this chapter.

Call Pseudo-instructions

The `callj` and `calljx` pseudo instructions let you assemble a call instruction, allowing the linker to perform call optimization, when possible. The linker transforms call pseudo-instructions into the appropriate instruction at link time, depending on the type (default, leaf, or system) of the called procedure. See page 7-8 for more information.

Compare-and-jump Pseudo-instructions

For compare-and-branch instructions, the assembler provides a convenient, symbolic way to specify the operation by using a set of compare-and-jump pseudo-instructions.

In the compare (ordinal or integer) and branch-on-condition instructions (such as the `cmpobe` instruction), the branch target must be fewer than 2^{12} bytes from the instruction pointer (IP). As an alternative, you can use the compare-and-jump pseudo-instructions provided by the assembler. These pseudo-instructions generate a compare-and-branch (e.g., `cmpobe`) instruction if the target is fewer than 2^{12} bytes away, or separate compare and branch instructions otherwise.

Form the compare-and-jump pseudo-instructions by substituting a `j` for the `b` in the corresponding instruction's mnemonic. For example, the instruction `be` becomes pseudo-instruction `je`; `cmpobe` becomes `cmpoje`. As another example, when you used the pseudo-instruction:

```
cmpije r5, r6, target
```

the assembler generates:

```
compibe r5, r6, target
```

if the label is within 2^{12} bytes, or:

```
compi r5, r6
be target
```

otherwise.

NOTE. *These pseudo-instructions never generate a branch-extended instruction. If you cannot guarantee that the branch address is fewer than 2^{23} bytes away from the instruction pointer, you must use the equivalent extended instruction sequence.*

The compare-and-jump pseudo-instructions appear in Table 7-2. Each pseudo-instruction is paired with the operation it performs.

Table 7-2   Compare-and-jump Pseudo-instructions

Pseudo-instruction	Full Function Name
cmpije	compare integer and jump if equal
cmpijg	compare integer and jump if greater
cmpijge	compare integer and jump if greater or equal
cmpijl	compare integer and jump if less
cmpijle	compare integer and jump if less or equal
cmpijne	compare integer and jump if not equal
cmpoje	compare ordinal and jump if equal
cmpojg	compare ordinal and jump if greater
cmpojge	compare ordinal and jump if greater or equal
cmpojl	compare ordinal and jump if less
cmpojle	compare ordinal and jump if less or equal
cmpojne	compare ordinal and jump if less not equal

Two pseudo-instructions never branch:

`cmpijno`   compare integer and jump if not ordered.

`cmpojno`   compare ordinal jump if not ordered. The equivalent instruction is `cmpibno`.

Two pseudo-instructions always branch:

`cmpijo`   compare integer and jump if ordered.

`cmpojo`   compare ordinal and jump if ordered. The equivalent instruction is `cmpibo`.

Ordered relationships apply only to real numbers on i960 processors with on-chip floating-point capability. The branch instructions for ordered and unordered numbers are consistent ways to provide null operations (no-ops).

7

Pseudo-instructions Reference

This section describes the pseudo-instructions in alphabetical order.

The syntax descriptions use the placeholder *targ* for any operand that is an expression representing a memory address. The assembler treats a *targ* operand as a signed displacement value representing an IP-relative address, as follows:

Format	Displacement in Words
COBR	-2^{10} through 2^{10}-1
CTRL	-2^{21} through 2^{21}-1

BRANCH

Branch if false or
branch if true

```
bf   targ
bt   targ
```

targ   specifies the branch target.

Discussion

Both the `bt` (branch if true) and `bf` (branch if false) directives check the condition code and branch to the location specified by *targ* based upon the result of the test.

The assembler recognizes the following correspondence:

Directive	Instruction
`bt` (branch if true)	`bo` (branch if ordered)
`bf` (branch if false)	`bno` (branch if unordered)

The syntax for the two directives is the same as the syntax for the corresponding machine instructions.

Example

The assembler changes the pseudo-instruction below to the instruction `bo process`:

`bt  process`

BRANCH REAL

*Branch on the result of
a floating-point
comparison*

`bre`	branch real if equal
`brg`	branch real if greater
`brge`	branch real if greater or equal
`brl`	branch real if less
`brle`	branch real if less or equal
`brlg`	branch real if less or greater
`bro`	branch real if ordered
`bru`	branch real if unordered
`brue`	branch real if unordered equal
`brug`	branch real if unordered greater
`bruge`	branch real if unordered greater or equal
`brul`	branch real if unordered less
`brule`	branch real if unordered less or equal
`brulg`	branch real if unordered less or greater
`br* targ`	specifies the branch target
`targ`	

7

Discussion

The branch real directives check the results of floating-point comparisons and branch to the location specified by *targ* based upon the result of the test. These instructions generate the appropriate compare instructions for unordered cases.

Table 7-1 shows the correspondences between pseudo-instructions and machine instructions.

Use the same syntax for pseudo-instructions you do for the corresponding machine instructions.

Example

The assembler changes the pseudo-instruction below to the instruction `bno process`:

```
bru  process
```

callj, calljx

Optimizable linker calls

```
callj   targ
calljx  targ
```

targ   is the name of the procedure to be called.

Discussion

The `callj` and `calljx` pseudo-instructions assemble a `call` or `callx` instruction, respectively, and a relocation entry instructing the linker to perform call optimization, when possible. The linker can also be instructed to ignore call optimization. See the utilities user's guide for more information about linker controls.

When the referenced procedure, represented by `targ`, is a `.leafproc`, the linker replaces the pseudo-instruction with a branch-and-link (`bal` or `balx`) instruction. When the target is a `.sysproc`, the linker replaces the pseudo-instruction with a `calls` instruction.

For example, inserting a `calljx` instruction while using the `-AJD` setting might produce the following linker output depending upon whether the target is a default call, leaf procedure, or system call:

Table 7-3   Call Pseudo-instruction Substitutions

Default Call	Leaf Procedure	System Call
callx _target	balx _target,g14	lda _sysprocIndex,g13 calls (g13)

Since `callj` and `calljx` are optimized at link time, examination of the object module generated by the assembler with the disassembler (dumper) displays the assembled instruction as a `call` instruction.

The assembler optimizes `callj` or `calljx` to `bal` or `balx`, respectively, when the referenced procedure is a C language static function.

`callj` can be optimized during assembly when the target of the `callj` is in the same object module and section as the call site.

Example

This sample optimizes a call for procedure `_subx`.

```
callj  _subx
```

7

Changes to the calljx Pseudo Instruction with the i960 RP Architecture

When used with the `-ARP` option, `calljx` uses the syntax:

```
calljx _target, tmpreg
```

where *tmpreg* is a local or global register. This change results in the following sequences in the linker:

Default Call	Leaf Procedure	System Call
lda _target,tmpreg	lda _target,tmpreg	lda _sysprocIndex,g13
callx (tmpreg)	balx (tmpreg),g14	calls (g13)

Notice that with the 80960RP `calljx` format all three call types result in a three-word instruction sequence, whereas the previous `calljx` format requires only two words.

Related Topics

```
bal             .leafproc
balx            .sysproc
```

Compare-and-jump

*Branch to specified
target*

cmpije	compare integer and jump if equal
cmpijg	compare integer and jump if greater
cmpijge	compare integer and jump if greater or equal
cmpijl	compare integer and jump if less

`cmpijle`	compare integer and jump if less or equal
`cmpijne`	compare integer and jump if not equal
`cmpijno`	compare integer and jump if not ordered
`cmpijo`	compare integer and jump if ordered
`cmpoje`	compare ordinal and jump if equal
`cmpojg`	compare ordinal and jump if greater
`cmpojge`	compare ordinal and jump if greater or equal
`cmpojl`	compare ordinal and jump if less
`cmpojle`	compare ordinal and jump if less or equal
`cmpojne`	compare ordinal and jump if less not equal

The syntax for these pseudo-instructions is:

```
cmpij*   src1,   src2,   targ
cmpoj*   src1,   src2,   targ
```

src1	specifies a global or local register, or literal value
src2	specifies a global or local register
targ	specifies the branch target

Discussion

Both the integer and ordinal compare-and-jump directives check the results of a comparison of the contents of the source operands and branch to the location specified by *targ* based upon the resulting condition code (CC). Shown below are the instructions assembled as a result of each of these directives. The assembler recognizes the following correspondences:

Table 7-4   Compare and Jump Substitutions

Directive	When Target is < 2^{12} Bytes	When Target is ≥ 2^{12} Bytes Away
cmpije	cmpibe	cmpi + be
cmpijg	cmpibg	cmpi + bg
cmpijge	cmpibge	cmpi + bge
cmpijl	cmpibl	cmpi + bl
cmpijle	cmpible	cmpi + ble
cmpoje	cmpobe	cmpo + be
cmpojg	cmpobg	cmpo + bg
cmpojge	cmpobge	cmpo + bge
cmpojl	cmpobl	cmpo + bl
cmpojle	cmpoble	cmpo + ble

As Table 7-4 shows, the assembler only generates a compare integer or compare ordinal followed by a branch instruction when the destination is 2^{12} bytes or more away.

Two pseudo-instructions never branch:

cmpijno   compare integer and jump if not ordered.

cmpojno   compare ordinal and jump if not ordered. The
                       equivalent instruction is cmpibno.

Two pseudo-instructions always branch:

cmpijo   compare integer and jump if ordered.

cmpojo   compare ordinal and jump if ordered. The
                       equivalent instruction is cmpibo.

Ordered relationships apply only to real numbers on i960 processors with on-chip floating-point capability. The branch instructions for ordered and unordered numbers are consistent ways to provide null operations (no-ops), when not used with floating-point values.

The syntax for these directives is the same as the syntax for the corresponding machine instructions in the core architecture.

Example

This sample pseudo-instruction uses compare and branch

```
cmpije  r4, g4, process
```

During assembly, the pseudo-instruction becomes the following:

```
cmpi r4, g4
be process
```

FAULT

Fault if false or fault if true

```
faultf
faultt
```

Discussion

The `faultt` (fault if true) and `faultf` (fault if false) directives raise a fault condition based upon a test of the condition code.

7

The assembler recognizes the following correspondence:

Directive	Instruction
faultt	faulto
(fault if true)	(fault if ordered)
faultf	faultno
(fault if false)	(fault if not ordered)

The syntax for the two directives is the same as the syntax for the corresponding machine instructions in the core architecture.

Example

The following pseudo-instruction becomes `faultnof` during assembly:

```
faultf
```

ldconst

Load constant

```
ldconst    src,   dst
```

src   is an expression specifying a constant value to load.

dst   specifies a global or local register.

Discussion

Immediate values that cannot be expressed as literals must be explicitly loaded into a register before they can be used as operands for machine instructions. For integer and ordinal operands, this loading can be done with the `ldconst` directive.

The assembler selects the most efficient instruction available to place the value in the register. This instruction can be a move, add, subtract, shift, or load address, depending on the value of `src`.

NOTE. *The listing file generated by the assembler does not indicate what instruction (in the object module) substitutes for the* `ldconst` *directive specified in the source file. To determine what is assembled, display the instruction in the object module with the disassembler (dumper).*

Example

In the following lines, you can see some of the various ways to load constants with this pseudo-instruction:

```
# ldconst and assembled instruction
    ldconst 0, g5           /* mov 0,g5 */
    ldconst 31, g5          /* mov 31,g5 */
    ldconst 32, g5          /* addo 1,31,g5 */
addr:
    ldconst 62, g5          /* addo 31,31,g5 */
    ldconst 3<<8, g5        /* shlo 8,3,g5 */
    ldconst 0x1234, g5      /* lda 0x1234,g5 */
    ldconst -1, g5          /* subo 1,0,g5 */
    ldconst -31, g5         /* subo 31,0,g5 */
    ldconst addr, g5        /* lda addr,g5 */
```

Example Programs

This chapter contains sample code, in two sections. The examples in the first section use the core instructions, and those in the second section use floating-point instructions. See the processor user's manuals for complete lists of the instructions supported by each i960 architecture.

Note that the code shown in this chapter has not been tested on the current version of the assembler toolset. Therefore it is shown for general learning purposes only, and is not provided on the distribution media.

Examples Using the Core Instruction Set

The examples in this section use the core instructions described in Chapter 6. The example programs show:

- Code to enable interrupts to the i960 processor from an 8259A Programmable Interrupt Controller.
- Sending a breakpoint IAC message to the processor using an assembly language block in a C routine.
- Performing a bitblt code routine.
- Matrix multiplication with core instructions only.
- C-style string comparisons speed-optimized for a K-series i960 processor.

Enable and Count Interrupts From 8259A

The following source code shows how to initialize an 8259A
Programmable Interrupt Controller to interrupt the i960 processor. The
routine counts the number of interrupts generated.

```
/****                    Enable Interrupts                  ****/

.globl  _enable_ints
_enable_ints:

  lda  cr0_address, r3          /* cntrl stat reg addr */
  ldos  (r3), r4                /* cr0 is a 16 bit reg */
  lda  0xff7f, r5              /* mask for enints# bit */
  and  r5, r4, r4              /* set enints# bit low */
  stos  r4, (r3)
  ret

/* NOTE:the EXV complements and rotates the data bus  */
/* left 3 bits.  This is compensated for in 8259 read */
/* and write routines.  The bits below are those that */
/* 8259 must see.                                      */

/*------------------------------------------------------*/
/* Initialize the 8259                                  */
/*------------------------------------------------------*/

.globl _init_8259
_init_8259:

/* Write ICW1:  ICW4 req., 1 8259 level triggered */
  lda  ICW1_ADR, g0
  lda  ICW1_DATA, g1
  call  _write_8259

/* Write ICW2: Vector base of 08 */
  lda  ICW2_ADR, g0
  lda  ICW2_DATA, g1
  call  _write_8259

/* Write ICW4: 86/88 mode, normal EOI, non-buffered
 not special fully nested */
  lda  ICW4-ADR, g0
  lda  ICW4-DATA, gl
  call  _write_8259
```

```
/* Write OCW1, this is the interrupt mask register, a 0
in a bit in this register means that the interrupt is
enabled.        */
  lda   OCW1_ADR, g0
  lda   OCW1_DATA, g1
  call  _write_8259
  ret

/*-------------------------------------------------------*/
/* WRITE 8259 ROUTINE                                    */
/*---------------------- ----------------------------*/
/* Write_8259 routine.  Pass 8259 port address in g0 and
the data as it should appear to the 8259 in the lower
byte of g1.  This routine will invert and rotate the
data, write it to the 8259 and pause so that any
subsequent accesses to the 8259 will not violate the
recovery time. */

_write_8259:
  lda   0x000000ff, r3     /* mask to clear bytes 1,2,3 */
  and   r3, g1, g1
  shlo  03, g1, g1          /* shift data left 3 bits. */
  lda   0x00000700, r3      /* mask all bits but 8,9,10 */
  and   r3, g1, r3      /* bits 8,9,10 become bits 0,1,2 */
  shro  08, r3, r3     /* shift bits down to byte 0      */
  or   r3,g1, g1           /* combine upper 5 bits in g1 */
  not   g1, g1              /* invert data               */
  stob  g1, (g0)           /* write byte to the 8259 */
  bal   waiting_loop       /* wait so 8259 recovery time
                                            guaranteed */

  ret

.globl  _write_count
_write_count:
  lda   cra_address, r3
  lda   0x2a, r4
  stob  r4, (r3)
  bal   waiting_loop

  lda   cra_address, r3
  lda   0x3a, r4
  stob  r4, (r3)
  bal   waiting_loop
```

```
lda   cra_address, r3
lda   0x1a, r4
stob  r4, (r3)
bal   waiting_loop
lda   mra_address, r3
lda   0x02, r4
stob  r4,  (r3)
bal   waiting_loop

lda   mrb_address, r3
lda   0x07, r4
stob  r4, (r3)
bal   waiting_loop

lda   crb_address, r3
lda   0x2a, r4
stob  r4, (r3)
bal   waiting_loop

lda   crb_address, r3
lda   0x3a, r4
stob  r4, (r3)
bal   waiting_loop

lda   crb_address, r3
lda   0x1a,r4
stob  r4, (r3)
bal   waiting_loop

lda   mra_address, r3
lda   0x02, r4
stob  r4, (r3)
bal   waiting_loop

lda   mrb_address, r3
lda   0x07, r4
stob  r4, (r3)
bal   waiting_loop

lda   sra_address, r3
lda   0xbb, r4
stob  r4, (r3)
bal   waiting_loop
```

```
lda   srb_address, r3
lda   0xbb, r4
stob  r4, (r3)
bal   waiting_loop

lda   input_port_address, r3
lda   0xf4, r4
stob  r4, (r3)
bal   waiting_loop
lda   acr_address, r3
lda   0xf0, r4
stob  r4, (r3)
bal   waiting_loop

lda   imr_address, r3
lda   0x44, r4
stob  r4, (r3)
bal   waiting_loop

lda   ctur_address, r3
lda   ctur_data, r4
stob  r4, (r3)
bal   waiting_loop

lda   ctlr_address, r3
lda   ctlr_data, r4
stob  r4, (r3)
bal   waiting_loop

lda   cra_address, r3
lda   0x05, r4
stob  r4, (r3)
bal   waiting_loop

lda   CLOCK_ADR, r3  /* zero out clock count */
lda   0, r4
st   r4, (r3)
ret

/*
 Wait loop required after each access to DUART registers.
*/
```

```
waiting_loop:
  lda  sr0_address, r8  /* BST access;
       DUART recovery time */

waiting_loopl:
  ldob  (r8), r8
  bx  (gl4)              /* bal return */
#include "fractal.h"
#include "ints.h"
#include "mp_system.h"

  .text
.globl  _clock_int
_clock_int:
  mov  g14, r14          /* save bal register */
  lda  cr0_address, r3
  ldos  (r3), r4
  lda  0x20, r5
  or  r4, r5, r4
  stos  r4, (r3)

  /* update clock */

  lda  CLOCK_ADR, r6
  atadd  r6, 1, r7
  bal  waiting_loop

  not  r5, r5
  and  r5, r4, r4
  stos  r4, (r3)

/*
 check clock, if time is 1 second, then signal somebody
*/

  lda  SECONDS_DIVIDE, r10
  modi  r10, r7, r7
  cmpibne  0, r7, cont_here

  lda  0xffffffff,r7
  lda  8(r6), r10
  lda  24(r6), r8
  stl  r10, 24(r6)  /* store to previous answer */
  subc  r8, r10, r8
  subc  r9, r11, r9
```

```
  ldl  16(r6), r10
  ldl  32(r6), r4
  stl  r10, 32(r6)  /* store to previous answer */
  subc  r4, r10, r4
  subc  r5, r11, r5
  addc  r4, r8, r4
  addc  r5, r9, r5
  /* do the fp shuffle --- */

  movrl  fp3, r8
  cvtilr  r4, fp3
  movrl  fp3, r4

  lda  40,r11
  addo  r11, r6, r11
  atmod  r11, r7, r4  /* cumulate idle time */

  movrl  r8, fp3
  lda  44, r11

  addo  r11, r6, r11
  atmod  r11, r7, r5  /* cumulate idle time */
  lda  CLOCK_PORT,r10
  signal  r10
/* acknowledge to 8259 that all is well */

cont_here:
  lda  ADJUSTED_EOI, r4
  lda  OCW2_ADR, r5
  stob  r4, r5)
  mov  r14, g14
  ret

waiting_loop:
  lda  sr0_address, r8

waiting_loop1:
  ldob  (r8), r8
  bx  (g14)

.globl  _no_int
_no_int:
  lda  BASE_ADR, r5
  lda  ADJUSTED_EOI, r4
  stob  r4, (r5)
  ret
```

8

Send an IAC to the Processor

Although written in the C language, this source listing includes an ASM
block that actually sends a breakpoint IAC to the processor. The code
assumes that breakpoint trace mode is set in the trace controls word and
that the trace enable flag of the process controls word is also set.

Figure 8-1 shows the format of the data structure used in the program.

Figure 8-1   IAC Message Structure
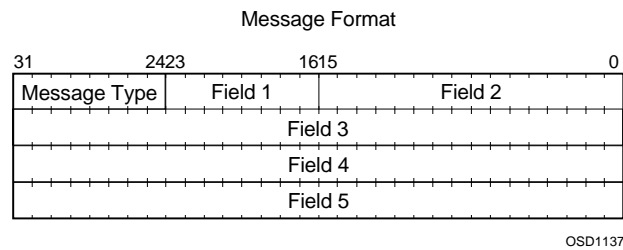

```
/* iac structure */
struct x iac_msg {
  unsigned short  field2;
  unsigned char   field1;
  unsigned char   message_type;
  unsigned int  field3;
  unsigned int  field4;
  unsigned int  field5;
} iac_struct;

/* This routine issues an IAC message to the local
processor on which the program resides.  It accepts a
pointer to a preformed IAC message as input, and uses the
synmovq instruction to send the IAC to the processor. */

asm send_iac (struct iac_msg * base_msg)
{
  %reglit base_msg; tmpreg myreg;
```

```
  lda  0xff000010, myreg  /* load local IAC address */
  synmovq  myreg, base_msg
      /* issue IAC message */

  %error;
}
/*************************************************/
/* Send a breakpoint IAC to the processor       */
/*                                               */
/* (don't forget to turn on breakpoints in the  */
/* trace control register )                      */
/*************************************************/
set_breakpt(addr1, addr2)
unsigned int addr1;
unsigned int addr2;
{
  iac_struct.message_type = 0x8f;
  iac_struct.field3 = addr1;
  iac_struct.field4 = addr2;
  send_iac(&iac_struct);
}
```

Perform a BitBlt Operation

The following example shows a `bitblt` code routine. The typical size of a character stored in memory is 32 x 40 bits. Optimization techniques include:

- use of the ldconst pseudo-instruction
- use of ldq and stq to move data blocks
- register bypassing for the or instructions within the loop
- instructions are placed between compare-and-branch; the branch instruction therefore uses 0 clocks
- register loading is done before the data is actually used; other instructions are executed while waiting for the load

i960 Processor Assembler User's Guide

```
.text
.globl _main

_main:
  lda   0x30000, r4    /* source address in r4 */
  lda   0x40000, r5    /* destination address in r5 */
  ldconst  7, r6       /* word count in r6 */
  divi  4, r6, r7      /* quad count in r7 */
  modi  4, r6, r6      /* remainder word count in r6 */
  ldconst  4, r8       /* offset in r8 */
  ldq  (r4), g0
  addi  0x10, r4, r4   /* increment source addr 4 words */
  ldconst  32, r9
  subo  r8, r9, r9     /* 32 - offset */
  ldconst  0, g4       /* clear g4 for carry in */

  cmpibge  0, r7, single
                       /* no quad words jump to single */

loop:
  shro  r9, g4, g5     /* shift carry rt. by 32-offset */
  shlo  r8, g0, g6     /* shift src1 left by offset */
  or  g5, g6, g8       /* combine */
  shro  r9, g0, g12    /* shift src1 right by 32-offset */
  shlo  r8, g1, g13    /* shift src2 left by offset */
  or  g12, g13, g9     /* combine */
  shro  r9, g1, g13    /* shift src2 right by 32-offset */
  shlo  r8, g2, r14    /* shift src3 left by offset */
  or  g13, r14, g10    /* combine */
  shro  r9, g2, g7     /* shift src3 right 32-offset */
  shlo  r8, g3, g11    /* shift src4 left by offset */
  mov  g3, g4          /* save src 4 for carry in */
  ldq  (r4), g0        /* start next load */
  or  g7, g11, g11     /* combine */
  addi  0x10, r4, r4   /* increment src addr by 4 words */
  subi  1, r7, r7      /* decrement quad count */
  cmpi  0, r7          /* test if done */
  stq  g8, (r5)        /* store 4 words in dest */
  addi  0x10, r5, r5   /* increment dest addr 4 words */
  bl  loop             /* if not done loop back */

  cmpibge  0, r6, end  /* if no remainder jump to end */
single:
  subi  0xc, r4, r4    /* get rid of extra loads */
```

```
cont:
  shro  r9, g4, g5  /* shift carry right by 32-offset */
  mov  g0, g4        /* save src for carry in */
  shlo  r8, g0, g6  /* shift src left by offset */
  ld  (r4), g0       /* start next load */
  addi  0x4, r4, r4  /* increment src addr */
  or  g5, g6, r14    /* combine */
  subi  1, r6, r6    /* decrement remainder */
  cmpi  0, r6        /* test if done */
  st  r14, (r5)      /* store word in dest */
  addi  0x4, r5, r5  /* increment dest addr */
  bl  cont           /* if not done loop back */

end:  ret
  fmark
  .word 0x00000000
  .word 0x00000000
```

Perform Matrix Multiplication

The following example shows an optimized version of a 1 x 3 matrix multiply, using only ordinal and integer arithmetic.

```
/*
        g7       input image vector pt
        g3       output sum
        r12      output line vector pt
        g0-2     a11,a12,a13     (kernel)
        g4-6     a21,a22,a23
        g8-10    a31,a32,a33
        r8-10    i1,i2,i3        (input image vector)
*/

.text
.globl  _fast3x3
_fast3x3:
  mov  g0, r8      /* 3x3 vector */
  mov  g1, g7      /* image pointer */
  subo  1, g2, r3  /* image size */
  mov  g3, r12     /* output vector point */
```

```
  ldt  (r8), g0      /* input 3x3 kernel */
  ldt  16(r8), g4
  ldt  32(r8), g8
.loop1:
  ldob  (g7), r8     /* load in image and convolve */
  xor  g3, g3, g3
  muli  r8, g0, g3
  ldob  1(g7), r9
  muli  r9, g1, r4
  addi  r4, g3, g3
  ldob  2(g7), r10
  muli  r10, g2, r4
  addi  r4, g3, g3

  ldob  640(g7), r8
  muli  r8, g4, r4
  addi  r4, g3, g3
  ldob  641(g7), r9
  muli  r9, g5, r4
  addi  r4, g3, g3
  ldob  642(g7), r10
  muli  r10, g6, r4
  addi  r4, g3, g3

  ldob  1280(g7), r8
  muli  r8, g8, r4
  addi  r4, g3, g3
  ldob  1281(g7), r9
  muli  r9, g9, r4
  addi  r4, g3, g3
  ldob  1282(g7), r10
  muli  r10, g10, r4
  addi  r4, g3, g3

  addo  1, g7, g7     /* increment image pointer */
  addo  1, r12, r12   /* increment output line pointer */
  cmpi  0, g3         /* if sum < 0, sum = 0 */
  ble   cont
  lda   0, g3
cont:
  stob  g3, (r12)
  cmpdeco  0, r3, r3
  bl   .loop1
  ret
```

Compare Strings

The following subroutine compares two C-style null-terminated strings
and returns an indication of the outcome of the comparison. The
application uses the `scanbyte` instruction to search for the null string
terminator.

```
  .globl  _strcmp
  .leafproc _strcmp,__strcmp
  .align 2

.rett:
  ret
_strcmp:
  lda   .rett,g14
__strcmp:
  ld  (g0), g5      # fetch first word of source_1
  mov  g14,g7       # preserve return address
  ldconst  0,g14    # conform to register conventions
  ldconst  0xff,g4  # byte extraction mask
.wloop:
  addo  4,g0,g0   # post-increment source_1 byte ptr
  ld  (g1), g3    # fetch word of source_2
  scanbyte 0,g5   # does word have a null byte?
  mov  g5,g2      # save a copy of the source_1 word
  be  .cloop      # branch if null byte encountered
  cmpo  g2,g3     # are the source words the same?
  addo  4,g1,g1   # post-increment source_2 byte ptr
  ld  (g0), g5    # fetch ahead next word of source_1
  be  .wloop      # fall thru if words are unequal

.cloop:
  and  g4,g2,g5  # extract and compare individual bytes
  and  g4,g3,g6
  cmpobne  g5,g6,.diff # if they diff, go return 1 or -1
  cmpo  0,g6      # they are the same. Are they null?
  shlo  8,g4,g4   # position mask for next extraction
  bne  .cloop     # loop if null not encountered
```

```
  mov  0,g0  # return equality
  bx  (g7)
.diff:
  bl  .neg
  mov  1,g0
  bx  (g7)
.neg:
  subi  1,0,g0
.exit:
  bx  (g7)
```

Examples Using Floating-point Instructions

The examples in this section use the on-chip numerics instructions
described in Chapter 6. The examples show:

- code optimization by reordering
- matrix multiplication with real arithmetic
- basic numerics operations using load, move, and store
- exponentiate with arbitrary exponent using rounding and scaling
- rectangular to polar conversions using trigonometric functions
- a call to the fault handler

Optimize a Numerics Application

This example shows two programs. The second, _testfast, is a
speed-optimized version of the first routine, _testslow.

```
  .text
  .align  4
  .globl  _testslow
_testslow:
  ldconst  999999, g3
  mov  g0, g13    # load address pointer
  mov  g1, r12    # load address pointer
  ldconst  0, r3  # store loop counter
  ldl  (g13),r14
  ldl  three_point_four,r10
```

```
loop_begin:

  ldl  (g13), r14
  mulrl  r14, r10, r8
  stl  r8, (r12)
  ldl  8(g13),r6
  mulrl  r6, r10, r4
  stl  r4, 8(r12)
  addo  1,r3,r3
  cmpi  r3,g3
  ble  loop_begin
  ret

  .data
  .align  4
three_point_four:
#
# below value is 3.4 in 64 bit real format
#
  .word  858993459
  .word  1074475827
  .text
  .align  4
  .globl  _testfast
_testfast:
  ldconst 999999, g3
  mov  g0, g13   # load address pointer
  mov  g1, r14   # load address pointer
  ldconst 0, r3  # store loop counter
  ldl  (g13),r12
  ldl  three_point_four,r10

loop_begin:

  ldl  8(g13), r4
  mulrl  r10, r12, r8
  stl  r8, (r14)
  mulrl  r4, r10, r6
  stl  r6, 8(r14)
  ldl  (g13), r12
  addo  1,r3, r3
  cmpi  r3, g3
  ble  loop_begin
  ret
```

```
  .data
  .align 4
three_point_four:
#
# below value is 3.4 in 64 bit real format
#
  .word  858993459
  .word  1074475827
```

Perform Matrix Multiplication

The following source code shows an optimized version of a 1 x 4 matrix multiply routine using real-valued arithmetic. The C program in the example sets up a sample matrix and uses the C version of the matrix multiply. Compare the C and assembly language versions.

Assembly Code

```
/*
  r3  no. of vectors
  g7  input vector pt
  g3  output vector pt
  g0-2  a11,a12,a13,a14
  g4-6  a21,a22,a23,a24
  g8-10  a31,a32,a33,a34
  r4-7  a41,a42,a43,a44  / translation vectors /
  r8-11  i1,i2,i3,i4     /input vector/
  r12-15  o1,o2,o3,o4    /output vector/

  fast1x4 does translation and rotation of the
  image supplied
*/

.text

.globl  _fast1x4
_fast1x4:
  mov  g0,r8     /* 4x4 vector */
  subo  1,g2,r3  /* image size */
  movrl  g4,fp0  /* translate x */
  movr  fp0,r4
  movrl  g6,fp0  /* translate y */
```

```
  movr  fp0,r5
  movrl  g8,fp0   /* translate z */
  movr  fp0,r6
  mov  g1,g7      /* image pointer */
  ldt  (r8), g0
  ldt  16(r8), g4
  ldt  32(r8), g8

  mov  r4,r4

.loop:
  ldt  (g7),r8

  mulr  r8, g0, fp0
  mulr  r9, g4, fp1
  addr  fp1, fp0, fp0
  mulr  r10, g8, fp1
  addr  fp1, fp0, fp0
  addr  r4, fp0, r12

  mulr  r8, g1, fp2
  mulr  r9, g5, fp3
  addr  fp3, fp2, fp2
  mulr  r10, g9, fp3
  addr  fp3, fp2, fp2
  addr  r5, fp2, r13

  mulr  r8, g2, fp0
  mulr  r9, g6, fp1
  addr  fp1, fp0, fp0
  mulr  r10, g10, fp1
  addr  fp1, fp0, fp0
  addr  r6, fp0, r14

  stt  r12, (g3)
  addo  12, g3, g3
  addo  12, g7, g7
  cmpdeco  0, r3, r3
  bl  .loop

  ret
```

C Code

```c
#include <stdio.h>
main ()
{
static float a[4][4] = {
  {0.0, 0.1, 0.2, 0.3},
  {1.0, 1.1, 1.2, 1.3},
  {2.0, 2.1, 2.2, 2.3},
  {3.0, 3.1, 3.2, 3.3}};

static float b[4] = {0.0, 0.1, 0.2, 0.3};
float c[4];

  fast1x4(a, b, c);

}
/*********************************************/
/* FAST1X4                               */
/* outer loop is the index for each column   */
/* of the kernel                             */
/*                                           */
/* inner loop is the index for each row of   */
/* the kernel, and the index for the source  */
/* matrix                                    */
/*                                           */
/* results are stored in a 1x4 matrix       */
/*                                           */
/*  input: kernel - 4x4 matrix              */
/*         source - 1x4 matrix              */
/*                                           */
/*  output: dest  - 1x4 matrix              */
/*********************************************/
fast1x4 (kernel, source, dest)
float kernel [4][4];
float source[];
float dest[];
{
int i,j;
float temp;
```

```
  for (i=0; i<=3; i++) {
    temp = 0.0;

    for (j=0; j<=3; j++) {
      temp += source[j] * kernel[j][i];
    }
    dest[i] = temp;
  }
}
```

Perform Basic Numerics Operations

This example represents a source code fragment that does many of the basic numerics operations.

```
# Assume: src1 = 32-bit real value in memory
#         src2 = 96-bit extended real
#         dst uninitialized in .bss section
# (all should be appropriately aligned)
  ld   src1, g0    # load 32-bit real
  ldt  src2, g4    # load 96-bit extended real
  movr  g0, fp0    # convert 32 to 80-bit
  cpysre  fp2, g4, fp3  # copy sign
  movrl  fp3, g0  # convert 80 to 64-bit real
  stl  g0, dst     # store dual register long
```

Exponentiate With an Arbitrary Exponent

This example shows an assembly language code fragment to handle exponentiation with an arbitrary exponent.

```
# Assume register g0 = real exponent
  roundr  g0, fp0    # fp0 = integer part
  subr  fp0, g0, g0  # g0 = fractional part
  expr  g0, g0       # g0 = 2^g0 - 1
  addr  1.0, g0, g0  # compensate for -1
  cvtri  fp0, g1 g0  # exponentiate integer
                     # and scale result
```

Convert Between Coordinate Systems

This source code fragment converts from a rectangular to a polar coordinate system and vice-versa. These routines use several of the real arithmetic and trigonometric functions.

```
# Rectangular to polar conversion
# Assume x, y are 64-bit reals in memory
# r, theta are quad-aligned 96-bit locations

rect_to_polar:
  ldl  x, g0           # load x coordinate
  ldl  y, g2           # load y coordinate
  atanrl  g0, g2, fp0  # fp0 = arctan y/x
  mulrl  g0, g0, g0    # square x
  mulrl  g2, g2, g2    # square y
  addrl  g0, g2, g4    # g4 = x^2 + y^2
  sqrtrl  g4, fp1      # fp1 = sqrt g4
  movre  fp0, g8       # convert theta to 96-bit
  movre  fp1, g12      # convert r to 96-bit
  stt  g8, theta       # store extended angle
  stt  g12, r          # store extended radius
  ret
#
# Polar to rectangular conversion
# Assume:
# r, theta quad-aligned 64-bit real values
# x, y are 96-bit locations in memory

polar_to_rect:
  ldl  r, g0           # load radius
  ldl  theta, g2       # load angle
  cosrl  g0, fp0       # fp0 = cos theta
  sinrl  g0, fp1       # fp1 = sin theta
  mulrl  fp0, g0, fp0  # fp0 = r cos theta
  mulrl  fp1, g0, fp1  # fp1 = r sin theta
  movre  fp0, g8       # convert x to 96-bit
  movre  fp1, g12      # convert y to 96-bit
  stt  g8, x           # store extended x
  stt  g12, y          # store extended y
  ret
```

Retrieve Fault Record Pointer

The following routine demonstrates how to retrieve the fault record from the stack after a floating-point fault has occurred. The fault handler calls this routine immediately after the fault is signaled. The routine continues execution at the point of interruption afterwards.

The procedure `return_fault_ptr` returns the information caused by a fault to the programmer, as follows:

- The procedure returns a pointer to the fault record.
- The procedure copies all global/local registers at the time of the fault into a global structure. This structure is an array of 32 unsigned integers, which contain g0 through g15 and r0 through r15. Use a global structure to avoid passing parameters and corrupting the registers. The programmer assumes that this routine is called directly by the fault handler so it uses that knowledge to unwind the stack.
- The stack provides the linkage that you use to find the fault data, as shown in Figure 8-2.

Figure 8-2   Stack For Fault Handler
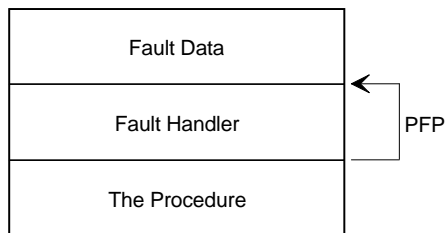
OSD1136

```
    .globl _return_fault_ptr
_return_fault_ptr:
  lda  0x001f0000, r8         # load pc mask
  lda  0x001f0001, r9         # load pc mask
  modpc  r8, r9, r8           # set priority to MAX
                              # to avoid interrupts
  flushreg                    # make stack current

  lda  _register_set, r5
  stq  g0, (r5)               # store global registers
  stq  g4, 16(r5)
  stq  g8, 32(r5)
  stq  g12, 48(r5)
  lda  0xffffffc0, r13    # PFP mask

  ld  (pfp), r6               # chain back past previous call
  and  r6, r13, r6            # mask off return bits
  ldq   (r6), r8              # load local registers
  stq   r8, 64(r5)            # store local registers
  ldq   16(r6), r8            # load local registers
  stq   r8, 80(r5)            # store local registers
  ldq   32(r6), r8            # load local registers
  stq   r8, 96(r5)            # store local registers
  ldq   48(r6), r8            # load local registers
  stq   r8, 112(r5)           # store local registers

  ldconst  48, r3             # length of fault record
  subo  r3, pfp, g0           # store start of fault to g0
  ldq  32(g0), r8             # get pc, ac, ip
  stl  r8, 128(r5)            # store pc, ac
  st  r11, 136(r5)            # store ip
  ldconst  0xffffffff, r13  # load mask
  ldconst  0, r14             # turn off tracing in monitor
  modtc  r13, r14, r14    # get old trace controls
  st  r14, 140(r5)            # and store to memory
  ret                         # and return it to handler

  .globl _begin
_begin:

  ldconst  _register_set, r5
  ldconst  0xffffffff, r6  # load mask
  ld  140(r5), r14           # load program trace
  modtc  r6, r14, r14      # set trace controls
```

```
  ldconst  1, r7      # load bit
  modpc  r7, r7, r7   # and restore

  callx  (g0)         # vector off to routine
  ret                 # should never return,
                      # but just in case

  .globl _continue_execution
_continue_execution:
  call restore_state
  ret                 # return to procedure

restore_state:

  flushreg            # make stack current
                      # AND.. Invalidate cache
  lda  _register_set, r5
  ld  60(r5), r15     # get frame ptr

  lda  0xffffffff, r6  # load mask
  ld  132(r5), r7     # bring in stored ac
  modac  r6, r7, r7   # and restore
  st  g0, 8(r15)      # store ip in return ptr
  ldq  (r5), g0       # load 1st 4 globals
  ldq  16(r5), g4     # load next 4 globals
  ldq  32(r5), g8     # load next 4 globals
  ldq  48(r5), g12    # and restore
  ret

  .data

  .globl  _register_set

_register_set:
  .space 160          # reserve storage for registers
```

Glossary

absolute expression	A valid assembly language symbol or expression that, when evaluated, produces a value that does not change with relocation at link time.
absolute value	A fixed number directly calculated by the assembler and used in the assembly. Absolute values can be used in assembly language expressions.
address space	The range of addresses available to a process.
addressing modes	Methods available for instructions to specify a memory address as an operand. The range of addressing modes for each instruction depends on the instruction type.
alignment (memory)	The allocation of data in memory relative to appropriate boundaries for efficient processing. For example, data words (4 bytes) must be located at memory addresses divisible by 4.
alignment (register)	When a single instruction accesses a dual-register group, the register specified in the instruction must be even numbered (e.g. `g0`, `r2`, `g6`). If an instruction accesses a triple- or quad-register group, the number of the register specified must be a multiple of four (e.g. `g0`, `g4`, `r8`).
ASCII-coded decimal	A data word containing a decimal digit (0 - 9) encoded in the four low-order bits.
assembler directive	A source code statement that indicates assembly information other than machine instructions to the assembler (e.g., debug information and data entries).

big-endian architecture	The bytes follow a left-to-right order from the most significant bit to least significant bit (example: HP 9000 Series 300 workstations).
bit field	A contiguous series of up to 31 bits in a data word, specified by the starting bit position and field length.
burst access	A technique that allows the processor to execute multiple data cycles after a single address cycle.
calling convention	The set of instructions inserted in the object code by a language processor to handle parameter passing, stack and register use, and return values in a function call.
COFF (Common Object File Format)	A format for storing file and section headers, relocation information, symbol tables, and other components of an object file. When you invoke the assembler as `gas960c`, the assembler generates output in this format.
comparand	Instruction operand used in a comparison that sets the condition code.
condition code	Three bits that can be set by the processor as a result of comparisons and other operations. The condition code bits can be tested by running programs.
core architecture	A set of processor features available across all i960 processors for supporting ordinal and integer arithmetic, faults, interrupts, etc.
directive	See *assembler directive*.
double-word	64 bits of data. Double-word data is also called long data, and must be aligned to 8 byte boundaries for efficient use by load and store instructions.

ELF (Executable and Linkable Format)	The Intel 80960 ABI-compliant object module format. When invoked with the gas960e command, the assembler emits this format.
exception	An unusual condition that detected by the processor as the result of instruction execution. See also *fault*.
extended-real	IEEE standard 80-bit real number that can be processed in an 80-bit floating-point register. A 96-bit extended-real value is the same as the 80-bit extended-real value with the most-significant 16 bits ignored. A 96-bit extended-real value can be loaded into an aligned global or local triple-register group.
external reference	A symbol in an object module that refers to a location in another object module. The linker resolves external references when creating an executable module.
fatal error	An error encountered during assembly that terminates the assembly process without producing object code.
fault	An event that the processor generates to indicate that, while executing a program, a condition arose that could cause the processor to go down a wrong and possible disastrous path. One example of a fault condition is a divisor operand of zero in a divide operation: another example is an instruction with an invalid opcode.
floating-point format	IEEE standard formats for floating-point, or real, numbers. See also real number formats.
floating-point literals	The values +0.0 and +1.0.
floating-point register	80-bit registers fp0 through fp3, available on the i960 KB processor only.
global register	32-bit registers g0 through g15.

half-word	16-bit integer or ordinal value. Half-word data is also called short data. Half-word data must be aligned on even boundaries for efficient use by the load and store instructions.
identifier	A symbol or name used in the source code for any purpose.
Immediate value	A value that is contained in the machine instruction itself (e.g., the value 10 in the instruction `mov 10, r5`) The value must be known at assembly time (i.e., cannot be unresolved).
in-circuit emulator	A software/hardware product used to debug embedded applications or hardware systems by emulating a particular processor.
include file	A source text file inserted by the assembler into the primary source text file.
instruction pointer	An internal processor register that contains the address of the instruction currently being executed.
instruction set	The set of executable instructions in a given i960 architecture.
integer	A positive or negative whole number or zero. The range of values that an integer can represent depends on its width (for example, short, word, or double-word).
interactive mode	An assembler mode of operation that allows direct input from the standard input device.
interrupt	A signal to the processor that an external condition requires immediate attention. An interrupt initiates a predefined handler, defined in the interrupt table, to service the condition.
J bit	In IEEE real number formats, a bit which is set (1) for zero and denormalized finite numbers and clear (0) otherwise. This bit can be used to detect invalid real numbers.

leaf procedure	A local procedure that can be executed by a branch and link instruction because it doesn't require that local registers be saved (rather than a call instruction).
linker	A utility used in preparing object code for execution by combining object files and resolving external references.
list file	A text file generated by the assembler, containing source code listing, symbol information, and other information.
literal value	A value in a source operand that can be used as immediate data in the instruction.
little-endian architecture	The bytes follow a right-to-left order from the most significant bit to least significant bit, as they do on Intel processors.
local register	32-bit registers `r0` through `r15`.
location counter	The current address of an instruction. The location counter starts at zero and is incremented by the length of each instruction or data value in the program.
long data	64-bit integer or ordinal value. Long data is also called double-word data.
long-real	IEEE standard 64-bit floating number that can be loaded into an aligned global or local register pair.
numerics architecture	Processor architecture supporting hardware floating-point arithmetic and trigonometric operations available on the i960 SB/KB processors.
object code	Instructions and associated data for a program, in binary format. This is the output generated by the assembler and consumed by the linker.

object file	The file containing the object module generated by the assembler when assembly is successful. The output can be in different formats based on how you invoke the assembler (COFF for `gas960c`, ELF for `gas960e`, and b.out for `gas960`).
object module	The formatted object code resulting from assembly.
opcode	The portion of each machine language instruction that determines the action caused by the instruction.
operand	The argument of an assembly language directive or instruction that represents data used in the operation.
ordinal	An unsigned whole number or zero. The range of values that an ordinal can represent depends on its width (for example, short, word, or double-word).
physical address	The address of a specific hardware memory location, as sent over the bus.
pipelining	A technique that allows the processor to output the address of the next bus request during the current data cycle, maximizing bus efficiency.
position-independent code and data	The code (`.text` section) or data (`.data` or `.bss` section) is loaded at a run-time address that is computed as an offset from a specific location in memory.
precision	A measure of the accuracy with which a real number can be represented.
preprocessor	A program that processes an assembly language source file before the actual assembly process (for example, the macro processor mpp960).
process	An executable module that represents a complete task to the system.

program sections	Parts of a program containing code (text section), initialized data (data section), and uninitialized data (bss section). Each section is handled separately by the linker.
protected extension	Filename extensions that protect the file from being overwritten by the assembler. The assembler-protected extensions are: `.s`, `.as`, and `.asm`.
quad-word	128 bits of data. Quad-word data must be aligned on 16-byte boundaries for efficient use by load and store instructions.
real	IEEE standard 32-bit real value that can be loaded into a single global or local register.
real number formats	IEEE standard formats for floating-point, or real, numbers: 32-bit (real), 64-bit (long-real), 80- and 96-bit (extended-real).
register	Any global register (`g0` - `g15`), local register (`r0` - `r15`), floating-point register (`fp0` - `fp3`), or special function register (`sf0` - `sf4`).
register group	A set of 2, 3, or 4 registers that participate in an instruction. See also *alignment (register)*.
search path	A list of directories used as possible pathnames to a file.
short data	16-bit integer or ordinal value. Short data is also called half-word data. Short data must be aligned on even byte boundaries for efficient use by the load and store instructions.
source directory	The directory containing your primary source file.
source file	The assembly language input to the assembler.
special function register	A 32-bit register (sf0 - sf4) used to control specific sections of the processor. These registers can be manipulated like any other register, but the contents affect the processor's behavior directly.

stack	A portion of memory used by the processor to store call and return information.
stack frame	A portion of the stack allocated by a procedure for storing temporary values until the procedure returns.
symbol table	A table in the object file containing information about the symbols used in a program.
system procedure	A procedure executed by a call system (`calls`) instruction. The entry point for each system procedure appears in the system procedure table.
triple-word	128 bits of data. Triple-word data must be aligned on 16-byte boundaries for efficient use by load and store instructions.
warning	An indication of an unusual condition encountered during assembly. In these situations, the assembler issues a message but continues processing the source file.
word	32 bits of data. Word data must be aligned on 4-byte boundaries for efficient use by the load and store instructions.

Index