Intel® Itanium® Architecture Assembly Language Reference Guide

Order Number: 248801-004

World Wide Web: http://developer.intel.com

Disclaimer

Overview

This document describes the programming conventions used to write an assembly program for the Itanium® architecture.

As prerequisites, you should be familiar with the Itanium architecture, and have assembly language programming experience.

About This Document

This document contains the following sections:

- This section lists related documentation and notation conventions.
- Program Elements Overview describes the basic elements and language specifications of an assembly-language program for the Itanium® architecture.
- Program Structure describes the directives used to structure the program.
- Declarations describes the directives used to declare symbols in the program.
- Data Allocation describes the statements used to allocate initialized and unitialized space for data objects, and align data objects in the program.
- Miscellaneous Directives describes directives not used to structure a program or to declare symbols.
- Annotations describes the assembler annotations.
- Register Names by Type lists the Itanium architecture registers.
- Pseudo-ops lists the Itanium architecture pseudo operations and their equivalent machine instructions, and pseudo-ops with missing operands.
- Link-relocation Operators lists the link-relocation operators and describes their functionality.
- List of Assembly Language Directives lists the assembly language directives according to category.
- Glossary

Related Documentation

The following documents, available at http://developer.intel.com, provide additional information:

- *Intel® Itanium® Architecture Software Developer's Manual*

  Volume 1: Application Architecture, order number 245317-001

  Volume 2: System Architecture, order number 245318-001

  Volume 3: Instruction Set Reference, order number 245319-001

  Volume 4: Itanium Processor Programmer's Guide, order number 245320-001

- *Software Conventions and Runtime Architecture Guide*, order number 245256-002

Notation Conventions

This notation is used in syntax descriptions:

`This type style`	Indicates an element of syntax, a reserved word, keyword, a filename, computer output, or part of a program example. The text appears in lowercase, unless uppercase is significant.
`This type style`	Indicates the text you enter as input.
`This type style`	Indicates a placeholder for an identifier, an expression, a string, a symbol or a value. Substitute one of these items for the placeholder.
`[items]`	Indicates optional items.
`[items | item]`	Indicates the possible choices. A vertical bar (|) separates the items. Choose one of the items enclosed in brackets.

Program Elements Overview

This section describes the basic elements and language specifications of an assembly-language program for the Itanium® architecture. The basic program elements are:

- identifiers
- symbols
- name spaces
- constants
- expressions
- statements.

Identifiers

In Itanium® architecture assembly language, objects such as machine instructions, registers, memory locations, sections in the object file, and constants, have symbolic names. In the source code these names are represented syntactically by identifiers.

An identifier may contain letters, digits, and a few special characters. Identifiers may not begin with a digit.

The following table summarizes the rules for character usage in identifiers.

Character Usage in Identifiers

Character Types	First Characters	Remaining Characters
Letters	a-z or A-Z	a-z or A-Z
Special Characters	`@ _ $ ? .`	`@ _ $ ? .`
Digits	not allowed	0-9

The assembler may place a limit on the length of an identifier, but this limit must be no less than 256 characters.

Name Spaces

There are three classes of names in the Itanium® architecture assembly language:

- Symbols, which refer to memory locations, sections, and symbolic constants. These names are case sensitive.
- Registers, which refer to registers defined in the Itanium architecture. These names are not case sensitive. Some register names consist of multiple syntactic elements rather than a single identifier.
- Mnemonics, which refer to machine instructions, pseudo-ops, directives, and completers. These names are not case sensitive.

The assembler places names in three separate name spaces, according to their class. A name may not be defined twice in the same namespace, but it may be defined once in each namespace. When a name is defined in both the register and symbol namespaces, the register name takes precedence over the symbol unless the identifier is “protected” by terminating it with the `#` operator; this forces the assembler to look up the identifier in the symbol namespace.

The `#` operator in conjunction with a symbol is legal only when the symbol is an operand.

The following examples illustrate the correct use of the # operator:

```
r5:            //label named r5, where label is the symbol name

movl r4=r5#    //;moves the r5 label address to register r4

.global r5#    //declares label r5 as global
```

The `#` operator is unnecessary and illegal when included in the symbol definition, as shown:

```
r5#:           //illegal
```

Symbols

A symbol refers to a location in memory, an object file section, a numeric constant, or a register. A symbol has the following attributes:

- name
- type
- value

The special symbols dollar sign ($) and period (`.`) when used in expressions, always refer to the current location counter. The current location counter points to the address of a bundle containing the current instruction, or to the address of the first data object defined by the current assembly statement. There is no difference between these symbols, either can be used.

In the following example, the `movl` instruction moves the address of the bundle containing the current instruction ($) into register `r1`:

```
movl r1=$
```

In the following data allocation statement, the period (`.`) is the address of the first data object defined by the assembly statement:

```
data4 2, 3, .
```

Symbol Names

Symbol names are case-sensitive identifiers. Symbols whose names begin with a period (`.`) are temporary. Temporary symbols are not placed in the object file symbol table. Symbols whose names begin with two periods (`..`) are temporary, and local. Local symbols are scope restricted symbols. Local symbols are recognized only within the scope in which they are defined. See the Symbol Scope Declaration section for more information about local symbol scopes.

The following table summarizes the rules for using temporary and scope-restricted indicators in different types of symbol names.

Temporary and Scope-restricted Indicators in Symbol Names

Symbol Type	Temporary (.)	Temporary and Scope-Restricted (..)
Labels	Allowed	Allowed
Instruction tags	Allowed	Allowed
Function names	Not allowed	Not allowed
Symbolic constants	Not allowed	Not allowed
Section names	Allowed	Not allowed

Symbols whose names begin with an "at" sign (`@`) are reserved as predefined constants. The assembler provides predefined symbolic constants for special operand values for several instructions, for example, `fclass` and `mux` instructions. The following tables list the predefined symbolic constant names for the operands of these instructions. These symbolic constants can be used in expressions as any user-defined symbolic constant.

`fclass` Condition Predefined Operand Names

Category	`fclass` Conditions	Predefined Name
NaT test	NaT	`@nat`
Sign test	Positive Negative	`@pos` `@neg`
Class test	Normalized	`@norm`
	Unnormalized	`@unorm`
	Signaling NaN	`@snan`
	Quiet NaN	`@qnan`
	Zero	`@zero`
	Infinity	`@inf`

`mux` Bytes Operation Predefined Type Operand Names

`mux` Bytes Operation Type (*`mbtype`*)	Predefined Name
Reverse	`@rev`
Mix	`@mix`
Shuffle	`@shuf`

Alternate	`@alt`
Broadcast	`@brcst`

Symbol Types

A symbol's type indicates the class of object to which it refers. A symbol type can be any of the following:

label	Refers to a location of code or data in memory. A label cannot refer to a procedure entry point. A code label refers to the address of a bundle. An instruction that follows a code label always starts a new bundle. The Bundles section provides more information about instruction bundling.
instruction tag	A symbol that refers to an instruction. An instruction tag is used in branch prediction instructions, and in unwind information directives. Unlike a label, an instruction tag does not cause the instruction to start a new bundle
function name	A symbol that refers to a procedure entry point.
section name	Represents an existing section that is active in the output object file.
symbolic constant	A constant assigned or equated to a number, symbol, or expression

Symbol Values

A symbol is defined when it is assigned a value. A symbol value can also be a number or expression assigned to a symbolic constant. The value of a symbol identifies the object to which it refers. If the symbol refers to a location in memory, the assigned value is the address of that memory location. In most cases, this address is resolved only in link time.

Register Names

All registers have predefined names, which are listed in Appendix A. Predefined register names are not case-sensitive. You can assign new register names to some of the predefined registers with a register assignment statement, or a rotating register directive. See the Assignment Statements, Equate Statements, and Rotating Register Directives sections, for more details. Registers that use the value of a specified general-purpose register as an index into the register file consist of the register file name followed by the name of a general register enclosed in brackets, such as `pmc[r]`.

The assembler determines the register type according to the form of its name, as shown in the following table. Some registers appear in name and number form. For example, `ar.bsp` is the name form of an application register, which also has a number form, `ar17`.

Register Number and Name Forms

Register Form	Register Name	Register Type
Number form	`r0 - r127`	General-purpose 64-bit registers
	`in0 - in95`	
	`loc0 - loc95`	
	`out0 - out95`	
	`f0 - f127`	Floating-point registers
	`p0 - p63`	Predicate registers (1-bit)
	`b0 - b7`	Branch registers
	`ar0 - ar127`	Application registers
	`cr0 - cr127`	Control registers
Name form	e.g. `ar.bsp`	Named application registers
	e.g. `cr.dcr`	Named control registers
	`pr`	All-predicate register (64-bits)
	`pr.rot`	All rotating registers
	`ip`	Instruction pointer
	`psr.l`	Processor-status registers
	`psr.um`	
Indirect file registers	e.g. `pmc[r2]`	Register file with general-purpose register as index.

User-defined registers	`user-name`	Registers assigned new names with assignment statements or rotating register directives.

Mnemonics

Mnemonics are predefined assembly-language names for machine instructions, pseudo-ops, directives, and data allocation statements. Mnemonics are not case-sensitive.

Machine Instruction Mnemonics

Machine instruction mnemonics specify the operation to be performed. For example, `br` is the mnemonic for the branch predict instruction. Some instruction mnemonics include suffixes and optional completers that indicate variations on the basic operation. The suffixes and completers are separated from the basic mnemonic by a period (`.`). For example, the instructions `br.call` (branch call), and `br.ret` (branch return) include suffixes, and are variations of the basic branch (`br)` instruction.

In this manual, completers are italicized to distinguish them from the instruction mnemonic suffixes. For example, in the instruction `brp.ret.sptk.imp b0,L`, the optional completers appear in italics to set them apart from the `.ret` suffix. For a full description of the instructions, see the *Intel® Itanium® Architecture Software Developer's Manual*.

Pseudo-op Mnemonics

Pseudo-op mnemonics represent assembler instructions that alias machine instructions. They are equivalent to instruction mnemonics and are provided for the convenience of the programmer. See Pseudo-ops section for a list of the assembler pseudo-ops.

The following is an example of a pseudo-op:

```
mov r5=2
```

The assembler translates this pseudo-op into the equivalent machine instruction:

```
add1 r5=2,r0
```

For more details about the pseudo-ops, see the *Intel® Itanium® Architecture Software Developer's Manual*.

Directive Mnemonics

Directives are assembler instructions to the assembler during the assembly process; they do not produce executable code. To distinguish them from other instructions, directive mnemonics begin with a period (`.`).

The following sections, Program Structure through Annotations, describe the assembler directives and explain how to use them.

Data Allocation Mnemonics

Data allocation mnenonics specify the types of data objects assembled in data allocation statements. See Data Allocation for a list of these mnemonics. Data allocation statements are used to allocate initialized memory areas.

Constants

Constants can be numeric or string.

- Numeric constants contain integers and floating-point numbers.
- String constants contain one or more characters.

Numeric Constants

A numeric constant contains integer and floating-point numbers. The assembler supports C and Microsoft Macro Assembly language (MASM) numeric constants. C numeric constants are the default.

C Numeric Constants

C numeric constants can be any of the following:

- **Decimal integer constants** (base 10) consist of one or more digits, 0 through 9, where 0 cannot be used as the first digit.
- **Binary constants** (base 2) begin with a `0b` or `0B` prefix, followed by one or more binary digits (0, 1).
- **Octal constants** (base 8) consist of one or more digits 0 through 7, where the first digit is 0.
- **Hexadecimal constants** (base 16) begin with a `0x` or `0X` prefix, followed by a hexadecimal number represented by a combination of digits 0 through 9, and characters A through F.
- **Floating-point constants** consist of:

— an optional sign - or +

— an integer part a combination of digits 0 through 9

— a period .

— a fractional part a sequence of digits 0 through 9

— an optional exponent `e` or `E`, followed by an optionally signed sequence of one or more digits

For example, the following floating-point constant contains both the optional and required parts: `+1.15e-12`.

The following floating-point constant contains only the required parts: `1.0`

The following formal grammar summarizes the rules for the C numeric constants:

```
C-constant:

  C-integer-constant
  floating-point-constant
  character-constant

C-integer-constant:

  [1-9][0-9]*
  0[bB][01]*
  0[0-7]*
  0[xX][0-9a-fA-F]*
```

```
floating-point-constant:
  integer-part.[ fractional-part ] [ exponent-part ]

integer-part:
  [0-9]*

fractional-part:
  [0-9]*

exponent-part:
  [eE][+-][0-9]*
  [eE][0-9]*
```

MASM Numeric Constants

MASM numeric constants can be any of following:

- **Radix constants** are numeric constants that also specify the radix of the value. They consist of one or more digits, 0 through 9, followed by a radix indicator. The radix indicators of MASM numeric constants define them as decimal (D), hexadecimal (H), octal (O), or binary (B). If the current radix is hexadecimal, the letters B and D are interpreted as digits. In this case, T specifies a decimal radix, and Y specifies a binary radix. See MASM Radix Indicators table below.

  Radix indicators are not case-sensitive.

  See the Radix Indicator Directive section for more information about how to specify a radix.

- **Integer constants** in the current radix consist of one or more digits, 0 through 9, A through F. If the current radix is not hexadecimal, the characters A through F are not applicable.
- **Floating-point constants** have the same syntax as in C. See the C Numeric Constants section.

MASM Radix Indicators

Radix	Radix Indicator Suffix
Decimal	`D (d), or T (t)` when the current radix is hex
Hexadecimal	`H (h)`
Octal	`O (o) or Q (q)`
Binary	`B (b), or Y (y)` when the current radix is hex

The following formal grammar summarizes the rules for the MASM numeric constants:

```
MASM-constant:

  MASM-integer-constant
MASM-radix-constant
floating-point-constant
character-constant

MASM-integer-constant:

  [0-9][0-9a-fA-F]*

MASM-radix-constant

 [0-9][0-9a-fA-F]*[tTdDhHOoqQbByY]

floating-point-constant: (as in C)
```

Characters in Numeric Constants

An underscore (_) can be inserted in a numeric constant to improve readability, as follows `1_000_000`. An underscore can be inserted anywhere except before the first character. The assembler ignores underscores.

Characters can represent numeric constants. For instance, a single ASCII character can represent a numeric constant by enclosing it in single quotes (`''`). The numeric constant is the ASCII value of the specified characterise use other special characters to represent numeric constants, use the character escapes defined in the ANSI C language, and enclose them in single quotes. Table below lists the common character escapes. To use the single quote (`'`) to represent a numeric constant, insert a backslash (`\`) before it, and enclose both in single quotes (`''`), as such,`'\'`.

Common Character Escapes

Escape Character	Definition	ASCII Value
`\'`	Single quote	39
`\"`	Double quote	34
`\b`	Backspace	8
`\t`	Tab	9
`\n`	New line	10
`\f`	Form feed	12
`\r`	Carriage return	13
`\\`	Backslash	92
`\num`	Character with octal value num (maximum three digits)	–
`\Xhh`	Character with the hexadecimal value hh (maximum two digits)	–

String Constants

String constants consist of a sequence of characters enclosed in double quotes (`""`).

To specify double-quotes (`"`) in a string constant, insert a backslash (`\`) before it, as such, `"\"`.

To include other special characters in a string constant, use the character escapes defined in the ANSI C language. See table Common Character Escapes for a list of common character escapes.

Expressions

An expression is a combination of symbols, numeric constants, and operators that uses standard arithmetic notation to yield a result. Expressions can be absolute or relocatable.

Absolute Expressions

An expression is absolute when it is not subject to link-time relocation. An absolute expression may contain relocatable symbols, but they must reduce to pairs of the form (`R1 - R2`), where `R1` and `R2` are relocatable symbols defined in the same section in the current source file.

Relocatable Expressions

An expression is relocatable when it is subject to link-time relocation. A relocatable expression contains a relocatable symbol, and may contain an absolute expression. If a relocatable expression contains an absolute expression, it must be reducible to the form (`R+K`), where `R` is either a relocatable symbol defined in the current source file, or an undefined symbol, and `K` is an absolute expression. The address of the relocatable symbol is defined in link time.

Operators

The assembly operators indicate arithmetic or bitwise-logic calculations. Parentheses (()) determine the order in which calculations occur. The assembler evaluates all operators of the same precedence from left to right.

The assembler evaluates all operators according to their level of precedence. Table below lists the operator precedence rules from lowest to highest.

Precedence of Arithmetic and Bitwise Logic Operations

Precedence	Operator Symbol	Operation
0 (Low)	+	Addition
	-	Subtraction
	|	Bitwise inclusive OR
	^	Bitwise exclusive OR
1 (Medium)	*	Multiplication
	/	Division
	%	Remainder
	<<	Shift Left
	>>	Arithmetic shift right

	&	Bitwise A ND
2 (High)	-	Unary negation
	~	Unary one’s complement

Link-relocation Operators

Link-relocation operators generate link-relocation entries in expressions. See Link-relocation Operators for a list of the link-relocation operators.

Statements

An assembly-language program consists of a series of statements separated by a semicolon (;). Multiple statements may be on the same line.

To separate lines, use the standard line termination convention on the local host system, typically CR (carriage return) and LF (line feed). To separate elements within a statement, use the CR, LF, FF (form feed), VT (vertical tab), Space, or Tab that represent white space.

To separate a comment from the code at the end of a statement, insert the comment before the semi colon (;) and precede it with a double-backslash (//). The assembler ignores comments.

The assembler may place a limit on the length of an input line, but this limit must be no less than 256 characters.

The types of assembly-language statements are as follows:

- label statements
- instruction statements
- directive statements
- assignment statements
- equate statements
- data allocation statements
- cross-section data allocation statements

The topics that follow detail each of the statement types, their components and syntax, and provide an example of each.

Label Statements

A label statement has the following syntax:

```
[label]:   // comments
```

Where:

`label`	Defines a symbol whose value is the address of the current location counter. If the assembler inserts padding to align the location counter to an implied alignment boundary, the value of the label is not affected. The assembler interprets a label followed by a double colon (::) as a global symbol. See the Symbol Scope Declaration section for more information about global symbols.

The following is an example of a global label statement:

```
foo::
```

Instruction Statements

An instruction statement has the following syntax:

```
[label:] [[tag:]] [(qp)] mnemonic[.completers]
dests=sources //comments
```

Where:

label	Defines a symbol whose value is the address of a bundle. When a label is present, the assembler always starts a new bundle. If the assembler inserts padding to align the location counter to a bundle boundary, the label is assigned the address of the newly-aligned bundle. The assembler interprets a label followed by a double colon (`::`) as a global symbol. See Symbol Scope Declaration for more information about global symbols.
[tag]	Defines a symbol whose value is the bundle address and slot number of the current instruction.
(qp)	Represents a predicate register symbol, which must be enclosed in parentheses. If this field is not defined, predicate register 0 (`p0`) is the default.
mnemonic.completers	Represents the instruction mnemonic or pseudo-op. Instructions may optionally include one or more completers. Completers must appear in the specified order in the instruction syntax. Mnemonics and completer mnemonics are not case-sensitive. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for a description of the machine instructions, pseudo-ops, and completers.
`dests =sources`	Represents the destination and source operands. The operands are register names, expressions, or keywords, depending on the instruction. Some instructions can have two destination operands, and one or more source operands. When there are multiple operands they are separated by a comma (,). In cases where all operands are destination operands or all operands are source operands, the equal (=) sign is omitted.

The following is an example of an instruction statement with a label and

`(qp)`:

```
L5: (p7) addl r14 = @gprel(L0), r1
```

The following is an example of an instruction statement with a `tag`:

```
[t1:] fclass.m.unc p4, p5 = f6, @pos
```

`@pos` is a predefined constant representing the fclass operation. `p4` is true if `f6` is positive.

Directive Statements

A directive statement has the following syntax:

```
.directive   [operands]   // comments
```

Where:

.directive	Represents the directive mnemonic. Directives always begin with a period (`.`). Directive mnemonics are not case-sensitive.
operands	The operands are optional and determined by the directive. Where multiple operands are present in directives, separate them with commas.

The following is an example of a directive statement:

```
.proc foo
```

Assignment Statements

Assignment statements enable the programmer to define or redefine a symbol by assigning it a value. This value may be a reference to another symbol, register name, or expression. The new value takes effect immediately and remains in effect until the symbol is redefined. Symbols defined in assignment statements do not have forward references.

In addition, symbols defined in assignment statements cannot:

- appear in the symbol table of an output object file.
- be declared global.
- be defined in an equate statement.

There are two types of assignment statements:

- Symbol assignment statements, which define or redefine a symbol in the symbol name space.
- Register assignment statements, which define or redefine a register name in the symbol name space.

Symbol Assignment Statements

A symbol assignment statement has the following syntax:

```
identifier=expression          // comments
```

Where:

identifier	Represents a symbol in the symbol name space.
expression	Specifies the type and value of the identifier. The expression cannot contain forward references.

The following is an example of an assignment statement that defines a symbol:

```
C = L0+2
```

Register Assignment Statements

A register assignment statement has the following syntax:

```
identifier=register name       // comments
```

Where:

identifier	Represents a register name in the symbol name space.

register name	Specifies an alternate register name. If the register name is a stack or rotating register name, the new register name continues to reference the previously-defined register name, even if the name is no longer in effect. See the Register Stack Directive and Rotating Register Directives sections.

The following is an example of an assignment statement that defines a register name:

```
A = r1
```

Equate Statements

Equate statements enable the programmer to define a symbol by assigning it a value. This value may be a reference to another symbol, register name, or expression. In equate statements, a symbol can be defined only once throughout the source file. These symbols may have forward references, except when referencing a register name. A symbol name defined in an equate statement cannot be defined in an assignment statement.

Equate statements have the same syntax as assignment statements, except for the operator.

There are two types of equate statements:

- symbol equate statements
- register equate statements

Symbol Equate Statements

A symbol equate statement has the following syntax:

```
identifier==expression    // comments
```

Where:

identifier	Represents a symbol in the symbol name space.
expression	Specifies the type and value of the identifier. The expression can contain forward references.

The following is an example of an equate statement that defines a symbol:

```
A == 5
```

Register Equate Statements

A register equate statement has the following syntax:

```
identifier==register name // comments
```

Where:

identifier	Represents a register name in the symbol name space.
register name	Specifies an alternate register name. The register name cannot contain forward references. If the register name is a stack or rotating register name, the new register name continues to refer to the previously-defined register, even if the name is no longer in effect. See the Register Stack Directive and Rotating Register Directives sections.

The following is an example of an equate statement that defines a register name:

```
A == r1
```

Data Allocation Statements

A data allocation statement has the following syntax:

```
[label:] dataop operands     // comments
```

Where:

label	Defines a symbol whose value is the address of the first data object defined by the statement. If the assembler inserts padding to align the location counter to an implied alignment boundary, the label is assigned the value of the newly-aligned address. The assembler interprets a label followed by a double-colon (`::`) as a global symbol. See the Symbol Scope Declaration section for more information about global symbols.
dataop	Defines the type and size of data objects that are assembled. Data object mnemonics are not case-sensitive. The Data Allocation Statements section lists the data object mnemonics.
operands	Contain multiple expressions separated by commas. Each expression defines a separate data object of the same type and size. The assembler puts the data objects into consecutive locations in memory, and automatically aligns each to its natural boundary.

The following is an example of a data-allocation statement with a label:

```
L2: data4.ua L1, L1+7, .t1+0x34, $-15
```

Cross-section Data Allocation Statements

A cross-section data allocation statement has the following syntax:

```
xdataop section-name, operands     //comments
```

Where:

xdataop	Defines the type and size of data objects that are assembled. Cross-section data object mnemonics are not case-sensitive.
section-name	Refers to a predefined name of an existing section in the object file.
operands	Contain multiple expressions that are separated by commas. Each expression defines a separate data object of the same type and size. The assembler puts the data objects into consecutive locations in memory, and automatically aligns each to its natural boundary.

The following is an example of a cross-section data allocation statement:

```
.xdata8 .data, 0x123, L1
```

Program Structure

This section describes the Itanium® architecture assembly language directives associated with symbol declarations. These directives can be used to perform the following functions:

- Declare symbol scopes
- Specify symbol types
- Specify symbol sizes
- Override default file names
- Declare common symbols
- Declare aliases for labels, function names, symbolic constants, or sections

---

Sections

The output object file of an assembly program is made up of named sections that contain code and data. The assembler allows any number of sections to be created in parallel within the output object file, one of which can be accessed at a time. The section currently accessed is referred to as the current section.

The assembler maintains a separate location counter for each existing section. The assembler always adds new code or data to the end of the current section, moving the location counter in that section ahead to incorporate the new code or data. The Cross-section Data Allocation Statements section explains how to add data to a section that is not the current section.

Section directives and predefined section directives are used to define and switch between sections. Some section directives have flag and type operands that specify the flag and type attributes of a section.

Section Flags and Section Type Operands

The `flags` operand specifies one or more flag attributes of a section. The `flags` operand is a string constant composed of one or more characters. Table Section Flag Characters lists the valid flag characters. The `flags` operand is case-sensitive. The assembler does not detect invalid specifications made by the programmer, such as stores to a section that is a non-writable section. A non-writable section is not flagged by the `w` flag character.

Section Flag Characters	
Flag Characters	**Description**
`w`	Write access allowed.
`a`	Section is allocated in memory.
`x`	Section contains executable instructions.
`s`	Section contains "short" data.
`o`	Section adds ordering requirement. The '`o`' flag is only for `ELF` (Unix*) files.

The `type` operand specifies a section's type attribute. The `type` operand is a string constant containing one of the valid section types listed in Table Section Types. The section types listed in the table correspond directly to `ELF` (UNIX*) section types, except for the "`comdat`" section type, which corresponds to `COFF32` (Windows NT). The `type` operand is case-sensitive.

Section Types	
Section Type	**Description**
`"progbits"`	Sections with initialized data or code.
`"nobits"`	Sections with uninitialized data (`bss`).
`"comdat"`	`COMDAT` sections, Windows NT specific. See Windows NT (COFF32) Specific Section Flag Operands.
`"note"`	Note sections.

Windows NT (COFF32) Specific Section Flag Operands

In addition to the section flags described in Section Flags and Section Type Operands, the assembler recognizes the flags listed in table COMDAT Section Flag Characters (below) when the section type is `"comdat"` and the object file format is `COFF32` (Windows NT).

These flags represent link-time selection criteria, and are case-sensitive.

COMDAT Section Flag Characters	
Flag	**Description**
`D`	Allow only one instance of this section.
`Y`	Select any one instance of this section.
`E`	Select any one instance of this section; all instances must have identical contents.
`L`	Select the largest instance of this section.
`A`	Select an instance of this section only if the associated section name is selected. See Associated Section Name Flag section that follows.

Associated Section Name Flag

When the `A` flag is present, the assembler identifies an associated section name. Use the `A` flag in conjunction with an associated section operand.

The associated section operand is a section name. A section name can only be loaded in link time if the associated section is already loaded.

To select the `A` flag, use the `.section` or `.pushsection` directive with an additional `assoc-section` operand in one of the following formats:

```
.section section-name [,"flags","type" [,assoc-section]]

.section section-name = "flags","type" [,assoc-section]

.pushsection section-name [,"flags","type"

         [,assoc-section]]

.pushsection section-name = "flags","type" [,assoc-section]
```

Where:

`section-name`	Represents a user-defined name using any valid identifier. Section names are case-sensitive.
`flags`	Represents a string constant composed of

	one or more characters that specify the attributes of a section. See table Section Flag Characters for a list of the valid flag characters.
`type`	Represents a string constant specifying a type attribute of a section. See table Section Types for a list of the section types
`assoc-section`	Represents a user-defined section name.

Section Definition Directive

The `.section` directive defines new sections, switches from one section to another, and sets the current section. The `.section` directive has the following formats, with a different functionality for each format:

```
.section section-name
```

```
.section section-name,"flags","type"
```

```
.section section-name = "flags","type"
```

Where:

`section-name`	Represents a user-defined name using any valid identifier. Section names are case-sensitive.
`flags`	Represents a string constant composed of one or more characters that specify the attributes of a section. See Table Section Flag Characters for a list of the valid flag characters.
`type`	Represents a string constant specifying a type attribute of a section. See Table Section Types for a list of the section types.

In the first format, the `.section` directive sets the ***`section-name`*** as the current section. In the second format, the `.section` directive defines a new section, assigns flags and type attributes, and makes the newly-defined section the current section. If the newly-defined section has the same name, flag attributes, and type attribute as a previously-defined existing section, the assembler switches to the previously-defined section without defining a new one. For example, the following `.section` directive defines a new section (`my_section`), assigns flags (`"aw"`) and type (`"progbits"`) attributes, and makes it the current section.

```
.section my_section, "aw","progbits"
```

In the third format, the .section directive creates a new section with a previously-defined section name, and assigns it new flags and type attributes. The newly-created section becomes the current section; any reference to this section name refers to the newly-created section. The Using Section Directives section illustrates how to use the `.section` directive.

Section Return Directive

The `.previous` directive returns to the previously-defined section of the current section and makes it the current section. This directive does not affect the section stack. The Using Section Directives section illustrates how to use this directive.

Absolute Sections

Absolute sections are only supported by ELF object file formats. To define an absolute section with a fixed starting address, use the `.section` and `.pushssection` directives with an optional *`origin`* operand. The *`origin`* operand must be an absolute expression. Absolute section addresses cannot overlap. The linker does not merge absolute sections with other section types, or with other absolute sections.

The following example defines a new section name and assigns it new *`flags`* and *`type`* attributes, with a starting address specified by the *`origin`* parameter.

```
.section new_name, "aw","progbits",0x1000
```

Section Stack Directives

The assembler maintains a section stack, which is defined by the `.pushsection` and `.popsection` directives. These directives push and pop previously-defined sections to and from the section stack. The assembler may limit the depth of a section stack, but it must allow at least ten levels. The `.pushsection` directive pushes the current section onto the stack and switches to the section specified in the directive. The `.pushsection` directive, like the `.section` directive, has one of the following formats:

`.pushsection` *`section-name`*

`.pushsection` *`section-name,"flags","type"`*

`.pushsection` *`section-name = "flags","type"`*

Where:

`section-name`	Represents a user-defined name using any valid identifier. Section names are case-sensitive.
`flags`	Represents a string constant composed of one or more characters that specify the attributes of a section. See table Section Flag Characters for a list of the valid flag characters.
`type`	Represents a string constant specifying a type attribute of a section. See table SectionTypes for a list of the section types The `.popsection` directive pops the previously-pushed section from the top of the stack, and makes it the current one.

The Using Section Directives section illustrates how to use the `.pushsection` and `.popsection` directives.

Predefined Section Directives

The predefined section directives define and switch between commonly-used sections. A predefined section directive creates a new section with the default *flags* and *type* attributes, and makes that section the current section.

The predefined section directive mnemonics are the same as the section names. The assembler generates section names in lower case, even though directive mnemonics are not case-sensitive.

On some platforms the assembler automatically creates a local symbol with a `"section"` type attribute for each defined section in the object file. See the Symbol Type Directive section for more information about symbol types.

The linker combines sections with the same name, *flags* and `type` attributes. The linker creates two separate output sections for sections with the same name, but different *flags* and *type* attributes.

To define a section without the default flags and type attributes, use the `.section` directive.

The predefined section directives cannot define a new section using the same name as a previously-defined section.

Table Predefined Section Directives below lists the predefined section directives, and their default *flags* and *type* attributes. A predefined section directive can have the same name as a section name.

Predefined Section Directives			
Directive/Section Name	**Flags**	**Type**	**Usage**
`.text`	`"ax"`	`"progbits"`	Read-only object code.
`.data`	`"wa"`	`"progbits"`	Read-write initialized long data.
`.sdata`	`"was"`	`"progbits"`	Read-write initialized short data.
`.bss`	`"wa"`	`"nobits"`	Read-write uninitialized long data.
`.sbss`	`"was"`	`"nobits"`	Read-write uninitialized short data.
`.rodata`	`"a"`	`"progbits"`	Read-only long data (literals). `ELF` (Unix) format only.
`.comment`	`" "`	`"progbits"`	Comments in the object file. `ELF` format, and `COFF` format only when used with the `-Qy` command-line option.

Sections Linking Directive

The `.seclink` directive declares a link between one section to another section. This directive can be used to link an unwind information section with the user-defined executable section.

The `.seclink` directive has the following syntax:

`.seclink` *`section-name, linked-to-section-name`*

Where

`section-name`	Represents the name of a section that links to another section.
`link-to-section-name`	Represents the name of a section the *`section-name`* links to.

Using Section Directives

The following code illustrates the use and behavior of the section directives `.text`, `.section`, `.pushsection`, `.popsection`, and `.previous`:

Example: Code Sequence Using Section Directives

```
.text                //Default

.section A           //Makes A the current section.
                    //.text is A's previous section.

.pushsection B       //Pushes A onto the stack and makes B the
                    //current section. A is B's previous section.

.pushsection C       //Pushes B onto stack and makes C the current
                    //section, B is C's previous section.

.popsection          //Pops B from stack and makes it current.

.popsection          //Pops A from stack and makes it current.
                    //.text is A's previous section.

.previous            //Makes A's previously current section .text the
                    //current section. A becomes .text's previous
                    //section.

.previous            //Makes A the current section, .text becomes A's
                    //previous section.
```

Include File Directive

To include the content of another file in the current file, use the `.include` directive (see Preprocessor Support) or use the `#include` directive of the standard C preprocessor.

To include the contents of another file in the current source file, use the `.include` directive in the following format:

```
.include "filename"
```

Where:

"filename"	Specifies a string constant. If the specified filename is an absolute pathname, the file is included. If the specified filename is a relative pathname, the assembler performs a platform-dependent search to locate the include file.

Bundles

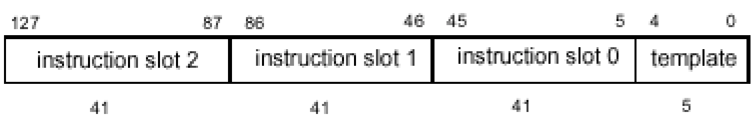
Itanium® architecture instructions are grouped together in 128-bit aligned containers called bundles. Each bundle contains three 41-bit instruction

slots, and a 5-bit template field. The template field specifies which type of execution unit processes each instruction in the bundle. Bit 0 is set to 1 if

there is a stop at the end of a bundle. There is no fixed relation between the boundaries of an instruction group and the boundaries of a bundle.

Figure below illustrates the format of a bundle.

Bundle Format



Multiway branch bundles contain more than one branch instruction. When the first branch instruction of a multiway bundle is taken, the subsequent branch instruction does not execute.

Bundles are always aligned at 16-byte boundaries. The assembler automatically aligns sections containing bundles to at least 16-bytes.

Bundling can be:

- implicit (automatically performed by the assembler)
- explicit (specified by the programmer)

— with automatic selection of the template

— with explicit selection of the template

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for more details about bundles.

Implicit Bundling

The assembler bundles instructions automatically by default.

In the implicit-bundling mode, section directives do not terminate a partially-filled bundle of a previously-defined section. This means that the assembler can return to the previous section and continue to fill the bundle.

In implicit-bundling mode, a label forces the assembler to start a new bundle.

Explicit Bundling

The programmer can explicitly assemble bundles by grouping together up to three instructions, and enclosing them in braces (`{ }`). The assembler places these instructions in one bundle, separate from all preceding and subsequent instructions. Stops at the end of an explicit bundle can be placed before or after the closing brace.

Section directives and data allocation statements cannot be used within an explicit bundle. Cross-section data allocation statements can be used within an explicit bundle. See the Cross-section Data Allocation Statements section for more information.

In explicit-bundling mode, labels can be inserted only as the first statement of an explicit bundle. Instruction tags can be applied to any instruction.

When using explicit-bundling, the appropriate template can be selected in one of the following ways:

- automatically by the assembler.
- explicitly by the programmer, using the explicit-template directives.

Auto-template Selection

By default, the assembler searches and selects a matching template for a bundle. The template fields specify intra-bundle instruction stops. When two templates consist of the same sequence of instruction types, they are distinguished by stops. The assembler selects the appropriate template field based on the stops within the bundle. If no template is found, the assembler produces a diagnostic message. Instruction group stops may occur in a bundle.

Explicit Template Selection

To explicitly select a specific template, use one of the directives listed in table Explicit Template Selection Directive (below) as the first statement of your code within the braces. For example, the `.mii` directive selects the memory-integer-integer (`mii`) template.

Explicit Template Selection Directives

Directive	Template Selection		
	Slot 0	**Slot 1**	**Slot 2**
`.mmi`	memory	integer	integer
`.mfi`	memory	floating point	integer
`.bbb`	branch	branch	branch
`.mlx`	memory	long immediate	
`.mib`	memory	integer	branch
`.mmb`	memory	memory	branch
`.mmi`	memory	memory	integer
`.mbb`	memory	branch	branch
`.mfb`	memory	floating point	branch

`.mmf`	memory	memory	floating point

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for more information about template field encoding and instruction slot mapping.

Note:

Select the `.mlx` directive for the move long immediate instruction and for the long branch instruction. These instructions operate on 64-bit data types and are too large to fit into one of the 41-bit bundle slots. This directive selects the `mlx` template and inserts the instruction in slot 1 and slot 2 of the bundle.

Example below is the code that shows an explicit bundle using explicit template selection, and a stop.

Example: Bundle with Explicit Template Selection and a Stop

```
{.mmi    //use the mmi template for this bundle
m inst  //memory instruction
;;      //stop
m inst  //memory instruction
i inst  //integer instruction
}
```

Instruction Groups

Itanium® architecture instructions are organized in instruction groups. Each instruction group contains one or more statically contiguous instruction(s) that can execute in parallel. An instruction group must contain at least one instruction; there is no upper limit on the number of instructions in an instruction group.

An instruction group is terminated statically by a stop, and dynamically by taken branches. Stops are represented by a double semi-colon (`;;`). The programmer can explicitly define stops. Stops immediately follow an instruction, or appear on a separate line. They can be inserted between two instructions on the same line.

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for more detailed information about instruction groups.

Dependency Violations and Assembly Modes

Dependency violations occur when instructions within an instruction group access the same resource register, including registers that appear as implicit operands. Dependency violations result in architecturally undefined behavior. The assembler can detect and eliminate dependency violations that occur within instruction groups, depending on its mode.

The assembler reads and processes assembly code in one of two modes: explicit and automatic.

Use explicit mode if you are an expert user with profound knowledge of Itanium® architecture or performance is important. In explicit mode, you are responsible for bundling and stops (`;;`), and the assembler generates errors where it finds dependency violations.

Use automatic mode if you are a novice user or performance is not the highest consideration. In automatic mode, the assembler bundles the code and adds stops to avoid dependency violations. It ignores existing stops and annotations.

You can mix code from both modes in the one file. Set the mode using the command-line option or the directives `.auto` and `.explicit`. The directive `.default` causes the assembler to revert to the mode of operation defined in the command line.

For a complete description of the rules of data dependencies, see the *Intel® Itanium® Architecture Software Developer's Manual*.

This feature may not be currently supported by all assemblers.

Procedures

Software conventions require that instructions belong to a declared procedure, and that procedure prologues be separated from the main body within the procedure. These conventions ensure that the proper stack unwind information is placed in the object file. Refer to the *Software Conventions and Runtime Architecture Guide* for details about the software conventions.

Procedure Directives

The `.proc` and `.endp` directives combine code belonging to the same procedure. The `.proc` directive marks the beginning of a procedure, and the `.endp` directive marks the end of a procedure. A single procedure may consist of several disjointed blocks of code. Each block should be individually bracketed with these directives. Name operands within a procedure can be used only for that specific procedure.

The `.proc` directive declares a symbol as a function. The `.proc` directive does not define the symbol by assigning it a value. Symbols must be defined as a label within the procedure. When *`name`* is defined, it is automatically assigned a `"function"` type.

The following code sequence shows the basic format of a procedure:

```
.proc name,...

 name:            //label

...                //instructions in procedure

.endp name, ...
```

Where:

`name`	Represents one or more entry points of the procedure. Each entry point has a different name. The assembler ignores the *`name`* operands of the `.endp` directive.

Procedure Label (PLabel)

When the object file format is `COFF32` (Windows NT), the assembler creates two symbols for a defined procedure. One symbol represents the procedure entry point and appears in the object file symbol table with the original symbol name preceded by a dot. For example, the label named `foo` becomes `.foo` in the object file symbol table. The other symbol represents the procedure label, also referred to as the function descriptor or PLabel, and is implicitly generated by the assembler using the original symbol name. Refer to the *Software Conventions and Runtime Architecture Guide* for more information about the procedure label.

Stack Unwind Directives

Stack unwind directives are used to generate unwind information for a procedure.

The *Software Conventions and Runtime Architecture Guide* describes stack unwind elements and their semantics. Refer to this document for information about the semantics of the stack unwind directives described in this section.

Procedures Used for Stack Unwind Directives

Procedures are bound by the `.proc` and `.endp` directives. See the Procedure Directives section for more information about these directives. Procedures are section-sensitive. The assembler interprets stack unwind directives according to the procedure in which they appear.

Procedures contain prologue and body regions that are divided by headers. These headers are specified using the `.prologue` and `.body` directives.

The `.prologue` directive introduces a prologue region within a procedure. Each prologue region must be introduced by the `.prologue` directive.

The `.body` directive separates the procedure prologue from the main body of the procedure. You can use the `.body` directive more than once within procedures with multiple body regions.

For language specific data, use the `.handlerdata` directive followed by handler data allocations with the `.endp` directive after the handler data allocations. The assembler places the handler data in the `.xdata` section.

See the Stack Unwind Directives Usage Guidelines section for more information about using this directive.

These directives may not be currently supported by all assemblers.

Example below, Procedure Format in a Code Sequence, illustrates the format of a procedure with two prologues, two body regions, and language specific data.

Example: Procedure Format in a Code Sequence
.proc *name*,... //start of procedure .prologue //instructions in first prologue .body //instructions in first body region .prologue //instructions in second prologue .body //instructions in second body region .handlerdata //data allocations go to .xdata section .endp *name*,... //end of procedure

List of Stack Unwind Directives

Stack unwind directives, except for the `.endp` directive, do not break bundles. When a ***tag*** operand is present in a stack unwind directive, the ***tag*** refers to a location of an instruction slot. If the ***tag*** is omitted, the location default is the location counter of the next instruction. More than one directive can refer to the same location of an instruction slot.

Generally, functions have unwind table entries. A stack unwind directive must be present between the `.proc` and `.endp` directives to write function entries and unwind information to the unwind table.

To create a function entry for unwind information when there is no stack unwind information, use the `.unwentry` directive.

The table that follows, Stack Unwind Directives, lists the stack unwind directives and their operands. The right-most column of the table summarizes the records and fields that are affected by these directives. For more information about the affected records and fields, refer to the *Software Conventions and Runtime Architecture Guide*.

Stack Unwind Directives				
Directive Name	**First Operand**	**Second Operand**	**Third Operand**	**Affected Record and Fields**
.proc	*symbol*			entry-start
.endp				entry-end
.handlerdata				handler data allocation
.unwentry				entry generation
.prologue				prologue head previous head
.prologue	*imm-mask*	*grsave*		prologue head previous head
.body				body header previous header
.personality	*symbol*	[*phases*]		personality
.fframe	*size*	[*tag*]		mem_stack_f
.vframe	*gr-location*	[*tag*]		mem_stack_v psp_gr
.vframesp	*spoff*	[*tag*]		mem_stack_v psp_sprel

.vframepsp	*pspoff*	[*tag*]		mem_stak_v pso_psprel
.restore	*sp*	[*ecount*]	[*tag*]	epilogue
.copy_state	state_no			copy_state
.label_state	state_no			label_state
.save	rp	*gr-location*	[*tag*]	rp_when rp_gr
.altrp	*br-location*			rp_br
.savesp rp	rp	*imm-location*	[*tag*]	rp_when rp_sprel
.savepsp rp	rp	*imm-location*	[*tag*]	rp_when rp_psprel
.save	ar.fpsr	*gr_location*	[*tag*]	fpsr_when fpsr_gr
.savesp	ar.fpsr	*imm_location*	[*tag*]	fpsr_when fpsr_sprel
.savepsp	ar.fpsr	*imm_location*	[*tag*]	fpsr_when fpsr_psprel
.save	ar.bsp	*gr_location*	[*tag*]	bsp_when bsp_gr
.savesp	ar.bsp	*imm_location*	[*tag*]	bsp_when bsp_sprel
.savepsp	ar.bsp	*imm_location*	[*tag*]	bsp_when bsp_psprel
.save	ar.bsp store	*gr_location*	[*tag*]	bspstore_when bspstore_gr
.savesp	ar.bsp store	*imm_location*	[*tag*]	bspstore_when bspstore_spre
.savepsp	ar.bsp store	*imm_location*	[*tag*]	bspstore_when bspstore_pspr
.save	ar.rnat	*gr_location*	[*tag*]	rnat_when rnat_gr

.savesp	ar.rnat	*imm_location*	[*tag*]	rnat_when rnat_sprel
.savepsp	ar.rnat	*imm_location*	[*tag*]	rnat_when rnat_psprel
.save	ar.pfs	*gr-location*	[*tag*]	pfs_when pfs_gr
.savesp	ar.pfs	*imm-location*	[*tag*]	pfs_when pfs_sprel
.savepsp	ar.pfs	*imm-location*	[*tag*]	pfs_when pfs_psprel
.save	ar.unat	*gr-location*	[*tag*]	natcr_when natcr_gr
.savesp	ar.unat	*imm-location*	[*tag*]	natcr_when natcr_sprel
.savepsp	ar.unat	*imm-location*	[*tag*]	natcr_when natcr_psprel
.save	ar.lc	*gr-location*	[*tag*]	lc_when lc_gr
.savesp	ar.lc	*imm-location*	[*tag*]	lc_when lc_sprel
.savepsp	ar.lc	*imm-location*	[*tag*]	lc_when lc_psprel
.save	pr	*gr-location*	[*tag*]	preds_when preds_gr
.savesp	pr	*imm-location*	[*tag*]	preds_when preds_sprel
.savepsp	pr	*imm-location*	[*tag*]	preds_when preds_psprel
.save	@priunat	gr_location	[*tag*]	priunat_when priunat_gr
.savesp	@priunat	imm_location	[*tag*]	priunat_when

				priunat_sprel
.savepsp	@priunat	imm_location	[*tag*]	priunat_when priunat_pspre
.save.g	*imm-grmask*			gr_mem spill_imask
.save.g	*imm_grmask*	*gr_location*	[*tag*]	gr_gr imask
.save.f	*imm-frmask*			fr_mem spill_imask
.save.b	*imm-brmask*			br_mem spill_imask
.save.gf	*imm-grmask*	*imm-frmask*		frgr_mem spill_imask
.save.b	*imm-brmask*	*gr-location*		br_gr spill_imask
.spill	*imm-location*			spill_base
.spillreg	*reg*	*treg*	[*tag*]	spill_reg
.restorereg		*reg*	[*tag*]	spill_reg
.spillsp	*reg*	*imm_location*	[*tag*]	spill_sprel
.spillpsp	*reg*	*imm_location*	[*tag*]	spill_psprel
.spillreg.p[1]	*qp*	*reg*	*treg*	spill_reg_p
.restorereg.p	*qp*	*reg*	*[tag]*	spill_reg_p
.spillsp.p	*qp*	*reg*	*imm_location*	spill_sprel_p
.spillpsp.p	*qp*	*reg*	*imm_location*	spill_psprel_
.unwabi	*os-type*	*imm_context*		abi

[1] .spillreg.p, .spillsp.p, and .spillpsp.p have an optional fourth operand: [*tag*].

Stack Unwind Directives Operands

The following alphabetical list defines the stack unwind directive operands listed in the table Stack Unwind Directives:

- `ar.pfs`, `ar.unat`, and `ar.lc` are explicit register names.
- `br-location` is the alternative branch register used to get the return link. By default, `b0` is the return link.
- `ecount` is the number of prologues -1 specified by the assembler if this field is not specified by the user.
- `gr-location` is a general-purpose register that specifies the destination of the save operation. For example, registers `r1` and `loc1`.
- `grsave` saves the `rp`, `ar.pfs`, `psp`, and `pr` register contents to the first general-purpose register.
- `imm-location` (immediate location) is the offset between the `sp` or `psp`, and the *`save_address`*, specified in bytes. This offset is always positive and specified as follows:
- *`imm-mask`* (immediate mask) is an integer constant specifying a bit pattern for the preserved registers, as follows:

  — The immediate mask *(`imm-mask`*) of the `.prologue` directive is specified as follows: `rp` (return link) (bit 3), `ar.pfs` register (bit 2), `psp` (previous stack pointer) (bit 1), `pr` register (bit 0)

  — The immediate mask *(`mm-frmask`*) of the `.save.f` and `.save.gf` directives refer to the preserved floating-point registers.

  — The immediate mask *(`imm-grmask`*) of the `.save.g` and `.save.gf` directives refer to the preserved general registers.

  — The immediate mask (`imm-brmask`) of the `.save.b` directive refers to the preserved branch registers.

- *`os-type`* is one of `@svr4`, `@hpux`, or `@nt`. It specifies the operating system type.
- *`phases`* is the number of phases, ranging from 0 to 3.
- `pr` is an explicit register name.
- `@priunat` is a predefined symbol and indicates a primary `unat`.
- *`psp`* is the location of the previous stack frame.

- *`psp_offset`*: `imm-location = psp_address - save_address`. See also *`imm-mask`*.
- `qp` is one of the following predicate registers: `p1-p63`.
- `reg` is one of the following registers: `r4-r7`, `f2-f5`, `f16-f31`, `b1-b5`, `pr`, `@psp`, `@priunat`, `rp`, `ar.bsp`, `ar.bspstore`, `ar.rnat`, `ar.unat`, `ar.fpsr`, `ar.pfs`, or `ar.lc`.
- `rp` is an explicit register name.
- *`size`* is the fixed frame size in bytes.
- `sp` is an explicit register name.
- *`sp_offset`*: `imm-location = save_address - sp_address`. See also *`imm-mask`*.
- `state_no` is the state copied or restored.
- *`symbol`* is an assembly label.
- *`tag`* is an optional operand, which specifies a `"when"` attribute of the operation described by the directive.
- `treg` is one of the following registers: `r1-r127`, `f2-f127`, or `b0-b7`.

Syntax for the .save.x Directives

The directives, `.save.f`, `.save.g`, `.save.gf`, and `.save.b`, define 2-bit fields for each save operation in the imask descriptor. The assembler interprets the instruction that immediately follows a save directive as a save instruction.

Example Code Sequence Using the .save.g Directive illustrates the use of the `.save.g` directive. Each `.save.g` directive describes the subsequent store instruction. The operand is a mask where only one bit is set. This bit specifies the preserved saved register. The assembler produces a `gr_mem` descriptor with a 0x5 mask. In addition, the assembler marks the 2-bit fields of the imask descriptor, corresponding to the slots of the two store instructions.

Example: Code Sequence Using the `.save.g` Directive

```
.save.g 0x1
st8... = r4
...
.save.g 0x4
st8... = r6
```

Example Code Sequence Using the .save.gf Directive illustrates the use of the `.save.gf` directive. The `.save.gf` directive describes the subsequent store instruction. The operands is a mask where only one bit is set. This bit specifies the preserved saved register. The assembler produces a frgr_mem descriptor with a 0x42 mask for the floating-point registers and a 0x2 mask for the general-purpose registers. In addition, the assembler marks the 2-bit fields of the imask descriptor, corresponding to the slots of the three store instructions.

Example: Code Sequence Using the `.save.gf` Directive

```
.save.gf 0, 0x2
fst... = f3
...
.save.gf 0, 0x40
fst... = f18
...
.save.gf 0x2, 0
st8... = r5
```

Stack Unwind Directives Usage Guidelines

Follow these guidelines when using the stack unwind directives:

- Place stack unwind directives between the unwind entry point of the function declared in `.proc` and `.endp`.
- The first directive in each region in a procedure must be one of the following region header directives, `.prologue` or `.body`.
- The first directive in the procedure must point to the same address as the first unwind entry point of the function.
- No two consecutive prologue regions are allowed.
- When none of the stack unwind directives listed in the Stack Unwind Directives table are specified, optionally use the `.unwentry` directive to create an unwind entry for the function. Do not use this directive if the unwind records are filled by the compiler.
- Use tags only within the current region. A tag operand cannot be specified out of the scope region. If a tag is omitted, the directive refers to the next instruction, which resides in the same region.
- Use only one `.personality` directive at any point within each procedure.
- Always precede the `.handlerdata` directive with the `.personality` directive.
- Follow these guidelines for prologue regions:

— Use one of the following frame directives if the procedure creates a new stack frame: `.fframe`, `.vframe`, or `.vframesp`.

— Use each of the `.save` directives only once. For example: `.save rp`, `ar.pfs`, `ar.unat`, `ar.lc`, and `pr`.

— Multiple usage of the directives, `.save.g`, `.save.f`, `.save.b`, and `.save.gf` is allowed. The number of bits set in the bit-mask operand specifies the number of the consecutive save instructions that immediately follow the directive.

— A single unwind record is built for one or more occurrences of the following directives: `.save.g`, `.save.f`, `.save.b`, and `.save.gf`. The bit-mask field of the record is a bitwise OR of all the masks that appear in the directives.

— Use only one `.save.b` with the `gr-location` operand.

— Use only one `.spill` directive.

— The `.prologue <imm_mask>` directive with the `psp` bit set and the `.vframe` directive both define the `psp` location. Use only one of them.

- Use only one `.restore` directive for body regions.

Using Stack Unwind Directives Example

The example below is a simple “Hello World” function that shows the usage of local and output registers. For comparison, the first part (A) does not include unwind directives, and the second part (B) includes stak unwind directives and DV detection clues.

Using Unwind Directives

A. "Hello World" Function Without Unwind Directives

```
// The string is defined in the read only data section
.section .rdata, ”a”, ”progbits”
.align 8
.STRING1:
stringz ”Hello World!!!\n”

// The definition of the function hello is in the text section
// The following registers are saved in local registers:
//    gp = r1   - loc0 = r32
//    rp = b0   - loc1 = r33
//    ar.pfs    - loc2 = r34
//    sp = r12 - loc3 = r35
.text
.global hello
.proc hello
hello:
alloc loc2 = ar.pfs, 0, 4, 1, 0
mov loc3 = sp
mov loc1 = b0
addl out0 = @ltoff(.STRING1), gp
;;
ld8 out0 = [out0]
mov loc0 = gp
br.call.sptk.many b0 = printf
;;
mov gp = loc0
mov ar.pfs = loc2
mov b0 = loc1
mov sp = loc3
br.ret.sptk.many b0
.endp hello

.global printf
.type printf, @function
```

B. "Hello World" Function With Unwind Directives

```
    .file ”hello.c”
.pred.safe_across_calls p1-p5,p16-p63

.section .rdata, ”a”, ”progbits”
.align 8
.STRING1:
stringz ”Hello World!!!\n”
.text
.align 16
.global hello#
```

```
.proc hello#
hello:
.prologue
.save ar.pfs, r34
alloc r34 = ar.pfs, 0, 4, 1, 0
.vframe r35
mov r35 = r12
.save rp, r33
mov r33 = b0
.body
addl r36 = @ltoff(.STRING1), gp
;;
ld8 r36 = [r36]
mov r32 = r1
br.call.sptk.many b0 = printf#
;;
mov r1 = r32
mov ar.pfs = r34
mov b0 = r33
.restore sp
mov r12 = r35
br.ret.sptk.many b0
.endp hello#

.global printf#
.type printf#, @function
```

Windows NT (COFF32) Symbolic Debug Directives

When the object file format is COFF32 (Windows NT), the symbolic debug directive `.ln` stores the line number table entry of a function in the symbolic debug information. The symbolic debug directive `.ln` must be enclosed within a function defined by the `.bf` and `.ef` directives. The `.bf` and `.ef` directives define the beginning and the end of a function.

The `.ln` directive has the following format:

```
.ln   line-number[,function]
```

Where:

line-number	Specifies the source line number associated with the next assembled instruction.
function	Is the name of the current function.

The `.bf` and `.ef` directives have the following format:

```
.bf   function,line
```

```
.ef   function,line,code-size
```

Where:

function	Represents the function name.
line	Is an integer number corresponding to the first source line of the function.
code-size	Is an integer number representing line group code size, which is written as debug information.

Declarations

This section describes the Itanium® architecture assembly language directives associated with symbol declarations. These directives can be used to perform the following functions:

- Declare symbol scopes
- Specify symbol types
- Specify symbol sizes
- Override default file names
- Declare common symbols
- Declare aliases for labels, function names, symbolic constants, or sections

---

Symbol Scope Declaration

Symbols are declared as global, weak, or local scopes. Symbol scopes are used to resolve symbol references within one object file or between multiple object files. The symbol scope attribute is placed in the object file symbol table and any reference to a symbol is resolved in link time. By default, symbols have a local scope, where they are available only to the current assembly-language source file in which they are defined.

Local Scope Declaration Directive

References to symbols with a local scope are resolved from within the object file in which the symbols are declared. Local symbols with the same name in different object files do not refer to the same entity. Symbols have a local scope by default, so it is not necessary to declare symbols with local scopes. However, the `.local` directive is available for completeness. The `.local` directive has the following format:

```
.local   name,name, ...
```

Where:

name	Represents a symbol name.

Global Scope Declaration Directive

References to symbols with a global scope are resolved within the object file in which the symbols are declared, and within other object files. Global symbols with the same name in different object files refer to the same entity.

To declare one or more symbols with a global scope, use the `.global` directive. These symbols are flagged as global symbols for the linkage editor. The `.global` directive has the following format:

```
.global   name,name, ...
```

Where:

name	Represents a symbol name.

Weak Scope Declaration Directive

References to symbols with a weak scope are resolved within the object file in which the symbols are declared, and within other object files. Weak symbols with the same name in different object files may not refer to the same entity. When a symbol name is declared with a weak scope as well as a global or local scope, the global or local scope will take precedence over the weak scope in link time.

To declare one or more symbols with a weak scope, use the `.weak` directive. These symbols are flagged as weak symbols for the linkage editor. The weak scope declaration format for UNIX* (`ELF`) and Windows NT (`COFF32`) differ and are described in the sections that follow.

Weak Scope Declaration for UNIX (ELF)

For UNIX (`ELF`), use the `.weak` directive in the following format:

```
.weak   name1,name2, ...
```

Where:

name	Represents a symbol name.

The following example illustrates how to declare an undefined symbol with a weak scope. The defined symbol `x:` has a local scope. `y` has the attributes of `x` and has a local scope. The symbol `y` can then be declared with a weak scope using the `.weak` directive while keeping the other attributes of `x`.

```
x:
 y == x
 .weak y
```

Weak Scope Declaration for Windows NT (COFF32)

For Windows NT (`COFF32`), use the `.weak` directive in the following format to declare a symbol with a weak scope and search for defined symbols within other object files and libraries:

```
.weak   identifier1 = identifier2
```

Where:

identifier1	Represents a symbol name that is assigned a weak symbol scope, which is resolved in link time.
identifier2	Represents a symbol name that holds the symbol definition.

Use the following syntax to declare a symbol with a weak scope and search for defined symbols within other object files and **not** within libraries:

```
.weak    identifier1 == identifier2
```

*W*here:

identifier1	Represents a symbol name that is assigned a weak symbol scope, which is resolved in link time.
identifier2	Represents a symbol name that holds the symbol definition.

The following example illustrates a weak scope declaration where `x:` is a local defined symbol. `x` is the associated symbol for `y`. The `.weak` directive assigns `y` a weak scope.

```
x:
.weak y = x
```

Symbol Visibility Directives

A symbol's visibility, although it may be specified in an object file, defines how that symbol may be accessed once it has become part of an executable or shared object.

The default visibility of symbols is specified by the symbol's binding type. That is, global and weak symbols are visible outside of their defining component (executable file or shared object). Local symbols are hidden, as described below.

A symbol defined in the current executable file or shared object is protected if it is visible in other components but not preemptable. Preemptable means that any reference to such a symbol from within the defining component must be resolved to the definition in that component, even if there is a definition in another component.

A symbol defined in the current component is hidden if its name is not visible to other components. Such a symbol is necessarily protected. This directive may be used to control the external interface of a component.

To declare one or more symbols as protected, use the `.protected` directive. These symbols are flagged as protected symbols for the linkage editor. The `.protected` directive has the following format:

`.protected` *`name,name, ...`*

Where:

name	represents a symbol name.

To declare one or more symbols as hidden, use the `.hidden` directive. These symbols are flagged as hidden symbols for the linkage editor. The `.hidden` directive has the following format:

`.hidden` *`name,name, ...`*

Where:

name	represents a symbol name.

To declare one or more symbols as exported, use the `.export` directive. These symbols are flagged as exported symbols for the linkage editor. The `.export` directive has the following format:

`.export` *`name,name, ...`*

Where:

name	represents a symbol name.

Symbol Type Directive

The default type of a symbol in an object file is based on the assembly-time type of the symbol. See table Symbol Types below for a list of the symbol types and their predefined names. To explicitly specify a symbol's type, use the .type directive in the following format:

```
.type    name,type
```

Where:

name	Represents a symbol name.
type	Specifies the symbol type using one of the predefined symbols listed in table Symbol Types that follows.

Symbol Types	
Symbol Types	**Predefined Symbol Name of Type**
Symbolic constants and undefined symbols	`@notype`
Labels and common symbols	`@object`
Function names	`@function`
Section names	Created by the assembler.

Note:

The assembler automatically creates a symbol of type `name` for section names. When the object file format is `COFF32` (Windows NT) the assembler creates a function symbol name for `@function`. For more information see the Procedure Label (PLabel) section.

Symbol Size Directive

To explicitly specify the size attribute of a symbol, use the `.size` directive. The `.size` directive has the following format:

```
.size   name,size
```

Where:

name	Represents a symbol name.
size	Represents an absolute integer expression with no forward references.

To implicitly specify the default size attribute of a symbol, use a data allocation statement. The default symbol size is written to the symbol table. See the Data Allocation Statements section for more information.

Note:

When the object file format is `COFF32` (Windows NT), the `.size` directive is only effective for common symbols.

File Name Override Directive

By default, the file name is the name of the source file. To override the default file name use the `.file` directive. If you use the `.file` directive more than once in a source file, the assembler places multiple file names in the output object file. The `.file` directive has the following format:

```
.file   "name"
```

Where:

name	Represents a string constant specifying a source file name.

Common Symbol Declarations

Common and local common symbol declarations enable you to define a symbol with the same name in different object files. The difference between a common symbol and local common symbol is as follows:

- The linker merges two or more common symbol declarations for the same symbol.
- The assembler merges two or more local common symbol declarations for the same symbol.

If a symbol is declared as both common and local common, the common declaration overrides the local common declaration. Any definition of a symbol supersedes either type of common declaration.

Common Symbol Directive

To declare a symbol as a common symbol, use the `.common` directive. Common symbols have a global scope, and do not necessarily have the same size and alignment attributes. The `.common` directive has the following format:

`.common` ***`name,size,alignment`***

Where:

`name`	Represents a symbol name.
`size`	Represents an absolute integer expression.
`alignment`	Represents an absolute integer expression to the power of two. Not supported in COFF32 format.

Note:

When the object file format is COFF32 (Windows NT), the alignment operand is not supported.

Local Common Symbol Directive

To declare a symbol as a local common symbol use the `.lcomm` directive. The `.lcomm` directive has the following format:

`.lcomm` ***`name`***`,size,`***`alignment`***

Where:

`name`	Represents a symbol name.
`size`	Represents an absolute integer expression.
`alignment`	Represents an absolute integer expression to the

	power of two.

The assembler allocates storage in the `.bss` or `.sbss` sections for undefined symbols declared as local `common`. The `.bss` or `.sbss` sections are chosen according to the size of the local common symbol. The assembler defines the symbol with the relocatable address of the allocated storage. The symbol is declared with a local scope, and assigned the largest size and alignment attributes of the local common declarations for that symbol.

Alias Declaration Directives

The `.alias` directive declares an alias for a label, a function name, or a symbolic constant. This directive can be used to reference an external symbol whose name is not legal in the assembly language. The `.alias` directive has the following format:

```
.alias  symbol,"alias"
```

Where:

symbol	Represents a symbol name that the assembler can recognize. This name must be a valid name for the type of symbol.
"alias"	Represents a string constant, which is the name the assembler exports to the object file symbol table.

The `.secalias` directive declares an alias for a section name. This directive can be used to reference an external section whose name is not valid in the assembly language.
The `.secalias` directive has the following format:

```
.secalias  section-name,"alias"
```

Where:

section-name	Represents a section name that the assembler can recognize. This name must be a valid name for the type of section.
"alias"	Represents a string constant, which is the name the assembler exports to the object file symbol table.

Data Allocation

This section describes the Itanium® architecture assembly language statements used to allocate initialized and unitialized space for data objects in current sections and in cross sections, and to align data objects in sections of the code.

Data Allocation Statements

Data allocation statements allocate space for data objects in the current section, and initialize the space by assigning it a value. Data objects can be integer numbers, floating-point numbers, or strings. Integer numbers and floating point numbers are aligned according to their size. A data allocation statement with a *`label`*, defines a symbol of `@object` type, and sets the size attribute for that symbol.

Data allocation statements have any of the following formats:

```
[label: ]   data1     expression,   ...

[label: ]   data2     expression,   ...

[label: ]   data4     expression,   ...

[label: ]   data8     expression,   ...

[label: ]   data16    expression,   ...

[label: ]   real4     expression,   ...

[label: ]   real8     expression,   ...

[label: ]   real10    expression,   ...

[label: ]   real16    expression,   ...

[label: ]   string    "string",     ...

[label: ]   stringz   "string",     ...
```

Where:

`label`	Specifies the data allocation address of the first data object.
`expression`	Represents any of the valid expression types listed in the Data Allocation Statements table, see below. Data alocation statements can have more than one epression operand.
`string`	Represents any of the valid string expression type values listed in the Data Allocation Statements table, see below.

The table below summarizes the data allocation mnemonics, and their expression type, memory format, data-object size, and alignment boundary for each.

Data Allocation Statements

Mnemonic	Expression Type	Memory Format	Size (in bytes)	Alignment
`data1`	Integer	Integer	1	1
`data2`	Integer	Integer	2	2
`data4`	Integer	Integer	4	4
`data8`	Integer	Integer	8	8
`data16`	Integer	Integer	16	16
`real4`	Floating point or Integer	IEEE single-precision floating point	4	4
`real8`	Floating point or Integer	IEEE double-precision floating point	8	8
`real10`	Floating point or Integer	IEEE extended-precision floating point (80-bit)	10	10
`real16`	Floating point or Integer	IEEE extended-precision floating point (80-bit)	16	16
`string`	String constant	Array of ASCII characters	Length of string	1
`stringz`	String constant	Array of ASCII characters with null terminator	Length of string + 1	1

To disable the automatic alignment of data objects in data allocation statements, add the `.ua` completer after the mnemonic, for example, `data4.ua`. These statements allocate unaligned data objects at the current location within the current section.

The default byte order for data allocation statements is platform dependent. To specify the byte order for data allocation statements, use the `.msb`, or `.lsb` directives described in the Byte Order Specification Directives section.

Uninitialized Space Allocation

The `.skip` and `.org` statements reserve uninitialized space in a section without assigning it a value. The `.skip` and `.org` statements enable the assembler to reserve space in any section type, including a `"nobits"` section. During program execution, the contents of a `"nobits"` section are initialized as zero by the operating system program loader. When using the `.skip` and `.org` statements in any other section type, the assembler initializes the reserved space with zeros.

The `.skip` statement reserves a block of space in the current section. The size of the block is specified in bytes, and is determined by an expression operand. The expression operand specifies the size of space reserved in the current section. The `.skip` statement with a label, defines a symbol of `@object` type, and sets the size attribute for that symbol.

The `.skip` statement has the following format:

```
[label: ]  .skip expression
```

Where:

label	Specifies the data allocation address of the beginning of the reserved block.
expression	Represents an absolute integer expression with no forward references. The location counter advances to a location relative to the current location within the section. This operand cannot have a negative value since the location counter cannot be reversed.

The `.org` statement reserves a block of space in the current section. The `.org` statement advances the location counter to the location specified by the expression operand. The `.org` statement with a label defines a symbol of `@object` type, and sets the size attribute for that symbol. The `.org` statement has the following format:

```
[label:]  .org expression
```

Where:

label	Specifies the data allocation address of the beginning of the reserved block.
expression	Represents an integer, or a relocatable expression, with no forward references. If the expression is relocatable, it must be reducible to the form `R+K`, where `R` is a symbol previously defined in the current section, and `K` is an absolute constant. The location counter is set to the indicated offset relative to the beginning of the section. Since the location counter cannot be reversed, this operand must be greater than, or equal to, the current

	location counter.

Alignment

Instructions and data objects are aligned on natural alignment boundaries within a section. To disable automatic alignment of data objects in data allocation statements, add the `.ua` completer after the data allocation mnemonic, for example, `data4.ua`. Bundles are aligned at 16-byte boundaries, and data objects are aligned according to their size. The assembler does not align string data, since they are byte arrays.

Each section has an alignment attribute, which is determined by the largest aligned object within the section.

Section location counters are not aligned automatically. To align the location counter in the current section to a specified alignment boundary use the `.align` statement. The `.align` statement has the following format:

```
.align     expression
```

Where:

`expression`	Is an integer number that specifies the alignment boundary of the location counter in the current section. The integer must be a power of two.

The `.align` statement enables the assembler to reserve space in any section type, including a `"nobits"` section. During program execution time the contents of a `"nobits"` section are initialized as zero by the operating system program loader. When using the `.align` statement in any other section type, the assembler initializes the reserved space with zeros for non-executable sections, and with a NOP pattern for executable sections.

Note:

When the object file format is `COFF32` (Windows NT) the section alignment boundary is limited to 8KB. The assembler does not guarantee alignment for requests above 8KB.

Cross-section Data Allocation Statements

Cross-section data allocation statements add data to a section that is not the current section. These statements save the overhead of switching between sections using the `.section` directive. See the Sections section for more information about switching between sections. Cross-section data allocation statements may be used within an explicit bundle. All data objects are aligned to their natural boundaries in the cross section. Cross-section data allocation statements have any of the following formats:

```
.xdata1    section,expression, ...

.xdata2    section,expression, ...

.xdata4    section,expression, ...

.xdata8    section,expression, ...

.xstring   section,"string",   ...

.xstringz  section,"string",   ...
```

Where:

section	Represents the name of a previously-defined section that is not the current section.
expression	Represents an absolute or relocatable integer expression. When these expressions reference a location counter, they refer to the location counter within the cross section, not within the current section.
string	Represents any of the valid string expression type values listed in the Data Allocation Statements table .

To disable automatic alignment of data objects in a cross-section data allocation statement, add the `.ua` completer to the statement, for example, `.xdata4.ua`. These statements allocate unaligned data objects at the current location counter of the cross section, not the current section.

The default byte order for cross-section data allocation statements is platform dependent. The byte order is determined by the cross section, not by the current section.

Miscellaneous Directives

This section describes the following Itanium® architecture assembly language directives:

- Register stack directive
- Rotating register directives
- Byte-order specification directive
- Ident string specification directive
- Radix indicator directive
- Preprocessor support

Register Stack Directive

The Itanium® architecture provides a mechanism for register renaming. Register renaming is implemented by allocating a register stack frame consisting of input, local, and output registers. These registers can be renamed. These renamable registers map to the general registers `r32` through `r127`. The assembler provides predefined alternate register names for the input, local, and output register areas of the register stack frame. The mapping of these registers to the general registers is determined by the nearest preceding `alloc` instruction.

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for detailed information about register renaming and for a full description of the `alloc` instruction.

The `.regstk` directive replaces the default register mappings defined by a preceding `alloc` instruction with new mappings. The `.regstk` directive does not allocate a new register stack frame.

The `.regstk` directive has the following format:

```
.regstk   ins, locals, outs, rotators
```

Where:

ins	Represents the number of input registers in the general register stack frame. `in0` through in_{ins-1} represent `r32` through r_{31+ins} for `ins` > 0.
locals	Represents the number of local registers in the general register stack frame state. `loc0` through $loc_{locals-1}$ represent r_{32+ins} through $r_{31+ins+locals}$ for `locals` > 0.
outs	Represents the number of output registers in the general register stack frame. `out0` through out_{outs-1} represent $r_{32+ins+locs}$ through $r_{31+ins+locals+outs}$ for `outs` > 0.
rotators	Represents the number of rotating registers in the general register frame. `rotators` must be <= *ins+locals+outs*.

The `in`, `loc`, and out register names defined by a previous `.regstk` directive or `alloc` instruction are visible by all subsequent instructions until the next `.regstk` directive or `alloc` instruction is specified.

The alternate register names specified by the operands of the `.regstk` directive refer to registers in the current register stack frame. If you reference input, local, or output registers using the alternate register names that are not within the current stack frame, the assembler produces an error message.

To prevent referencing the alternate register names, use the `.regstk` directive without the operands. The operands of a subsequent `.regstk` directive or `alloc` instruction redefine the mappings of the alternate register names.

The `alloc` instruction and `.regstk` directive do not affect the names of the general registers, `r32` through `r127`.

Stacked Registers in Assignment and Equate Statements

To define an alternate register name for a stacked register, use an assignment statement. The alternate register name is not affected by any subsequent changes to the rotating register. See the Assignment Statements and Equate Statements sections for more details about assignment and equate statements.

Example Defining a Stacked Register in an Assignment Statement illustrates how to define an alternate register name using an assignment statement, so that the alternate register name is not affected by a subsequent `.regstk` directive. The local register name `loc0` maps to the general register `r36`. `loc0` is assigned to `tmp`. The subsequent `add` instruction refers to `loc0`, which is currently mapped to `r40`. The next `add` instruction refers to `tmp` which is mapped to `r36`, not `r40`.

Example: Defining a Stacked Register in an Assignment Statement

```
    .regstk   4,4,2,0
tmp = loc0                  //loc0 is currently r36
   ...
   .regstk   8,1,3,0
   add       loc0 = r1,r7  //loc0 is currently r40
   add       r1 = r2,tmp   //tmp = r36!
```

Rotating Register Directives

General registers, floating-point registers, and predicate registers contain a subset of rotating registers. This subset of rotating registers can be renamed.

The following directives enable the programmer to provide names for one or more registers within each rotating register region:

- `.rotr` for general registers
- `.rotf` for floating-point registers
- `.rotp` for predicate registers

The `.rotx` directives assign alternate names and generation numbers for the rotating registers. One generation corresponds to one iteration of a software-pipelined loop. Each copied register is numbered with an index, where the most recent copy of a register has a zero index, such as `b[0]`. For every loop iteration, the registers within the group are renamed, and become one generation older by incrementing the index by one.

The `.rotx` directives define the number of instances of each pipeline variable and allocate them in the appropriate rotating register region. You can use an arbitrary name with a subscript-like notation for referencing the current and previous generations of each variable.

The rotating register directives have the following format:

```
.rotr    name [expression], ...

.rotf    name [expression], ...

.rotp    name [expression], ...
```

Where:

`name`	Represents a register name specified by the user, and represents a pipelined variable.
`expression`	Specifies the number of generations needed for the variable. The expression must be an absolute integer expression with no forward references.

When the alias rotating register names are used as instruction operands, they have the following format:

```
name[expression]
```

Where:

`name`	Represents an alias rotating register name defined by one of the rotating register directives.

`expression`	Represents an absolute integer expression with no forward references. The index must be between `0` and `(n-1)`, where *n* is the number of generations defined for that name. If the index is negative, or greater than `(n-1)`, the assembler produces an error message.

The `.rotr`, `.rotf`, and `.rotp` directives cancel all previous alias names associated with the appropriate register file, before defining new register names. The register files include the general, floating-point, and predicate registers.

If the number of rotating general registers implied by a `.rotr` directive exceeds the number of rotating registers declared by the nearest preceding `alloc` instruction, or `.regstk` directive, the assembler issues a warning.

Using Rotating Register directives

Examples Using the .rotp Directive and Using the .rotf Directive illustrate the behaviour of the `.rotp` and `.rotf` directives, respectively.

Example Using the .rotp Directive illustrates how the `.rotp` directive declares alternate rotating predicate register names for two predicate registers, `p[2]`, and three predicate registers `q[3]`. Instructions subsequent to the `.rotp` directive refer to `p[0]` for the current generation of `p`, and `p[1]` for the previous generation of `p`. For the current generation of `q`, the subsequent instructions refer to `q[0]`, `q[1]` for the previous generation, and `q[2]` for the one before the previous generation.

Example: Using the .rotp Directive

```
.rotp  p[2],q[3]
//The alternate predicate register names map to the
// predicate registers as follows:

p[0] = p16; p[1] = p17
q[0] = p18; q[1] = p19; q[2] = p20
```

Example Using the .rotf Directive illustrates how the `.rotf` directive declares alternate floating-point register names for three floating-point registers `x[3]`, two floating-point registers `y[2]`, and three floating-point registers `z[3]`.

Example: Using the .rotf Directive

```
.rotf  x[3],y[2],z[3]

//The alternate floating-point register names map to the
//floating-point registers as follows:

x[0]=f32;x[1]=f33;x[2]=f34
y[0]=f35;y[1]=f36
z[0]=f37;z[1]=f38;z[2]=f39
```

Rotating Registers in Assignment and Equate Statements

To define an alias name for a rotating register, use an assignment statement. The alias register name is not affected by any subsequent changes to the rotating register. See the Assignment Statements and Equate Statements sections for more details about assignment and equate statements.

Example Defining an Alias Name in an Assignment Statement illustrates how to define an alias name using an assignment statement so that the alias name is not affected by a subsequent `.rotr` directive. The `.rotr` directive maps `b[1]` to general register `r36`. `b[1]` is assigned to `tmp`. The second `.rotr` directive defines the new mapping of `b[1]` to `r33`. The subsequent `add` instruction that refers to `b[1]` is currently mapped to `r33`. The second `add` instruction refers to `tmp`, which is mapped to `r36`, not `r33`.

Example: Defining an Alias Name in an Assignment Statement

```
    .rotr   a[3],b[2],c[4]
tmp = b[1]                  //b[1] is currently r36
   ...
   .rotr   b[4],c[3],d[2]
   add     b[1] = r1,r7    //b[1] is currently r33
   add     r1 = r2,tmp     //tmp = r36!
```

Byte Order Specification Directives

The `.msb` and `.lsb` directives determine the byte order of data assembled by the `datan`, `realn`, and `.xdatan` data allocation statements. The values of *n* for `data` and `.xdata` are `1`, `2`, `4`, and `8`. The values of *n* for real are `4`, `8`, `10`, and `16`. See Data Allocation section for more information about data allocation statements.

The `.msb` and `.lsb` directives change the byte order for current sections only. They do not affect the instructions that are assembled. They only affect the data created. The default byte order is little-endian.

The `.msb` directive switches to MSB, where the most-significant byte is stored at the lowest address (big-endian). The `.lsb` directive switches to LSB, where the least-significant byte is stored at the lowest address (little-endian).

The byte order is a property of each section. If the byte order is changed in one section, it remains in effect for that section until the byte order is redefined. This change does not affect the byte order of other sections in the assembly program.

String Specification Directive

The `.ident` directive places a null-terminated string in the `.comment` section of an output object file. See the use of `.comment` in Program Structure. The `.ident` directive has the following format:

```
.ident "string"
```

Where:

`"string"`	Represents a string.

Radix Indicator Directive

The `.radix` directive selects the numeric constant style.

To select a MASM numeric constant and specify a radix indicator, use the `.radix` directive in the following format:

```
.radix   [radix-indicator]
```

Where:

`radix-indicator`	Indicates a MASM (Microsoft * macro assembler) numeric constant and specifies the radix. See table MASM Radix Indicators, for a list of the radix indicators.

The MASM numeric constant and radix remain in effect until redefined.

To select a C numeric constant, use the `.radix` directive in the following format:

```
.radix    [C]
```

Where:

`C`	Indicates a C numeric constant.

The `.radix` directive used with an operand, pushes the previous numeric constant style and radix onto a radix stack. The `.radix` directive without the radix-indicator operand, pops and restores the previous style and radix from the stack. The assembler may limit the depth of a radix stack, but this limit must be no less than 10 levels.

Preprocessor Support

The assembler recognizes a special filename and the line number directive (`#line`) inserted by the standard C preprocessor, and sets its record as the current filename and line number accordingly. The `#line` directive has the following format:

```
#line line_number, filename
```

Where:

line_number	Specifies the source line number
filename	Identifies the name of the current filename.

Additionally, the assembler supports the following built-in symbols:

`@line` Current line number

`@filename` Current filename

`@filepath` Current file path

Annotations

Annotations are a subset of the assembler directives. They explicitly provide additional information for the assembler during the assembly process. These annotations have the same format and syntax as all other directives. This section describes these annotations and their functionality.The annotations covered in this section include:

- `.pred.rel`
- `.pred.vector`
- `.mem.offset`
- `.entry`

Predicate Relationship Annotation

The predicate relationship annotation `.pred.rel` provides information for the assembler about a logical relationship between the values of predicate registers. It is relevant only for explicit code.

The annotation `.pred.rel` takes the following forms:

`"mutex"` mutual exclusion

`"imply"` implication

`"clear"` clear existing relations

When conflicting instructions follow an entry point, IAS ignores all existing predicate relationships defined before the entry point.

Predicate Vector Annotation

The predicate vector annotation .pred.vector explicitly specifies the predicate register contents using a user-defined value. The user-defined value is represented by a 64-bit binary number and each bit corresponds to a predicate register, respectively. A second optional operand can be used as a mask to selectively set only some of the predicate registers. Currently this annotation is ignored by the Intel® Itanium® Assembler.

This annotation takes effect at the point of insertion and the assembler may use this information for further analysis. The `.pred.vector` annotation has the following syntax:

```
.pred.vector   val [,mask]
```

Where:

val	Specifies a number represented as a 64-bit binary number. Each bit represents a 1-bit value in each of the corresponding 64 predicate registers. If *val* is not within the 64-bit range, this annotation is ignored.
mask	Represents an optional mask value used to define a subset of the predicate register file.

Example Using a Predicate Vector Annotation with a Mask illustrates a predicate vector annotation that sets the predicate registers according to the specified value `0x9`, and uses a mask of `0xffff` to define a subset of the predicate register file.

Example: Using a Predicate Vector Annotation with a Mask

```
.pred.vector 0x9, 0xffff   //only refers to lowest 16-bits that
                          //are set in the mask.
                          //Values of p0-p15 are defined.
```

Memory Offset Annotation

The memory offset annotation `.mem.offset` provides hints about the address that memory operations address, when the exact address is unknown. The annotation is useful for avoiding false reports of dependency violations. The annotation affects the instruction that follows.

The `.mem.offset` annotation has the following syntax:

`.mem.offset` *`off_val, base_ind`*

Where:

`off_val`	The relative offset for the memory region where the data is stored or retrieved.
`base_ind`	A number that identifies the memory region where the information is stored or retrieved. The number is an arbitrary method of distinguishing between different memory regions.

Example Using the Memory Offset Annotation illustrates a `.mem.offset` annotation.

Example: Using the Memory Offset Annotation

```
.proc foo
foo::
FOO_STACK_INDEX=0
...                            //code...
.mem.offset 0,FOO_STACK_INDEX  //Suppose r3 contains the stack pointer
st8.spill [r3]=r32,8           //We want to save r32-r34
.mem.offset 8,FOO_STACK_INDEX
st8.spill [r3]=r33,8
.mem.offset 16,FOO_STACK_INDEX
st8.spill [r3]=r34,8
.endp

.proc bar
bar::
.BAR_STACK_INDEX=1
...                            //code...
.mem.offset 0,BAR_STACK_INDEX  //Suppose r3 contains the stack pointer
st8.spill [r3]=r40             //We want to save r40
```

Entry Annotation

The entry annotation `.entry` notifies the assembler that a label can be entered from another function. By default, only global labels, designated by `<label>::`, are considered entry points. The annotation and the label need not be consecutive.

The `.entry` annotation has the following syntax:

```
.entry   label [, labels...]
```

Where:

`label`	Represents the associated label.

Example: Using the Entry Annotation

```
.entry A                //entry annotation
 A: mov r1=r2
```

Register Names by Type

This section contains eight tables that list the following the Itanium® architecture registers and their names:

- General Registers
- Floating-point Registers
- Predicate Registers
- Branch Registers
- Application Registers
- Control Registers
- Other Registers
- Indirect-register Files

General Registers

Register	Register Name
Fixed general registers	r0 - r31
Stacked general registers	r32 - r127
Alternate names for input registers	in0 - in95
Alternate names for local registers	loc0 - loc95
Alternate names for output registers	out0 - out95
Global pointer (r1)	gp
Return value registers (r8-r11)	ret0 - ret3
Stack pointer (r12)	sp

Floating-point Registers

Register	Register Name
Floating-point registers	`f0 - f127`
Argument registers (`f8-f15`)	`fret0 - fret7`
Return value registers (`f8-f15`)	`fret0 - fret7`

Predicate Registers

Register	Register
Predicates	`p0 - p63`
All predicates	`pr`
Rotating predicates	`pr.rot`

Branch Registers

Register	Register Name
Branch registers	`b0` - `b7`
Return pointer (`b0`)	`rp`

Application Registers

Register	Register Number	Register Name
Application registers by number	0 - 127	ar0 - ar127
Kernel registers	0 - 7	ar.k0 - ar.k7
RSE control register	16	ar.rsc
Backing store pointer	17	ar.bsp
Backing store “store” pointer	18	ar.bspstore
RSE NaT collection register	19	ar.rnat
Compare & Exchange comparison value	32	ar.ccv
User NaT collection register	36	ar.unat
Floating-point status register	40	ar.fpsr
Interval time counter	44	ar.itc
Previous frame state	64	ar.pfs
Loop counter	65	ar.lc
Epilog counter 66	66	ar.ec

Control Registers

Register	Register Number	Register Name
Control registers by number	0 - 127	cr0 - cr127
Default control register	0	cr.dcr
Interval time match 1	1	cr.itm
Interruption vector address 2	2	cr.iva
Page table address 8	8	cr.pta
Guest page table address	9	cr.gpta
Interruption processor status register	16	cr.ipsr
Interruption status register	17	cr.isr
Interruption instruction pointe	19	cr.iip
Interrupt faulting address	20	cr.ifa
Interrupt TLB insertion register	21	cr.itir
Interruption instruction previous address	22	cr.iipa
Interruption frame state	23	cr.ifs
Interruption immediat	24	cr.iim
Interruption hash address	25	cr.iha
External interrupt registers	64 65 66 67 68-71 72 73 74 80-81	cr.lid cr.ivr cr.tpr cr.eoi cr.irr0-cr.irr3 cr.itv cr.pmv cr.cmcv cr.lrr0-cr.lrr1

Other Registers

Register	Register Name
Processor status register	`psr`
Processor status register, lower 32 bits	`psr.l`
User mask psr.um	`psr.um`
Instruction pointer	`ip`

Indirect-register Files

Register	Register Name
Performance monitor control registers	`pmc[r]`
Performance monitor data registers	`pmd[r]`
Protection key registers	`pkr[r]`
Region registers	`rr[r]`
Instruction breakpoint registers	`ibr[r]`
Data breakpoint registers	`dbr[r]`
Instruction translation registers	`itr[r]`
Data translation registers	`dtr[r]`
Processor identification register	`CPUID[r]`

Pseudo-ops

This section contains two tables of pseeudo-ops:

- Pseudo-ops Listed by Opcode
- Pseudo-ops with Missing Operands

Pseudo-ops Listed by Opcode

The table that follows lists the assembly language pseudo-ops for the Itanium® architecture according to their opcodes. The table lists pseudo-ops with missing operands. The opcodes are listed alphabetically, with their operands, and the equivalent machine instructions. The table lists mnemonics converted to other mnemonics.

Opcode	Instruction Description	Operands	Equivalent Machine Instruction
`add`	Add immediate	*r1 =imm,r3*	`adds` *r1 =imm14,r3* `addl` *r1 =imm22,r3*
`break`	Break	*imm21*	`break.b` *imm21 (B)* `break.i` *imm21 (I)* `break.m` *imm21 (M)* `break.f` *imm21 (F)*
`chk.s`	Speculation check	*r2,target25*	`chk.s.i` *r2,target25(I)* `chk.s.m` *r2,target25(M)*
`fabs`	Floating-point absolute value	*f1 =f3*	`fmerge.s` *f1 =f0,f3*
`fadd.pc.sf`	Floating-point add	*f1 =f3,f2*	`fma.pc.sf` *f1 =f3,f1,f2*
`fcvt.xuf`	Convert integer to float unsigned	*f1 =f3*	`fma.pc.sf` *f1 =f3,f1,f0*
`fmpy.pc.sf`	Floating-point multiply	*f1 =f3,f4*	`fma.pc.sf` *f1 =f3,f4,f0*
`fneg`	Floating-point negate	*f1 =f3*	`fmerge.ns` *f1 =f3,f3*
`fnegabs`	Floating-point negate absolute value	*f1 =f3*	`fmerge.ns` *f1 =f0,f3*
`fnorm.pc.sf`	Floating-point normalize	*f1 =f3*	`fma.pc.sf` *f1 =f3,f1,f0*
`fsub.pc.sf`	Floating-point	*f1 =f3,f2*	`fms.pc.sf` *f1 =f3,f1,f2*

	subtract		
`ld8.mov`	`ld8` that can be translated to `mov`. It is used to support link time rewriting of indirect addressing code sequences. In ELF format only.	*r2=[r3],* Symbol+Addend	`ld8` *r2=[r3]* `mov` *r2=r3*
`mov`	Move to application register immediate	*ar3 =imm8*	`mov.i` *ar3 =imm8 (I)* `mov.m` *ar3 =imm8 (M)*
`mov`	Move to application register	*ar3 =r2*	`mov.i` *ar3 =r2 (I)* `mov.m` *ar3 =r2 (M)*
`mov`	Move floating-point register	*f1 =f3*	`fmerge.s` *f1 =f3,f3*
`mov`	Move from application register	*r1 =ar3*	`mov.i` *r1 =ar3 (I)* `mov.m` *r1 =ar3 (M)*
`mov`	Move immediate	*r1 =imm22*	`addl` *r1 =imm22,r0*
`mov`	Move general register	*r1 =r2*	`adds` *r1 =0,r2*
`mov`	Move to branch register	*b1 =r2*	`mov` *b1 =r2*
`nop`	No operation	*imm21*	`nop.b` *imm21 (B)* `nop.i` *imm21 (I)* `nop.m` *imm21 (M)* `nop.f` *imm21 (F)*
`shl`	Shift left	*r1=r2,count6*	`dep.z` *r1=r2,count6, 64-count6*
`shr`	Shift right signed	*r1=r3,count6*	`extr` *r1=r3,count6, 64-count6*
`shr.u`	Shift right unsigned	*r1=r3,count6*	`extr.u` *r1=r3,count6, 64-count6*
`xma.lu`	Fixed-point	*f1=f2,f3,f4,*	`xma.l` *f1 =f2,f3,f4*

	multiply low unsigned			

Pseudo-ops with Missing Operands

The table below lists pseudo-ops that omit one or more operands of the machine instruction. The assembler substitutes the missing operand with a predefined value. The missing operand(s) appear as bold text. In addition to omitting many operands, many completers may also be omitted.

Pseudo-op	Missing Operand(s)		Substitute Value
alloc	alloc	r1=ar.pfs,i,l,o,r	ar.pfs
cmp	cmp.*crel.ctype*	p1,p2=imm8,r3	p0
cmp	cmp.*crel.ctype*	p1,p2=r2,r3	p0
cmp4	cmp4.*crel.ctype*	p1,p2=imm8,r	p0
cmp4	cmp4.*crel.ctype*	p1,p2=r2,r3	p0
cmpxchg	cmpxchgsz.*sem.ldhint*	r1=[r3],r2,ar.ccv	ar.ccv
fclass	fclass.m.*fctype* fclass.nm.*fctype*	p1,p2=f2,f3 p1,p2=f2,f3	p0
fcmp	fcmp.*fcrel.fctype.sf*	p1,p2=f2,f3	p0
mov	mov pr=*r2,mask17*		all ones
tbit	tbit.*trel.ctype*	p1,p2=r3,pos6	p0
tnat	tbit.*trel.ctype*	p1,p2=r3	p0

Link-relocation Operators

The table below lists and describes the link-relocation operators and their usage. Unless otherwise specified, the usage is for both COFF and ELF formats.

Operator	Generates a Relocation For:	Usage
@dtpmod(*expr*)	The current instruction requests the linker to put the load module index for expr. It is used in dynamically-bound programs.	data8 statement in ELF format.
@dtprel(*expr*)	The current instruction or data object that calculates the static dtv-relative offset to the address given by *expr*. It is used in dynamically-bound programs.	The adds, addl, and movl instructions and data8 statement in ELF format.
@gprel(*expr*)	The current instruction or data object that calculates the gp-relative offset to the address given by *expr*.	The addl instruction, and data8 (and data4 in ELF format) statements.
@ltoff(*expr*)	The current instruction that instructs the linker to create a linkage table entry for *expr*, and calculates the gp-relative offset to the new linkage table entry.	add long immediate instructions.
@ltoff (@dtpmod *(expr)*)	The current instruction requests the linker to allocate a linkage table entry to hold the load module index for *expr*. The linker processes this relocation by substituting the gp-relative offset for the new linkage table entry. It is used in dynamically- bound programs.	The add long immediate instruction in ELF format.
@ltoff (@dtprel *(expr)*)	The current instruction that instructs the linker to create a linkage table entry to hold the dtv-relative offset for *expr* and calculates the gp-relative offset to the new linkage table entry. It is used in dynamically-bound programs.	The add long immediate instruction in ELF format.
@ltoff(@tprel *(expr)*)	The current instruction that instructs the linker to create a linkage table entry to hold the tp-relative offset for *expr*, and calculates the gp-relative offset to the new linkage table entry. It is used in statically-bound programs.	The add long immediate instruction in ELF format.
@ltoffx*(expr)*	The current instruction that instructs the linker to create a linkage table entry for *expr*, and calculates the gp-relative offset to the new linkage table entry. It is used to support link-time rewriting of the indirect addressing code sequences.	The add long immediate instruction in ELF format.
@secrel(expr)	The current data object that calculates the	data4 and data8

	offset, relative to the beginning of the section, to the address given by *expr*.	statements, and the `addl` instruction.
@segrel(expr)	The current data object that calculates the offset, relative to the beginning of the segment, to the address given by *expr*.	`data4` and `data8` statements, in `ELF` format.
`@imagerel (`*expr*`)`	The current data object that calculates the offset, relative to the beginning of the image, to the address given by *expr*.	`data4` statements, in `COFF` format.
`@fptr(`*sym*`)`	The current instruction or data object that calculates the address of the official *plabel* descriptor for the symbol *sym*, which must be a procedure label (function descriptor) name.	`data4` and `data8` statements, and move long immediate instructions. Requires function symbol in `COFF` format. It can be used in add long immediate instructions when combined with the `@ltoff` operator in the `@ltoff(@fptr(`*sym*`)` form.
`@pltoff(`*sym*`)`	The current instruction or data object that calculates the gp-relative offset to the procedure linkage table entry for the symbol `sym`, which must be a function name.	`data8` statements and add long immediate instructions. The `PLT` entry referenced by this operator should be used only for a direct procedure call. It does not serve as a function descriptor name.
`@iplt(`*sym*`)`	The current data object that calculates the *plabel* descriptor for the symbol *sym*, which must be a procedure label (function descriptor) name.	`data16` statements in `ELF` format.
`@ltv(`*expr*`)`	The current data object that calculates the address of the relocatable expression *expr*, with one exception; while it is expected that the addresses created will need further relocation at run-time, the linker should not create a corresponding relocation in the output executable or shared object file. The run-time consumer of the information provided is expected to relocate these values.	`data4` statements in `ELF` format.
`@section(`*sec*`)`	The current data object that provides the section header number of section *sec*. Used for debug information.	`data2` statements in `COFF` format.
`@tprel(`*expr*`)`	The current instruction or data object that calculates the tp-relative offset to the address given by *expr*. It is used in	The `adds`, `addl`, and `movl` instructions and `data8` statement in `ELF`

	statically-bound programs.	format.

List of Assembly Language Directives

The table below summarizes the Itanium® architecture assembly language directives by category.

Category	Directive
Alias declaration directives	`.alias` `.secalias`

Assembler annotations	`.pred.rel`
	`.pred.vector`
	`.mem.offset`
	`.entry`

Assembler modes	`.auto`
	`.explicit`
	`.default`

Byte order specification directive	`.msb` `.lsb`
Common symbol declaration directives	`.common` `.lcomm`

Cross-section data allocation statements	`.xdata1`
	`.xdata2`
	`.xdata4`
	`.xdata8`
	`.xstring`
	`.xstringz`

Data-allocation statements data1	`data1`
	`data2`
	`data4`
	`data8`
	`real4`

	real8 real10 real16 string stringz
Explicit template selection directives	.mii .mfi .bbb .mlx .mib .mmb .mmi .mbb .mfb .mmf
File symbol declaration directive	.file
Ident string directive	.ident
Include file directive	.include
Language specific data directive (Windows NT * specific)	.handlerdata
Procedure declaration directives	.proc .endp
Radix indicator directive	.radix
Register stack directive	.regstk
Reserving uniniatialized space statements	.skip .org
Rotating register directives	.rotr .rotp .rotf
Section directives	.section

	`.pushsection`
	
	`.popsection`
	
	`.previous`
	
	`.text`
	
	`.data`
	
	`.sdata`
	
	`.bss`
	
	`.sbss`
	
	`.rodata`
	
	`.comment`

Section and data alignment directive	`.align`
Stack unwind information directives	See Stack Unwind Directives table

Symbol scope declaration directives	`.global`
	
	`.weak`
	
	`.local`

Symbol visibility directives	`.protected`
	
	`.hidden`

Symbol type and size directives	`.type`
	
	`.size`

Symbolic debug directive	`.ln`

Symbolic debug directive Windows NT
 	
specific
	`.bf`
	
	`.ef`

Virtual register allocation directives	`.vreg.allocatable`
	
	`.vreg.safe_across_calls`
	
	`.vreg.family`
	
	`.vreg.var`
	
	`.vreg.undef`

Glossary

absolute address	A virtual (not physical) address within the process' address space that is computed as an absolute number.
absolute expression	An expression that is not subject to link-time relocation.
alias	Two identifiers referring to the same element.
assembler	A program that translates assembly language into machine language.
assembly language	A low level symbolic language closely resembling machine-code language.
binding	The process of resolving a symbolic reference in one module by finding the definition of the symbol in another module, and substituting the address of the definition in place of the symbolic reference. The linker binds relocatable object modules together, and the DLL loader binds executable load modules. When searching for a definition, the linker and DLL loader search each module in a certain order, so that a definition of a symbol in one module has precedence over a definition of the same symbol in a later module. This order is called the binding order.
bundle	128 bits that include three instructions and a template field.
COFF	Common Object File Format, an object-module

	format.
directive	An assembler instruction that does not produce executable code.
execution time	The time during which a program is actually executing, not including the time during which the program and its DLLs are being loaded.
expression	A sequence of symbols that represents a value.
function name	A label that refers to a procedure entry point.
global symbol	Symbol visible outside the source file in which it is defined.
IA-32	Intel Architecture-32: the name for Intel’s current 32-bit Instruction Set Architecture (ISA).
identifier	Syntactic representation of symbol names using alphabetic or special characters, and digits.
instruction	An operation code that performs a specific machine operation.
instruction group	Itanium® architecture instructions are organized in instruction groups. Each instruction group contains one or more statically contiguous instructions that execute in parallel. An instruction group must contain at least one instruction; there is no upper limit on the number of instructions in an instruction group. An instruction group is terminated statically by a stop, and dynamically by taken branches. Stops are represented by a double semi-colon (;;). You can explicitly define stops. Stops immediately follow an instruction, or appear on a separate line. They can be inserted between two instructions on the same line, as a semi-colon (;) is used to separate two instructions.

Instruction Set Architecture	The architecture that defines application level resources which include: user-level instructions, addressing modes, segmentation, and user visible register files. instruction tag A label that refers to an instruction.
ISA	See Instruction Set Architecture
Itanium processor	Name of Intel's first 64-bit processor.
label	A location in memory of code or data.
link time	The time when a program, dynamic-link library (DLL), or starred object is processed by the linker. Any activity taking place at link time is static.
linkage table	A table containing text, unwind information, constants, literals, and pointers to imported data symbols and functions.
local symbol	Symbol visible only within the source file in which it is defined.
location counter	Keeps track of the current address when assembling a program. It starts at zero at the beginning of each segment and increments appropriately as each instruction is assembled. To adjust the location counter of a section, use the `.align` directive, or the `.org` directive.
memory stack	A contiguous array of memory locations, commonly referred to as "the stack", used in many processors to save the state of the calling procedure, pass parameters to the called procedure and store local variables for the currently executing procedure.
mnemonic	A predefined assembly-language name for machine instructions, pseudo-ops, directives, and data-allocation statements.
multiway branch bundle	A bundle that contains more than one branch instruction.
name space	A virtual (not physical) file. The assembler assigns names to a symbol, register, or

	mnemonic name space. Usually a name is defined only once in each separate name space. A name can be defined twice, in the symbol and register name space. In this case the register name takes precedence over the symbol name.
operator	The assembly-language operators indicate arithmetic or bitwise-logic calculations.
plabel	See procedure label.
predicate registers	64 1-bit predicate registers that control the execution of instructions. The first register, `p0`, is always treated as 1.
predication	The conditional execution of an instruction used to remove branches from code.
procedure label	A reference or pointer to a procedure. A procedure label (PLabel) is a special descriptor that uniquely identifies the procedure. The PLabel descriptor contains the address of the function's actual entry point, and the linkage table pointer.
pseudo-op	An instruction aliasing a machine instruction, provided for the convenience of the programmer.
qualifying predicate	The execution of most instructions is gated by a qualifying predicate. If the predicate is true, the instruction executes normally; if the instruction is false the instruction does not modify architectural state or affect program behaviour.
register rotation	Software renaming of registers to provide every loop iteration with its own set of registers.
register stack configuration	A 64-bit register used to control the register stack engine (RSE).
relocatable expression	An expression that is subject to link-time relocation
rotating registers	Registers which are rotated by one register position at each loop execution so that the content of register X is in register X+1 after one rotation. The predicate, floating-point, and general registers can be rotated. The

	registers are rotated in a wrap-around fashion.
section	Portions of an object file, such as code or data, bound to one unit.
software pipelining	Pipelining of a loop by way of allowing the processor to execute, in any given time, several instructions in various instructions of the loop.
stacked registers	Stacked general registers, starting at r32, used to pass parameters to the called procedure and store local variables for the currently executing procedure.
statement	An assembly-language program consists of a series of statements. The following are primary types of assembly-language statements: • label statements • instruction statements • directive statements • assignment statements • equate statements • data allocation statements • cross-data allocation statements
stop	Indicates the boundary of an instruction group. It is placed in the code by the assembly writer or compiler.
symbol declaration	The symbol address is resolved, not necessarily based on the current module. Declare symbols using a `.global` or `.weak` directive.
symbol definition	The symbol address is resolved based on the

	current module. A symbol is defined by assigning it a type and value. You can define a symbol either in an assignment statement, by using it as a label, or with a `.common` directive.
temporary symbol	A symbol name that is not placed in the object-file symbol table. To define a temporary symbol name, precede the name with a period (.).
weak symbol	Undefined symbol in object file, resolved during link time.