April 1993
Order Number: 312546-001

PARAGON™ XP/S

i860™ 64-BIT MICROPROCESSOR

ASSEMBLER

REFERENCE MANUAL

Intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	4/93

LIMITED RIGHTS

Preface

This manual describes the Paragon™ XP/S i860™ 64-bit microprocessor assembler (**as860**). The assembler is designed for direct use by programmers and for indirect use as a postprocessor for the output of high-level language translators. It supports the entire instruction set of the i860 microprocessor.

This manual assumes you are an application programmer proficient in the use of some assembly language and that you are familiar with the architecture and instruction set of the i860 microprocessor as presented in the *i860™ 64-Bit Microprocessor Family Programmer's Reference Manual.*

Organization

Chapter 1	Introduces the assembler, command-line syntax, command-line options, and assembler directives.
Chapter 2	Describes the syntax of an assembly language program.
Chapter 3	Describes the syntax of individual assembly language instructions and pseudo-instructions. (For a detailed description of the machine instructions, refer to the *i860™ 64-Bit Microprocessor Family Programmer's Reference Manual.*)
Chapter 4	Describes assembler directives.
Chapter 5	Tells how to use the macro preprocessor (**mac860**).

Notational Conventions

This manual uses the following notational conventions:

Bold — Identifies command names and switches, system call names, reserved words, and other items that must be entered exactly as shown.

Italic — Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace` — Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

Bold-Italic-Monospace — Identifies user input (what you enter in response to some prompt).

`Bold-Monospace` — Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>` **`<s>`** **`<Ctrl-Alt-Del>`**

`[ ]` — Surround optional items.

... — Indicate that the preceding item may be repeated.

| — Separates two or more items of which you may select only one.

`{ }` — Surround two or more items of which you must select one.

Applicable Documents

For more information, refer to the following manuals:

- *i860™ 64-Bit Microprocessor Family Programmer's Reference Manual*, Intel order number 240875
- *Paragon™ OSF/1 User's Guide*, Intel order number 312489
- *Paragon™ OSF/1 C Compiler User's Guide*, Intel order number 312490
- *Paragon™ OSF/1 Fortran Compiler User's Guide*, Intel order number 312491

Comments and Assistance

Intel Supercomputer Systems Division is eager to hear of your experiences with our new software product. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
phone: 800-421-2823
email: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Japan Intel Corporation K.K.
Supercomputer Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Supercomputer System Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bel Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 629-7600

Table of Contents

Chapter 1
Getting Started

Chapter 2
Assembly Language Syntax

Chapter 3
Instruction Syntax

Chapter 4
Assembler Directives

Chapter 5
Using Macros

List of Tables

Getting Started 1

This chapter introduces the **as860** assembler and discusses the command-line syntax, command-line options, and assembler directives it recognizes.

Using the Assembler

In addition to supporting the entire instruction set of the i860™ microprocessor, the assembler provides:

- Common object file format (COFF) output modules
- Long identifiers (up to 80 characters)
- Completely relocatable object modules
- Enforcement of coding rules unique to the i860 processor
- Optional source and code listings
- Symbolic debugger support

Command-line Syntax

The **as860** command-line syntax is:

as860 [*options*] [*source_file*]

Where:

options — Represent command-line options that control the assembly process. Most options regulate the output created by the assembler. Each option must be preceded by a hyphen (-). Some options are followed by arguments. Arguments are usually shown separated from the option by a space, but the spaces are optional.

If you do not declare an option, its default setting determines the function of the assembler.

source_file — Represents the complete path, filename, and extension of the assembly-language source file. If you do not specify the path, the assembler searches for the filename in the current directory. If you do specify a source file name, the assembler uses standard input as the source until you type <Ctrl-D>. You can also use the UNIX redirection operator (<), to specify the source. (If you use standard input or the redirection operator, the output file is *a.out*.)

Command-line Options

Command-line options affect input conditions, the assembly process, and output selections. Table 1-1 lists the assembler command-line options.

Table 1-1. Assembler Options

Option	Function
–a	Prohibit the importing of any symbols that are referenced but are otherwise undefined.
–l[listfile]	Write the source listing to listfile or to standard output if no listfile argument is specified.
–L	Preserve text symbols starting with ".L" in the debug section.
–m	This option is recognized but ignored.
–o objfile	Put the output object file in objfile. If you omit this switch, the default object file name is produced by stripping any directory prefixes from filename, stripping any suffixes, and appending .o. An existing file with the same name is silently overwritten.
–R	Suppress all **.data** directives. Code and data are both assembled into the **.text** section.
–V	Display assembler version information.
–x	Enable additional checks of the program to find illegal or dangerous sequences of instructions.

Case Significance

Case is significant for assembler options. The following example shows the warning message that results when an uppercase *A*, which is not a valid option, is used by mistake.

```
as860 -A myprog.s

as860 *command line*: Fatal: unknown flag: '-A'

Usage: as860 [ options ] file
```

Arguments

Arguments for options must follow the options that require them. A space between the option and its argument is optional (except for the **–l** option, which does not allow a space if an argument is present). If the command line contains a syntax error, the assembler does not complete processing of the command and issues an error message.

Filename Specifications

For options that require a filename specification, you can provide a complete pathname, filename, and extension. If you do not specify a pathname, the assembler uses the current directory.

If the specified file already exists for the listing-file option **–l** or the object-file option **–o**, the assembler overwrites it after checking that the command syntax is in order.

Multiple Options

The following examples show valid uses of multiple assembler options. The first example specifies an object file (**–o**), a listing file (**–l**), and an input file.

```
as860 -o test1.o -ltest.lst source.s
```

The next example suppresses **.data** directives (**–R**), and enables checking for out-of-sequence instructions (**–x**). The resulting object file is named *test.* Separate options with a space, as in the following example:

```
as860 -R -x test.s
```

Input Files

The **as860** input file is an ASCII source file consisting of assembly language statements, including mnemonics for i860 microprocessor instructions and assembler directives. The input file for **as860** is an output of the **icc** compiler, the **if77** compiler, the **mac860** preprocessor, or is created with an editor.

If you fail to specify a source file at invocation, the assembler accepts input until you type a `<Ctrl-D>`.

Output Files

Upon successful assembly, the assembler produces an object file and, optionally, a listing file as output.

By default, the object filename is the source filename with a .o extension, but you can choose a specific object filename using the **–o** option. The assembler overwrites an existing file with the same specified or default name. If you use standard input or the redirection operator (<) for input, the output file is *a.out*.

Object Files

After successful assembly, the assembler produces an object file in the common object file format (COFF). The **as860** object file contains i860 instructions, relocation information, a symbol table, and, optionally, symbolic debugging information. After the linker, **ld860**, processes the object file it becomes part of an executable program.

NOTE

The assembler generates only the standard COFF file header, not the optional header.

The primary output of the assembler is an object module with four sections: **.text**, **.data**, **.bss**, and **.abs**. When the linker combines several object modules, the sections from each input module are concatenated to form a single output module consisting of the combined **.text** sections, the combined **.data** sections, and any **.abs** sections (the **.bss** section is not physically present in object files).

Operating systems make distinctions between text and data memory. The assembler treats both sections identically; however, it supports the linker and operating system by assigning different type information to symbols in the different sections.

.text section — When assembly begins, output is directed to the **.text** section. The **.text** section customarily contains instructions and constant data. Use the directive **.text** to return to the **.text** section, after output has been diverted to another section. Refer to Chapter 4 for more information.

.data section — The **.data** section customarily contains variable data to which the program assigns initial values. Use the directive **.data** to divert output to the **.data** section. Refer to Chapter 4 for more information.

.bss section — The **.bss** section is not physically present in object files; rather, it serves as a type for symbols that are assigned addresses in an area of memory that is allocated only when the program is loaded. This memory is initialized with zeros. Use the directives **.lcomm** and **.comm** to allocate **.bss** section memory. Refer to Chapter 4 for more information.

.abs section — There can be zero, one, or more absolute (**.abs**) sections. An absolute section is assigned to a specific virtual address and, possibly, to a specific physical address. An absolute section can contain either text or data. Use the directive **.abs** to allocate an **.abs** section. Refer to Chapter 4 for more information.

Assembly-language programmers must be aware of these sections to ensure that instructions, constants, and variables are correctly located by the linker.

The object-file symbol table contains an entry for each symbol defined in the assembly, thus providing for linking to referenced modules by symbol. Assembler directives define symbols in the assembly language source code. Directives control the format, processing, and content of the assembler output.

Listing Files

The **as860** listing file is an optional, line-numbered listing of assembly-language statements with their hexadecimal representations and any error messages generated by the assembler. Use the *–l* option to generate the listing. You can enable or disable the listing using the directives **.list** and **.nlist**. Refer to Chapter 4 for more information.

The following sample invocation creates a listing file named *mylist.lst*.

```
as860 -lmylist.lst myprog.s
```

The listing file is an ASCII file containing four columns as follows:

instruction location line # source code

Where:

instruction — Is the machine instruction or data, in hexadecimal, generated by the line of source code.

location — Is the current location-counter value in hexadecimal. This value indicates the offset in bytes from the start of the object file.

line # — Is the line number in decimal.

source code Is the assembly-language statement from the source code. This column can include any of the following elements: a label, the instruction mnemonic or assembly directive, the operands, and any comments.

Figure 1-1 shows a portion of a listing file produced by the assembler.

```
                          1          .file   "testa.f"
                          2 // PGFTN Rel 2.1 -opt 1 -norecursive
                          3          .text
                          4          .globl  __unnamed_
                          5          .align  8
                          6 __unnamed_:
                          7          .globl  _MAIN_
                          8 _MAIN_:
                          9 .a1 = 0
                         10 .f1 = 48
00001fec 00000000        11          orh h%.STACK1+.f1-16, r0, r31
4002ffe7 00000004        12          or l%.STACK1+.f1-16, r31, r31
0118e01f 00000008        13          st.l fp, 0(r31)
0000e3a3 0000000c        14          mov r31, fp
0508e01f 00000010        15          st.l r1, 4(r31)
.
.
.
                         24 // lineno: 0
00001fec 00000014        25          orh h%.C1_265, r0, r31
6801f1e7 00000018        26          or l%.C1_265, r31, r17
06001094 0000001c        27          adds 6, r0, r16
07001294 00000020        28          adds 7, r0, r18
34121c94 00000024        29          adds 4660, r0, r28
fde77f1c 00000028        30          st.l r28, -4(fp)
.
.
.
```

Figure 1-1. Sample Listing File

The location counter starts at 00000000 and is incremented each time a machine instruction is generated. The value of the location counter indicates the number of bytes from the start of the object file. For lines containing comments, the assembler does not increment the counter.

Listing-file error messages appear preceding the line in which the error was detected. Error messages also appear on the standard error-reporting device, usually the monitor screen.

```
                          1          .file   "testa.f"
                          2 // PGFTN Rel 2.1           -opt 1 -norecursive
                          3          .text
                          4          .globl  __unnamed_
                          5          .align  8
                          6 __unnamed_:
                          7          .globl  _MAIN_
                          8 _MAIN_:
                          9 .a1 = 0
                         10 .f1 = 48
 00001fec 00000000        11          orh h%.STACK1+.f1-16, r0, r31
 4002ffe7 00000004        12          or l%.STACK1+.f1-16, r31, r31
 0118e01f 00000008        13          st.l fp, 0(r31)
 0000e3a3 0000000c        14          mov r31, fp
 0508e01f 00000010        15          st.l r1, 4(r31)
.
.
.
                         24 // lineno: 0
 00001fec 00000014        25          orh h%.C1_265, r0, r31
 6801f1e7 00000018        26          or l%.C1_265, r31, r17
       ***** error ****             line 27: Error:
'r100' syntax error
06001094 0000001c    27          adds 6, r100, r16
07001294 00000020    28          adds 7, r0, r18
34121c94 00000024    29          adds 4660, r0, r28
fde77f1c 00000028    30          st.l r28, -4(fp)
.
.
.
```

Figure 1-2. Listing File With Error Message

Assembler Directives

In addition to the assembler command-line options shown in Table 1-2, you can use a set of optional assembler directives to determine the format, processing, and content of the assembler output (refer to Chapter 4 for more information on assembler directives). Assembler directives govern the following operations of the assembler:

- alignment on boundaries
- dual-mode instruction interpretation
- section control
- block and common space definition
- record and structure definition
- memory initialization and allocation
- enumeration
- external symbol definition
- temporary register assignment
- listing management
- debugging support

Assembly Language Syntax 2

Statements

The assembler accepts five general statement formats:

1. An instruction statement results in the generation of one (and sometimes two or three) machine instructions. The instructions are defined in Chapter 3.

   [*labels:*]... *instruction* [*operands*][*// comment*]

2. A directive statement controls the operation of the assembler or the macro preprocessor. The assembler directives are defined in Chapter 4. The macro preprocessor directives are defined in Chapter 5.

   [*labels:*]... *directive* [*parameters*][*// comment*]

3. An assignment statement defines a symbol that can be used in place of the given constant integer expression. Assignments are defined in this chapter.

   [*labels:*]... *symbol* =[:] *expr*[*// comment*]

4. An empty statement contains nothing other than spaces, tab characters, and comments. Empty statements have no meaning to the assembler. They can be inserted freely to improve the appearance of a source file and to clarify the code.

5. Two adjacent slashes(/ /) introduce a comment. The slashes can appear anywhere in the line; the comment extends from the slashes to the end of the line.

   [*// comment*]

An assembler statement is contained within one line of an input file. Multiple statements can be entered on a single line if each statement is separated from the previous statement by a semicolon (;). For example:

```
pfadd.ss f23, f31, f31 ; fld.d 8(r16)++, f22
```

For statements, the input character sequence is separated into lines by the line-feed (LF) character (also called newline). A carriage return (CR) can precede the LF, in which case the CR-LF pair is treated as a single newline.

NOTE

The assembler accepts only lowercase machine instructions and register names. For example, this instruction is acceptable:

```
addu r1, r2, r3
```

but these are not:

```
Addu r1, r2, r3
ADDU r1, r2, r3
addu R1, r2, r3
```

Constants

The assembler accepts both numeric and alphanumeric constants.

Numeric

Numeric constants may be integers or floating-point numbers. Integer constants can be expressed according to any of the following bases:

Decimal	A sequence of the digits 0-9. The sequence may optionally be prefixed by **0t** or **0T**. If the prefix is not used, the digit sequence must not begin with a zero.
Hexadecimal	A sequence of the digits 0-9, A-F, a-f prefixed by **0x** or **0X**
Binary	A sequence of the digits 0-1 prefixed by **0b** or **0B**
Octal	A sequence of the digits 0-7 either beginning with the digit **0** (zero) or prefixed by **0o** or **0O** (zero, letter oh)

A floating-point constant has the form:

[0f | 0F][*integer*][.[*fraction*]][**e**{[+][-]} *exponent*]

where *integer*, *fraction*, and *exponent* are decimal integers. The prefix **0f** (or **0F**) may be omitted when the presence of a decimal point makes it clear that a floating-point number is intended.

NOTE

Although a token such as **.2e12** is also a legal symbol, the assembler recognizes it as a floating-point constant. Do not use such tokens as identifiers.

Example

This example shows numeric constants in storage-allocation statements. You can use numeric constants in a variety of other places.

```
//Valid numeric constants
mask:           .byte           0b01101001      //Binary
year:           .short          365             //Decimal
                .long           0t1950344       //Decimal
                .short          0xffff          //Hexadecimal
factor:         .float          1.2
                .float          1.2e12
                .float          1.2e+12
                .float          1.2e-12
                .double         1.e12
                .double         .2e12
                .double         .2e+12
                .double         .2e-12
```

Alphanumeric

There are two types of alphanumeric constants:

Character Constant — A single character enclosed within single quotation marks ('). A character constant is treated as an integer numeric constant with a value equal to the code of the ASCII character specified.

String Constant — A sequence of character specifications enclosed in double quotation marks ("). A string constant supplies a sequence of values for the data storage directives. A NUL character is *not* automatically appended to the string by the assembler. Refer to the **.byte** and **.string** directives in Chapter 4 for more details about strings.

Character and string constants can contain any ASCII character. Use the backslash character (\) within character and string constants to enter apostrophes and quotation marks repetitively and to specify certain control characters symbolically. An apostrophe is valid within a quotation mark enclosed string constant, and likewise, a quotation mark is valid within an apostrophe enclosed character constant. The symbolic character specifications are defined in Table 2-1. A backslash followed by any character not shown in the first column of Table 2-1 is equivalent to the character itself. For example, `\c` is equivalent to `c`, because Table 2-1 does not define `\c` as specifying a special character.

Table 2-1. Symbolic Character Specification

Symbolic Form	Character	ASCII Code
\0	NUL	0x0
\b	BS backspace	0x8
\t	TAB	0x9
\n	LF linefeed	0xA
\r	CR carriage return	0xD
****	Backslash	0x5C
\"	Double quote in string	0x22
\'	Single quote in constant	0x27
\f	Form Feed	0xC
\a	Bell	0x7
\v	Vertical tab	0x0B

Example

This example shows string constants in storage-allocation statements.

```
// Valid alphanumeric constants
        .byte     '*'                      // Character constant
        .byte     "***"                    // String constant
        .byte     "Ape\tBadger\tCamel\0" // Special characters
```

Integer

The term integer constant refers either to a numeric constant that is not a floating-point constant or to a character constant. The value of a character constant is the value of its ASCII code.

Symbols

You can use symbols to label memory locations or integer values. Symbols are composed of letters, digits, and the period (.), dollar sign ($), and underscore (_) characters. The first character of a symbol may not be a digit, a period, or a dollar sign. Both uppercase and lowercase letters are accepted, but are treated distinctly; for example, the symbol `a` is unrelated to the symbol `A`. Symbols may be up to 80 characters long; all characters are significant.

Symbols are defined by:

- The *label* part of statements
- The **.comm** and **.lcomm** directives
- Assignment statements
- The **.enum** directive

A symbol not defined by one of the preceding methods is considered *undefined.* A symbol that is used but not defined in the current module either is an external symbol (i.e., is declared in another module) or is an error. If the *–a* command-line option is not specified when you invoke the assembler, an undefined symbol is considered to be external. When *–a* is specified, such a symbol is treated as an error.

For external symbols (those declared in other modules), the assembler generates information in the output module that identifies them to the linker.

Labels

A *label* is a symbol that represents a location in either the *text* or *data* section. You can define multiple labels for the same location. A label can be either *named* or *temporary*.

Named Labels

A named label is a symbol followed by one or two colons. Labels defined with a single colon cannot be referenced from another module. Two colons specify that the label is global, so that it can be referenced by other modules. (The directive **.globl** provides another way to make a label global. Refer to Chapter 4 for more information.)

Example

```
xyz:                    // A local named label
abc::          .byte 1  // A global label
x1:x2:x3:      .byte 1  // Three labels on a line
```

Temporary Labels

A temporary label consists of a nonzero integer constant followed by a single colon. Any number of these labels may be present in a source program, even if there are duplicates.

A reference to a temporary label consists of the label's constant value followed immediately (i.e., with no intervening space) by an **f** or **b**. The trailing letter specifies that the reference is *forward* or *backward*, respectively. The integer specifies that the reference is to the nearest temporary label in the given direction that has the same integer value.

Use temporary labels only in text sections and only as operands of control transfer instructions.

Example

```
1:      br 1f           // skips the next three instructions
        nop
17:     br 1b           // selects prior branch instruction
        nop
1:                      // continue
```

Other Address-valued Symbols

The **.comm** and **.lcomm** directives assign the symbol *id* to a **.bss** section location. Symbols thus defined differ from labels, because labels refer to locations in the **.text** or **.data** section. The **.comm** directive establishes an undefined external symbol; the directive **.lcomm** establishes a local symbol. (Refer to the definitions of **.comm** and **.lcomm** in Chapter 4 for more information.)

Assignments

Assignments have the form

symbol =[:] *expr*

An assignment defines a symbol that can be used in place of the given constant integer expression. An assignment using = defines a local constant. An assignment using **=:** defines a global constant, whose 32-bit integer value is placed in the output symbol table so that it can be referenced by other modules. For example:

```
a=1           // A local constant
xyz =: 123    // A global constant
```

The .enum Directive

The **.enum** directive can be considered a form of assignment that also defines local symbols. Refer to Chapter 4 for more information.

Expressions

The assembler supports expressions formed of integers and of floating-point numbers.

You can use integer expressions in assembler statements where an integer value is required. Integer values are represented by the assembler in 32-bit, twos complement form. A basic integer expression can be any of the following:

- An integer constant
- An integer-valued symbol
- An address-valued symbol

A basic floating-point expression is a floating-point constant.

Given that *exp1*, *exp2*, and *exp3* are integer or floating-point expressions, the following are also expressions:

(exp)	Paired parentheses can be used freely to clarify or override operator precedence.
uop exp	*uop* is a unary operator.
exp1 bop exp2	*bop* is a binary operator.

Any of the arithmetic, bit, and Boolean operators listed in Table 2-2 can be used in integer expressions. All integer arithmetic is performed with 32 bits of precision. The operators defined for floating-point expressions are unary + and -, and binary +, -, *, and /. Operands do not need to be separated from operators by spaces. When an integer expression is combined with a floating point expression by a binary operator, the result is floating-point.

Table 2-2. Operators

Class	Operator	Operator Type	Function
Arithmetic	+	Unary	(none)
	-	Unary	Negation
	+	Binary	Addition
	-	Binary	Subtraction
	*	Binary	Multiplication
	/	Binary	Division
Bit	&	Binary	Logical AND
	^	Binary	Logical Exclusive OR
	|	Binary	Logical OR
	<<	Binary	Shift left
	>>	Binary	Arithmetic shift right
Boolean	!	Unary	Not
	<	Binary	Less than
	>	Binary	Greater than
	=	Binary	Equal
Type	**l%**	Unary	Select low-order half
	h%	Unary	Select high -order half
	ha%	Unary	Select high-order half and adjust

Bit and Boolean Operators

For the shift operators, *exp2* (the right-hand operator) specifies the number of bit positions to shift. The value of *exp2* must lie in the range 0 through 31.

The right shift is an arithmetic shift. It does not change the sign bit; rather, it propagates the sign bit to the right *exp2* bits.

Boolean operators return only the integers zero (FALSE) and one (TRUE). The *not* operator **!** returns one (1) if its operand is zero and returns zero (0) if its operand has any nonzero value.

Operand Types

The operators in Table 2-2 use the operand type information maintained by the assembler. To aid the linker in combining object files, the assembler associates the value of every expression with a type. There are both *primary types* and *special types*. The primary types deal with the operand characteristics defined by the assembly language. The primary types are:

absolute — An absolute expression is one whose value is based on a constant or on the difference between two relocatable expressions of the same subtype (as defined below). The values of absolute expressions are never affected by the linker.

relocatable — The value of an expression is relocatable if it is based on a label (but not on the absolute difference of two labels) or upon an undefined external symbol. Relocatable expressions are further classified by the following subtypes:

- *text* — Value is relative to the *text* section.
- *data* — Value is relative to the *data* section.
- *bss* — Value is relative to the *bss* section.
- *undefined* — Based on a symbol that is not defined except for its appearance in a **.global**, **.extern**, or **.comm** directive.

The special types deal with operand characteristics of the machine instructions. These types are explained in the section "32-Bit Relocatable Expressions" on page 2-10.

32-Bit Constant Expressions

The assembly language supports 32-bit constant expressions; however, instructions for the i860 microprocessor do not directly accept 32-bit immediate constants. The assembler provides three methods for converting a 32-bit constant into a 16-bit constant:

1. Selection operators that allow the programmer to specify either the high- or low-order half of a 32-bit constant.

2. Automatic expansion of an assembler pseudo-instruction into a multiple-instruction sequence, one or more instructions of which handle each half of the 32-bit constant.

3. Automatic conversion of 32-bit constants to 16 bits. See "Automatic Conversion of 32-bit Constants" on page 2-13 for more information.

The operators **l%** and **h%** (refer to Table 2-2), select the low- or high-order half respectively of a 32-bit constant expression. The following example illustrates their use.

Example

```
            LongMask = 0xFF00C7F3
// Case 1
            or l%LongMask,      r0,  r4
            orh h%LongMask,     r4,  r4
// Case 2
            or LongMask,        r0,  r4
```

The first case reconstructs the 32-bit constant in a register, by loading 16 bits at a time. In the second case, the assembler automatically expands the given instruction into a similar two-instruction sequence. Note that instruction expansion causes undesirable effects after a delayed branch instruction or within dual-instruction mode. If you use the **-x** assembler option, the assembler detects these situations and indicates an error.

32-Bit Relocatable Expressions

Relocatable expressions are adjusted by the linker using 32-bit arithmetic. However, the i860 microprocessor has no instructions that directly accept 32-bit address constants. To accommodate this situation, the assembler and linker recognize an additional set of special relocatable types. With these types, the assembler instructs the linker to relocate 32-bit addresses 16 bits at a time.

The assembler provides three methods for converting from the primary types to the special types:

1. Selection operators that allow the programmer to control type conversion.

2. Automatic type conversion of a pseudo-instruction that generates multiple-instruction sequences.

3. Automatic conversion of 32-bit constants to 16 bits. See "Automatic Conversion of 32-bit Constants" on page 2-13 for more information.

The operators **l%**, **h%**, and **ha%** (refer to Table 2-2), in addition to selecting the high- or low-order half of a relocatable 32-bit expression, convert a primary relocatable type to a special relocatable type.

l% Selects the low-order 16 bits of an expression.

h% Selects the high-order 16 bits of an expression. Does not perform any adjustment; therefore, is suitable for combination with a subsequent **or** instruction.

ha% Selects the high-order 16 bits of an expression, and, if bit 15 of the expression is set, performs the necessary adjustment. This is suitable for combination with a subsequent register/offset instruction (**ld.l**, for example).

The following examples illustrate the need for adjustment.

Example

```
            .align    .float
ST:         .float    0f1.659463
// Case 1
            or        l%ST,           r0,   r5
            orh       h%ST,           r5,   r5
            ld.l      0(r5),          r6
// Case 2
            orh       ha%ST, r0,      r5
            ld.l      l%ST(r5),       r6
// Case 3
            ld.l      ST,             r6
```

The first case forms the complete address in **r5**, then loads the data item. The immediate value placed into the **orh** instruction by the linker is precisely the upper 16 bits of the address after relocation.

The second case first extracts the high-order part of the address, then loads the data item by combining the low-order bits of the address using the immediate offset form of the load instruction. However, the processor sign-extends the immediate offset of this instruction. If bit 15 of the address is set after relocation, the effect is that of a *negative* offset. The **ha%** operator identifies this potential condition to the linker. When bit 15 is set, the linker adjusts the value of the high-order 16 bits so that the correct result is produced even with the "negative" offset in the load instruction.

In the third case, the assembler automatically expands the given pseudo-instruction into a similar multiple-instruction sequence using r31 as the temporary address register. (The addressing temporary register can be changed by the **.atmp** directive, as described in Chapter 4.) Note that pseudo-instruction expansion causes undesirable effects after a delayed branch instruction or within dual-instruction mode. If the assembler option –x is selected, the assembler detects these situations and indicates an error.

Type Combinations

When a complex expression is formed with one of the operators in Table 2-3 on page 12, the type of the resulting expression depends on the types of the original expressions and upon the operator, as defined by Table 2-3. All other type combinations are invalid. For example:

Example

```
      .data; .align 4
Array:       .short[8]0
             .long[10]0
             .byte "Miscellaneous message"
             .align 4
End_Array:
      .text
ld.l Array+4, r4           //Relocatable+constant=relocatable
or End_Array-Array, r0, r5 //Relocatable-relocatable=constant
```

Table 2-3. Type Combination

Type of Operand 1	Type of Operator	Type of Operand 2	Result
Absolute	any	Absolute	Absolute
Relocatable	+	Absolute	Relocatable
Absolute	+	Relocatable	Relocatable
Relocatable	-	Absolute	Relocatable
Relocatable	-	Relocatable	Absolute

Automatic Conversion of 32-bit Constants

The assembler automatically converts 32-bit immediate constants to 16 bits whenever possible. When conversion is not possible, the assembler generates pseudo-instructions to accommodate the 32-bit operand. Note in the following examples that the assembler issues a warning to indicate when pseudo-instructions have been substituted.

```
AS860 ASSEMBLER, Vx.y
  8488ffff 00000000       1 addu  0xffff,r4,     r8  // Case 1 0xFFFF
  8488ffff 00000004       2 addu  0xffffffff,r4, r8  // Case 2 0xFFFFFFFF
  8488ffff 00000004       3 addu  -1,r4,         r8  // Case 3 0xFFFFFFFF
  8488ffff 0000000c       4 addu  l%-1,r4,       r8  // Case 4 0x0000FFFF
  8488ffff 00000010       5 addu  0x0000ffff,r4, r8  // Case 5 0x0000FFFF
**line 6: Warning:Constant not representable in 16 bits, pseudo-inst generated
e7ff0000ec1f0001 00000014 6 addu  0x00010000,r4, r8  // Case 6 0x00010000
        8088f800 0000001c
```

Operator Precedence

In the absence of overriding parentheses, binary operators are evaluated according to the following precedence groups. Group one is the group with highest precedence (the first to be evaluated).

1. *, /
2. +, -
3. <, >, =
4. <<, >>, &, |, ^

Unary operators have precedence over binary operators, except for the unary operators **h%**, **l%**, and **ha%**, which have lower precedence. For example, `l%main+0x4` gives the lower 16 bits of `main+0x4`.

Instruction Syntax 3

The instructions of the assembly language correspond one-to-one with the machine instructions of the i860 microprocessor (except for the "pseudoinstructions" presented in the section "Pseudoinstructions" on page 3-16). The general syntax of an instruction is:

mnemonic source_operand_1, source_operand_2, destination

NOTE

The assembler accepts only lowercase machine instructions and register names. For example, this instruction is acceptable:

```
addu r1, r2, r3
```

but these are not:

```
Addu r1, r2, r3
ADDU r1, r2, r3
addu R1, r2, r3
```

Mnemonics for machine instructions are defined in lowercase only.

Not all instructions have two source operands. In all cases, the actual destination appears to the right of the source.

This chapter presents only the syntax for specifying machine instructions. For details regarding instruction semantics, format, and encoding, refer to the *i860™ Family Microprocessor Programmer's Reference Manual.*

Key to Abbreviations

For register operands, the abbreviations that describe the operands are composed of two parts. The first part describes the type of register.

c — One of the predefined names of control registers: **fir**, **psr**, **epsr**, **dirbase**, **db**, **fsr**, **bear**, **ccr**, **p0**, **p1**, **p2**, or **p3**.

f — One of the floating-point registers: **f0** through **f31**

i — One of the integer registers: **r0** through **r31**

The second part identifies the field of the machine instruction into which the operand is to be placed:

src1 — The first of the two source-register designators, which may be either a register or a 16-bit immediate constant or address offset. The immediate value is zero-extended for logical operations and is sign extended for add and subtract operations (including **addu** and **subu**) and for all addressing calculations.

src1ni — Same as *src1* except that no immediate constant or address offset value is permitted.

src1s — Same as *src1* except that the immediate constant is a 5-bit value that is zero-extended to 32 bits.

src2 — The second of the two source-register designators.

dest — The destination register designator.

Thus, the operand specifier *isrc2*, for example, means that an integer register is used and that the encoding of that register must be placed in the *src2* field of the generated machine instruction.

Other (nonregister) operands are specified by a one-part identifier that represents both the type of operand required and the instruction field into which the value of the operand is placed:

const32 — A 16-bit immediate constant or address offset that the i860 microprocessor sign-extends to 32 bits when computing the effective address.

lbroff — A signed, 26-bit, immediate, relative branch offset.

sbroff — A signed, 16-bit, immediate, relative branch offset.

brx — A function that computes the target address by shifting the offset (either *lbroff* or *sbroff*) left by two bits, sign-extending it to 32 bits, and adding the result to the current instruction pointer plus four. The resulting target address may lie anywhere within the address space.

Other abbreviations include:

.p Precision specification **.ss**, **.sd**, or **.dd** (**.ds** not permitted). Refer to Table 3-1.

.r Precision specification **.ss**, **.sd**, **.ds**, or **.dd**. Refer to Table 3-1.

.v **.sd** or **.dd**

.w **.ss** or **.dd**

.x **.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)

.y **.l** (32 bits, **.d** (64 bits), or **.q** (128 bits)

mem.x(address) The contents of the memory location indicated by *address* with a size of *x*.

port.x(address) The I/O port indication by *address* with a size of *x*.

int_vector.x(address)
The interrupt vector with a size of *x* returned from I/O port *address*.

PM The pixel mask, which is considered as an array of eight bits (PM(7)..PM(0), where PM(0) is the least-significant bit.

Table 3-1. Precision Specification

Suffix	Source Precision	Result Precision
.ss	single	single
.sd	single	double
.dd	double	double
.ds	double	single

NOTE: Unless otherwise specified, floating-point operations accept single- or double-precision source operands and produce a result of equal or greater precision. Both input operands must have the same precision. The source and result precision are specified by a two-letter suffix to the mnemonic of the operation.

Instruction Definitions in Alphabetical Order

adds *isrc1, isrc, idest* ..... **Add Signed**

idest ← *isrc1* + *isrc2*
OF ← (bit 31 carry ≠ bit 30 carry)
CC set if *isrc1* + *isrc2* < 0 (signed)
CC clear if *isrc2* + *isrc1* ≥ 0 (signed)

addu *isrc1, isrc2, idest* ..... **Add Unsigned**

idest ← *isrc1* + *isrc2*
OF ← bit 31 carry
CC ← bit 31 carry

and *isrc1, isrc2, idest* ..... **Logical AND**

idest ← *isrc1* and *isrc2*
CC set if result is zero, cleared otherwise

andh *#const, isrc2, idest* ..... **Logical AND High**

idest ← (*#const* shifted left 16 bits) and *isrc2*
CC set if result is zero, cleared otherwise

andnot *isrc1, isrc2, idest* ..... **Logical AND NOT**

idest ← (not *isrc1*) and *isrc2*
CC set if result is zero, cleared otherwise

andnoth *#const, isrc2, idest* ..... **Logical AND NOT High**

idest ← (not (*#const* shifted left 16 bits)) and *isrc2*
CC set if result is zero, cleared otherwise

bc *lbroff* ..... **Branch on CC**

IF CC = 1
THEN continue execution at *brx*(*lbroff*)
FI

bc.t *lbroff* ..... **Branch on CC, Taken**

IF CC = 1
THEN execute one more sequential instruction
continue execution at *brx*(*lbroff*)
ELSE skip next sequential instruction
FI

bla *isrc1ni, isrc2, sbroff* ..... **Branch on LCC and Add**

LCC-temp clear if $isrc2 + isrc1ni < 0$ (signed)
LCC-temp set if $isrc2 + isrc1ni \geq 0$ (signed)
$isrc2 \leftarrow isrc1ni + isrc2$
Execute one more sequential instruction
IF LCC
THEN LCC ← LCC-temp
continue execution at *brx(sbroff)*
ELSE LCC ← LCC-temp
FI

bnc *lbroff* ..... **Branch on Not CC**

IF CC = 0
THEN continue execution at brx(lbroff)
FI

bnc.t *lbroff* ..... **Branch on Not CC, Taken**

IF CC = 0
THEN execute one more sequential instruction
continue execution at brx(lbroff)
ELSE skip next sequential instruction
FI

br *lbroff* ..... **Branch Direct Unconditionally**

Execute one more sequential instruction
Continue execution at *brx(lbroff)*

bri [*iscr1ni*] ..... **Branch Indirect Unconditionally**

```
Execute one more sequential instruction
IF      any trap bit in psr is set
THEN    copy PU to U, PIM to IM in psr
        clear trap bits
        IF      DS is set and DIM is reset
                THEN    enter dual-instruction mode after executing one
                            instruction in single-instruction mode
                ELSE    IF      DS is set and DIM is set
                        THEN    enter single instruction mode after executing one
                                    instruction in single-instruction mode
                        ELSE    IF      DIM is set
                                THEN    enter dual-instruction mode
                                            for next instruction pair
                                ELSE    enter single-instruction mode
                                            for next instruction pair
                                FI
                        FI
                FI
        FI
FI
Continue execution at address in isrc1ni
   (The original contents of isrc1ni is used even if the next instruction
    modifies isrc1ni. Does not trap if isrc1ni is misaligned.)
```

bte *isrc1s, isrc2, sbroff* ..... **Branch If Equal**

```
IF      isrc1s = isrc2
THEN    continue execution at brx(sbroff)
FI
```

btne *isrc1s, isrc2, sbroff* ..... **Branch If Not Equal**

```
IF      isrc1s ≠ isrc2
THEN    continue execution at brx(sbroff)
FI
```

call *lbroff* ..... **Subroutine Call**

```
r1 ← address  of next sequential instruction + 4 (or + 8 in dual mode)
Execute one more sequential instruction
Continue execution at brx(lbroff)
```

calli (*isrc1ni*) ..... **Indirect Subroutine Call**

```
r1 ← address  of next sequential instruction + 4 (or + 8 in dual mode)
Execute one more sequential instruction
Continue execution at brx(lbroff)
   (The original contents of isrc1ni is used even if the next instruction
    modifies isrc1ni. Does not trap if isrc1ni is misaligned. The register
    isrc1ni must not be r1.)
```

fadd.p *fsrc1, fsrc2, fdest* ..... **Floating-Point Add**
fdest ← *fsrc1* + *fsrc2*

faddp *fsrc1, fsrc2, fdest* ..... **Add with Pixel Merge**
fdest ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift MERGE right 16 and load fields 31..16 and 63..48 from *fsrc1* + *fsrc2*

Table 3-2. FADDP MERGE Update

Pixel Size (from PS)	Fields Load from Result into MERGE				Right Shift Amount (Field Size)
8	63..56,	47..40,	31..24,	15..8	8
16	63..58,	47..42,	31..26,	15..10	6
32	63..56,		31..24		8

famov.a *fsrc1, fdest* ..... **Floating-Point Adder Move**
fdest ← *fsrc1*

fiadd.w *fsrc1, fsrc2, fdest* ..... **Long-Integer Add**
fdest ← *fsrc1* + *fsrc2* (2's complement integer arithmetic)

fisub.w *fsrc1, fsrc2, fdest* ..... **Long-Integer Subtract**
fdest ← *fsrc1* − *fsrc2* (2's complement integer arithmetic)

fix.v *fsrc1, fdest* ..... **Floating-Point to Integer Conversion**
fdest ← 64-bit value with low-order 32 bits equal to integer part of *fsrc1* rounded

Floating-Point Load
fld.y *isrc1(isrc2), fdest* ..... **(Normal)**
fld.y *isrc1(isrc2)++, fdest* ..... **(Autoincrement)**
fdest ← mem.y (*isrc1* + *isrc2*)
IF autoincrement
THEN *isrc2* ← *isrc1* + *isrc2*
FI

Cache Flush

flush *#const(isrc2)* ..... **(Normal)**
fld.y *#const(isrc2)++* ..... **(Autoincrement)**

Write back (if modified) the line in data cache that has address (*#const* + *isrc2*)
80860XR: and set tag value to (*#const* + *isrc2*).
80860XP: and invalidate its virtual and physical tags. The Paragon XP/S system uses the 80860XP.
Contents of line undefined.
IF autoincrement
THEN $isrc2 \leftarrow \#const + isrc2$
FI

fmlow.dd *fsrc1, fsrc2, fdest* ..... **Floating-Point Multiply Low**

$fdest \leftarrow$ low-order 53 bits of ($fsrc1$ mantissa $\times fsrc2$ mantissa)
$fdest$ bit 53 $\leftarrow$ most significant bit of ($fsrc1$ mantissa $\times fsrc2$ mantissa)

fmov.r *fsrc1, fdest* ..... **Floating-Point Reg-Reg Low**

Assembler pseudo-operation

fmov.ss *fsrc1, fdest*	= **fiadd.ss** *fsrc1*, **f0**, *fdest*
fmov.dd *fsrc1, fdest*	= **fiadd.dd** *fsrc1*, **f0**, *fdest*
fmov.sd *fsrc1, fdest*	= **fiadd.sd** *fsrc1, fdest*
fmov.ds *fsrc1, fdest*	= **fiadd.ds** *fsrc1, fdest*

fmul.p *fsrc1, fsrc2, fdest* ..... **Floating-Point Multiply**

$fdest \leftarrow fsrc1 \times fsrc2$

fnop ..... **Floating-Point No Operation**

Assembler pseudo-operation
fnop = shrd r0, r0, r0

form *fsrc1, fdest* ..... **OR with MERGE Register**

$fdest \leftarrow fsrc1$ OR MERGE
MERGE $\leftarrow 0$

frcp.p *fsrc2, fdest* ..... **Floating-Point Reciprocal**

$fdest \leftarrow 1 / fsrc2$ with maximum mantissa error $< 2^{-7}$

fsqr.p *fsrc2, fdest* ..... **Floating-Point Reciprocal Square Root**

$fdest \leftarrow 1 / \sqrt{(fsrc2)}$ with maximum mantissa error $< 2^{-7}$

Floating-Point Store

fst.y *fdest, isrc1(isrc2)* ..... **(Normal)**

fst.y *fdest, isrc1(isrc2)++* ..... **(Autoincrement)**

mem.y $(isrc2 + isrc1) \leftarrow fdest$
IF autoincrement
THEN $isrc2 \leftarrow isrc1 + isrc2$
FI

fsub.p *fsrc1, fsrc2, fdest* ..... **Floating-Point Subtract**

$fdest \leftarrow fsrc1 - fsrc2$

ftrunc.v *fsrc1, fdest* ..... **Floating-Point to Integer Conversion**

fdest ← 64-bit value with low-order 32 bits equal to integer part of *fsrc1*

fxfr *fsrc1, idest* ..... **Transfer F-P to Integer Register**

$idest \leftarrow fsrc1$

fzchkl *fsrc1, fsrc2, fdest* ..... **32-Bit Z-Buffer Check**

Consider the 64-bit operands as arrays of two 32-bit fields
fsrc1(1)..*fsrc1*(0), *fsrc2*(1)..*fsrc2*(0), and *fdest*(1)..*fdest*(0)
where zero denotes the least-significant field.
PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
$PM[i + 6] \leftarrow fsrc2\,(i) \leq fsrc1\,(i)$ (unsigned)
fdest (i) ← smaller of *fsrc2*(i) and *fsrc1*(i)
OD
MERGE ← 0

fzchks *fsrc1, fsrc2, fdest* ..... **16-Bit Z-Buffer Check**

Consider the 64-bit operands as arrays of four 16-bit fields
fsrc1(3)..*fsrc1*(0), *fsrc2*(3)..*fsrc2*(0), and *fdest*(3)..*fdest*(0)
where zero denotes the least-significant field.
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
$PM[i + (4)] \leftarrow fsrc2\,(i) \leq fsrc1\,(i)$ (unsigned)
fdest (i) ← smaller of *fsrc2*(i) and *fsrc1*(i)
OD
MERGE ← 0

intovr ..... **Software Trap on Integer Overflow**

IF OF = 1
THEN generate trap with IT set in **psr**
FI

ixfr *isrc1ni, fdest* ..... **Transfer Integer to F-P Register**
fdest ← *isrc1ni*

ld.c *csrc2, idest* ..... **Load from Control Register**
idest ← *csrc2*

ld.x *isrc1(isrc2), idest* ..... **Load Integer**
idest ← *mem.x* (*isrc1* + *isrc2*)

ldint.x *isrc2, idest* ..... **Load Interrupt Vector**
idest ← *int_vector.x* (*isrc2*)
NOTE: Not available with the i860 XR CPU. Available on a Paragon XP/S system.

ldio.x *isrc2, idest* ..... **Load I/O**
idest ← *port.x* (*isrc2*)
NOTE: Not available with the i860 XR CPU. Available on a Paragon XP/S system.

lock ..... **Begin Interlocked Sequence**
Set BL in **dirbase**
The next data load or store that appears on the bus locks that location
Disable interrupts until the bus is unlocked.

mov *isrc2, idest* ..... **Register-Register Move**
Assembler pseudo-operation
mov *isrc2, idest* = **shl r0**, *isrc2, idest*

mov *const32, idest* ..... **Constant-to-Register Move**
Assembler pseudo-operation
adds *l%const32*,**r0**, *idest*
... when 0xFFFF8000 *const32* < 0x8000

orh *h%const32*, **r0**, *idest*
or *l%const32*, *idest*, *idest*
... otherwise

nop ..... **Core-Unit No Operation**
Assembler pseudo-operation
nop = shl r0, r0, r0

or *isrc1, isrc2, idest* ..... **Logical OR**
idest ← *isrc1* OR *isrc2*
CC set if result is zero, cleared otherwise

orh *#const, isrc2, idest* ..... **Logical OR high**
idest ← (*#const* shifted left 16 bits) OR *isrc2*
CC set if result is zero, cleared otherwise

pfadd.p *fsrc1, fsrc2, fdest* ..... **Pipelined Floating-Point Add**

fdest ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← *fsrc1* + *fsrc2*

pfaddp *fsrc1, fsrc2, fdest* ..... **Pipelined Add with Pixel Merge**

fdest ← last stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2*
(using integer arithmetic; 8-byte operands and destination)
Shift and load MERGE register from *fsrc1* + *fsrc2* as defined in Table 3-2 on page 3-7

pfaddz *fsrc1, fsrc2, fdest* ..... **Pipelined Add with Z Merge**

frdest ← last stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2*
(using integer arithmetic; 8-byte operands and destination)
Shift MERGE right 16 and load fields 31..16 and 63..48 from *fsrc1* + *fsrc2*

pfam.p *fsrc1, fsrc2, fdest* ..... **Pipelined Floating-Point Add and Multiply**

fdest ← last stage adder result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 + A-op2
M pipeline first stage ← M-op1 + M-op2

pfamov.r *fsrc1, fdest* ..... **Pipelined Floating-Point Adder Move**

fdest ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← *fsrc1*

pfeq.p *fsrc1, fscr2, fdest* ..... **Pipelined Floating-Point Equal Compare**

fdest ← last stage adder result
CC set if *fsrc1* = *fsrc2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfgt.p *fsrc1, fscr2, fdest* ..... **Pipelined Floating-Point Greater-Than Compare**

(Assembler clears R-bit of instruction)
fdest ← last stage adder result
CC set if *fsrc1* > *fsrc2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfiadd.w *fsrc1, fsrc2, fdest* ..... **Pipelined Long-Integer Add**

fdest ← last stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2*
(2's complement integer arithmetic)

pfisub.w *fsrc1, fsrc2, fdest* ..... **Pipelined Long-Integer Subtract**

fdest ← last stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* – *fsrc2*
(2's complement integer arithmetic)

pfix.v *fsrc1, fdest* ..... **Pipelined Floating-Point to Integer Conversion**

fdest ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← 64-bit value with low-order 32 bits
equal to integer part of *fsrc1* rounded

Pipelined Floating-Point Load

pfld.y *isrc1(isrc2), fdest* ..... **(Normal)**
pfld.y *isrc1(isrc2)++, fdest* ..... **(Autoincrement)**

fdest ← mem.y (third previous **pfld**'s (*isrc1* + *isrc2*))
(where .y is precision of third previous **pfld.y**)
IF autoincrement
THEN *isrc2* ← *isrc1* + *isrc2*
FI
NOTE: pfld.q is not available with the i860 XR CPU. Available on a Paragon XP/S system.

pfle.p *fsrc1, fsrc2, fdest* ..... **Pipelined F-P Less-Than or Equal Compare**

Assembler pseudo-operation, identical to **pfgt.p** except that
assembler sets R-bit of instruction
fdest ← last stage adder result
CC clear if *fsrc1* *fsrc2*, else set
Advance A pipeline one state
A pipeline first stage is undefined, but no result exception occurs

pfmam.p *fsrc1, fsrc2, fdest* ..... **Pipelined Floating-Point Add and Multiply**

fdest ← last stage multiplier result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 + A-op2
M pipeline first stage ← M-op1 × M-op2

pmov.r *fsrc1, fdest* ..... **Pipelined Floating-Point Reg-Reg Move**

Assembler pseudo-operation

pfmov.ss *fsrc1, fdest*	= **pfiadd.ss** *fsrc1,* **f0**, *fdest*
pfmov.dd *fsrc1, fdest*	= **pfiadd.dd** *fsrc1,* **f0**, *fdest*
pfmov.sd *fsrc1, fdest*	= **pfiadd.sd** *fsrc1, fdest*
pfmov.ds *fsrc1, fdest*	= **pfiadd.ds** *fsrc1, fdest*

pfmsm.p *fsrc1, fdest* .....**Pipelined Floating-Point Subtract and Multiply**

fdest ← last stage multiplier result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 – A-op2
M pipeline first stage ← M-op1 × M-op2

pfmul.p *fsrc1, fsrc2, fdest* .....**Pipelined Floating-Point Multiply**

fdest ← last stage multiplier result
Advance M pipeline one stage
M pipeline first stage ← *fsrc1* × *fsrc2*

pfmul3.dd *fsrc1, fsrc2, fdest* .....**Three-Stage Pipelined Multiply**

fdest ← last stage multiplier result
Advance 3-Stage M pipeline one stage
M pipeline first stage ← *fsrc1* × *fsrc2*

pform *fsrc1, fdest*.....**Pipelined OR to MERGE Register**

fdest ← last stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* OR MERGE
MERGE ← 0

pfsm.p *fsrc1, fsrc2, fdest* .....**Pipelined Floating-Point Subtract and Multiply**

fdest ← last stage multiplier result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 – A-op2
M pipeline first stage ← M-op1 × M-op2

pfsub.p *fsrc1, fsrc2, fdest* .....**Pipelined Floating-Point Subtract**

fdest ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← *fsrc1* – *fsrc2*

pftrunc.v *fsrc1, fdest* .....**Pipelined Floating-Point to Integer Conversion**

fdest ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← 64-bit with low-order 32 bits equal to integer part of *fsrc1*

pfzchkl *fsrc1, fsrc2, fdest* ..... **Pipelined 32-Bit Z-Buffer Check**

Consider the 64-bit operands as arrays of two 32-bit fields
fsrc1(1)..*fsrc1*(0), *fsrc2*(1)..*fsrc2*(0), and *fdest*(1)..*fdest*(0)
where zero denotes the least-significant field.

PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO

PM[i + 6] ← *fsrc2* (i) ≤ *fsrc1* (i) (unsigned)

fdest (i) ← last-stage graphics-unit result

last-stage graphics-unit result ← smaller of *fsrc2*(i) and *fsrc1*(i)

OD
MERGE ← 0

pfzchks *fsrc1, fsrc2, fdest* ..... **Pipelined 16-Bit Z-Buffer Check**

Consider the 64-bit operands as arrays of four 16-bit fields
fsrc1(3)..*fsrc1*(0), *fsrc2*(3)..*fsrc2*(0), and *fdest*(3)..*fdest*(0)
where zero denotes the least-significant field.

PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO

PM[i + 4] ← *fsrc2* (i) ≤ *fsrc1* (i) (unsigned)

fdest ← last-stage graphics-unit result

last-stage graphics-unit result(i) ← smaller of *fsrc2*(i) and *fsrc1*(i)

OD
MERGE ← 0

pst.d *#const(isrc2)* ..... **Pixel Store**

pst.d *#const(isrc2)++* ..... **Pixel Store Autoincrement**

Pixels enabled by PM in *mem.d* (*isrc2* + *#const*) ← *fdest*
Shift PM right by 8/pixel size (in bytes) bits
IF autoincrement
THEN *isrc2* ← *#const* + *isrc2*
FI

scyc.x *isrc2* ..... **Special Cycles**

Generate a special bus cycle (D/C#=0, W/R#=1, M/IO#=0) and set BE7#-BE0 according to the value contained in the register *isrc2*

NOTE: Not available with the i860 XR CPU. Available on a Paragon XP/S system.

shl *isrc1, isrc2, idest* ..... **Shift Left**

idest ← *isrc2* shifted left by *isrc1* bits

shr *isrc1, isrc2, idest* ..... **Shift Right**

SC (in **psr**) ← *isrc1*
idest ← *isrc2* shifted right by *isrc1* bits

shra *isrc1, isrc2, idest* ..... **Shift Right Arithmetic**

idest ← *isrc2* arithmetically shifted right by *isrc1* bits

shrd *isrc1ni, isrc2, idest* ..... **Shift Right Double**

idest ← low-order 32-bits of *isrcni:isrc2* shifted right by SC bits

st.c *isrc1ni, csrc2* ..... **Store to Control Register**

csrc2 ← *srclni*

st.x *isrc1ni, #const(isrc2)* ..... **Store Integer**

mem.x (*isrc2* + *#const*) ← *isrclni*

stio.x *isrc1ni, isrc2* ..... **Store I/O**

port.x (*isrc2*) ← *isrclni*

NOTE: Not available with the i860 XR CPU. Available on a Paragon XP/S system.

subs *isrc1, isrc2, idest* ..... **Subtract Signed**

idest ← *isrc1* – *isrc2*
OF ← (bit 31 carry ≠ bit 30 carry)
CC set if *isrc2* > *isrc1* (signed)
CC clear if *isrc2* *isrc1* (signed)

trap *isrc1ni, isrc2, idest* ..... **Software Trap**

Generate trap with IT set in **psr**

unlock ..... **End Interlocked Sequence**

Clear BL in **dirbase**. The next load or store
unlocks the bus. Interrupts are enabled.

xor *isrc1, isrc2, idest* ..... **Logical Exclusive OR**

idest ← *isrc1* XOR *isrc2*
CC is set if result is zero, cleared otherwise

xorh *#const, isrc2, idest* ..... **Logical Exclusive OR High**

idest ← (*#const* shifted left 16 bits) XOR *isrc2*
CC is set if result is zero, cleared otherwise

Dual-instruction Mode

One of the ways to indicate dual-instruction mode is with the **d.** prefix before the mnemonic of an instruction of the floating-point unit. The **d.** prefix sets the dual-mode bit in that instruction. For example:

d.fadd.ss f4, f5, f6

The other way is with the **.dual**... **.enddual** directives (refer to Chapter 4). It may be necessary to use the **d.** prefix to create the preamble before a **.dual**... **.enddual** block.

Pseudoinstructions

A pseudoinstruction is an assembly-language instruction that does not correspond directly to a machine instruction. Some pseudoinstructions are aliases for instructions that could be specified with a different, but longer and less mnemonic, syntax. Others are like macro instructions; they are expanded into a two-or three-instruction sequence.

Integer Register to Register Move

mov *isrc2, idest*

This pseudoinstruction is implemented as:

shl *r0, isrc2, idest*

Integer Constant to Register Move

mov *const32, idest*

If the value of *const32* is 0xFFFF8000≤const32<0x8000, then this pseudoinstruction is implemented as:

adds *l%const32, r0, idest*

otherwise it is implemented as:

orh h%*const32, r0, idest*
or l%*const32, idest, idest*

Floating-point Register to Register Moves

$$\left\{\begin{array}{l}\text{fmov}\\ \text{pfmov}\end{array}\right\}\left\{\begin{array}{l}\text{.ss}\\ \text{.dd}\end{array}\right\}\ \textit{fsrc1,fdest}$$

These pseudoinstructions are implemented as:

$$\left\{\begin{array}{l}\text{fiadd}\\ \text{pfiadd}\end{array}\right\}\left\{\begin{array}{l}\text{.ss}\\ \text{.dd}\end{array}\right\}\ \textit{fsrc1, f0, fdest}$$

$$\left\{\begin{array}{l}\text{fmov}\\ \text{pfmov}\end{array}\right\}\left\{\begin{array}{l}\text{.sd}\\ \text{.ds}\end{array}\right\}\ \textit{fsrc1,fdest}$$

These pseudoinstructions are implemented as:

$$\left\{\begin{array}{l}\text{famov}\\ \text{pfamov}\end{array}\right\}\left\{\begin{array}{l}\text{.sd}\\ \text{.ds}\end{array}\right\}\ \textit{fsrc1,fdest}$$

No Operation

nop

The core no-op pseudoinstruction is implemented as:

shl r0, r0, r0

The floating-point no-op pseudoinstruction is implemented as:

shrd r0, r0, r0

fnop can be prefixed with "**d.**", but cannot be used to enter or exit DIM.

32-bit Address Expression

ld { .b, .s, .l } *addr_expr, idest*

st { .b, .s, .l } *idest, addr_expr*

fld { .l, .d, .q } *addr_expr, idest*

pfld { .l, .d } *addr_expr, idest*

fst { .l, .k, .q } *fdest, addr_expr*

pst.d *fdest, addr_expr*

flush *addr_expr*

When the memory reference of any of these instructions is a label or other relocatable expression, the instruction is expanded into a two-instruction sequence. The following example serves to illustrate the remaining cases as well:

```
orh     ha%addr_expr, r0, r31
ld.l    l%addr_expr(r31), idest
```

Unsigned 32-bit Constant

and *const32, isrc2, idest*

andnot *const32, isrc2, idest*

or *const32, isrc2, idest*

xor *const32, isrc2, idest*

When *const32* cannot be represented in 16 bits, these become pseudoinstructions, which are expanded into a two-instruction sequence. The following example illustrates the expansion of **or**; expansion of **andnot** and **xor** are similar.

```
orh      h%const32, isrc2, r31
or       %const32, r31, idest
```

The expansion of **and** complements *const32*, then uses the **andnot** instruction:

```
andnoth  h%(-1-const32), isrc2, r31
andnot   l%(-1-const32), r31, idest
```

Signed 32-bit Constant

```
adds     const32, isrc2, idest
addu     const32, isrc2, idest
subs     const32, isrc2, idest
subu     const32, isrc2, idest
```

When the value of *const32* cannot be represented in 16 bits, these become pseudoinstructions, which are expanded into a three-instruction sequence; for example:

```
orh      h%const32, r0, r31
or       l%const32, r31, r31
adds     r31, isrc2, idest
```

Assembler Directives 4

Assembler directives do not directly generate instruction codes; rather, they control operation of the assembler, define and initialize data, or change the way instructions are generated. Assembler directives are defined in lowercase only.

The following keywords for data formats are used in this section, both as directives and as parameters:

.byte	8 bits
.short	16 bits
.long	32 bits
.quad	128 bits
.float	single-precision floating point (32 bits)
.double	double-precision floating point (64 bits)

When a directive calls for a list of parameters, commas are used to separate the parameters.

Alignment

.align *type*

.align *exp1* [,*exp2*]

The **.align** directive causes the assembler to advance the location counter to the boundary specified by the first parameter. The *type* may be any of the following:

.short
.long
.float
.double
.quad

The parameter *exp1* is a constant integer expression that specifies the alignment boundary in number of bytes. As the location counter is advanced, the section is normally filled with **nop** instructions in the text section and with NULs (binary zeros) in the data section. When the constant integer expression *exp2* is supplied, its value is used as the byte pattern for filling. Any symbols used in the expressions must be previously defined.

The value of the *exp1* parameter must be a power of two less than or equal to 32. If the value is not a power of two, the assembler produces an error message. When the value of *exp1* is a power of two greater than 32, the assembler produces a warning message.

Dual Mode

.dual ... .endual

The assembler offers two different methods for generating dual-mode instruction sequences. The first method, the **d.** prefix to the instruction mnemonics has already been presented in Chapter 3. The second method is to enclose normal core and F-unit instruction mnemonics within the **.dual** and **.enddual** directives. After a **.dual** directive, the assembler aligns the instruction stream to a 64-bit boundary. For all instructions until the corresponding **.enddual**, the assembler sets the dual-mode bit in F-unit instructions.

Proper generation of the preamble (for entering dual-instruction mode) and the postamble (for exiting) is the responsibility of the programmer. In some cases it is necessary to use the **d.** prefix to create the preamble before a **.dual....enddual** block.

When the **-x** command-line option is set, the assembler enforces the dual-instruction mode rules defined in the *i860™ Microprocessor Family Programmer's Reference Manual*. If the assembler encounters any kind of error, then to avoid additional misleading error messages, the alignment checking is disabled until the end of dual-instruction mode. These checks are performed whether the dual-instruction mode is generated by the **d.** prefix or by a **.dual....enddual** block.

In Example 3-1, note that dual mode does not begin until three instructions after the **.dual** directive and does not end until three instructions after the **.enddual** directive.

Example

```
// SINGLE-PRECISION VECTOR SUM
//      input:     r16 - vector address
//                 r17 - vector size (must be > 5)
//      output:    f16 - sum of vector elements
        fld.d      r0(r16),  f20            // Load first two elements
        mov        -2,       r21            // Loop decrement for bla
        .dual                               // Enter dual-instruction mode
        pfadd.ss   f0,       f0,      f0    // Clear adder pipe (1)
        adds       -6,       r17,     r17   // Decrement size by 6
        pfadd.ss   f0,       f0,      f0    // Clear adder pipe (2)
        bla        r21,      r17,     L1    // Initialize LCC
        pfadd.ss   f0,       f0,      f0    // Clear adder pipe (3)
        fld.d      8(r16)++, f22            // Load 3rd and 4th elements
L1::    pfadd.ss   f20,      f30,     f30   // Add f20 to pipeline
        bla        r21,      r17,     L2    // If more, go to L2 after
        pfadd.ss   f21,      f31,     f31   // adding f21 to pipeline and
        fld.d      8(r16)++, f20            // loading next f20:f21
       // If we reach this point, at least one element remains to be loaded.
        // r17 is either -4 or -3.  f20, f21, f22, and f23 still contain
        // vector elements.  Add f20 and f22 to the pipeline, too.
```

```
        pfadd.ss   f20,       f30,      f30
        br         sumup                      // Exit loop after adding
        pfadd.ss   f21,       f31,      f31   // f21 to the pipeline
        nop
L2::    pfadd.ss   f22,       f30,      f30   // Add f22 to pipeline
        bla        r21,       r17,      L1    // If more, go to L1 after
        pfadd.ss   f23,       f31,      f31   // adding f23 to pipeline and
        fld.d      8(r16)++,  f22             // loading next f22:f23
       // If we reach this point, at least one element remains to be loaded.
        // r17 is either -4 or -3.  f20, f21, f22, and f23 still contain
        // vector elements.  Add f20 and f21 to the pipeline, too.
        pfadd.ss   f20,       f30,      f30
        nop
        pfadd.ss   f21,       f31,      f31
        nop
        .enddual                              // Initiate exit from dual mode
sumup:: pfadd.ss   f22,       f30,      f30   // Still in dual mode
        mov        -4,        r21
        pfadd.ss   f23,       f31,      f31   // Last dual-mode pair
        bte        r21,       r17,      done  // If there is one more
        fld.l       8(r16)++, f20             // element, load it and
        pfadd.ss   f20,       f30,      f30   // add to pipeline
        // Intermediate results are sitting in the adder pipeline.
        // Let A1:A2:A3 represent the current pipeline contents.
done::  pfadd.ss   f0,        f0,       f30   // 0:A1:A2          f30=A3
        pfadd.ss   f30,       f31,      f31   // A2+A3:0:A1       f31=A2
        pfadd.ss   f0,        f0,       f30   // 0:A2+A3:0        f30=A1
        pfadd.ss   f0,        f0,       f0    // 0:0:A2+A3
        pfadd.ss   f0,        f0,       f31   // 0:0:0         f31=A2+A3
        fadd.ss    f30,       f31,      f16   //          f16 = A1+A2+A3
```

Section Control

.text
.data
.abs *vaddr* (*paddr*)

These directives specify in which section assembly is to take place. The directive **.text** assigns output to the *text* section, the directive **.data** assigns output to the *data* section, and each **.abs** directive creates an absolute-address section. In the absence of a section control directive at the beginning of a program, assembly begins in the *text* section.

In an **.abs** directive, *vaddr* is an integer expression that specifies the logical address of the section. The optional *paddr* is an integer expression that specifies the physical address. The same address may not belong to two absolute sections.

Block Space Definition

.blks [*expr*]

The block space directive reserves space for an object of the size indicated by *s*. When **.blk** is used in the *text* or *data* section, bytes of zeros are assembled into the object module. When **.blk** is used within a dummy section (i.e., in a **.dsect** or **.ndsect** block), no space is actually allocated; its only effect is to increase or decrease the location counter. The size specifier *s* may take the following values:

b	Byte
s	Short
l	Long
f	Single precision floating-point
d	Double precision floating-point

The *expr* specifies the number of objects of the given size. If *expr* is not present, one such location is reserved.

The block space reserved is aligned according to the size of the *S* specifier.

See "Common Space Definition" on page 7 for methods of defining zero-filled objects that do not allocate space in the output object module.

The following example shows space being reserved by the **.blk** directive for 128 double-precision floating-point objects and 16 long objects:

Example

```
            .data
darray:     .blkd  128   // Array big enough for 128 doubles
iarray:     .blkl  16    // Array big enough for 16 longs
```

Common Space Definition

.comm *id, expr*

.lcomm *id, expr*

These directives reserve space in the memory image without requiring space in the object file.

.comm

The **.comm** directive establishes the symbol *id* as an undefined external symbol. The size of the symbol is set to *expr* bytes. The **.comm** directive is useful for defining storage that is shared among two or more modules, where the storage area:

- Does not need to be initialized, or
- Must be initialized to zero, or
- Must be initialized to a nonzero value by precisely one of the sharing modules.

A symbol defined by a **.comm** directive may be redefined (in the same module or in another module) as a *text* or *data* section symbol. This is accomplished by using the same symbol as a label for a **.blkS** directive or for one of the storage-definition directives. Redefining the symbol assigns it a location in the *text* or *data* section and gives it an initial value. All references to the symbol then refer to this location.

If the *id* symbol is not redefined as a *text* or *data* section symbol (i.e., all other modules that reference the symbol do so via a **.comm** directive), the linker assigns the symbol to the *bss* section.

When several identically named common symbols are present, the linker defines a single area with the size of the largest common symbol.

The space allocated for the *.comm* symbol is aligned according to the size of the *expr* in the request. The space allocated is aligned according to the most restrictive type possible (byte, short, word, double, quad) whose size is less than or equal to *expr*. That is, if *expr* were in the range 4-7(bytes), the alignment would be by word and if *expr* were in the range 8-15 (bytes), the alignment would be by double word. If *expr* were greater than 16, alignment would be by quad (16 byte alignment).

.lcomm

The **.lcomm** directive defines *id* as a local symbol, assigns it to a *bss* section location, and reserves *expr* bytes. This directive is useful for allocating objects that are not initialized (or are initialized to zero) and not exported.

The space allocated for the *.lcomm* symbol is aligned according to the size of the *expr* in the request. The space allocated is aligned according to the most restrictive type possible (byte, short, word, double, quad) whose size is less than or equal to *expr*. That is, if *expr* were in the range 4-7 (bytes), the alignment would be by word and if *expr* were in the range 8-15 (bytes), the alignment would be by double word. If *expr* were greater than 16, alignment would be by quad (16 byte alignment).

Records and Structures

.dsect....end

.ndsect....end

Records and structures are defined in a dummy section. The purpose of a dummy section is to assign relative address values to the labels so that they may be used with an indexed addressing mode. Only assignments, labels, storage-definition directives, and the directives **.align**, and **.blkS** are allowed. (No code may be generated in a dummy section.)

Dummy sections are said to be *ascending* or *descending*. An ascending dummy block begins with **.dsect** and ends with **.end**. After **.dsect**, the assembler's location counter is set to zero and increases after each directive that allocates storage.

Example

```
// Employee record
            .dsect
      id:   .short
      name: .blkb 30
      ss_no: .byte [11]0        // same as blkb when in dsect
            .align .long
      salary:.long
            .end
```

A descending dummy block begins with **.ndsect** and ends with **.end**. The descending dummy block is useful for defining stack frames. In such a block, the assembler's location counter decreases after each directive that allocates storage. Note that, because that location counter decreases after each storage allocation, you must place the label for a storage location *after* the statement that allocates that location.

Example

```
      // Stack frame
            .ndsect
      x:                    // has value zero
            .long
      y:                    // refers to the long
            .byte
      z:                    // refers to the byte
            .end
```

Storage Definition

.byte	[[[*rcount*]] *is_expr*] [, [[*rcount*]] *is_expr*]...
.short	[[[*rcount*]] *i_expr*] [, [[*rcount*]] *i_expr*]...
.long	[[[*rcount*]] *i_expr*] [, [[*rcount*]] *i_expr*]...
.float	[[[*rcount*]] *ir_expr*] [, [[*rcount*]] *ir_expr*]...
.double	[[[*rcount*]] *ir_expr*] [, [[*rcount*]] *ir_expr*]...
.string	[[[*rcount*]] *si_expr*] [, [[*rcount*]] *is_expr*]...

NOTE

Brackets shown in boldface are required punctuation. For example, replace [rcount] by a number with brackets, e.g., [3].

The directives allocate and initialize areas of memory. They may be used either in the *text* section or in the *data* section. An area of the indicated size is allocated and is initialized with the value of the following expression. The repeat count *rcount* is an optional constant integer expression enclosed in square brackets. When *rcount* is given, *rcount* areas of the indicated size are allocated each with the value of *expr*. When *rcount* is not given, one area of the indicated size is allocated.

The *i_expr* is an integer expression. The *ir_expr* may be either an integer or floating-point expression. The *is_expr* may be either a constant integer expression or a string constant. If it is a string constant, each character within the string generates an area of the specified size, and the area is initialized with the value of that character. If no initialization expression is given, the allocated area is initialized to zero.

The directive **.string** is equivalent to **.byte** except that **.string** adds a final byte with the value NUL.

The storage allocated for the allocation directives begins at the current location in the current section. No default alignment rules apply. If you need a specific alignment, use the **.align** directive.

Example

```
// Valid storage definitions
  .byte     '*','*','*'              // Three stars
  .byte     [3]'*'                   // Three stars
  .byte     "***"                    // Three stars
  .byte     [3]"***"                 // Nine stars
  .string "Aardvaark\tBadger\tCamel"// NUL-terminated string
  .long                              // Initialized to zero
  .long     1                        // 32-bit word, value 1
  .long     1,2,3                    // Three 32-bit words
  .long     [3]1,[3]2,[3]3           // Nine 32-bit words
  .float                             // Uninitialized storage
  .float    3.14159                  // 32-bit real
     .double   3.14159265            // 64-bit real
// Invalid storage definitions
   .long     "XYZ"                 // Must be integer expression
   .float    "XYZ"                 // Must be numeric expression
```

Enumeration

.enum [*symbol* [=*expr*] [,*symbol* [=*expr*]]...

This directive assigns integer constants with increasing values to a list of symbols. If = *expr* is not given, the first symbol's value is zero, and subsequent values are each greater by one. Any symbol may be followed by the assignment = *expr* to set the sequence to another value. The *expr* is an integer expression.

Example

```
// Valid enumerations
    .enum a,b,c                 // define a=0,b=1,c=2
    .enum x=5,y,z               // define x=5,y=6,z=7
// Invalid enumerations
    .enum x=2.0, y, z           // Not an integer
```

External Symbols

.extern *symbol* [,*symbol*]...

.globl *symbol* [,*symbol*]...

The **.extern** and **.globl** directives declare a list of symbols as external. If a symbol is defined within the module as a constant or label, the effect is to make the value and type available to the linker. (In other words, **.global labelx** is equivalent to **.extern labelx**, which is equivalent to **labelx**::).

If a symbol is referenced but not declared or defined within the same module, then

1. If the **-a** option is specified in the command line, the undefined symbol causes an error message.

2. If the **-a** option is **not** specified in the command line, the symbol is an *undefined external*, and the linker is instructed to import the symbol and relocate any references to it.

Change Addressing Temporary

.atmp *reg*

The **.atmp** directive selects the register that is to be used temporarily by pseudoinstruction expansions that perform address computation (and other pseudo-instruction expansions as well). (For more about pseudoinstructions, refer to Chapter 3.)

The register *reg* must be an integer register. Programmers should be careful when using this register (Refer to the description of the Application Binary Interface (ABI) in Chapter 11 of the *i860 Programmer's Reference Manual* for details). The default addressing temporary register is `r31`.

Listing Control

list [**.macro**] [**.rept**][**.if**]

nlist [**.macro**] [**.rept**][**.if**]

.page

.title "*string*"

sbttl "*string*"

If listing is enabled with a command-line option, then these directives control the listing from within the source module.

The directives **.list** and **.nlist** enable and disable the assembly listing for subsequent statements. They work by incrementing (**.list**) or decrementing (.**nlist**) a counter; the listing is produced as long as the counter is greater than zero.

Using **.list** or **.nlist** by itself affects the entire listing. Using **.list** or **.nlist** with **.macro** affects listing of expanded macros. Using **.list** or **.nlist** with **.rept** controls listing of repeated blocks.

The **.page** directive begins a new page in the listing.

The **.title** directive specifies a string to appear in the page header as a title. The **.sbttl** directive specifies a string for the page header subtitle.

Symbolic Debugging

```
.file name
.ln ln_num
.def symbol
        .val expr
        .scl expr
        .type expr
        .tag name
        .line expr
        .size expr
        .dim expr1 [,expr2] ...
.endef
```

These directives create symbol-table entries with specific values for the various fields. Normally, these directives are generated only by translators, which intersperse them among generated assembly-language instructions. The values are defined by the COFF specifications.

The **.file** directive creates a symbol table entry of type file and value *name*, which is normally the name of the source file.

The line-number directive **.ln** uses the location counter in the *text* section as the address of the line.

You can repeat the **.def... .endef** block once for each symbol to be defined. The items within a **.def... .endef** block correspond to the COFF as follows:

.val	Value of the symbol
.scl	Storage class of the symbol
.type	Type of the symbol
.tag	Tag name for auxiliary table entries
.line	Line number for auxiliary table entries
.size	The total size of an array, structure, union, etc.
.dim	The number of elements in each dimension of an array

Example

This example shows the C source code and the actual assembly-language program created as the output of a C compilation. (The **-g** compiler option was set to cause the C compiler to generate symbolic debug information.)

C Source Code

```
struct record {
   char name[30];
   int mileage
   };

float vel(distance,time)
float distance;
float time;
{
  double mat[4][5];
  return (distance/time);
}
```

Assembly Language Output of C Compiler

```
   .file          "symdeb.c"
// PGC Rel 2.1 -opt 0 -debug
   .text
   .def record;   .scl 10;    .type 8;    .size 36;      .endef
   .def name;     .val 0;     .scl 8;     .type 50;      .dim 30;    .size 30;   .endef
   .def mileage;  .val 32;    .scl 8;     .type 4;       .endef
   .def .eos;     .val 36;    .scl 102;   .tag record;   .size 36;   .endef
   .globl         _vel
   .align         32
_vel:
   .def _vel;     .val _vel;  .scl 2;     .type 38;      .endef
.a1 = 80
.f1 = 192
   addu -(.a1+.f1), sp, sp
   st.l fp,(.f1-16)(sp)
   addu (.f1-16), sp, fp
   st.l r1, 4(fp)


   fmov.ds f8, f8


   fmov.ds f10, f10


   fst.l f8, 16(fp)
   fst.l f10, 24(fp)
   .ln            1
   .def    .bf;    .val     .;    .scl     101;     .line     7;        .endef
// lineno: 7
// lineno: 11
   .ln            5
   fld.l 16(fp), f8
```

```
   call _mth_i_rdiv
   fld.l 24(fp), f9
   br .B62
   nop
// lineno: 12
   .ln            6
.B62://.R0000
```

```
   .def distance; .val 16;    .scl 9;     .type 6;       .size 4;    .endef
   .def time;     .val 24;    .scl 9;     .type 6;       .size 4;    .endef
   .def mat;      .val -160;  .scl 1;     .type 247;     .dim 4,5;   .size 160;  .endef
   .ln            6
   .def    .ef;    .val     .;    .scl     101;    .line    12;      .endef
   adds .a1+16, fp, r31
   ld.l 4(fp), r1
   ld.l 0(fp), fp
   bri r1
   mov r31, sp
   .def _vel;     .val .;     .scl -1;    .endef
   .extern        _mth_i_rdiv
```

Using Macros 5

You can use the macro preprocessor (**mac860**) to expand macros in your assembly language files. The macro preprocessor features:

- Definition and replacement of symbols with strings
- Macro definition and expansion with parameters
- Repeat statement expansion
- File inclusion
- Conditional expansion (conditional assembly)

The syntax of macro statements is similar to that of assembler assignment and directive statements; therefore, programs appear to be written in a single homogeneous language.

Use of the macro preprocessor is optional. You invoke the macro preprocessor as explained in the following section. If you do not use the macro preprocessor, your macros will not be expanded.

NOTE

All instruction names and other keywords are reserved. Do not use these strings in your assembly code.

Macro Preprocessor Command-line Syntax

To invoke the macro preprocessor, use the following syntax:

```
mac860 [switches] source_file
```

where:

switches Is one or more of the switches listed in Table 5-1.

source_file Is the name of the file you want to process. The macro preprocessor reads the specified input file and produces a single output file. This file is ready to be processed by the assembler. The extension of the output file name is *.mac*.

Table 5-1. Macro Options

Switch	Function
-D *sym=val*	Defines *sym* as a local symbol with the value *val* in the macro processor.
-I *incfile*	Includes the file *incfile* before the first statement of *source_file*. You can use at most one **-I** switch in a single **mac860** command.
-o *objfile*	Specifies an output filename to replace the default output filename (*sourcefile.mac*).
-V	Displays **mac860** version information.
-y	Outputs special directives that the assembler uses for better reporting of the lines in the source file where errors are detected.

Each switch is processed in the order in which it is listed. If no command-line files or options are entered, the macro preprocessor simply displays the command-line syntax and quits.

Macro Symbols

The macro preprocessor associates a symbol with an arbitrary string. Once a macro symbol is defined, any use of that symbol causes the symbol to be replaced by the associated string.

Local Symbol Definition

A local symbol definition associates a symbol with either an integer expression (*int_exp*) or with a string (*mstring*):

symbol = *int_expr*
symbol = *mstring*

An *int_expr* can be any previously defined integer expression, as defined in Chapter 2. However, *int_expr* cannot use type operators. Any expression that is not defined at the time of macro expansion is treated as a string (except in the condition of an *if* macro statement, where the undefined expression is treated as zero).

The *mstring* is not enclosed in quotation marks. The string starts with the first non-space character after the = sign and ends with the last character of the line (except for comments). Space characters at the end of the string are part of the symbol definition.

A local symbol definition is used only during macro expansion; the symbol definition is not carried on to the assembly passes.

Global Symbol Definition

A global symbol definition has the form:

symbol =: *int_expr*

The *int_expr* can be any previously defined integer expression, as defined in Chapter 2. Any global symbol expression that is not defined at the time of macro expansion is treated as a syntax error.

A global symbol definition is used both during the macro pass and during the assembly passes. The assembly passes place the symbol and its value in the output symbol table.

Symbol Replacement

After a macro symbol definition, any occurrence of *symbol* causes *symbol* to be replaced by the *int_expr* or *mstring* that it represents. To be accepted as such, a *symbol* must be properly delimited from surrounding text. The macro preprocessor accepts the following as delimiters:

space
tab
newline
comma
operators defined for expressions
? (the symbol concatenation operator)

Note that, if an arithmetic expression contains an uninitialized symbol, the expression is treated as a string.

The expression that results from symbol replacement is also scanned for occurrences of macro symbols. Replacement is carried out at most four times. If the resulting expression, after four replacements, still contains macro symbols, these symbols are not replaced.

The following example shows a local symbol definition in source code, and the resulting macro expansion:

The following source code:

```
f_offset = 4
ld.l     f_offset(r1Ø)++, r15
ld.l     f_record+f_offset+8, r15
```

would be expanded by **mac860** as follows:

```
mac86Ø myprog.s
cat myprog.mac
ld.l     4(r1Ø)++, r15
ld.l     f_record+4+8, r15
```

Symbol Concatenation

You can use the macro operator ? to separate two or more symbols so that each is recognized as a symbol, checked for replacement, and replaced if possible. The results are then concatenated, without the ? operator. The syntax is:

```
symbol?symbol[?symbol]...
```

The following example shows source code with the ? operator on two lines and the resulting macro expansion.

The following source code:

```
base=(r1Ø)
offset=4
ld.l offset?base, r14
base=_n_rec
offset=4
ld.l offset?base, r14
```

would be expanded by **mac860** as follows:

```
mac86Ø myprog.s
cat myprog.mac
ld.l 4(r1Ø), r14
ld.l 4_n_rec, r14
```

Macro Definition

Use the following syntax to define a macro:

```
.macro [-]name[param] [[,]param]
statement
  .
  .
  .
.endm
```

A macro definition assigns a name and a formal parameter list to a sequence of assembler statements. Each parameter *param* is a symbol. Symbols in the parameters are expanded before expansion of the macro. There can be at most 30 parameters.

After a matching **.endm** directive is processed, **mac860** recognizes the name of the macro and substitutes the saved assembly language statements. This procedure, called macro expansion, invokes the macro. Actual parameters are supplied when the macro is invoked. The invocation must supply the same number of actual parameters as there are formal parameters in the definition.

The **.endm** directive must be the first symbol on its line; no labels are permitted.

During macro expansion, all references to a parameter in the macro definition are replaced by the corresponding actual parameter. The replacement is done only once; the resulting assembly-language statement is not scanned for further parameter matches. The expansion of the macro along with the generated object code is listed. The directive **.nlist .macro** disables listing of macro expansions.

One macro can invoke another to a depth of six macros. When one macro invokes another, the parameters of the first invocation are hidden from the inner invocation, and therefore the inner macro cannot reference them.

A macro *name* that is preceded in the definition by the dash character (–) defines a macro that is not expanded recursively. Note that the macro processor does not recognize assembler keywords; therefore, you can define a nonrecursive macro that has the same name as an instruction mnemonic.

Redefinition of macros is permitted. The latest version is the one that is saved for further expansion.

A macro definition can itself contain macro definitions. In this case, an inner definition is processed only when the outer macro is expanded. The inner macro cannot be called until the outer macro has been expanded at least once. If the outer macro is expanded again, the inner macro is redefined.

Comments in the macro definition are replicated without change each time the macro is invoked. No expansion is done on comments. Comments on the invocation line are discarded upon expansion.

The following example shows a macro definition and its expansion by the macro preprocessor. The following source code:

```
.macro st5freg rx freg1 freg2 freg3 freg4 freg5
  fst.l freg1, 4(rx)++
  fst.l freg2, 4(rx)++
  fst.l freg3, 4(rx)++
  fst.l freg4, 4(rx)++
  fst.l freg5, 4(rx)++
.endm
  ld.l 1ØØØ(r12), r31
  st5freg r31 f7 f8 f9 f6 f15
```

would be expanded by **mac860** as follows:

```
mac86Ø myprog.s
cat myprog.mac
ld.l 1ØØØ(r12), r31
fst.l f7, 4(r31)++
fst.l f8, 4(r31)++
fst.l f9, 4(r31)++
fst.l f6, 4(r31)++
fst.l f15,4(r31)++
```

Repetition

For repetition, use the following syntax:

```
.rept expr
statement
  .
  .
  .
.endr
```

The block of statements contained within these directives is expanded *expr* times. The integer expression *expr* must be a positive integer constant and must be previously defined.

Repeat blocks can be nested within repeat blocks, up to eight levels deep. In this case, the inner repeat block is expanded once for each expansion of the next outer block. The repeat count of an inner block is evaluated at each expansion of the inner block.

The **.endr** directive must be the first symbol on its line; no labels are permitted.

Repeat blocks can contain macro calls, and macro definitions can contain repeat loops. Repeat loops, however, cannot contain macro definitions.

The following example shows a macro with the repeat directive and the resulting macro expansion. The following source code:

```
.macro idn size// Define identity matrix
     tmp = Ø
.rept size .float [tmp]Ø;
     .float 1; .float [size-tmp-1] Ø
     tmp = tmp + 1
.endr
.endm
// Define 16x16 identity matrix
     idn 16
```

would be expanded by **mac860** as follows:

```
mac86Ø myprog.s
cat myprog.mac
// Define 16x16 identity matrix
.float [Ø]Ø;  .float 1;.float [16-Ø-1]Ø
.float [1]Ø;  .float 1;.float [16-1-1]Ø
.float [2]Ø;  .float 1;.float [16-2-1]Ø
.float [3]Ø;  .float 1;.float [16-3-1]Ø
.float [4]Ø;  .float 1;.float [16-4-1]Ø
.float [5]Ø;  .float 1;.float [16-5-1]Ø
.float [6]Ø;  .float 1;.float [16-6-1]Ø
.float [7]Ø;  .float 1;.float [16-7-1]Ø
.float [8]Ø;  .float 1;.float [16-8-1]Ø
.float [9]Ø;  .float 1;.float [16-9-1]Ø
.float [1Ø]Ø; .float 1;.float [16-1Ø-1]Ø
.float [11]Ø; .float 1;.float [16-11-1]Ø
.float [12]Ø; .float 1;.float [16-12-1]Ø
.float [13]Ø; .float 1;.float [16-13-1]Ø
.float [14]Ø; .float 1;.float [16-14-1]Ø
.float [15]Ø; .float 1;.float [16-15-1]Ø
```

File Inclusion

The syntax for file inclusion is:

```
.include "file_name"
```

The **.include** directive causes the assembler to temporarily read input from *file_name* instead of the current input file. (The quotation marks around *file_name* are required.) Upon reaching the end of *file_name*, the assembler resumes reading from the current file at the statement that follows the **.include** directive. Include files can be nested up to eight levels deep.

Conditional Assembly

You can obtain conditional assembly by using an **.if/.else/.endif** sequence or an **.if/.elseif/.endif** sequence:

```
.if expr                    .if expr
statement                      statement
   .                              .
   .                              .
   .                              .
.else expr                  .elseif expr
statement                      statement
   .                              .
   .                              .
   .                              .
.endif                         .endif
```

The **.if** and **.endif** directives specify a block of assembly-language statements that are to be assembled only if *expr* is true (nonzero). If the *expr* after **.if** is false (zero), assembly of statements is suspended until a matching **.else**, **.elseif**, or **.endif** is found. If the *expr* is undefined, it is treated as false.

The **.else** directive assembles subsequent statements only if the *expr* is false.

The **.elseif** directive is equivalent to a **.else** followed by a second **.if**, except that you need only one **.endif** to terminate the block.

Conditional blocks can be nested within conditional blocks up to 32 levels deep.

A conditional block is listed regardless of the value of *expr*.

The following macro example shows conditional assembly that depends on *single1* being nonzero. Macro expansion results are shown for cases when *single1* is both `0` and `1`.

Example

Source Code

```
.macro ADD  x1,x2,res
.if single1
          fadd.ss   x1,x2,res
.else
          fadd.dd   x1,x2,res
.endif
.endm
          single1 = Ø
          ADD       f16,f18,f2Ø
          single1 = 1
          ADD       f17,f19,f21
```

Macro Expansion

```
mac86Ø myprog.s
cat myprog.mac
fadd.dd   f16,f18,f2Ø
fadd.ss   f17,f19,f21
```