intel

i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual

Version 3

January 1990

240436-003

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

376, 386, 387, 486, 4-SITE, Above, ACE51, ACE96, ACE186, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, DVI, ETOX, FaxBACK, Genius, i, i, i486, i750, i860, ICE, iCEL, ICEVIEW, iCS, iDBP, iDIS, I^2ICE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, intel, Intel386, intelBOS, Intel Certified, Intelevision, intelligent Identifier, intelligent Programming, Intellec, Intellink, iOSP, iPAT, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, Pro750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, ToolTALK, UPI, Visual Edge, VLSiCEL, and ZapCode, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

MULTIBUS is a patented Intel bus.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

UNIX is a trademark of AT&T Bell Labs.

OS/2, PS/2, AIX are trademarks of International Business Machines Corporation.

Additional copies of this manual or other Intel literature may be obtained from:

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

CG-010290

Preface

The Intel i860™ Microprocessor delivers supercomputing performance in a single VLSI component. The 64-bit design of the i860 Microprocessor balances integer, floating point, and graphics performance for applications such as engineering workstations, scientific computing, 3-D graphics workstations, and multiuser systems. Its parallel architecture achieves high throughput with RISC design techniques, pipelined processing units, wide data paths, large on-chip caches, million-transistor design, and fast one-micron CHMOS IV silicon technology.

The i860 Microprocessor Assembler is designed for use both directly by programmers and for use as a postprocessor for the output of high-level language translators. It supports the entire instruction set of the i860 Microprocessor.

The i860 Microprocessor Linker combines object files created by the Assembler into a single object file. The Linker supports the unique relocation requirements of the instructions of the i860 Microprocessor.

This manual is a complete source of information about the use of both the Assembler and the Linker. It is useful both to software managers and software engineers. Software managers will learn how these tools can accelerate development of applications for the i860 Microprocessor. Software engineers will find detailed descriptions of the use of these tools.

System Requirements

The software documented in this manual executes in the following environments:

- ❑ UNIX System V/386, release 3.2
- ❑ OS/2 version 1.1 or later
- ❑ AIX version 1.0 or later

Prerequisites

Because the Assembler deals with machine-level instructions, you should already be familiar with the architecture and instruction set of the i860 Microprocessor as presented in the *i860 64-Bit Microprocessor Programmer's Reference Manual.*

How to Use This Manual

- Chapter 1, "Introduction to the Assembler" presents an overview of the Assembler in terms of its basic features and its inputs and outputs.
- Chapter 2, "Assembly Language Syntax" identifies the elements of the assembly language.
- Chapter 3, "Instruction Syntax" defines all the assembly-language syntax that results in assembly of machine instructions. (For a detailed description of the machine instructions themselves, refer to the *i860 64-Bit Microprocessor Programmer's Reference Manual.*)
- Chapter 4, "Assembler Directives" explains additional syntax that controls the actions of the Assembler.
- Chapter 5, "Macro Processor" defines the syntax of the macro language.
- Chapter 6, "Operation" explains how to execute the Assembler and Linker, including command-line options.
- Chapter 7, "Object File Format" explains the object file format used by the Assembler and the Linker. It is explained in terms of extensions to the UNIX System V/386 standard COFF. For a complete definition of COFF, refer to the UNIX System V/386 documentation. The information in this chapter is needed only by those who need to write programs that create or process object files for the i860 microprocessor. Knowledge of the object file format is not generally needed by applications programmers.

Related Documentation

The following books contain additional material concerning the i860 Microprocessor:

- *i860 64-Bit Microprocessor* (Data Sheet), Intel order number 240296
- *i860 64-Bit Microprocessor Programmer's Reference Manual,* Intel order number 240329
- *i860 64-Bit Microprocessor Simulator and Debugger Reference Manual,* Intel order number 240437

Notation and Conventions

gbold	When a name, symbol, or other sequence of characters is used in exactly the same way that it is intended to be entered into the Assembler, Linker, or other software product, it is printed in a contrasting type style, gothic bold; for example, **dirbase**.
gitalic	Gothic italic type indicates a metasymbol that is to be replaced with an item that fulfills the rules for that symbol; for example, *objfil* is to be replaced by an object-file identifier that fulfills the rules for file names.
[ ]	Brackets indicate optional arguments or parameters.
[ ]	Brackets, when printed in the contrasting type style, are required, and must be entered as shown.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ }...	At least one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order, unless otherwise noted.
...	Ellipses indicate that the preceding argument or parameter may be repeated. When an ellipsis follows a right bracket or brace, the entire unit enclosed by the brackets or braces may be repeated.
SMALLCAPS	Small capital letters indicate the ASCII name of one of the nondisplayable characters; for example, LF is the linefeed character.
^Z	The caret (**^**) indicates the use of the control key (usually labeled **Ctrl**) together with the key for the letter that follows the caret. For example, to type **^C**, depress and hold **Ctrl**, and type the **C** key while holding **Ctrl**.
.,/$	Punctuation other than brackets must be entered as shown.

Table of Contents

Tables

Chapter 1
Introduction to the Assembler

In addition to its support for 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
- Conditional assembly
- Macro language
- Enforcement of coding rules unique to the i860 Microprocessor
- Optional source and code listings
- Symbolic debugger support
- Very high-speed assembly

1.1 Source Input

The Assembler interprets input as a sequence of characters encoded according to the American Standard Code for Information Interchange (ASCII). The syntax for the input is the subject of Chapters 2, 3, 4, and 5. The character ^Z (SUB or 0x1A), if present, marks the end of an input file.

1.2 Object Code Output

The primary output of the Assembler is an object module of three types sections: *text*, *data*, *bss*, and *abs*. When the Linker combines several object modules, the *text* and *data* 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.

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. The directive **.text** returns to the text section, after output has been diverted to another section.
Data Section	The data section customarily contains variable data to which the program assigns initial values. The directive **.data** diverts output to the data section.
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. The directives **.lcomm** and **.comm** allocate *bss* section memory. (These directives are explained in Chapter 4.)
Abs Section	There may be zero, one, or more absolute sections. An absolute section is one that is assigned to a specific logical address and, possibly, to a specific physical address. An *abs* section may contain either text or data. The directive **.abs** allocates an *abs* section.

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

1.3 Listing

Other outputs of the Assembler include error messages and an optional listing file. Error messages are directed to **stderr**. When a listing is being produced, error messages are also included in the listing.

The listing is created only when the **-l** command-line option is set. The listing includes source statements, generated object code, and error messages. Selective control over the listing is provided by the directives **.list** and **.nlist** (refer to Chapter 4 for more information).

Chapter 2
Assembly Language Syntax

2.1 Statements

The input character sequence is separated into *lines* by the LF character (also called *newline*). A CR can precede the LF, in which case the CR-LF pair is treated as a single *newline*.

There are four general statement formats:

1. Instruction statement...

   [*label:*]... i*nstruction [operands]* [*// comment]*

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

2. *Directive statement...*

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

   ...controls the operation of the Assembler or macro processor. The assembly directives are defined in Chapter 4. Macro directives are defined in Chapter 5.

3. *Assignment statement...*

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

   ...defines a symbol that can be used in place of the given constant integer expression. Assignments are defined in Chapter 2.

4. *Empty statement...*

   [*// comment*]

   ...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.

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
```

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.

2.2 Constants

The Assembler accepts both numeric and alphanumeric constants.

2.2.1 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 may 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 oh)

A floating-point constant has the form...

[0f][*integer*][.[*fraction*]]**[e**{ [+] [–] } *exponent*]

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

Note that, although a token such as **.2e12** is also a legal symbol, the Assembler recognizes it as a floating-point constant. No such tokens should be used as identifiers.

Example:

(This example shows numeric constants in storage-allocation statements. Numeric constants can also be used 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                 // Floating-Point
            .float    1.2e12
            .float    1.2e+12
            .float    1.2e-12
            .double   1.e12
            .double   .2e12
            .double   .2e+12
            .double   .2e-12
```

2.2.2 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 may contain all characters of the ASCII set. The backslash character \ is used within character and string constants to permit entry of quotation marks and to specify certain control characters symbolically. The symbolic character specifications are defined in Table 2-1. A backslash followed by any other character is equivalent to the other character itself.

Table 2-1. Symbolic Character Specification

Symbolic Form	Character	ASCII Code
\0	NUL	0x0
\b	BS backspace	0x8
\t	TAB	0x9
\n	LF newline	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

Example:

(This example shows string constants in storage-allocation statements.)

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

2.2.3 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.

2.3 Symbols

Symbols can be used to label memory locations or integer values. Symbols are be composed of letters, digits, and the **.**, **$**, and **_** characters. The first character of a symbol may not be a digit or a **$**. Both uppercase and lowercase letters are accepted, but are treated distinctly; e.g., 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, such a symbol is considered by the Assembler to be external. If

-a is specified, such a symbol is an error.

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

2.3.1 Labels

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

2.3.1.1 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 to be made 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.)

Example:

```
xyz:
abc::           .byte 1       // a global label
x1:x2:x3:       .byte 1       // three labels on a line
```

2.3.1.2 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.

Temporary labels may be used 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
```

2.3.2 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 also to the definitions of **.comm** and **.lcomm** in Chapter 4.)

2.3.3 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 by = defines a local constant. An assignment by **=:** 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
```

2.3.4 The .enum Directive

The **.enum** directive (refer to Chapter 4) can be considered a form of assignment that also defines local symbols.

2.4 Expressions

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

Integer expressions may be employed in all kinds of Assembler statements where an integer value is required. Integer values are represented by the Assembler in 32-bit, two's 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 *exp* is an integer or floating-point expression, 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 half
	h%	Unary	Select high half
	ha%	Unary	Select high half and adjust

2.4.1 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 zero if its operand has any nonzero value.

2.4.2 Type Operators

The type operators in Table 2-2 utilize 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 section 2.4.2.2.

2.4.2.1 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 two 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 instruction into a two-instruction sequence, each instruction of which handles half of the 32-bit constant.

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. The Assembler detects these situations and indicates an error.

2.4.2.2 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 two 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 by generating two-instruction sequences.

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 instruction into a similar two-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 instruction expansion causes undesirable effects after a delayed branch instruction or within dual-instruction mode. The Assembler detects these situations and indicates an error.

2.4.2.3 Type Combinations

When a complex expression is formed with one of the above operators, 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.

Table 2-3. Type Combination

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

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
```

2.4.2.4 16-Bit Constant Expressions

The Assembler does all arithmetic with 32-bit operands and, normally, interprets all constants as 32-bit values. To specify a 16-bit constant, the l% operator must be used. Consider the following coding examples:

```
ld.l      0x8000(r4),       r5    // Case 1 (an error)
ld.l      l%0x8000(r4),     r5    // Case 2
and       -1,r4,            r5    // Case 3 (0xFFFFFFFF)
and       l%-1,r4,          r5    // Case 4 (0x0000FFFF)
```

Case 1 is an error, because, using 32-bit arithmetic, 0x8000 is the same as 0x00008000 or 32768, which is out of the permissible range of immediate constants for the load instruction.

Case 2, by using the l% operator indicates that the constant is a 16-bit value; therefore it is interpreted as –32768, a value that can be used as the immediate constant in the load instruction.

Case 3 and 4 are both correct, but case 3 may programmed when case 4 was intended, if the programmer is not aware that all arithmetic is done in 32 bits.

2.4.3 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`.

Chapter 3
Instruction Syntax

The instructions of the assembly language correspond one-to-one with the machine instructions of the i860 Microprocessor (except for the "pseudoinstructions" presented in section 3.5). The general syntax of an instruction is:

mnemonic source_operand_1, source_operand_2, destination

Mnemonics for machine instructions are defined in lowercase only.

Not all instructions have two source operands, but, 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 64-Bit Microprocessor Programmer's Reference Manual.*

3.1 Notation

For register operands, the abbreviations that describe the operands are composed of two parts: the first part descibes the type of operand required...

ctrlreg	One of the predefined names of control registers: **psr**, **epsr**, **db**, **dirbase**, **fir**, **fsr**
freg	One of the predefined names of floating-point registers: **f0** through **f31**
ireg	One of the predefined names of integer registers: **r0** through **r31** (sp can be used in place of **r2** and **fp** in place of **r3**)

The second part identifies the field of the machine instruction in which the operand is to be placed...

_src1	The first (lower addressed) of the two source-register designators.
_src2	The second (higher addressed) of the two source-register designators.
_dest	The destination-register designator.

Thus, the operand specifier *freg_src2*, for example, means that the name of a floating-point register is to be entered, and the encoding of that register will 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:

soffset	Short branch offset. An expression whose value is an address in 16 the text section of the current module less than 2^{16} bytes distant from the current instruction.
loffset	Long branch offset. An expression whose value after linking is an 26 address less than 2^{26} bytes distant from the current instruction.
const	An absolute or relocatable integer expression. The value is signed for add, subtract, load, and store instructions; unsigned for logical instructions. If the value of *const* is absolute and cannot be stored in 16 bits or if the value of *const* is relocatable, the Assembler generates a two or three instruction sequence to load the 32-bit integer in two 16-bit parts.
const5	An absolute expression whose value can be stored as a signed 5-bit immediate.

3.2 Core Unit Instructions

3.2.1 Integer Load

ld { .b / .s / .l } *ireg_src1(ireg_src2), ireg_dest*
onst(ireg_src2), ireg_dest

3.2.2 Integer Store

st { .b / .s / .l } *ireg_dest, const(ireg_src2)*

3.2.3 Integer to Floating-Point Register Transfer

ixfr *ireg_src1, freg_dest*

3.2.4 Floating-Point Load

fld { .l / .d / .q } { *ireg_src1(ireg_src2), freg_dest* / *ireg_src1(ireg_src2)++, freg_dest* / *const(ireg_src2), freg_dest* / *const(ireg_src2)++, freg_dest* }

3.2.5 Pi;elined Floating-Point Load

pfld { .l / .d } { *ireg_src1(ireg_src2), freg_dest* / *ireg_src1(ireg_src2)++, freg_dest* / *const(ireg_src2), freg_dest* / *const(ireg_src2)++, freg_dest* }

3.2.6 Floating-Point Store

pfld { .l / .d } { *freg_dest, ireg_src1(ireg_src2)* / *freg_dest, ireg_src1(ireg_src2)++* / *freg_dest, const(ireg_src2)* / *freg_dest, const(ireg_src2)++* }

3.2.7 Pixel Store

pst.d { *freg_dest, const(ireg_src2)* / *freg_dest, const(ireg_src2)++* }

3.2.8 Cache Flush

flush { *const(ireg_src2)* / *const(ireg_src2)++* }

3.2.9 Control Register Load/Store

ld.c *ctrlreg_src2, ireg_dest*
st.c *ireg_src1, ctrlreg_src2*

3.2.10 Long Branch and Call

{ bc, bnc / bc.t, bnc.t / br, call } *loffset*

3.2.11 Branch if Equal/Not Equal

{ bte / btne } { *ireg_src1, ireg_src2, soffset* / *const5, ireg_src2, soffset* }

3.2.12 Branch on LCC and Add

bla *ireg_src1, ireg_src2, soffset*

3.2.13 Branch and Call Indirect

{bri / calli} *ireg_src1*

3.2.14 Arithmetic, Logical, Shift

{adds / addu / subs / subu} {and / andnot / or / xor} {shl / shr / shra} {*ireg_src1, ireg_src2, ireg_dest* / *const, ireg_src2, ireg_dest*}

3.2.15 Double-Register Shift

shrd *ireg_src1, ireg_src2, ireg_dest*

3.2.16 High-Half Logical

{andh / andnoth / orh / xorh} *const, ireg_src2, ireg_dest*

3.2.17 Bus Lock

lock
unlock

3.2.18 Software Traps

trap / intovr {*ireg_src1, ireg_src2, ireg_dest* / *const, ireg_src2, ireg_dest*}

3.3 Floating-Point Unit Instructions

3.3.1 Compare

{pfgt / pfle / pfeq} {.ss / .dd} *freg_src1, freg_src2, freg_dest*

3.3.2 Add, Subtract, and Multiply

{ fadd, fsub, fmul, pfadd, pfsub, pfuml } { .ss, .sd, .dd } *freg_src1, freg_src2, freg_dest*

3.3.3 Special Multiply

{ fmlow.dd, pfmul3.dd } *freg_src1, freg_src2, freg_dest*

3.3.4 Special Add

{ famov, pfamov } { .ss, .sd, .ds, .dd } *freg_src1,freg_dest*

3.3.5 Add and Multiply Dual Operation

{ i2ap1, i2apt, i2p1, i2pt, ia1p2, iat1p2, m12apm, m12tpa, m12tpm, m12ttpa, r2ap1, r2apt, r2p1, r2pt, ra1p2, rat1p2 } { *.ss, .sd, .dd* } *freg_src1, freg_src2, freg_dest*

{ mi2ap1, mi2apt, mi2p1, mi2pt, mia1p2, miat1p2, mm12apm, mm12tpa, mm 12tpm, mm12ttpa, mr2ap1, mr2apt, mr2p1, mr2pt, mra1p2, mrat1p2 } { *.ss, .sd, .dd* } *freg_src1, freg_src2, freg_dest*

3.3.6 Subtract and Multiply Dual Operation

{ i2as1, i2ast, 92s1, 92st, ia1s2, iat1s2, m12asm, m12tsa, m12tsm, m12ttsa, r2as1, r2ast, r2s1, r2st, ra1s2, rat1s2 } { *.ss, .sd, .dd* } *freg_src1, freg_src2, freg_dest*

{ mi2as1, mi2ast, mi2s1, mi2st, mia1s2, miat1s2, mm12asm, mm12tsa, mm 12tsm, mm12ttsa, mr2as1, mr2ast, mr2s1, mr2st, mra1s2, mrat1s2 } { *.ss, .sd, .dd* } *freg_src1, freg_src2, freg_dest*

3.3.7 Reciprocal and Reciprocal Square Root

{frcp, frsqr} {.ss, .sd, .dd} *freg_src2, freg_dest*

3.3.8 Floating-Point to Integer Register Transfer

fxfr *freg_src1, ireg_dest*

3.3.9 Floating-Point to Integer Conversion

{ftrunc, fix pftrunc, pfix} {.sd, .dd} *freg_src1, freg_dest*

3.3.10 Long Integer Arithmetic

{fiadd, fisub pfiadd, pfisub} {.ss, .dd} *freg_src1, freg_src2, freg_dest*

3.3.11 Graphics

{faddp pfaddp, faddz pfaddz, fzchksp fzchks, fzchkl pfzchkl} *freg_src1, freg_src2, freg_dest*

3.3.12 OR with MERGE Register

{form, pform} *freg_src1, freg_dest*

3.4 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.

3.5 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.

3.5.1 Integer Register to Register Move

mov *ireg_src2, ireg_dest*

This pseudoinstruction is implemented as...

shl *r0, ireg_src2, ireg_dest*

3.5.2 Integer Constant to Register Move

mov *const, ireg_dest*

If the unsigned value of const is less than 0x8000, then this pseudoinstruction is implemented as...

adds l%*const, r0, ireg_dest*

...otherwise it is implemented as...

orh h%*const, r0, ireg_dest*
or l%*const, ireg_dest, ireg_dest*

3.5.3 Address to Register Move

mov *addr_expr, ireg_dest*

This pseudoinstruction is implemented as...

orh h%*addr_expr, r0, ireg_dest*
or l%*addr_expr, ireg_dest, ireg_dest*

3.5.4 Floating-Point Register to Register Moves

{fmov / pfmov} {.ss / .dd} *freg_src1, freg_dest*

These pseudoinstructions

{fiadd / pfiadd} {.ss / .dd} *freg_src1, f0, freg_dest*

{fmov / pfmov} {.sd / .ds} *freg_src1, freg_dest*

These pseudoinstructions are implemented as...

famov / pfamov { .sd / .ds } *freg_src1, f0, freg_dest*

3.5.5 No Operation

```
nop
```

The core no-op pseudoinstruction is implemented as...

```
shl r0, r0, r0
fnop
```

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

```
shrd r0, r0, r0
```

3.5.6 32-Bit Address Expression

ld { .b / .s / .l } *addr_expr, ireg_dest*

st { .b / .s / .l } *ireg_dest, addr_expr*

fld { .l / .d / .q } *addr_expr, freg_dest*

pfld { .l / .d } *addr_expr, freg_dest*

fst { .l / .k / .q } *freg_dest, addr_expr*

pst.d *freg_dest, addr_expr*

flush *addr_exp*

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), ireg_dest
```

3.5.7 Unsigned 32-Bit Constant

```
and     const, ireg_src2, ireg_dest
andnot  const, ireg_src2, ireg_dest
or      const, ireg_src2, ireg_dest
xor     const, ireg_src2, ireg_dest
```

When const 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%const, ireg_src2, r31
or      l%const, r31, ireg_dest
```

The expansion of **and** complements const, then uses the **andnot** instruction...

```
andnot  h%(-1-const), ireg_src2, r31
andnot  l%(-1-const), r31, ireg_dest
```

3.5.8 Signed 32-Bit Constant

```
adds    const, ireg_src2, ireg_dest
addu    const, ireg_src2, ireg_dest
subs    const, ireg_src2, ireg_dest
subu    const, ireg_src2, ireg_dest
```

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

```
orh     h%const, r0, r31
or      l%const, r31, r31
adds    r31, ireg_src2, ireg_dest
```

Chapter 4
Assembler Directives

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.

4.1 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.

4.2 Dual Mode

.dual... .enddual

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 *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.

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
```

4.3 Section Control

.text
.data
.abs *vaddr [paddr]*

These directives specify in which section assembly is to take place. The directive **.text** causes output to be assigned to the *text* section; the directive **.data** causes output to be assigned to the *data* section; each **.abs** directive causes creation of an absolute-address section. The **-R** command-line parameter, if specified, overrides the **.data** directive, and assembly continues in the *text* 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. No address may belong to two absolute sections.

4.4 Block Space Definition

.blks [*exp*]

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.

Refer to the **.comm** and **.lcomm** directives for methods of defining zero-filled objects that do not allocate space in *objfil*.

Example:

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

4.5 Common Space Definition

.comm *id, expr*
.lcomm *id, expr*

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

4.5.1 .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 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.

4.5.2 .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.

4.6 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 **blk*s*** are allowed. (No code may be generated in a dummy section.)

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, it is necessary to 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
```

4.7 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*]] *is_expr*]	[, [[*rcount*]] *is_expr*]...

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.

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
```

4.8 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 a 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
```

4.9 External Symbols

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

The **.extern** and **.globl** directives define 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** is equivalent to **labelx**::.)

If a symbol is referenced but not 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.

4.10 Change Addressing Temporary

.atmp *reg*

The **.atmp** directive selects the register that is to be used temporarily by pseudoinstruction expansions that perform address computation. (For more about pseudoinstructions, refer to Chapter 3.) The *reg* must be an integer register. Programmers should avoid using this register. The default addressing temporary register is **r31**.

4.11 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** selectively enable and disable the assembly listing for subsequent statements. They work by respectively incrementing and decrementing a counter; the listing is produced as long as the counter is greater than zero. There is a separate counter for **.macro**, **.if**, and **.rept**.

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 to appear in the page header as a subtitle. The table of contents includes all titles and subtitles, and includes the line number for each.

4.12 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 with the name **.file** and the 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.

The **.def... .endef** block can be repeated, 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 actual assembly-language program created as the output of a C compilation. The **-g** option is set to cause the C complier to generate symbolic debug information.

----- C Source Code -----

```
float vel(distance,time)
    float    distance;
    float    time;
    {
    double  mat[4][5];
    struct  record {
        char  name[30];
        int   mileage
        };
    return (distance/time);
    }
```

----- Assembly Language Output of C Compiler -----

```
.file "symdeb.ss"
.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
// ccom -g -OM -X22 -X74 -X80 -X83 -X254 -X266 -X325 -X332 -X350 -X383 - X424
.text
.align 4
.def _vel; .val _vel; .scl 2; .type 39; .endef
_vel:
adds -176,sp,sp
//          .bf
.ln 1
.def .bf; .val .; .scl 101; .line 4; .endef
st.l r1,164(sp)
st.l fp,160(sp)
adds 160,sp,fp

.ln 2

.ln 3
//          .bs
.def _distance; .val 48; .scl 17; .type 7; .endef
.def _time; .val 50; .scl 17; .type 7; .endef
.def _mat; .val -160; .scl 1; .type 247; .dim 4,5; .size 160;.endef

.ln 8
orh 16384,r0,r28

frcp.dd f18,f24
fmul.dd f18,f24,f20
ixfr r28,f27
fmov.ss f0,f26
fsub.dd f26,f20,f20
fmul.dd f24,f20,f24
```

```
            fmul.dd f18,f24,f20
            fsub.dd f26,f20,f20
            fmul.dd f24,f20,f24
            fmul.dd f18,f24,f20
            fmul.dd f16,f24,f24
            fsub.dd f26,f20,f20
            br .L5
            fmul.dd f20,f24,f22
//          .es
            .ln 9
.L5:
//          .ef
            .def .ef; .val .; .scl 101; .line 9; .endef
            ld.l 160(sp),fp ld.l 164(sp),r1
            adds 176,sp,sp
            bri r1
                                fmov.dd f22,f16
                                .def vel; .scl -1; .endef
                                .align 4
                                .data
                                //_mat
-160(fp) local
.L1:
//_distance f16 local
//_time f18 local

            .text
            .data
            .glob
 _vel

            .text
```

Chapter 5
Macro Processor

The Assembler can be used in conjunction with the macro processor. When used, the macro processor is executed prior to the usual assembly passes. The macro processor 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 processor is optional. The default action is to skip the macro pass in order to achieve faster assembly. The Assembler has a command-line option **-z** to enable the macro prepass when required.

5.1 Macro Symbols

The macro processor enables a symbol to be associated 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.

5.1.1 Local Symbol Definition

A local symbol definition associates a symbol with either an integer expression...

symbol = int_expr

...or with a string...

symbol = mstring

The *int_expr* may be any integer expression, as defined in Chapter 2 (except that the type operators may not be used), that is previously defined. 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 it is treated as zero).

The *mstring* is *not* enclosed in quotation marks. The string starts with the first printable character after the equals sign and ends with the last printable character of the line (except for comments).

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

5.1.2 Global Symbol Definition

A global symbol definition has the form...

symbol =: *int_expr*

The *int_expr* may be any integer expression, as defined in Chapter 2, that is previously defined. Any 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.

5.1.3 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 recognized as such, a *symbol* must be properly delimited from surrounding text. The macro processor recognizes as delimiters NEWLINE. 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 executed at most four times. If the resulting expression, after four replacements, still contains macro symbols, these symbols are *not* replaced.

Example:

----- Source Code -----

```
f_offset = 4
ld.l        f_offset(r10)++, r15
ld.l        f_record+f_offset+8, r15
```

----- Macro Expansion -----

```
ld.l        4(r10)++, r15
ld.l        f_record+4+8, r15
```

5.1.4 Symbol Concatenation

The macro operator **?** can be used to artificially 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.

*symbol***?***symbol*[**?***symbol*]...

Example:

----- Source Code -----

```
base=(r10)
offset=4
ld.l offset?base, r14
base=fin_rec
offset=4+
ld.l offset?base, r14
```

----- Macro Expansion -----

```
ld.l 4(r10), r14
ld.l 4+fin_rec, r14
```

5.2 Macro Definition

```
.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 may be at most 30 parameters.

After a matching **.endm** directive is processed, the Assembler recognizes the name of the macro and substitutes the saved assembly-language statements. This procedure is called *macro expansion* and is said to *invoke* 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 of the 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 may invoke another, up to a depth of six. 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 preceeded in the definition by the character / defines a macro that is not expanded recursively. Note that the macro processor does not recognize Assembler keywords; therefore, it is possible to 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 may itself contain macro definitions. In this case an inner definition is processed only when the outer macro is later expanded. The inner macro may not 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 each time the macro is invoked; however, they are left as is. No expansion is done in comments. Comments on the invocation line are discarded upon expansion.

Example:

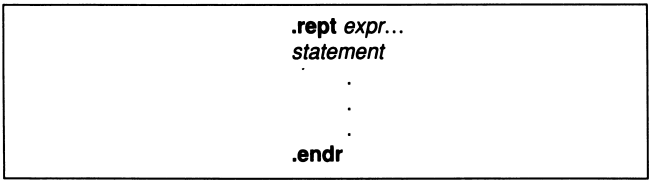
----- Source Code -----

```
.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 1000(r12), r31
          st5freg r31 f7 f8 f9 f6 f15
```

----- Macro Expansion -----

```
          ld.l 1000(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)++
```

5.3 Repetition



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 may occur within repeat blocks, up to eight levels. 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 may contain macro calls, and macro definitions may contain repeat loops. Repeat loops, however, may *not* contain macro definitions.

Example:

----- Source Code -----

```
.macro idn size                    // Define identity matrix
           tmp = 0
.rept size .float [tmp]0;
           .float 1; .float [size-tmp-1]0
           tmp = tmp + 1
.endr
.endm
// Define 16x16 identity matrix
           idn 16
```

----- Macro Expansion -----

```
// Define 16x16 identity matrix
.float [0]0;.float 1;              .float [16-0-1]0
.float [1]0;.float 1;              .float [16-1-1]0
.float [2]0;.float 1;              .float [16-2-1]0
.float [3]0;.float 1;              .float [16-3-1]0
.float [4]0;.float 1;              .float [16-4-1]0
.float [5]0;.float 1;              .float [16-5-1]0
.float [6]0;.float 1;              .float [16-6-1]0
.float [7]0;.float 1;              .float [16-7-1]0
.float [8]0;.float 1;              .float [16-8-1]0
.float [9]0;.float 1;              .float [16-9-1]0
.float [10]0;.float 1;             .float [16-10-1]0
.float [11]0;.float 1;             .float [16-11-1]0
.float [12]0;.float 1;             .float [16-12-1]0
.float [13]0;.float 1;             .float [16-13-1]0
.float [14]0;.float 1;             .float [16-14-1]0
.float [15]0;.float 1;             .float [16-15-1]0
```

5.4 File Inclusion

.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. Nesting of include files is supported up to eight levels.

----- Macro Expansion -----

```
// Define 16x16 identity matrix
.float [0]0;.float 1;            .float [16-0-1]0
.float [1]0;.float 1;            .float [16-1-1]0
.float [2]0;.float 1;            .float [16-2-1]0
.float [3]0;.float 1;            .float [16-3-1]0
.float [4]0;.float 1;            .float [16-4-1]0
.float [5]0;.float 1;            .float [16-5-1]0
.float [6]0;.float 1;            .float [16-6-1]0
.float [7]0;.float 1;            .float [16-7-1]0
.float [8]0;.float 1;            .float [16-8-1]0
.float [9]0;.float 1;            .float [16-9-1]0
.float [10]0;.float 1;           .float [16-10-1]0
.float [11]0;.float 1;           .float [16-11-1]0
.float [12]0;.float 1;           .float [16-12-1]0
.float [13]0;.float 1;           .float [16-13-1]0
.float [14]0;.float 1;           .float [16-14-1]0
.float [15]0;.float 1;           .float [16-15-1]0
```

5.4 File Inclusion

.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. Nesting of include files is supported up to eight levels.

5.5 Conditional Assembly

.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 only one **.endif** is needed to terminate the block.

Conditional blocks may be nested within conditional blocks, up to 32 levels.

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

Example:

----- Source Code -----

```
.macro ADD x1,x2,res
.if single1
          fadd.ss          x1,x2,res
.else
          fadd.dd          x1,x2,res
.endif
.endm

          single1 = 0
          ADD              f16,f18,f20
          single1 = 1
          ADD              f16,f18,f20
```

----- Macro Expansion -----

```
          fadd.dd          f16,f18,f20
          fadd.ss          f16,f18,f20
```

Chapter 6
Operation

6.1 Using the Assembler

To execute the Assembler, enter the following system command of the form:

mas860 [[[*options*]...] *input_file*]

If no command-line tail is entered, the Assembler simply displays the command line syntax and quits. The *input_file* is the name of a file of Assembler commands. The Assembler reads the specified input file and produces a single output module. This module is not directly executable; it must be processed by the Linker. The *options* can be any of the following. Options must be separated from one another by one or more spaces.

-a Do not automatically import any symbols that are referenced but are otherwise undefined. Issue an error message for this case.

-be Handle all data defined in data sections in "big endian" mode (i.e., with most significant byte as lowest memory address).

-D*sym=val* Define *sym* as a local symbol in the macro processor with value val. The **-z** option must also be set.

-I *inc_file* Include (via the macro processor) the file *inc_file* before the first statement of *input_file*. The **-z** option must also be set. Multiple **-I** options may not be specified.

-i386 Put the magic number for the Intel386™ architecture in the output object file instead of the magic number for the i860 Microprocessor, so that 80386-oriented tools can process the object module.

-l[filespec] Enable the source listing. If a *filespec* follows (without intervening space) it specifies the name of the listing file in the current disk directory. If no *filespec* follows, the listing is directed to the standard output.

-L Preserve text symbols that begin with **.L**. High-level languages generate labels that begin with **.L**.

-m Ignore (in the assembly pass) the special directives output (in the macro pass) by the -**y** option.

-o *output_file* — Set the name of the output module to *output_file.* If this option is not specified, the output module has a name derived by stripping the name of the *input_file* of its **.s** or **.860** suffix (if any) and appending **.o**. If a file with the same name as the output module already exists, it is deleted.

-R — Suppress all **.data** directives; all code is assembled in the text segment.

-x — Enable additional checks of the program to find illegal sequences of instructions.

-y — Output in the macro pass special directives that the assembly pass uses for better reporting of the lines in the source file where errors are detected. The **-z** option must also be set.

-z — Pass *input_file* through the macro processor before the first assembly pass. The default is no macro processing.

6.2 Using the Linker

To execute the Linker, enter a system command of the form:

ld860 [*option*]...[[**-B** *bss_addr*] { [**-p**] *in_file* }...]...

The Linker combines all the specified input object files and produces a single output module. This module is directly executable if no references to external symbols remain unresolved. An *option* can be any of the following. Options must be separated from one another by one or more spaces.

-B *bss_addr* — Specify the RAM address to be used for common blocks and for the *bss* section of all *in_file* modules that follow. This specification can be repeated to specify different addresses for different groups of modules. The *bss_addr* is a hexadecimal integer.

-d *integer* — Specify the address at which the data section is to be loaded. The default starting address is 0x1000.

-D *integer* — Specify the length of the data section as *integer* bytes. The data section is padded with NULs to the specified length. The *integer* may not be less than the original length of the data section.

-e *symbol* — Specify *symbol* as the entry-point. The default entry-point is **ld$start.**

-f *list_file* — Read *list_file* for a list of object files to be linked. Names in this file can be separated by comma, SP, TAB, or NEWLINE. Multiple **-f** options are permitted. When an **-f** option is used, the **-B** and **-p** options cannot be used.

-i386 — Put the magic number for the Intel386™ architecture in the output object file instead of the magic number for the i860 Microprocessor, so that 80386-oriented tools can process the object module.

-k Start the text and data sections exactly at the addresses specified by the **-T** and **-d** options (or at the default starting addresses if **-T** or **-d** is not given) without performing the normal modifications to those addresses to make the file pageable.

-M [type] Produce a load map. The *type* can be...

x Include a cross-reference.

The map goes to the standard output unless redirected by > *out_file* immediately following the **-M** option.

-o *out_file* Put the output module in a file named *out_file*. The default name is **a.out**.

-p Align the data section of the following module on a page boundary. This is useful for producing page directories and tables.

-r Retain relocation entries in *out_file*. This enables incremental linking. The *out_file* can be used as input to a subsequent execution of the Linker. When **-r** is used, **-o** must also be used to specify the name of the output file.

-s Strip all symbols from the output object file.

-t Print the name of each *in_file* as it is processed.

-T *integer* Specify the address at which the text section is to be loaded. The *integer* is the starting address. The default starting address is 0xF0400000.

-u *symbol* Initialize the symbol table with *symbol*. The linker considers *symbol* to be undefined; therefore, it may be used to trigger loading of first member of an archive library.

-v Print the names of modules and libraries as they are processed.

The following command line is an example of an invocation of the Linker:

```
ld860 -Mx > map -t -e entry_point -k -T 0xF0400000 -o my_prog
file1.o file2.o
```

Chapter 7
Object File Format

The object files produced by the Assembler and Linker follow the Common Object File Format (COFF) defined in the *Unix System V/386 Programmers' Guide*. Only a few differences are needed for the i860 Microprocessor.

The relocation items, line numbers, symbol table, and string table of an object file for the i860 Microprocessor are optional; they may not appear in all object files.

7.1 File Header

In the file header, the code for an i860 microprocessor target processor is 0x14D (0o515).

7.2 Optional Header

The i860 microprocessor magic number code is 0x14D (0o515).

7.3 Section Headers

Four kinds of sections may appear in object files for the i860 Microprocessor: text, data, *bss*, and *abs*. The sections text, data, and *bss* begin at 16-byte boundaries.

7.4 Relocation Directories

The text directory specifies how relocatable items within the text section should be adjusted when the module is linked. References to external symbols must be defined and references to other segments within the module must be adjusted because the span between segments changes when modules are linked. The data relocation directory contains the same information for the data section.

Table 7-1 defines the relocation types used for i860 microprocessor relocatable symbols. The terms *high* and *low* refer to the high-order and low-order halves of a 32-bit constant. The linker must relocate 32-bit constants in two stages, because instructions of the i860 Microprocessor handle only 16-bit constants.

Table 7-1. Relocation Directory Entry Types

Name	Value (in hex)	Description
PAIR	1C	The low half that follows relates to the preceeding HIGH
HIGH	1E	The high half of a 32-bit constant
LOW0	1F	The low half, instruction bits 15..0, byte aligned
LOW1	20	The low half, instruction bits 15..1, 2-byte aligned
LOW2	21	The low half, instruction bits 15..2, 4-byte aligned
LOW3	22	The low half, instruction bits 15..3, 8-byte aligned
LOW4	23	The low half, instruction bits 15..4, 16- byte aligned
SPLIT0	24	The low half, instruction bits 20..16 and 10..0, byte aligned
SPLIT1	25	The low half, instruction bits 20..16 and 10..1, 2-byte aligned
SPLIT2	26	The low half, instruction bits 20..16 and 10..2, 4-byte aligned
HIGHADJ	27	Similar to HIGH, but needs to be adjusted
BRADDR	28	24/26 bit branch displacement

The HIGH and HIGHADJ types extract the high-order bits of a 32-bit item for use as a 16 immediate. HIGHADJ is similar to HIGH except that the linker may need to adjust the value. Refer to the **ha%** operator in Chapter 2 for a complete explanation of adjustment.

The various LOW and SPLIT types extract the low-order bits. The linker must verify alignment of immediate offsets because the lower bits of the immediate are used to encode the operand length. Refer to Appendix B of the *Programmer's Reference Manual* for an explanation of instruction format.

The SPLIT types are used for the **st** instruction. They cause the immediate to be split between bits 20..16 and 10..0 of the instruction.