Variable-Length Encoding (VLE) Extension Programming Interface Manual

VLEPIM
Rev. 1, 2/2006

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: VLEPIM
Rev. 1, 2/2006

freescale™
semiconductor

Contents

Contents

Figures

Figures

Tables

Tables

Table Number	Title	Page Number

About this Book

The primary objective of this manual is to help programmers provide software that is compatible across the family of processors using variable-length encoding (VLE) extension.

Individual VLE technology implementations are beyond the scope of this manual. Each processor is unique in its implementation of the VLE extension.

The information in this book is subject to change without notice. As with any technical documentation, it is the reader's responsibility to ensure they are using the most recent version of the documentation. For more information, contact your sales representative.

Audience

This manual is for system software and application programmers who want to develop products using the VLE extension. An understanding of operating systems, microprocessor system design, the basic principles of RISC processing, and the VLE instruction set is assumed.

Organization

Following is a summary of the major sections of this manual:

- Chapter 1, "Overview," provides a general understanding of what the programming model defines in the VLE extension.
- Chapter 2, "Application Binary Interface (ABI)," describes the VLE extensions for the PowerPC™ e500 Application Binary Interface (e500 ABI) to support VLE technology.
- Chapter 3, "Instruction Set," provides an overview of the VLE instruction set architecture. For a detailed description of each instruction, including assembly language syntax, refer to the VLE section of the *EREF*.
- Appendix A, "Simplified Mnemonics for VLE Instructions," describes simplified mnemonics, which are provided for easier coding of assembly language programs using VLE technology.

Suggested Reading

This section lists background reading for this manual as well as general information on the VLE extension and PowerPC architecture.

Related Documentation

Freescale Semiconductor processor documentation is organized in the following types of documents:

- *EREF: A Reference for Freescale Book E* (Freescale order no. EREF)—A higher-level view of the programming model as it is defined by Book E, the Freescale Book E implementation standards
- User's manuals—Provide details on individual implementations and are for use with the *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture*
- Addenda/errata to user's manuals—Because some processors have follow-on parts, an addendum describes the additional features and changes to functionality. These addenda are for use with the corresponding user's manuals.
- Hardware specifications—Specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries—Overview of device features.
- Application notes—Address-specific design issues useful to programmers and engineers working with Freescale Semiconductor processors.

Additional literature is released as new processors become available.

General Information

The following documentation, published by Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information on the PowerPC architecture and computer architecture in general:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.

  For updates to the specification, see http://www.austin.ibm.com/tech/ppc-chg.html.
- *Computer Architecture: A Quantitative Approach*, Third Edition, by John L. Hennessy and David A. Patterson
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, David A. Patterson and John L. Hennessy

Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.

italics	Italics indicate variable command parameters, for example, **bcctr***x*. Book titles in text are set in italics. Internal signals are set in italics, for example, $\overline{qual\ BG}$.
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
x	An italicized *x* indicates an alphanumeric variable.
n	An italicized *n* indicates an numeric variable.
¬	NOT logical operator
&	AND logical operator
|	OR logical operator

Terminology Conventions

Table i lists certain terms used in this manual that differ from the architecture terminology conventions.

Table i. Terminology Conventions

Architecture Specification	This Manual
Change bit	Changed bit
Extended mnemonics	Simplified mnemonics
Out of order memory accesses	Speculative memory accesses
Privileged mode (or privileged state)	Supervisor level
Problem mode (or problem state)	User level
Reference bit	Referenced bit
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access

Acronyms and Abbreviations

Table ii contains acronyms and abbreviations that are used in this document.

Table ii. Acronyms and Abbreviated Terms

Term	Meaning
CR	Condition register
CTR	Count register
DCR	Data control register
DTLB	Data translation lookaside buffer
EA	Effective address
ECC	Error checking and correction
FPR	Floating-point register
GPR	General-purpose register
IEEE	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation lookaside buffer
L2	Secondary cache
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
PTE	Page table entry
RISC	Reduced instruction set computing
RTL	Register transfer language
SIMM	Signed immediate value
SPR	Special-purpose register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value

Table ii. Acronyms and Abbreviated Terms (continued)

Term	Meaning
UISA	User instruction set architecture
VA	Virtual address
VLE	Variable-length encoding
XER	Register used primarily for indicating conditions such as carries and overflows for integer operations

Chapter 1
Overview

This document defines a programming model for use with the variable-length encoding (VLE) instruction set extension. Three types of programming interfaces are described herein:

- An application binary interface (ABI) defining low-level coding conventions
- An assembly language interface
- A simplified mnemonic assembly language interface

1.1 Application Binary Interface (ABI)

The VLE programming model extends the existing PowerPC™ ABIs. This extension is independent of the endian mode with regard to data; however, VLE instructions are supported only in big-endian mode. The ABI reviews instruction and data representations for memory management and distinguishes between PowerPC Book E and VLE instructions. The ABI also discusses VLE section identification and relocation types used by the executable and linking format (ELF).

NOTE

Use this chapter in conjunction with the *PowerPC e500 Application Binary Interface* (e500 ABI). Except for the sections discussed in this chapter, the VLE ABI follows the e500 ABI standard. For information on register usage and availability, function calling sequence, parameter passing, stack frames, and other topics, refer to the e500 ABI.

1.2 Assembly Language Interface

The assembly language interface provides an overview of the VLE instructions. The description of each instruction along with the instruction mnemonic and operands can be found in the VLE section of the EREF.

1.3 Simplified Mnemonics Assembly Language Interface

Simplified mnemonics are provided for easier coding of assembly language programs. They are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the VLE extension. Some assemblers may define additional simplified mnemonics not listed in this document; however, all assemblers should support the VLE simplified mnemonics listed in Appendix A.

Chapter 2
Application Binary Interface (ABI)

NOTE

The ABI extensions described herein for VLE applications are still under review by the PowerPC ABI industry working group and may be subject to change. Any modifications will be highlighted in revisions of this document.

This chapter specifies VLE extensions to the PowerPC e500 Application Binary Interface (e500 ABI) that defines both a big-endian and a little-endian ABI. This VLE ABI extension is independent of the endian mode with regards to data; however, VLE instructions are supported only in big-endian mode.

NOTE

This chapter should be used in conjunction with the PowerPC e500 Application Binary Interface (e500 ABI). Except for the sections discussed in this chapter, the VLE ABI follows the e500 ABI standard. For information on topics not covered in this section, including function calling sequence, register usage and availability, stack frame layout, parameter passing, and other topics, please refer to the e500 ABI.

2.1 Instruction and Data Representation

The VLE extension includes additional operations with an alternate instruction encoding to enable reduced code footprint. This alternate encoding set is selected on an instruction page basis. A single page attribute bit selects between standard PowerPC Book E instruction encodings and the VLE instructions for the particular page of memory. This page attribute is an extension to the existing PowerPC Book E page attributes. Pages can be freely intermixed, allowing for a mixture of code with both types of encodings.

Instruction encodings in pages marked as using VLE are either 16 or 32 bits long and are aligned on 16-bit boundaries. Therefore, all instruction pages marked as VLE must use big-endian byte ordering.

The programmer's model uses the same register set when executing either instruction encoding, although certain registers are not accessible to VLE instructions using the 16-bit formats, and not all fields of the condition register (CR) are used by condition setting or conditional branch instructions when executing from a VLE instruction page. In addition, immediate fields and displacements differ in size and use, due to the more restrictive encodings imposed by VLE instructions.

Other than the requirement of big-endian byte ordering for instruction pages, and the additional page attribute to identify whether the instruction page corresponds to a VLE section of code, VLE uses the identical storage model, interrupts and exceptions, timer facilities, debug facilities, and special-purpose registers (SPRs) defined throughout Book E.

2.2 Executable and Linking Format (ELF) Object Files

Both VLE and Book E instructions can coexist in the same ELF binary separated into different ELF sections allowing easy identification for defining memory management page tables for run-time environments. Because implementations supporting VLE use an extension to the existing PowerPC Book E page attributes, providing a single additional page attribute to select between VLE and Book E encodings, memory pages of VLE and Book E instructions can be freely intermixed. Binding of VLE and Book E memory pages to different memory bounds requires separation of VLE and Book E encodings into different ELF sections.

The VLE encodings also require additional relocation types, which allow the linker to resolve immediate and branch displacement fields in the instruction encoding once a symbol or label address is known (at link time). The VLE encodings require additional relocation types to resolve fields not present in the PowerPC Book E encodings.

2.2.1 VLE Information Section

The e500 ABI defines an information section named .PPC.EMB.apuinfo having type SHT_NOTE and attributes of 0, which matches the format of a typical ELF note section as shown in Figure 2-1. The information section allows disassemblers and debuggers to interpret the instructions properly within the binary and can be used by operating systems to provide emulation or error checking of the VLE extension revisions.

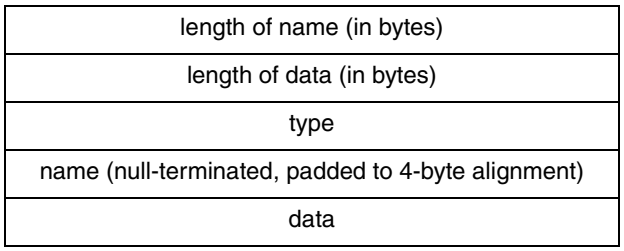


Figure 2-1. Typical Elf Note Section Format

For the .PPC.EMB.apuinfo section, the name is APUinfo, the type is 2 (as type 1 is already reserved), and the data contains a series of words providing information about the APU or extension, one per word. The information contains two unsigned half words: the upper half contains the unique identifier, and the lower half contains the revision number. The VLE identifier is shown in Table 2-1.

Table 2-1. VLE Identifier

Identifier (16 Bits)	APU/Extension
0x0104	VLE

Example 2-1. Object file a.o:

```
0    0x00000008      # 8 bytes in "APUinfo\0"
4    0x0000000C      # 12 bytes (3 words) of APU information
8    0x00000002      # NOTE type 2
12   0x41505569      # ASCII for "APUi"
16   0x6e666f00      # ASCII for "nfo\0"
20   0x00010001      # APU #1, revision 1
24   0x01040001      # VLE, revision 1
28   0x00040001      # APU #4, revision 1
```

Example 2-2. Object file b.o:

```
0    0x00000008      # 8 bytes in "APUinfo\0"
4    0x00000008      # 8 bytes (2 words) of APU information
8    0x00000002      # NOTE type 2
12   0x41505569      # ASCII for "APUi"
16   0x6e666f00      # ASCII for "nfo\0"
20   0x00010002      # APU #1, revision 2
24   0x00040001      # APU #4, revision 1
```

Linkers merge all .PPC.EMB.apuinfo sections in individual object files into one, with merging of per-APU information. For example, after linking file a.o and b.o, the merged .PPC.EMB.apuinfo is as shown in Example 2-3.

Example 2-3. PPC.EMB.apuinfo:

```
0    0x00000008      # 8 bytes in "APUinfo\0"
4    0x0000000C      # 12 bytes (3 words) of APU information
8    0x00000002      # NOTE type 2
12   0x41505569      # ASCII for "APUi"
16   0x6e666f00      # ASCII for "nfo\0"
20   0x00010002      # APU #1, revision 2
24   0x01040001      # VLE, revision 1
28   0x00040001      # APU #4, revision 1
```

Note that it is assumed that a later revision of any APU or extension is compatible with an earlier one, but not *vice versa*. Thus, the resultant .PPC.EMB.apuinfo section requires APU #1 revision 2 or greater to work, and does not work on APU #1 revision 1. If a revision breaks backwards compatibility, it must be given a new unique identifier.

A linker may optionally warn when different objects require different revisions, because moving the revision up may make the executable no longer work on processors with the older revision. In this example, the linker could emit a warning like “Warning: bumping APU #1 revision number to 2, required by b.o.”

2.2.2 VLE Identification

The executable and linking format (ELF) allows processor-specific section header and program header flag attributes to be defined. The following section header and program header flag attribute definitions are used to mark ELF sections containing VLE instruction encodings.

```
#define SHF_PPC_VLE 0x10000000      /* section header flag */
#define PF_PPC_VLE 0x10000000       /* program header flag */
```

The SHF_PPC_VLE flag marks ELF sections containing VLE instructions. Similarly, the PF_PPC_VLE flag is used by ELF program headers to mark program sections containing VLE instructions. If either the SHF_PPC_VLE flag or the PF_PPC_VLE flag is set, then instructions in those marked sections are interpreted as VLE instructions; Book E instructions reside in sections that do not have these flags set.

ELF sections setting the SHF_PPC_VLE flag that contain VLE instructions should also use the SHF_ALLOC and SHF_EXECINSTR bits as necessary. Setting the SHF_PPC_VLE bit does not automatically imply a section that is marked as *allocate* (*SHF_ALLOC*) or *executable* (*SHF_EXECINSTR*). The linker keeps sections marked as VLE (SHF_PPC_VLE) in separate output sections that do not contain Book E instructions.

Similarly, ELF program headers setting the PF_PPC_VLE flag should use the PF_X, PF_W, and PF_R flags to indicate executable, writable, or readable attributes. It is considered an error for a program header with PF_PPC_VLE set to contain sections that do not have SHF_PPC_VLE set.

A program loader or debugger can then scan the section headers or program headers to detect VLE sections in case anything special is required for section processing or downloading.

2.2.3 Relocation Types

Relocation entries describe how to alter the instruction relocation fields once symbols or labels are defined at link time. The VLE instruction set requires relocation types beyond those described in the PowerPC e500 Application Binary Interface (e500 ABI). Table 2-2 shows additional relocation fields used by the VLE instruction set.

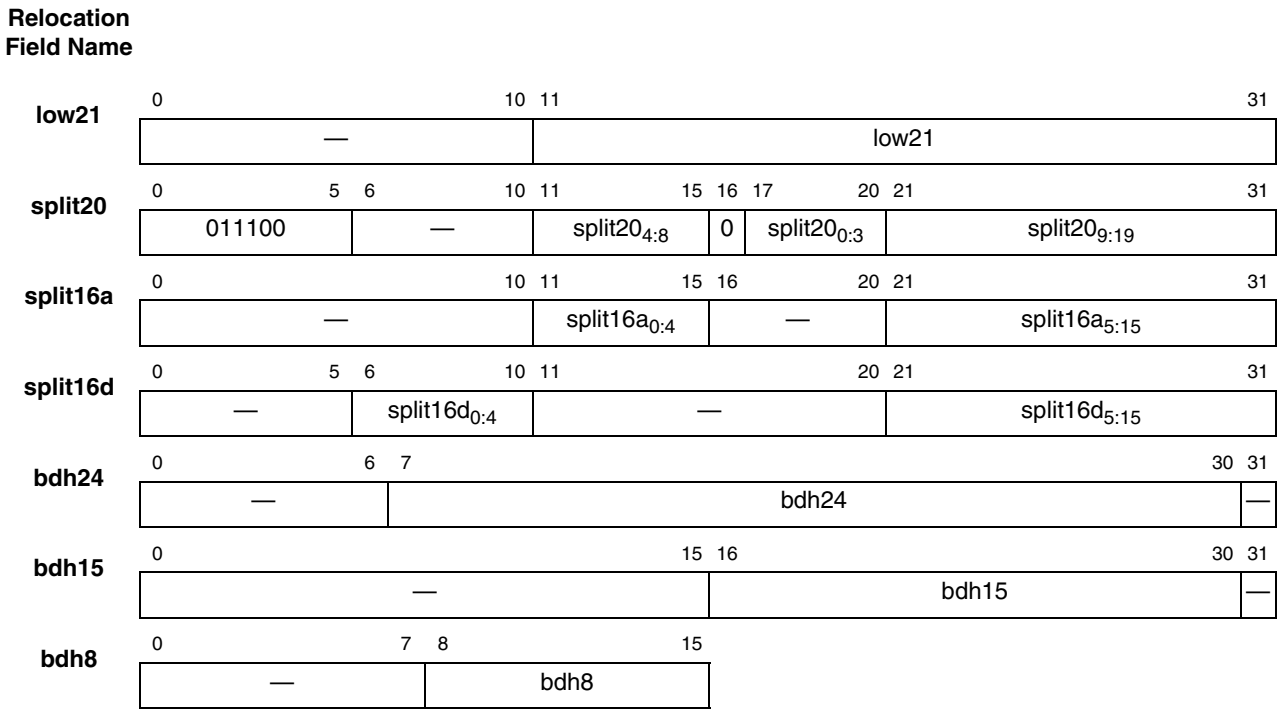


Figure 2-2. VLE Relocation Fields

Table 2-2 describes the additional relocation fields required by VLE instructions.

Table 2-2. VLE Relocation Field Descriptions

Field	Descriptions
low21	21-bit field occupying the lsbs of a word (bits 11–31).
split20	20-bit field with the 4 msbs occupying bits 17–20, the next 5 bits occupying bits 11–15, and the remaining 11 bits occupying bits 21–31. In addition, bits 0–5 in the destination word are encoded with the binary value 011100, bit 16 is encoded with the binary value 0. **Note:** This relocation field specifies the opcode for the VLE **e_li** instruction, allowing the linker to force the encoding of the **e_li** instruction, potentially changing the user's specified instruction. This functionality supports small data area relocation types. (R_PPC_VLE_SDA21 and R_PPC_VLE_SDA21_LO).
split16a	16-bit field with the 5 msbs occupying bits 11–15 (the **r**A field) and the remaining 11 bits occupying bits 21–31.
split16d	16-bit field with the 5 msbs occupying bits 6–10 (the **r**D field) and the remaining 11 bits occupying bits 21–31.
bdh24	24-bit field occupying bits 7–30 used to resolve branch displacements to half-word boundaries.
bdh15	15-bit field occupying bits 16–30 used to resolve branch displacements to half-word boundaries.
bdh8	8-bit field occupying bits 8–15 of a half-word. This field is used by a 16-bit branch instruction.

NOTE

Relocation entry types applied to VLE sections use half-word alignment boundaries, because the VLE instruction architecture mixes 16- and 32-bit encodings within a VLE section. Book E instruction encodings in non-VLE sections use e500 ABI alignment specifications.

Calculations in Table 2-4 assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It determines how to combine and locate the input files, updates the symbol values, and then performs relocations. Relocations applied to executable or shared object files are similar and accomplish the same result. The notations used in Table 2-4 are described in Table 2-3.

Table 2-3. Notation Conventions

Field	Descriptions
A	Represents the addend used to compute the value of the relocatable field.
P	Represents the place (section offset or address) of the storage unit being relocated (computed using r_offset).
S	Represents the value of the symbol whose index resides in the relocation entry.
X	Represents the offset from the appropriate base (_SDA_BASE_, _SDA2_BASE_, or 0) to where the linker placed the symbol whose index is in r_info.
Y	Represents a 5-bit value for the base register for the section where the linker placed the symbol whose index is in r_info. Acceptable values are: the value 13 for symbols in .sdata or .sbss, the value 2 for symbols in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, or the value 0 for symbols in .PPC.EMB.sdata0 or .PPC.EMB.sbss0.

Relocation entries apply to half words or words. In either case, the r_offset value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. Processors that implement the PowerPC architecture use only the Elf32_Rela relocation entries with explicit addends. For relocation entries, the r_addend member

serves as the relocation addend. In all cases, the offset, addend, and the computed result use the byte order specified in the ELF header.

The following general rules apply to the interpretation of the relocation types in Table 2-4:

- + and – denote 32-bit modulus addition and subtraction.

  || denotes concatenation of bits or bit fields.

  >> denotes arithmetic right-shifting (shifting with sign copying) of the value of the left operand by the number of bits given by the right operand.
- For relocation types associated with branch displacements, in which the name of the relocation type contains 8, the upper 24 bits of the computed value before shifting must all be the same (either all zeros or all ones—that is, sign-extended displacement). For relocation types in which the name contains 15, the upper 17 bits of the computed value before shifting must all be the same. For relocation types in which the name contains 24, the upper 7 bits of the computed value before shifting must all be the same. For relocation types whose names contain 8, 15, or 24, the low 1-bit of the computed value before shifting must be zero (half-word boundary).
- #hi(value) and #lo(value) denote the 16 msbs and lsbs of the indicated value. That is, #lo(x)=(x & 0xFFFF) and #hi(x)=((x>>16) & 0xFFFF).

  The high-adjusted value, #ha(value), compensates for #lo() being treated as a signed number: #ha(x)=(((x >> 16) + ((x & 0x8000) ? 1 : 0)) & 0xFFFF).
- _SDA_BASE_ is a symbol defined by the link editor whose value in shared objects is the same as _GLOBAL_OFFSET_TABLE_, and in executable programs is an address within the small data area. Similarly, _SDA2_BASE_ is a symbol defined by the link editor whose value in executable programs is an address within the small data 2 area.

Note that the relocation types in Figure 2-4 apply only to VLE sections. Sections containing Book E instructions should use the PowerPC e500 Application Binary Interface.

Table 2-4. VLE Relocation Types

Name	Value	Field	Calculation
R_PPC_VLE_REL8	216	bdh8	(S + A - P) >> 1
R_PPC_VLE_REL15	217	bdh15	(S + A - P) >> 1
R_PPC_VLE_REL24	218	bdh24	(S + A - P) >> 1
R_PPC_VLE_LO16A	219	split16a	#lo(S + A)
R_PPC_VLE_LO16D	220	split16d	#lo(S + A)
R_PPC_VLE_HI16A	221	split16a	#hi(S + A)
R_PPC_VLE_HI16D	222	split16d	#hi(S + A)
R_PPC_VLE_HA16A	223	split16a	#ha(S + A)
R_PPC_VLE_HA16D	224	split16d	#ha(S + A)
R_PPC_VLE_SDA21	225	low21 split20	Y || (X + A). See Table 2-5.

Table 2-4. VLE Relocation Types (continued)

Name	Value	Field	Calculation
R_PPC_VLE_SDA21_LO	226	low21 split20	Y || #lo(X + A). See Table 2-5.
R_PPC_VLE_SDAREL_LO16A	227	split16a	#lo(X + A)
R_PPC_VLE_SDAREL_LO16D	228	split16d	#lo(X + A)
R_PPC_VLE_SDAREL_HI16A	229	split16a	#hi(X + A)
R_PPC_VLE_SDAREL_HI16D	230	split16d	#hi(X + A)
R_PPC_VLE_SDAREL_HA16A	231	split16a	#ha(X + A)
R_PPC_VLE_SDAREL_HA16D	232	split16d	#ha(X + A)

Relocation types with special semantics are described in Table 2-5.

Table 2-5. Relocation Types with Special Semantics

Name	Description
R_PPC_VLE_SDA21[1]	The linker computes a 21-bit value with the 5 msbs having the value 13 (for GPR13), 2 (for GPR2), or 0. If the symbol whose index is in r_info is contained in .sdata or .sbss, a linker supplies a value of 13; if the symbol is in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, the linker supplies a value of 2; if the symbol is in .PPC.EMB.sdata0 or .PPC.EMB.sbss0, the linker supplies a value of 0; otherwise, the link fails. The 16 lsbs of this 21-bit value are set to the address of the symbol plus the relocation entry r_addend value minus the appropriate base for the symbol section: • _SDA_BASE_ for a symbol in .sdata or .sbss • _SDA2_BASE_ for a symbol in .PPC.EMB.sdata2 or .PPC.EMB.sbss2 • 0 for a symbol in .PPC.EMB.sdata0 or .PPC.EMB.sbss0 If the 5 msbs of the computed 21-bit value are non-zero, the linker uses the low21 relocation field, where the 11 msbs remain unchanged and the computed 21-bit value occupies bits 11–31. Otherwise, the 5 msbs of the computed 21-bit value are zero, with the following results: • The linker uses the split20 relocation field, where only bits occupying 6–10 remain unchanged • The 5 msbs of the 21-bit value are ignored • The next msb is copied to bit 11 and to bits 17–20 as a sign-extension • The next 4 msbs are copied to bits 12–15 • The 11 remaining bits are copied to bits 21–31. • In the destination word, bits 0–5 are encoded with the binary value 011100, and bit 16 is encoded with the binary value 0. **Note:** Use of the split20 relocation field forces the encoding of the VLE **e_li** instruction, which may change the user's specified instruction. See Table 2-2.
R_PPC_VLE_SDA21_LO [1]	Like R_PPC_VLE_SDA21, except that the #lo() operator obtains the 16 lsbs of the 21-bit value. The #lo() operator is applied after the address of the symbol plus the relocation entry r_addend value is calculated, minus the appropriate base for the symbol's section: _SDA_BASE_ for a symbol in .sdata or .sbss, _SDA2_BASE_ for a symbol in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, or 0 for a symbol in .PPC.EMB.sdata0 or .PPC.EMB.sbss0. The R_PPC_VLE_SDA21 entry describes applying the calculated 21-bit value to the destination word that uses either the low21 relocation field or the split20 relocation field. See Table 2-2.

[1] Note that if the opcode is changed, 27 bits are changed instead of 21.

NOTE

The relocations in Table 2-5 are not for load and store instructions (such as, **e_lwz** and **e_stw**), which should use the EABI relocation R_PPC_EMB_SDA21. These relocations, as written here, only start with an **e_add16i**. A linker might convert the instruction to an **e_li**. Although other relocations do not specify the instructions they apply to, it may be useful to know that these relocations can apply only to one instruction.

Chapter 3
Instruction Set

NOTE

This section provides an overview of the VLE instruction set architecture. For details on each instruction, including assembly mnemonic and operands, refer to the VLE section of the *EREF*.

The VLE extension allows PowerPC Book E implementations to support more efficient binary representations of applications for the embedded processor spaces where code density plays a major role in affecting overall system cost, and to a somewhat lesser extent, performance. The intent of the VLE extension is not to define an entirely different ISA nor to supplant the existing PowerPC ISA. Instead, it can be viewed as a supplement that is applied conditionally to an application, or to part of an application, to improve code density.

The major objectives of the VLE extension are as follows:

- Maintain coexistence and consistency with the existing PowerPC Book E ISA and architecture
- Maintain a common programming model and instruction operation model in the VLE extension
- Reduce overall code size by 30 percent over existing PowerPC text segments
- Limit the increase in execution path length to under 10 percent for most important applications
- Limit the increase in hardware complexity for implementations containing the VLE extension

By meeting these objectives, cost-sensitive markets may significantly benefit from the use of the VLE extension.

The VLE extension uses the same semantics as traditional Book E. Due to the limited instruction encoding formats, VLE instructions typically support reduced immediate fields and displacements, and not all Book E operations are encoded in the VLE extension. The basic philosophy is to capture all useful operations, with most frequent operations given priority. Immediate fields and displacements are provided to cover most ranges encountered in embedded control code. Instructions are encoded in either a 16- or 32-bit format, and these formats can be freely intermixed.

Book E floating-point registers are not accessible to VLE instructions. Book E GPRs and SPRs are used by VLE instructions with the following limitations:

- VLE instructions using the 16-bit formats are limited to addressing GPR0–GPR7 and GPR24–GPR31 in most instructions. Move instructions are provided to transfer register contents between these registers and GPR8–GPR23.
- VLE instructions using the 16-bit formats are limited to addressing CR0.
- VLE instructions using the 32-bit formats are limited to addressing CR0-CR3.

VLE instruction encodings generally differ from Book E instructions, except that most Book E instructions falling within Book E primary opcode 31 are encoded identically in 32-bit VLE instructions. Also, they have identical semantics unless they affect or access a resource not supported by the VLE extension. Primary opcode 4 is available to support additional instructions using identical encodings for both Book E and VLE. Therefore, an implementation of VLE can include additional APUs, such as the cache line locking APU, vector or scalar single-precision floating-point APU, and SPE extension and use the exact encodings.

The VLE extension does not currently fully encompass 64-bit operations, although the addition of such operations in a future version is envisioned. For future compatibility, and to avoid confusion with Book E, register bit numbering remains the same as in traditional Book E.

The description of each instruction is contained in the VLE section of the *EREF* and includes the mnemonic and a formatted list of operands. VLE instructions have either exact or similar semantics to Book E instructions. Where the semantics, side-effects, and binary encodings are identical, the Book E mnemonics and formats are used. Where the semantics are similar but the binary encodings differ, the Book E mnemonic is generally preceded with an ‘e_’. To distinguish similar instructions available in both 16- and 32-bit formats under VLE and standard Book E instructions, VLE instructions encoded with 16 bits have an ‘se_’ prefix. VLE instructions encoded with 32 bits that have different binary encodings or semantics than the equivalent Book E instruction have an ‘e_’ prefix. Some examples are the following:

```
stw      RS,D(RA)     // Standard Book E instruction
e_stw    RS,D(RA)     // 32-bit VLE instruction
se_stw   RZ,SD4(RX)   // 16-bit VLE instruction
```

For detailed functional descriptions of each VLE instruction, along with the assembly mnemonic and operands, refer to the VLE section of the *EREF*.

Appendix A
Simplified Mnemonics for VLE Instructions

This appendix describes simplified mnemonics for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the VLE extension.

The simplified mnemonics for the VLE extension are similar to those defined for the PowerPC programming environment. The result is a consistent programming view when working with VLE instructions on PowerPC architectures. Section A.11, “Comprehensive List of Simplified Mnemonics,” provides an alphabetical listing of VLE simplified mnemonics used by a variety of processors. Some assemblers may define additional simplified mnemonics not included here. The simplified mnemonics listed here should be supported by all compilers.

A.1 Overview

Simplified (or extended) mnemonics allow an assembly-language programmer to use more intuitive mnemonics and symbols than the instructions and syntax defined by the instruction set architecture. For example, to code the conditional call “branch to target if CR3 specifies a greater than condition, setting the LR” without simplified mnemonics, the programmer would write the branch conditional and link instruction **e_bcl 1,13,***target*. The simplified mnemonic, branch if greater than and link, **e_bgtl cr3,***target*, incorporates the conditions. Not only is it easier to remember the symbols than the numbers when programming, it is also easier to interpret simplified mnemonics when reading existing code.

Simplified mnemonics are not a formal part of the architecture, but rather a recommendation for assemblers that support the instruction set.

Simplified mnemonics for VLE instructions provide a consistent assembly-language interface with the PowerPC architecture. Many simplified mnemonics were originally defined in the PowerPC architecture documentation. Some assemblers created their own, and others have been added to support extensions to the instruction set (for example, AltiVec instructions and Book E auxiliary processing units (APUs)). Simplified mnemonics for new architecturally defined and new implementation-specific special-purpose registers (SPRs) are described here very generally.

A.2 Subtract Simplified Mnemonics

This section describes simplified mnemonics for subtract instructions.

A.2.1 Subtract Immediate

The effect of a subtract immediate instruction can be achieved by negating the immediate operand of the add immediate instructions, **e_add16i**, **e_add2i.**, **e_add2is**, and **e_addi**. Simplified mnemonics include this negation, making the intent of the computation clearer. These are listed in Table A-1.

Table A-1. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
e_sub16i **r**D,**r**A,value	**e_add16i** **r**D,**r**A,–value
e_sub2i. **r**A,value	**e_add2i.** **r**A,–value
e_sub2is **r**A,value	**e_add2is** **r**A,–value
e_subi **r**D,**r**A,value	**e_addi** **r**D,**r**A,–value
e_subic **r**D,**r**A,value	**e_addic** **r**D,**r**A,–value
e_subic. **r**D,**r**A,value	**e_addic.** **r**D,**r**A,–value

A.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in Table A-2 use the more common order in which the third operand is subtracted from the second.

Table A-2. Subtract Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic[1]
sub[**o**][.] **r**D,**r**A,**r**B	**subf**[**o**][.] **r**D,**r**B,**r**A
subc[**o**][.] **r**D,**r**A,**r**B	**subfc**[**o**][.] **r**D,**r**B,**r**A

1 **r**D,**r**B,**r**A is not the standard order for the operands. The order of **r**B and **r**A is reversed to show the equivalent behavior of the simplified mnemonic.

A.3 Rotate and Shift Simplified Mnemonics

Rotate and shift instructions provide powerful, general ways to manipulate register contents, but they can be difficult to understand. Simplified mnemonics are provided for the following operations:

- Extract—Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register.
- Insert—Select a left- or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged.
- Rotate—Rotate the contents of a register right or left *n* bits without masking.
- Shift—Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift).
- Clear—Clear the leftmost or rightmost *n* bits of a register.
- Clear left and shift left—Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

A.3.1 Operations on Words

The simplified mnemonics in Table A-3 do not support coding with a dot (**.**) suffix. In PowerPC instructions, a dot (.) suffix causes the Rc bit to be set in the underlying instruction. However, the following VLE instruction forms do not support this.

Table A-3. Word Rotate and Shift Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify word immediate	**e_extlwi rA,rS,***n*,*b* (*n* > 0)	**e_rlwinm rA,rS,***b*,**0,***n* – 1
Extract and right justify word immediate	**e_extrwi rA,rS,***n*,*b* (*n* > 0)	**e_rlwinm rA,rS,***b* + *n*,32 – *n*,**31**
Insert from left word immediate	**e_inslwi rA,rS,***n*,*b* (*n* > 0)	**e_rlwimi rA,rS**,32 – *b*,*b*,(*b* + *n*) **– 1**
Insert from right word immediate	**e_insrwi rA,rS,***n*,*b* (*n* > 0)	**e_rlwimi rA,rS**,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**e_rotlwi rA,rS,***n*	**e_rlwinm rA,rS,***n*,**0,31**
Rotate right word immediate	**e_rotrwi rA,rS,***n*	**e_rlwinm rA,rS**,32 – *n*,**0,31**
Shift left word immediate	**e_slwi rA,rS,***n* (*n* < 32)	**e_rlwinm rA,rS,***n*,**0**,31 – *n*
Shift right word immediate	**e_srwi rA,rS,***n* (*n* < 32)	**e_rlwinm rA,rS**,32 – *n*,*n*,**31**
Clear left word immediate	**e_clrlwi rA,rS,***n* (*n* < 32)	**e_rlwinm rA,rS,0,***n*,**31**
Clear right word immediate	**e_clrrwi rA,rS,***n* (*n* < 32)	**e_rlwinm rA,rS,0,0,31** – *n*
Clear left and shift left word immediate	**e_clrlslwi rA,rS,***b*,*n* (*n* ≤ *b* ≤ 31)	**e_rlwinm rA,rS,***n*,*b* – *n*,31 – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   e_extrwi rA,rS,1,0 equivalent to **e_rlwinm rA,rS,1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   e_insrwi rB,rA,1,0 equivalent to **e_rlwimi rB,rA,31,0,0**
3. Shift the contents of **r**A left 8 bits.
   e_slwi rA,rA,8 equivalent to **e_rlwinm rA,rA,8,0,23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   e_clrlwi rA,rS,16 equivalent to **e_rlwinm rA,rS,0,16,31**

A.4 Branch Instruction Simplified Mnemonics

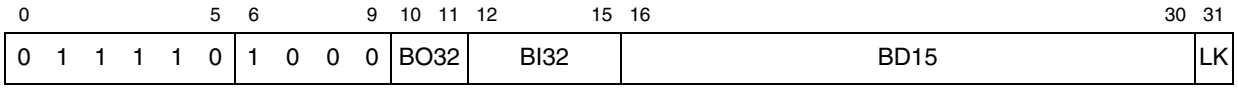
Branch conditional instructions can be coded with the operations and with a condition to be tested as part of the instruction mnemonic rather than as numeric operands (the BO32, BI32 and BO16, BI16 operands). Table A-4 shows the four general types of branch instructions. Simplified mnemonics are defined only for branch conditional instructions that include either the BO32, BI32 or BO16, BI16 operands; there is no need to simplify the other branch mnemonics.

Table A-4. Branch Instructions

Instruction Name	Mnemonic	Syntax
Branch	**e_b** (**e_bl**) **se_b** (**se_bl**)	target_addr target_addr
Branch Conditional	**e_bc** (**e_bcl**) **se_bc**	BO32,BI32,target_addr BO16,BI16,target_addr
Branch to Link Register	**se_blr** (**se_blrl**)	—
Branch to Count Register	**se_bctr** (**se_bctrl**)	—

The BO32, BI32, and BO16, BI16 operands correspond to fields in the instruction opcode, as Figure A-1 shows for Branch Conditional (**e_bc**, **e_bcl**, and **se_bc**) instructions.

e_bc (**e_bcl**)



se_bc

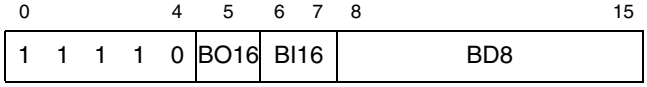


Figure A-1. Branch Conditional (e_bc, se_bc) Instruction Formats

Both the BO32 and BO16 operands allow testing whether a CR bit causes a branch to occur based on a true or false condition. The BO32 operand provides additional capability that allows branch operations that involve decrementing the CTR and testing for a zero or non-zero CTR value.

The BI32 and BI16 operands identify a CR bit to test (whether a comparison is less than or greater than, for example). The simplified mnemonics avoid the need to memorize the numerical values for BO32, BI32, and BO16, BI16 operands.

For example, **e_bc 2,0,***target* is a conditional branch that, as a BO32 value of 2 (0b10) indicates, decrements the CTR, then branches if the decremented CTR is not zero. The operation specified by BO32 is abbreviated as **d** (for decrement) and **nz** (for not zero), which replace the **c** in the original mnemonic; so the simplified mnemonic for **e_bc** becomes **e_bdnz**. The branch does not depend on a condition in the CR, so BI32 can be eliminated, reducing the expression to **e_bdnz** *target*.

In addition to CTR operations, the BO32 operand provides branch decisions based on true or false conditions. For example, if a branch instruction depends on an equal condition in CR0, the expression is **e_bc 1,2,***target*. To specify a true condition, the BO32 value becomes 1; the CR0 equal field is indicated by a BI32 value of 2. Incorporating the branch-if-true condition a '**t**' is used to replace the **c** in the original mnemonic, **e_bt**. The BI32 value of 2 is replaced by the **eq** symbol. Using the simplified mnemonic and the **eq** operand, the expression becomes **e_bt eq,***target*.

This example tests CR0[EQ]; however, to test the equal condition in CR3 (CR bit 14), the expression becomes **e_bc 1,14,***target*. The BI32 operand of 14 indicates CR[14] (CR3[2], or BI32 field 0b1110). This can be expressed as the simplified mnemonic, **e_bt 4 × cr3 + eq,***target*.

The notation, **4 × cr3 + eq** may at first seem awkward, but it eliminates computing the value of the CR bit. It can be seen that (4 × 3) + 2 = 14. Note that although 32-bit registers in Book E processors are numbered 32–63, only values 0–15 are valid (or possible) for BI32 operands. A Book E–compliant processor automatically translates the BI32 bit values; specifying a BI32 value of 14 selects bit 46 on a Book E processor, or CR3[2] = CR3[EQ].

To reduce code size, VLE provides a 16-bit conditional branch instruction that uses the BO16 and BI16 operands. For example, the 32-bit conditional branch **e_bc 1,2,***target* can be expressed using a 16-bit instruction format, **se_bc 1,2,***target*. In simplified mnemonic form this becomes **se_bt eq,**target. The BO16 operand only allows testing a true or false condition, unlike the BO32 operand that also allows decrementing the CTR. The BI16 operand allows testing of only CR0, unlike the BI32 operand, which allows testing CR0–CR3.

A.4.1 Key Facts about Simplified Branch Mnemonics

The following key points are helpful in understanding how to use simplified branch mnemonics:

- All simplified branch mnemonics eliminate the BO32 and BO16 operands, so if any operand is present in a branch simplified mnemonic, it is the BI32 or BI16 operand (or a reduced form of it).
- If the CR is not involved in the branch, the BI32 and BI16 operands can be deleted.
- If the CR is involved in the branch, the BI32 and BI16 operands can be treated in the following ways:
  - It can be specified as a numeric value, just as it is in the architecturally defined instruction, or it can be indicated with an easier to remember formula, **4 * cr***n* **+** [test bit symbol], where *n* indicates the CR field number. For BI16 operands only CR0 is allowed, for BI32 CR0–CR3 is allowed.
  - The condition of the test bit (eq, lt, gt, and so) can be incorporated into the mnemonic, leaving the need for an operand that defines only the CR field.
    - If the test bit is in CR0, no operand is needed.
    - If the test bit is in CR1–CR3, the BI32 operand can be replaced with a **cr**S operand (that is, **cr1**, **cr2**, **or cr3**). The BI16 operand cannot be used for test bits that are not in CR0.

A.4.2 Eliminating the BO32 and BO16 Operands

The 2-bit BO32 field, shown in Figure A-1, encodes the following operations in 32-bit conditional branch instructions:

- Decrement count register (CTR)
  - And test if result is equal to zero
  - And test if result is not equal to zero

- Test condition register (CR)
  - — Test condition true
  - — Test condition false

The 1-bit BO16 field, shown in Figure A-1, encodes the following operations in 16-bit conditional branch instructions:

- Test condition register (CR)
  - — Test condition true
  - — Test condition false

As shown in Table A-5, the '**c**' in the standard mnemonic is replaced with the operations otherwise specified in the BO32 or BO16 field, (**d** for decrement, **z** for zero, **nz** for non-zero, **t** for true, and **f** for false).

Table A-5. BO32 and BO16 Operand Encodings

BO32 Field	BO16 Field	Value (Decimal)	Description	Symbol
00	0	0	Branch if the condition is FALSE.[1]	**f**
01	1	1	Branch if the condition is TRUE.[1]	**t**
10[2]	—	2	Decrement the CTR, then branch if the decremented CTR ≠ 0.	**dnz**[3]
11[2]	—	3	Decrement the CTR, then branch if the decremented CTR = 0.	**dz**[3]

1 Instructions for which BO32 or BO16 are 0 (branch if condition true) or 1 (branch if condition false) do not depend on the CTR value and alternately can be coded by incorporating the condition specified by the BI32 or BI16 fields. See Section A.5.2, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS)."

2 Simplified mnemonics for branch instructions that do not test CR bits (BO32 = 2 or 3) should specify only a target. Otherwise a programming error may occur.

3 Notice that these instructions do not use the branch in condition true or false operations, so simplified mnemonics for these should not specify a BI32 operand.

A.4.3 The BI32 and BI16 Operand—CR Bit and Field Representations

With standard branch mnemonics, the BI32 and BI16 operands are used to test a CR bit, as shown in the example in Section A.4, "Branch Instruction Simplified Mnemonics,"

With simplified mnemonics, the BI32 and BI16 operands are handled differently depending on whether the simplified mnemonic incorporates a CR condition to test, as follows:

- Some branch simplified mnemonics incorporate only the BO32 or BO16 operand. These simplified mnemonics can use the architecturally defined BI32 or BI16 operand to specify the CR bit, as follows:
  - — The BI32 or BI16 operands can be presented exactly as it is with standard mnemonics—as a decimal number, 0–15 for the BI32 operand, and 0–3 for the BI16 operand.
  - — Symbols can be used to replace the decimal operand, as shown in the example in Section A.4, "Branch Instruction Simplified Mnemonics," where **e_bt 4 * cr3 + eq,***target* could be used instead of **e_bt 14,***target*. This is described in Section A.4.4.1, "Specifying a CR Bit."

The simplified mnemonics in Section A.5, “Simplified Mnemonics that Incorporate the BO32 and BO16 Operands,” use one of these two methods to specify a CR bit.

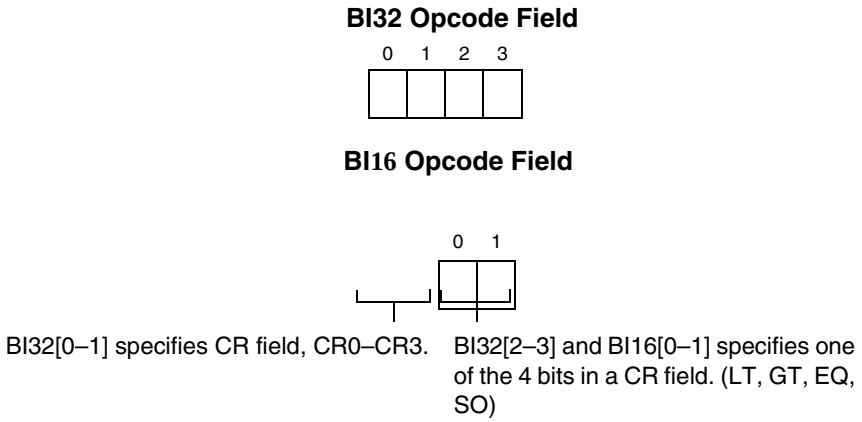
- Additional simplified mnemonics incorporate CR conditions that would otherwise be specified by the BI32 or BI16 operand, so the BI32 or BI16 operand is replaced by the **cr**S operand to specify the CR field. See Section A.4.4, “BI32 and BI16 Operand Instruction Encoding.”

These mnemonics are described in Section A.5.2, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS).”

A.4.4 BI32 and BI16 Operand Instruction Encoding

The entire 4-bit BI32 and 2-bit BI16 fields, shown in Figure A-2, represent the bit number for the CR bit to be tested. For standard branch mnemonics and for branch simplified mnemonics that do not incorporate a CR condition, the BI32 operand provides all 4 bits and the BI16 operand provides all 2 bits.

For simplified branch mnemonics described in Section A.5.2, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS),” the BI32 or BI16 operand is replaced by a **cr**S operand. To understand this, it is useful to view the BI32 operand as composed of two parts. As Figure A-2 shows, BI32[0–1] indicates the CR field and BI32[2–3] represents the condition to test. The 2-bit BI16 operand only has one part, BI16[0–1] represents the condition within CR0 to test.



Simplified mnemonics based on CR conditions but not CTR values —branch if true : BO32=1 or BO16=1 —branch if false: BO32=0 or BO16=0	Specified by a separate, reduced BI32 operand (**crS**)	Incorporated into the simplified mnemonic.
Standard branch mnemonics and simplified mnemonics based on CTR values	The BI32 operand specifies the entire 4-bit field and the BI16 operand specifies a 2-bit field. If CR0 is used, the bit can be identified by LT, GT, EQ, or SO. For BI32, if CR1–CR3 are used, the form 4 * **cr**S + LT|GT|EQ|SO can be used.	

Figure A-2. BI32 and BI16 Fields

Integer record-form instructions update CR0 and floating-point record-form instructions update CR1, as described in Table A-6.

A.4.4.1 Specifying a CR Bit

Note that the AIM version of the PowerPC architecture numbers CR bits 0–31 and Book E numbers them 32–63. However, no adjustment is necessary to the code; in Book E devices, 32 is automatically added to the BI32 and BI16 values, as shown in Table A-6 and Table A-7.

Table A-6. CR0 and CR1 Fields as Updated by Integer and Floating-Point Instructions

CR*n* Bit	CR Bits		BI32		BI16	Description
	AIM	Book E	0–1	2–3	0–1	
CR0[0]	0	32	00	00	00	Negative (LT)—Set when the result is negative.
CR0[1]	1	33	00	01	01	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	2	34	00	10	10	Zero (EQ)—Set when the result is zero.
CR0[3]	3	35	00	11	11	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	4	36	01	00	—	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	5	37	01	01	—	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	6	38	01	10	—	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	7	39	01	11	—	Copy of FPSCR[OX] at the instruction's completion.

Some simplified mnemonics incorporate only the BO32 or BO16 fields (as described Section A.4.2, "Eliminating the BO32 and BO16 Operands"). If one of these simplified mnemonics is used and the CR must be accessed, the BI32 or BI16 operand can be specified either as a numeric value or by using the symbols in Table A-7.

Compare word instructions (described in Section A.6, "Compare Word Simplified Mnemonics"), floating-point compare instructions, move to CR instructions, and others can also modify CR fields, so CR0 and CR1 may hold values that do not adhere to the meanings described in Table A-6.

Table A-7. BI32 and BI16 Operand Settings for CR Fields for Branch Comparisons

CR*n* Bit	Bit Expression	CR Bits		BI32		BI16	Description
		AIM BI Operand	Book E	0–1	2–3	0–1	
CR*n*[0]	**4 * cr0 + lt** (or **lt**) **4 * cr1 + lt** **4 * cr2 + lt** **4 * cr3+ lt**	0 4 8 12	32 36 40 44	00 01 10 11	00	00 — — —	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
CR*n*[1]	**4 * cr0 + gt** (or **gt**) **4 * cr1 + gt** **4 * cr2 + gt** **4 * cr3+ gt**	1 5 9 13	33 37 41 45	00 01 10 11	01	01 — — —	Greater than or floating-point greater than (GT, FG). For integer compare instructions: rA > SIMM or rB (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.

Table A-7. BI32 and BI16 Operand Settings for CR Fields for Branch Comparisons (continued)

CR*n* Bit	Bit Expression	CR Bits		BI32		BI16	Description
		AIM BI Operand	Book E	0–1	2–3	0–1	
CR*n*[2]	**4 * cr0 + eq** (or **eq)** **4 * cr1 + eq** **4 * cr2 + eq** **4 * cr3+ eq**	2 6 10 14	34 38 42 46	00 01 10 11	10	10 — — —	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	**4 * cr0 + so/un** (or **so/un)** **4 * cr1 + so/un** **4 * cr2 + so/un** **4 * cr3 + so/un**	3 7 11 15	35 39 43 47	00 01 10 11	11	11 — — —	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at instruction completion. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Only the most useful simplified mnemonics are found in Section A.5, “Simplified Mnemonics that Incorporate the BO32 and BO16 Operands.” Unusual cases can still be coded using a standard branch conditional syntax.

A.4.4.2 The crS Operand

The **cr**S symbols are shown in Table A-8. Note that either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table A-8. CR Field Identification Symbols

Symbol	BI32[0–1]	BI16	CR Bits
cr0 (default, can be eliminated from syntax)	00	Implied	32–35
cr1	01	—	36–39
cr2	10	—	40–43
cr3	11	—	44–47

To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used, (for example, **cr0 * 4 + eq**).

A.5 Simplified Mnemonics that Incorporate the BO32 and BO16 Operands

The mnemonics in Table A-9 allow common BO32 and BO16 operand encodings to be specified as part of the mnemonic, along with the set link register bit (LK). There are no simplified mnemonics for unconditional branches, branch to link register, and branch to count register. For these, the basic mnemonics **e_b**, **e_bl**, **se_b**, **se_bl**, **se_blr**, **se_blrl**, **se_bctr**, **and se_bctrl** are used.

Table A-9. Branch Simplified Mnemonics

Branch Semantics	LR Update Not Enabled		LR Update Enabled
	e_bc	**se_bc**	**e_bcl**
Branch if condition true	**e_bt**	**se_bt**	**e_btl**
Branch if condition false	**e_bf**	**se_bf**	**e_bfl**
Decrement CTR, branch if CTR ≠ 0[1]	**e_bdnz**	—	**e_bdnzl**
Decrement CTR, branch if CTR = 0[1]	**e_bdz**	—	**e_bdzl**

[1] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise, a programming error may occur.

Table A-10 shows the syntax for basic simplified branch mnemonics

Table A-10. Branch Instructions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**e_b** (**e_bl**) **se_b** (**se_bl**)	target_addr	N/A, syntax does not include BO32 or BO16	
Branch Conditional	**e_bc** (**e_bcl**) **se_bc**	BO32,BI32,target_addr BO16,BI16,target_addr	**e_b***x*[1] (**e_bxl**) **se_b**x[1]	BI32[2],target_addr BI16 [2],target_addr
Branch to Link Register	**se_blr** (**se_blrl**)	—	N/A, syntax does not include BO32 or BO16	
Branch to Count Register	**se_bctr** (**se_bctrl**)	—	N/A, syntax does not include BO32 or BO16	

[1] *x* stands for one of the symbols in Table A-5, where applicable.

[2] BI32 or BI16 can be a numeric value or an expression as shown in Table A-8.

The simplified mnemonics in Table A-9 that test a condition require a corresponding CR bit as the first operand (as the examples 2–5 in Section A.5.1, "Examples that Eliminate the BO32 and BO16 Operands," below illustrate). The symbols in Table A-8 can be used in place of a numeric value.

A.5.1 Examples that Eliminate the BO32 and BO16 Operands

The simplified mnemonics in Table A-9 are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR) (note that no CR bits are tested).

   e_bdnz *target* equivalent to **e_bc 2,**0,*target*

   Because this instruction does not test a CR bit, the simplified mnemonic should specify only a target operand. Specifying a CR (for example, **e_bdnz** 0,*target* or **e_bdnz cr0,***target*) may be considered a programming error. Subsequent examples test conditions).

2. Branch if condition in CR0 is equal.

   e_bt eq,*target* equivalent to **e_bc 1,2,***target*

   Other equivalents include **e_bt 2,***target* or the unlikely **e_bt 4 * cr0 + eq,***target*

3. Same as (2), but equal condition is in CR3.
   e_bt 4 * cr3 + eq,*target* equivalent to **e_bc 1,14,***target*
   e_bt 14,*target* would also work
4. Branch if bit 47 of CR is false.
   bf 15,*target* equivalent to **e_bc 0,15,***target*
   bf 4 * cr3 + so,*target* would also work
5. Same as (4), but set the link register. This is a form of conditional call.
   bfl 15,*target* equivalent to **bcl 4,15,***target*

Table A-11 lists simplified mnemonics and syntax for **e_bc** and **se_bc** without LR updating.

Table A-11. Simplified Mnemonics for e_bc and se_bc without LR Update

Branch Semantics	e_bc	Simplified Mnemonic	se_bc	Simplified Mnemonic
Branch if condition true	**e_bc 1,BI32,**target	**e_bt BI32,**target[1]	**se_bc 1,BI16,**target	**se_bt BI16,**target
Branch if condition false	**e_bc 0,BI32,**target	**e_bf BI32,**target[1]	**se_bc 0,BI16,**target	**se_bf BI16,**target
Decrement CTR, branch if CTR ≠ 0	**e_bc 2,0,**target	**e_bdnz** target[2]	—	—
Decrement CTR, branch if CTR = 0	**e_bc 3,0,**target	**e_bdz** target[2]	—	—

1 Instructions for which B032 is either 1 (branch if condition true) or 0 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI32 field, as described in Section A.5.2, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS).”

2 Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise, a programming error may occur.

Table A-12 provides simplified mnemonics and syntax for **e_bcl**.

Table A-12. Simplified Mnemonics for e_bcl with LR Update

Branch Semantics	e_bcl	Simplified Mnemonic
Branch if condition true[1]	**e_bcl 1,BI32,**target	**e_btl BI32,**target
Branch if condition false[1]	**e_bcl 0,BI32,**target	**e_bfl BI32,**target
Decrement CTR, branch if CTR ≠ 0	**e_bcl 2,0,**target	**e_bdnzl** target[2]
Decrement CTR, branch if CTR = 0	**e_bcl 3,0,**target	**e_bdzl** target [2]

1 Instructions for which B032 is either 1 (branch if condition true) or 0 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI32 field. See Section A.5.2, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS).”

2 Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise, a programming error may occur.

A.5.2 Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS)

The mnemonics in Table A-15 are variations of the branch-if-condition-true (BO32 = 1 or BO16 = 1) and branch-if-condition-false (BO32 = 0 or BO16 = 0) encodings. Because these instructions do not depend on the CTR, the true/false conditions specified by either BO32 or BO16 can be combined with the CR test bit specified by the BI32 or BI16 operand to create a different set of simplified mnemonics that eliminates the BO32 and BO16 operands and the portion of the BI32 and BI16 operands (BI32[2–3] and BI16[0–1]) that specifies one of the four possible test bits. However, for simplified mnemonics using the BO32 operand, the simplified mnemonic cannot specify in which of the four CR fields (CR0–CR3) the test bit falls, so the BI32 operand is replaced by a **cr**S operand.

The standard codes shown in Table A-13 are used for the most common combinations of branch conditions. For ease of programming, these codes include synonyms; for example, less than or equal (**le**) and not greater than (**ng**) achieve the same result.

NOTE

A CR field symbol, **cr0**–**cr3**, is used as the first operand after the simplified mnemonic. If the default, CR0, is used, no **cr**S is necessary.

Table A-13. Standard Coding for Branch Conditions

Code	Description	Equivalent	Bit Tested
lt	Less than	—	LT
le	Less than or equal (equivalent to **ng**)	**ng**	GT
eq	Equal	—	EQ
ge	Greater than or equal (equivalent to **nl**)	**nl**	LT
gt	Greater than	—	GT
nl	Not less than (equivalent to **ge**)	**ge**	LT
ne	Not equal	—	EQ
ng	Not greater than (equivalent to **le**)	**le**	GT
so	Summary overflow	—	SO
ns	Not summary overflow	—	SO
un	Unordered (after floating-point comparison)	—	SO
nu	Not unordered (after floating-point comparison)	—	SO

Table A-14 shows the syntax for simplified branch mnemonics that incorporate CR conditions. Here, **cr**S replaces a BI32 operand to specify only a CR field (because the specific CR bit within the field is now part of the simplified mnemonic. Note that the default is CR0; if no **cr**S is specified, CR0 is used.

Table A-14. Branch Instructions and Simplified Mnemonics that Incorporate CR Conditions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**e_b** (**e_bl**) **se_b** (**se_bl**)	target_addr	—	
Branch Conditional	**e_bc** (**e_bcl**) **se_bc**	BO32,BI32,target_addr BO16,BI16,target_addr	**e_bx** [1] (**e_bxl**) **se_bx**[1]	**cr**S[2],target_addr target_addr
Branch to Link Register	**se_blr** (**se_blrl**)	—	—	
Branch to Count Register	**se_bctr** (**se_bctrl**)	—	—	

1 *x* stands for one of the symbols in Table A-13, where applicable.

2 **cr**S can be a numeric value or an expression as shown in Table A-8.

Table A-15 shows the simplified branch mnemonics incorporating conditions.

Table A-15. Simplified Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled		LR Update Enabled
	e_bc	**se_bc**	**e_bcl**
Branch if less than	**e_blt**	**se_blt**	**e_bltl**
Branch if less than or equal	**e_ble**	**se_ble**	**e_blel**
Branch if equal	**e_beq**	**se_beq**	**e_beql**
Branch if greater than or equal	**e_bge**	**se_bge**	**e_bgel**
Branch if greater than	**e_bgt**	**se_bgt**	**e_bgtl**
Branch if not less than	**e_bnl**	**se_bnl**	**e_bnll**
Branch if not equal	**e_bne**	**se_bne**	**e_bnel**
Branch if not greater than	**e_bng**	**se_bng**	**e_bngl**
Branch if summary overflow	**e_bso**	**se_bso**	**e_bsol**
Branch if not summary overflow	**e_bns**	**se_bns**	**e_bnsl**
Branch if unordered	**e_bun**	**se_bun**	**e_bunl**
Branch if not unordered	**e_bnu**	**se_bnu**	**e_bnul**

Instructions using the mnemonics in Table A-15 indicate the condition bit, but not the CR field. If no field is specified, CR0 is used. For 32-bit instruction forms (denoted with the **e_** prefix) the CR field symbols defined in Table A-8 (**cr0**–**cr3**) are used, as shown in examples 2–3 of Section A.5.3, “Branch Simplified Mnemonics that Incorporate CR Conditions: Examples,” below. Note that the 16-bit instruction forms (denoted with the **se_** prefix) must use CR0.

A.5.3 Branch Simplified Mnemonics that Incorporate CR Conditions: Examples

The following examples use the simplified mnemonics shown in Table A-15:

1. Branch if CR0 reflects not-equal condition.
   e_bne *target* equivalent to **e_bc 0,2,***target*
   se_bne *target* equivalent to **se_bc 0,2,***target*
2. Same as (1) but condition is in CR3.
   e_bne cr3,*target* equivalent to **e_bc 0,14,***target*
3. Branch if CR2 specifies greater than condition, setting the LR. This is a form of conditional call.
   e_bgtl cr2,*target* equivalent to **e_bcl 1,9,***target*

A.5.4 Branch Simplified Mnemonics that Incorporate CR Conditions: Listings

Table A-16 shows simplified branch mnemonics and syntax for **e_bc** and **se_bc** without LR updating.

Table A-16. Simplified Mnemonics for e_bc and se_bc without Comparison Conditions or LR Updating

Branch Semantics	e_bc	Simplified Mnemonic	se_bc	Simplified Mnemonic
Branch if less than	**e_bc 1,**BI32[1]**,**target	**e_blt crS,**target	**se_bc 1,**BI16[1]**,**target	**se_blt** target
Branch if less than or equal	**e_bc 0,**BI32[2]**,**target	**e_ble crS,**target	**se_bc 0,**BI16[2]**,**target	**se_ble** target
Branch if not greater than		**e_bng crS,**target		**se_bng** target
Branch if equal	**e_bc 1,**BI32[3]**,**target	**e_beq crS,**target	**se_bc 1,**BI16[3]**,**target	**se_beq** target
Branch if greater than or equal	**e_bc 0,**BI32[1]**,**target	**e_bge crS,**target	**se_bc 0,**BI16[1]**,**target	**se_bge** target
Branch if not less than		**e_bnl crS,**target		**se_bnl** target
Branch if greater than	**e_bc 1,**BI32[2]**,**target	**e_bgt crS,**target	**se_bc 1,**BI16[2]**,**target	**se_bgt** target
Branch if not equal	**e_bc 0,**BI32[3]**,**target	**e_bne crS,**target	**se_bc 0,**BI16[3]**,**target	**se_bne** target
Branch if summary overflow	**e_bc 1,**BI32[4]**,**target	**e_bso crS,**target	**se_bc 1,**BI16[4]**,**target	**se_bso** target
Branch if unordered		**e_bun crS,**target		**se_bun** target
Branch if not summary overflow	**e_bc 0,**BI32[4]**,**target	**e_bns crS,**target	**se_bc 0,**BI16[4]**,**target	**se_bns** target
Branch if not unordered		**e_bnu crS,**target		**se_bnu** target

[1] The value in the BI32 or BI16 operand selects CR*n*[0], the LT bit.
[2] The value in the BI32 or BI16 operand selects CR*n*[1], the GT bit.
[3] The value in the BI32 or BI16 operand selects CR*n*[2], the EQ bit.
[4] The value in the BI32 or BI16 operand selects CR*n*[3], the SO bit.

Table A-17 shows simplified branch mnemonics and syntax for **e_bcl**.

Table A-17. Simplified Mnemonics for e_bcl with Comparison Conditions and LR Updating

Branch Semantics	e_bcl	Simplified Mnemonic
Branch if less than	**e_bcl 1,**BI32[1],target	**e_bltl crS,**target
Branch if less than or equal	**e_bcl 0,**BI32[2],target	**e_blel crS,**target
Branch if not greater than		**e_bngl crS,**target
Branch if equal	**e_bcl 1,**BI32[3],target	**e_beql crS,**target
Branch if greater than or equal	**e_bcl 0,**BI32[1],target	**e_bgel crS,**target
Branch if not less than		**e_bnll crS,**target
Branch if greater than	**e_bcl 1,**BI32[2],target	**e_bgtl crS,**target
Branch if not equal	**e_bcl 0,**BI32[3],target	**e_bnel crS,**target
Branch if summary overflow	**e_bcl 1,**BI32[4],target	**e_bsol crS,**target
Branch if unordered		**e_bunl crS,**target
Branch if not summary overflow	**e_bcl 0,**BI32[4],target	**e_bnsl crS,**target
Branch if not unordered		**e_bnul crS,**target

[1] The value in the BI32 operand selects CR*n*[0], the LT bit.
[2] The value in the BI32 operand selects CR*n*[1], the GT bit.
[3] The value in the BI32 operand selects CR*n*[2], the EQ bit.
[4] The value in the BI32 operand selects CR*n*[3], the SO bit.

A.6 Compare Word Simplified Mnemonics

In compare word instructions, the L operand indicates a word (L = 0) or double-word (L = 1). Simplified mnemonics in Table A-18 eliminate the L operand for word comparisons.

Table A-18. Word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	**e_cmpwi crD,rA,**SIMM	**e_cmpi crD,rA,**SIMM
	e_cmpwi cr0**,rA,**SIMM	**e_cmp16i rA,**SIMM
Compare Word	**cmpw crD,rA,rB**	**cmp crD,0,rA,rB**
Compare Logical Word Immediate	**e_cmplwi crD,rA,**UIMM	**e_cmpli crD,rA,**UIMM
	e_cmplwi cr0**,rA,**UIMM	**e_cmpl16i rA,**UIMM
Compare Logical Word	**cmplw crD,rA,rB**	**cmpl crD,0,rA,rB**

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 32-bit integers and place result in CR0.
   e_cmpwi rA**,100** equivalent to **e_cmp16i r**A**,100**
2. Same as (1), but place results in CR4.
   e_cmpwi cr3,rA**,100** equivalent to **e_cmpi 3,r**A**,100**
3. Compare **r**A and **r**B as unsigned 32-bit integers and place result in CR0.
   cmplw rA**,r**B equivalent to **cmpl 0,0,r**A**,r**B

A.7 Trap Instructions Simplified Mnemonics

The codes in Table A-19 are for the most common combinations of trap conditions.

Table A-19. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U[1]	>U[2]
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

1 The symbol ‘<U’ indicates an unsigned less-than evaluation is performed.
2 The symbol ‘>U’ indicates an unsigned greater-than evaluation is performed.

The mnemonics in Table A-20 are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table A-20. Trap Simplified Mnemonics

Trap Semantics	tw Register
Trap unconditionally	**trap**
Trap if less than	**twlt**
Trap if less than or equal	**twle**
Trap if equal	**tweq**
Trap if greater than or equal	**twge**
Trap if greater than	**twgt**
Trap if not less than	**twnl**
Trap if not equal	**twne**
Trap if not greater than	**twng**
Trap if logically less than	**twllt**
Trap if logically less than or equal	**twlle**
Trap if logically greater than or equal	**twlge**
Trap if logically greater than	**twlgt**
Trap if logically not less than	**twlnl**
Trap if logically not greater than	**twlng**

The following examples use the trap mnemonics shown in Table A-20:

1. Trap if **r**A is not equal to **r**B.

   twne rA**,r**B equivalent to **tw 24,r**A**,r**B

2. Trap unconditionally.

   trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition by comparing the contents of **r**A with the contents of **r**B. The comparison results in five conditions that are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. See Table A-21 for these conditions.

Table A-21. TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

A.8 Simplified Mnemonics for Accessing SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. The pattern for **mtspr** and **mfspr** simplified mnemonics is straightforward: replace the -**spr** portion of the mnemonic with the abbreviation for the spr (for example XER, SRR0, or LR), eliminate the SPRN operand, leaving the source or destination GPR operand, **r**S or **r**D.

Following are examples using the SPR simplified mnemonics:

1. Copy the contents of **r**S to the XER.
   mtxer rS equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**S.
   mflr rD equivalent to **mfspr r**D**,8**
3. Copy the contents of **r**S to the CTR.
   mtctr rS equivalent to **mtspr 9,r**S

The examples above show simplified mnemonics for accessing SPRs defined by the AIM version of the PowerPC architecture; however, the same formula is used for Book E, EIS, and implementation-specific SPRs, as shown in the following examples:

1. Copy the contents of **r**S to CSRR0.
   mtcsrr0 rS equivalent to **mtspr 58,r**S
2. Copy the contents of IVOR0 to **r**S.
   mfivor0 rD equivalent to **mfspr r**D**,400**
3. Copy the contents of **r**S to the MAS1.
   mtmas1 rS equivalent to **mtspr 625,r**S

There are additional simplified mnemonics for accessing SPRGs, which are not all supported by all assemblers. These mnemonics are shown in Table A-22 along with the equivalent simplified mnemonic using the formula described in this section.

Table A-22. Additional Simplified Mnemonics for Accessing SPRGs

SPR	Move to SPR		Move from SPR	
	Simplified Mnemonic	**Equivalent to**	**Simplified Mnemonic**	**Equivalent to**
SPRGs	**mtsprg** *n*,**r**S	**mtspr** 272 + *n*,**r**S	**mfsprg r**D,*n*	**mfspr r**D,272 + *n*
	mtsprg*n*,**r**S		**mfsprg***n* **r**D	

A.9 Recommended Simplified Mnemonics

This section describes commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

A.9.1 No-Op (nop)

Many instructions can be coded in such a way that, effectively, no operation is performed. Additional mnemonics are provided for the preferred forms of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred forms are the following:

e_nop	equivalent to	**e_ori 0,0,0**
se_nop	equivalent to	**se_or 0,0**

A.9.2 Load Address (la)

The **la** mnemonic permits computing the value of a base-displacement operand, using the **e_add16i** instruction that normally requires a separate register and immediate operands.

e_la rD**,**d(**r**A)	equivalent to	**e_add16i r**D**,r**A**,**d

The **e_la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *V* is located at offset d*V* bytes from the address in **r***V,* and the assembler is directed to use **r***V* as a base for references to the data structure containing *V,* the following line causes the address of *V* to be loaded into **r**D:

e_la rD**,***V*	equivalent to	**e_add16i r**D**,r***V***,**d*V*

A.9.3 Move Register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided to signify that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

mr rA**,r**S	equivalent to	**or r**A**,r**S**,r**S

A.9.4 Complement Register (not)

Several instructions can be coded to complement the contents of one register and place the result into another register. A simplified mnemonic allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

not rA**,r**S	equivalent to	**nor r**A**,r**S**,r**S

A.9.5 Move to Condition Register (mtcr)

The **mtcr** mnemonic permits copying the contents of a GPR to the CR, using the same syntax as the **mfcr** instruction.

mtcr rS	equivalent to	**mtcrf** 0xFF**,r**S

A.10 EIS-Specific Simplified Mnemonics

This section describes simplified mnemonics used by auxiliary processing units (APUs) defined as part of the Freescale Book E implementation standards (EIS).

A.10.1 Integer Select (isel)

The following mnemonics simplify the most common variants of the **isel** instruction that access CR0:

Integer Select Less Than

isellt **r**D,**r**A,**r**B equivalent to **isel** **r**D,**r**A,**r**B,**0**

Integer Select Greater Than

iselgt **r**D,**r**A,**r**B equivalent to **isel** **r**D,**r**A,**r**B,**1**

Integer Select Equal

iseleq **r**D,**r**A,**r**B equivalent to **isel** **r**D,**r**A,**r**B,**2**

A.11 Comprehensive List of Simplified Mnemonics

Table A-23 lists simplified mnemonics. Note that compiler designers may implement additional simplified mnemonics not listed here.

Table A-23. Simplified Mnemonics

Simplified Mnemonic	Mnemonic	Instruction
e_bdnz target[1]	**e_bc 2,0,**target	Decrement CTR, branch if CTR ≠ 0 (**e_bc** without LR update)
e_bdnzl target [1]	**e_bcl 2,0,**target	Decrement CTR, branch if CTR ≠ 0 (**e_bcl** with LR update)
e_bdz target [1]	**e_bc 3,0,**target	Decrement CTR, branch if CTR = 0 (**e_bc** without LR update)
e_bdzl target [1]	**e_bcl 3,BI32,**target	Decrement CTR, branch if CTR = 0 (**e_bcl** with LR update)
e_beq **cr**S,target	**e_bc 1,**BI32[2],target	Branch if equal (**e_bc** without LR updating)
se_beq target	**se_bc 1**,BI16[2],target	Branch if equal (**se_bc**)
e_beql **cr**S,target	**e_bcl 1,**BI32 [2],target	Branch if equal (**e_bcl** with LR updating)
e_bf BI32,target	**e_bc 0,BI32,**target	Branch if condition false [3] (**e_bc** without LR update)
se_bf BI16,target	se_bc 0,BI16,target	Branch if condition false[3] (**se_bc**)
e_bfl BI32,target	**e_bcl 0,BI32,**target	Branch if condition false [3] (**e_bcl** with LR update)
e_bge **cr**S,target	**e_bc 0,**BI32[4],target	Branch if greater than or equal (**e_bc** without LR updating)
se_bge target	**se_bc 0**,BI16[4],target	Branch if greater than or equal (**se_bc**)
e_bgel **cr**S,target	**e_bcl 0,**BI32[4],target	Branch if greater than or equal (**e_bcl** with LR updating)

Table A-23. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
e_bgt crS,target	**e_bc 1,**BI32[5]**,**target	Branch if greater than (**e_bc** without LR updating)
se_bgt target	**se_bc 1**,BI16[5],target	Branch if greater than (**se_bc**)
e_bgtl crS,target	**e_bcl 1,**BI32[5]**,**target	Branch if greater than (**e_bcl** with LR updating)
e_ble crS,target	**e_bc 0,**BI32[5]**,**target	Branch if less than or equal (**e_bc** without LR updating)
se_ble target	**se_bc 0**,BI16[5],target	Branch if less than or equal (**se_bc**)
e_blel crS,target	**e_bcl 0,**BI32[5]**,**target	Branch if less than or equal (**e_bcl** with LR updating)
e_blt crS,target	**e_bc 1,**BI32[4]**,**target	Branch if less than (**e_bc** without LR updating)
se_blt target	**se_bc 1**,BI16[4],target	Branch if less than (**se_bc**)
e_bltl crS,target	**e_bcl 1,**BI32[4]**,**target	Branch if less than (**e_bcl** with LR updating)
e_bne crS,target	**e_bc 0,**BI32[3]**,**target	Branch if not equal (**e_bc** without LR updating)
se_bne target	**se_bc 0**,BI16[3],target	Branch if not equal (**se_bc**)
e_bnel crS,target	**e_bcl 0,**BI32[3]**,**target	Branch if not equal (**e_bcl** with LR updating)
e_bng crS,target	**e_bc 0,**BI32[5]**,**target	Branch if not greater than (**e_bc** without LR updating)
se_bng target	**se_bc 0**,BI16[5],target	Branch if not greater than (**se_bc**)
e_bngl crS,target	**e_bcl 0,**BI32[5]**,**target	Branch if not greater than (**e_bcl** with LR updating)
e_bnl crS,target	**e_bc 0,**BI32[4]**,**target	Branch if not less than (**e_bc** without LR updating)
se_bnl target	**se_bc 0**,BI16[4],target	Branch if not less than (**se_bc**)
e_bnll crS,target	**e_bcl 0,**BI32[4]**,**target	Branch if not less than (**e_bcl** with LR updating)
e_bns crS,target	**e_bc 0,**BI32[6]**,**target	Branch if not summary overflow (**e_bc** without LR updating)
se_bns target	**se_bc 0**,BI16[6],target	Branch if not summary overflow (**se_bc**)
e_bnsl crS,target	**e_bcl 0,**BI32[6]**,**target	Branch if not summary overflow (**e_bcl** with LR updating)
e_bnu crS,target	**e_bc 0,**BI32[6]**,**target	Branch if not unordered (**e_bc** without LR updating)
se_bnu target	**se_bc 0,**BI16[6]**,**target	Branch if not unordered (**se_bc**)
e_bnul crS,target	**e_bcl 0,**BI32[6]**,**target	Branch if not unordered (**e_bcl** with LR updating)
e_bso crS,target	**e_bc 1,**BI32[6]**,**target	Branch if summary overflow (**e_bc** without LR updating)
se_bso target	**se_bso 1**,BI16[6],target	Branch if summary overflow (**se_bc**)
e_bsol crS,target	**e_bcl 1,**BI32[6]**,**target	Branch if summary overflow (**e_bcl** with LR updating)
e_bt BI32,target	**e_bc 1,BI32,**target	Branch if condition true[3] (**e_bc** without LR update)
se_bt BI16,target	**se_bc 1**,BI16,target	Branch if condition true[3] (**se_bc**)
e_btl BI32,target	**e_bcl 1,BI32,**target	Branch if condition true [3] (**e_bcl** with LR update)
e_bun crS,target	**e_bc 1,**BI32[6]**,**target	Branch if unordered (**e_bc** without LR updating)
se_bun target	**se_bc 1**,BI16[6],target	Branch if unordered (**se_bc**)

Table A-23. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
e_bunl crS,target	**e_bcl 1,**BI32[6],target	Branch if unordered (**e_bcl** with LR updating)
e_clrlslwi rA,rS,*b*,*n* (*n* ≤ *b* ≤ 31)	**e_rlwinm rA,rS,***n*,*b* – *n*,31 – *n*	Clear left and shift left word immediate
e_clrlwi rA,rS,*n* (*n* < 32)	**e_rlwinm rA,rS,0,***n*,**31**	Clear left word immediate
e_clrrwi rA,rS,*n* (*n* < 32)	**e_rlwinm rA,rS,0,0,31** – *n*	Clear right word immediate
cmplw crD,rA,rB	**cmpl crD,0,rA,rB**	Compare logical word
e_cmplwi crD,rA,UIMM	**e_cmpli crD,rA,**UIMM	Compare logical word immediate
e_cmplwi cr0,rA,UIMM	**e_cmpl16i rA,**UIMM	Compare logical word immediate
cmpw crD,rA,rB	**cmp crD,0,rA,rB**	Compare word
e_cmpwi crD,rA,SIMM	**e_cmpi crD,rA,**SIMM	Compare word immediate
e_cmpwi cr0,rA,SIMM	**e_cmp16i rA,**SIMM	Compare word immediate
e_extlwi rA,rS,*n*,*b* (*n* > 0)	**e_rlwinm rA,rS,***b*,**0,***n* – 1	Extract and left justify word immediate
e_extrwi rA,rS,*n*,*b* (*n* > 0)	**e_rlwinm rA,rS,***b* + *n*,32 – *n*,**31**	Extract and right justify word immediate
e_inslwi rA,rS,*n*,*b* (*n* > 0)	**e_rlwimi rA,rS,**32 – *b*,*b*,(*b* + *n*) **– 1**	Insert from left word immediate
e_insrwi rA,rS,*n*,*b* (*n* > 0)	**e_rlwimi rA,rS,**32 – (*b* + *n*),*b*,(*b* + *n*) – 1	Insert from right word immediate
iseleq rD,rA,rB	**isel rD,rA,rB,2**	Integer Select Equal
iselgt rD,rA,rB	**isel rD,rA,rB,1**	Integer Select Greater Than
isellt rD,rA,rB	**isel rD,rA,rB,0**	Integer Select Less Than
e_la rD,d(**rA**)	**e_add16i rD,rA,**d	Load address
e_nop	**e_ori 0,0,0**	No-op
se_nop	**se_or 0,0**	No-op
not rA,rS	**nor rA,rS,rS**	NOT (Complement register)
e_rotlwi rA,rS,*n*	**e_rlwinm rA,rS,***n*,**0,31**	Rotate left word immediate
e_rotrwi rA,rS,*n*	**e_rlwinm rA,rS,**32 – *n*,**0,31**	Rotate right word immediate
e_slwi rA,rS,*n* (*n* < 32)	**e_rlwinm rA,rS,***n*,**0,**31 – *n*	Shift left word immediate
e_srwi rA,rS,*n* (*n* < 32)	**e_rlwinm rA,rS,**32 – *n*,*n*,**31**	Shift right word immediate
sub rD,rA,rB	**subf rD,rB,rA**	Subtract from
sub. rD,rA,rB	**subf. rD,rB,rA**	Subtract from
subo rD,rA,rB	**subf rD,rB,rA**	Subtract from
subo. rD,rA,rB	**subf. rD,rB,rA**	Subtract from
subc rD,rA,rB	**subfc rD,rB,rA**	Subtract from carrying
subc. rD,rA,rB	**subfc. rD,rB,rA**	Subtract from carrying

Table A-23. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
subco rD,rA,rB	**subfco** rD,rB,rA	Subtract from carrying
subco. rD,rA,rB	**subfco.** rD,rB,rA	Subtract from carrying
e_sub16i rD,rA,value	**e_add16i** rD,rA,–value	Subtract immediate
e_sub2i. rA,value	**e_add2i.** rA,–value	Subtract 2 operand immediate and recorded
e_sub2is rA,value	**e_add2is** rA,–value	Subtract 2 operand shifted immediate
e_subi rD,rA,value	**e_addi** rD,rA,–value	Subtract immediate
e_subic rD,rA,value	**e_addic** rD,rA,–value	Subtract immediate carrying
e_subic. rD,rA,value	**e_addic.** rD,rA,–value	Subtract immediate carrying
trap	tw 31,0,0	Trap unconditionally
tweq rA,rB	**tw 4,**rA,rB	Trap if equal
twge rA,rB	**tw 12,**rA,rB	Trap if greater than or equal
twgt rA,rB	**tw 8,**rA,rB	Trap if greater than
twle rA,rB	**tw 20,**rA,rB	Trap if less than or equal
twlge rA,rB	**tw 12,**rA,rB	Trap if logically greater than or equal
twlgt rA,rB	**tw 1,**rA,rB	Trap if logically greater than
twlle rA,rB	**tw 6,**rA,rB	Trap if logically less than or equal
twllt rA,rB	**tw 2,**rA,rB	Trap if logically less than
twlng rA,rB	**tw 6,**rA,rB	Trap if logically not greater than
twlnl rA,rB	**tw 5,**rA,rB	Trap if logically not less than
twlt rA,rB	**tw 16,**rA,rB	Trap if less than
twne rA,rB	**tw 24,**rA,rB	Trap if not equal
twng rA,rB	**tw 20,**rA,rB	Trap if not greater than
twnl rA,rB	**tw 12,**rA,rB	Trap if not less than

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

2 The value in the BI32 or BI16 operand selects CR*n*[2], the EQ bit.

3 Instructions for which B032 or BO16 is either 1 (branch if condition true) or 0 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by BI32 or BI16, as described in Section A.5.2, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO32 and BO16 and Replaces BI32 with crS).”

4 The value in the BI32 or BI16 operand selects CR*n*[0], the LT bit.

5 The value in the BI32 or BI16 operand selects CR*n*[1], the GT bit.

6 The value in the BI32 or BI16 operand selects CR*n*[3], the SO bit.

Appendix B
Revision History

Table B-1 provides a revision history for this document.

Table B-1. Document Revision History

Rev. Number	Date	Editor/ Writer	Substantive Change(s)
1	02/21/2006	MC/JY	Edited language and formats throughout.
0	12/22/2005	MC/JY	Initial release.

Glossary

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Note that some terms are defined in the context of how they are used in this book.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

B

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte. *See* Little-endian.

C

Cache . High-speed memory component containing recently-accessed data and/or instructions (subset of main memory).

D

Denormalized number. A nonzero floating-point number whose *exponent* has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

Effective address (EA). The 32- or 64-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. *See also* Biased exponent.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

L

Least-significant bit (lsb). The bit of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. *See* Big-endian.

M

Mnemonic. The abbreviated name of an instruction used for coding.

Modulo. A value *v* which lies outside the range of numbers representable by an n-bit wide destination type is replaced by the low-order n bits of the two's complement representation of *v.*

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

N

NaN . An abbreviation for 'Not a Number'; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs (SNaNs) and quiet NaNs (QNaNs).

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits.

R

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation. Its presence is denoted by a "." following the mnemonic.

Reserved field. In a register, a reserved field is one that is not assigned a function. A reserved field may be a single bit. The handling of reserved bits is *implementation-dependent*. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Saturate. A value *v* which lies outside the range of numbers representable by a destination type is replaced by the representable number closest to *v*.

Signaling NaN. A type of *NaN* that generates an invalid operation program exception when it is specified as arithmetic operands. *See* Quiet NaN.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Sticky bit. A bit that when *set* must be cleared explicitly.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

T

Tiny. A floating-point value that is too small to be represented for a particular precision format, including *denormalized* numbers; they do not include ±0.

U

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or mantissa than the single-precision format can provide. In other words, the result is too small to be represented accurately.

User mode. The unprivileged operating state of a processor used typically by application software. In user mode, software can only access certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

W

Word. A 32-bit data element.