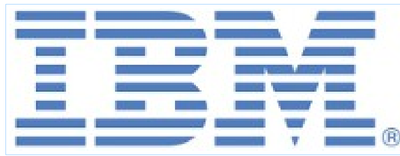


PPE 42 & PPE 42X

Embedded Processor Core

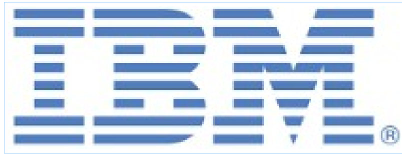
User's Manual

Version 4.0

September 19, 2019

IBM Corporation
Systems Group
11400 Burnet Road
Austin, Texas 78758
c/o Michael Floyd, mfloyd@us.ibm.com

Don't Panic



© Copyright International Business Machines Corporation 2019

Printed in the United States of America September 19, 2019

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

Note: This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design. While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com®**.

Version 4.0
September 19, 2019

Acknowledgments

This manual borrows heavily in organization, substance and style from two other public documents bearing IBM Copyrights:

- PowerPC 405-S Embedded Processor Core, User's Manual, Version 1.2, June 16, 2010. Available from: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores

- Power ISA^TM^, Version 2.07, May 3, 2013. Available from http://www.power.org

We would like to acknowledge the sizable contributions made to the PPE Architecture and to this document by Bishop Brock (https://github.com/bcbrock) while at IBM.

We also acknowledge the unattributed authors of the above for their contributions to this document.

Table of Contents

1 Overview....12
1.1 Audience....13
1.2 Notation....14
2 Programming Model....16
2.1 Registers....16
2.1.1 Programmer Visible Registers....16
2.1.2 Externally Visible Registers....17
2.2 Interface Signals....17
2.3 Privilege Levels....18
2.4 Memory Organization and Addressing....18
2.4.1 Data Types and Byte Ordering....18
2.4.2 Alignment....19
2.5 Instruction Processing....19
2.6 Exception Processing....20
2.7 Branch Processing....22
2.7.1 Branch Target Addressing Options....22
2.7.2 Conditional Branch Operations....22
2.7.3 Fused Compare-Branch Operations....22
2.8 Precise and Imprecise Memory Accesses....23
2.9 Synchronization....24
2.9.1 Synchronization and Storage Ordering....24
2.9.2 Synchronization, Interrupts and Error Reporting....25
2.10 Non-Maskable Interrupts....25
2.11 Special-Purpose Registers....26
2.11.1 Link Register – LR....26
2.11.2 Count Register – CTR....26
2.11.3 Condition Register – CR....26
2.11.3.1 CR[CR0] Fields After Comparison Instructions....27
2.11.3.2 CR[CR0] Fields After Update-Form Instructions....27
2.11.3.3 mtcr0 and mfcr....28
2.11.4 Fixed-Point Exception Register – XER....28
2.11.5 Machine State Register – MSR....29
2.11.5.1 Interrupt Processing and Control....29
2.11.5.2 WAIT mode....30
2.11.5.3 Imprecise Mode Enable....30
2.11.5.4 SIB Error Reporting and Accumulation....30
2.11.5.5 Low-Priority Mode....30
2.11.5.6 Instance-Specific Control....30
3 Initialization,Reset, and Starting Execution....32
3.1 Initial State....32
3.2 Reset Operations....32
3.3 Core State Subsequent to a Reset Event....33
3.4 Starting Instructions....34
3.5 System Reset Interrupt Handler....34
4 Interrupts and Exceptions....35
4.1 Architectural Definitions and PPE 42 Behavior....35
4.1.1 Interrupt Precision....35

4.1.2 Asynchronous, Synchronous, and Machine Check Interrupts....36
4.1.3 Interrupt Address Reporting....37
4.2 Interrupt Vector Offsets....37
4.3 Interrupt Handling....38
4.3.1 Interrupt Masking....38
4.3.2 Interrupt Priority....38
4.3.3 Interrupt Processing....40
4.3.4 Interrupt Halt Semantics....41
4.3.5 Unmaskable Interrupt Promotion....41
4.4 General Interrupt Handling Registers....42
4.4.1 Machine State Register – MSR....42
4.4.2 Save/Restore Registers 0 and 1 – SRR0/1....42
4.4.3 Interrupt Vector Prefix Register – IVPR....43
4.4.4 Interrupt Status Register – ISR....43
4.4.5 Error Data Register – EDR....43
4.5 Detailed Interrupt Descriptions....43
4.5.1 Machine Check Interrupt – PPE 42 Vector x'000'; PPE 42X Vector x'020'....44
4.5.1.1 Service Interface Bus (SIB) Error Reporting and Handling....44
4.5.1.2 Instruction Machine Check Handling....45
4.5.1.3 Data Machine Check Handling for Load-Type Operations....46
4.5.1.4 Data Machine Check Handling for Store-type Operations....47
4.5.1.5 Machine Checks Promoted from Other Unmaskable Interrupts....48
4.5.2 System Reset Interrupt – Vector x'040'....49
4.5.3 Data Storage Interrupt – Vector x'060'....50
4.5.4 Instruction Storage Interrupt – Vector x'080'....51
4.5.5 External Interrupt – Vector x'0A0'....52
4.5.5.1 External Interrupt Recognition; Phantom Interrupt Avoidance....52
4.5.6 Alignment Interrupt – Vector x'0C0'....53
4.5.7 Program Interrupt – Vector x'0E0'....54
4.5.8 Decrementer (DEC) Interrupt – Vector x'100'....55
4.5.9 Fixed Interval Timer (FIT) Interrupt – Vector x'120'....56
4.5.10 Watchdog Timer (WDT) Interrupt – Vector x'140'....57
5 Timer Facilities....58
5.1 The Decrementer (DEC)....59
5.1.1 Using DEC as a Programmable Interval Timer....59
5.1.2 Using DEC to Emulate a Timebase....60
5.2 The Fixed Interval Timer (FIT)....60
5.3 The Watchdog Timer (WDT)....61
5.3.1 Implications of TSR[ENW]....62
5.4 Debug Behavior....62
5.5 Reset Behavior....62
6 External Interface Registers....63
7 Debugging....66
7.1 External Debug Mode....66
7.2 Processor Control....66
7.3 Processor Status....67
7.3.1 Status outputs....67
7.3.1.1 Halted indication....67
7.3.1.2 Watchdog Timeout indication....67
7.3.1.3 Error indications....67
7.4 Debug Registers....69
7.4.1 DACR – Debug Address Compare Register....69

7.4.2 DBCR – Debug Control Register....69
7.4.3 EDR – Error Data Register....69
7.4.4 ISR – Interrupt Status Register....69
7.4.5 XCR – External Control Register....69
7.4.6 XSR – External Status Register....69
7.5 Debug Events....70
7.5.1 Trap Events....70
7.5.2 Instruction-Address Comparison Events....70
7.5.3 Data-Address Comparison Events....71
7.5.4 Zero Address Comparison....71
7.5.5 Data Address Comparison and Alignment....72
7.6 Halt Processing....72
7.6.1 Definition of Halted....72
7.6.2 Entering the Halted state....72
7.6.3 Halt Conditions and Error indication....73
7.6.4 Exiting the Halted state....73
7.6.5 Halting and Synchronization....74
7.7 Single-Stepping and Ramming....74
7.7.1 Single-Stepping....74
7.7.1.1 Single-stepping and Exceptions....74
7.7.2 Ramming....75
7.8 Debugging Procedures....77
7.8.1 Basic Debugging Procedures....77
7.8.1.1 Halting the Processor....77
7.8.1.2 Force-Halting the Processor....77
7.8.1.3 Clearing Debug Halt Status....77
7.8.1.4 Resetting the Processor....77
7.8.1.5 Restarting the Processor....77
7.8.1.6 Single-Stepping an Instruction....78
7.8.1.7 Ramming an Instruction....78
7.8.1.8 Low-overhead Ramming....79
7.8.1.9 Toggling XSR[TRH]....79
7.8.2 Advanced Debugging Procedures....79
7.8.2.1 Reading Status and IAR Contents Simultaneously....79
7.8.2.2 Reading Status and SPRG0 Simultaneously....80
7.8.2.3 Writing IR and SPRG0 Simultaneously....80
7.8.2.4 Writing XCR and SPRG0 Simultaneously....80
7.8.2.5 Writing XSR and IAR Simultaneously....80
7.8.2.6 Reading CTR....80
7.8.2.7 Reading SRR0 and LR Simultaneously....80
7.8.2.8 Reading GPR pairs (VDRs) Simultaneously....80
8 Register Summary....81
8.1 Reserved Registers....81
8.2 Reserved Fields....81
8.3 General Purpose Registers....81
8.4 Virtual Doubleword Registers....82
8.5 Machine State Register and Condition Register....83
8.6 Special Purpose Registers....83
8.6.1 Using SPRs as Scratch Registers....84
8.7 External Interface Registers....85
8.8 Simultaneous Update....85
8.9 Initialization and Reset....85
8.10 Alphabetical Listing of PPE 42 Registers....86

8.10.1 CR – Condition Register....87
8.10.2 CTR – Count Register....88
8.10.3 DACR – Debug Address Compare Register....89
8.10.4 DBCR – Debug Control Register....90
8.10.5 DEC – Decrementer....92
8.10.6 EDR – Error Data Register....93
8.10.7 IAR – Instruction Address Register....94
8.10.8 IR – Instruction Register....95
8.10.9 ISR – Interrupt Status Register....96
8.10.10 IVPR – Interrupt Vector Prefix Register....98
8.10.11 LR – Link Register....99
8.10.12 MSR – Machine State Register....100
8.10.13 PIR – Processor Identification Register....103
8.10.14 PVR – Processor Version Register....104
8.10.15 SPRG0 – SPR General 0....105
8.10.16 SRR0 – Save Restore Register 0....106
8.10.17 SRR1 – Save Restore Register 1....107
8.10.18 TCR – Timer Control Register....108
8.10.19 TSR – Timer Status Register....109
8.10.20 XCR – External Control Register....110
8.10.21 XER – Fixed Point Exception Register....112
8.10.22 XSR – External Status Register....113
9 Instruction Set....116
9.1 Instruction Set Origin and Portability....116
9.2 Rationale for the PPE 42 Instruction Set....118
9.2.1 PPE 42X Added Instructions....118
9.2.2 PPE 42 New Instructions....119
9.2.3 PPE 42X New Instructions....120
9.3 Instruction Formats....120
9.3.1 PPE 42 Specific Instruction Format....121
9.3.2 PPE 42X Specific Instruction Format....121
9.4 Alphabetical Instruction Listing....122
9.4.1 add....123
9.4.2 addc....124
9.4.3 adde....125
9.4.4 addi....126
9.4.5 addic....127
9.4.6 addic....128
9.4.7 addis....129
9.4.8 addme....130
9.4.9 addze....131
9.4.10 and....132
9.4.11 andc....133
9.4.12 andi....134
9.4.13 andis....135
9.4.14 b....136
9.4.15 bc....137
9.4.16 bcctr....139
9.4.17 bclr....141
9.4.18 bnbw....143
9.4.19 bnbwi....145
9.4.20 clrbwbc....147
9.4.21 clrbwibc....149

9.4.22 cmplw 151
9.4.23 cmplwbc 152
9.4.24 cmplwi 155
9.4.25 cmpw 156
9.4.26 cmpwbc 157
9.4.27 cmpwi 160
9.4.28 cmpwibc 161
9.4.29 cntlzw 164
9.4.30 dcbf 165
9.4.31 dcbi 166
9.4.32 dcbq 167
9.4.33 dcbt 169
9.4.34 dcbz 170
9.4.35 eqv 171
9.4.36 extsb 172
9.4.37 extsh 173
9.4.38 lbz 174
9.4.39 lbzu 175
9.4.40 lbzx 176
9.4.41 lcxu 177
9.4.42 lhz 180
9.4.43 lhzu 181
9.4.44 lhzx 182
9.4.45 lsku 183
9.4.46 lvd 186
9.4.47 lvdu 187
9.4.48 lvdx 188
9.4.49 lwz 189
9.4.50 lwzu 190
9.4.51 lwzx 191
9.4.52 mfcr 192
9.4.53 mfmsr 193
9.4.54 mfspr 194
9.4.55 mtcr0 195
9.4.56 mtmsr 196
9.4.57 mtspr 197
9.4.58 mullhw 198
9.4.59 mullhwu 199
9.4.60 mulli 200
9.4.61 mullw 201
9.4.62 nand 202
9.4.63 neg 203
9.4.64 nor 204
9.4.65 or 205
9.4.66 orc 206
9.4.67 ori 207
9.4.68 oris 208
9.4.69 rfi 209
9.4.70 rldicl 210
9.4.71 rldicr 212
9.4.72 rldimi 213
9.4.73 rlwimi 214
9.4.74 rlwinm 215
9.4.75 rlwnm 217

9.4.76 slvd........ 218
9.4.77 slw........ 219
9.4.78 sraw........ 220
9.4.79 srawi........ 221
9.4.80 srvd........ 222
9.4.81 srw........ 223
9.4.82 stb........ 224
9.4.83 stbu........ 225
9.4.84 stbx........ 226
9.4.85 stcxu........ 227
9.4.86 sth........ 230
9.4.87 sthu........ 231
9.4.88 sthx........ 232
9.4.89 stsku........ 233
9.4.90 stvd........ 236
9.4.91 stvdu........ 237
9.4.92 stvdx........ 238
9.4.93 stw........ 239
9.4.94 stwu........ 240
9.4.95 stwx........ 241
9.4.96 subf........ 242
9.4.97 subfc........ 243
9.4.98 subfe........ 244
9.4.99 subfic........ 245
9.4.100 subfme........ 246
9.4.101 subfze........ 247
9.4.102 sync........ 248
9.4.103 tw 249
9.4.104 wrtee........ 251
9.4.105 wrteei........ 253
9.4.106 xor........ 255
9.4.107 xori........ 256
9.4.108 xoris........ 257
9.5 Instruction Set Mnemonics List........ 258

Source Documents

The latest version of this document can be accessed internal to IBM via the following URL:

https://ibm.box.com/ Link for PPE 42X Core Users Manual.pdf

The source documents for this document can be accessed internal to IBM via the following URL:

https://ibm.box.com/ Link for PPE42X Users Manual folder

Note that change bars for sub-documents of Open Office master documents are treated inconsistently by Open Office in general, and may be lost when Open Office master documents are rendered into PDF. In order to view change markings of sub-documents it may be necessary to view the Open Office sub-documents directly.

Revision History

Version	Date	Changes	Author
4.0	09/16/2019	• First “official” PPE42X Release	mfloyd

Cross Reference Table

The following table is included as a reference for future editing. Cross-references between sub-documents must be inserted and named explicitly. Note that when viewing a sub-document that references another sub-document, these cross references will appear as “Error: Reference not found” even when they are valid references. It will be necessary to view a printed or PDF version of the document to check whether cross-sub-document references are correct.

Table 1: Sub-document Cross Reference Table

PPE 42 Core Overview	
Tag	**Reference**
PPE 42 Core Programming Model	
Tag	**Reference**
precise and imprecise memory accesses	1.8 Precise and Imprecise Memory Accesses
MSR	1.12.5 Machine State Register - MSR
wait mode	1.12.5.2 WAIT mode
PPE 42 Core Initialization and Reset	
Tag	**Reference**
initialization and reset	1 Initialization and Reset
PPE 42 Core Interrupts and Exceptions	
Tag	**Reference**
interrupts and exceptions	1 Interrupts and Exceptions
interrupt vector offsets	1.2 Interrupt Vector Offsets
general interrupt handling registers	1.6 General Interrupt Handling Registers
PPE 42 Core Timer Facilities	
Tag	**Reference**
timer facilities	1 Timer Facilities
PPE 42 Core External Interface Registers	
Tag	**Reference**
external interface registers	1 External Interface Registers
PPE 42 Core Debugging	
Tag	**Reference**
debugging	1 Debugging
PPE 42 Core Instruction Set	
Tag	**Reference**
PPE 42 Core Register Summary	
Tag	**Reference**
register summary	1 Register Summary
virtual doubleword registers	1.4 Virtual Doubleword Registers
special purpose registers	1.6 Special Purpose Registers

1 Overview

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

It seems that perfection is attained not when there is nothing left to add, but when there is nothing left to take away.

- Antoine de Saint-Exupéry

The **P**ower ISA Lite **P**rocessing **E**ngine (**PPE**, pronounced *"peppy"*) model 42 embedded processor core implements an extended subset of the Power ISA™ Version 2.07 specification. PPE 42 is a 32-bit processor, with extensions that provide for atomic loads and stores of 64-bit data. PPE 42 implements subsets or modified subsets of the Power ISA 32-bit User Instruction Set Architecture, Virtual Environment Architecture and Embedded Operating Environment Architecture. PPE 42X is fully based on the PPE 42 processor and additionally implements a modified subset of the Power ISA 64-bit User Instruction Set Architecture. PPE 42 and PPE 42X also implement several instructions and capabilities unique to each processor.

As an extended subset of the Power ISA, PPE 42 is *not* Power ISA compliant, and the architectural features specific to PPE will *not* be included in future Power ISA specifications. PPE 42 is derived from the Power ISA to take advantage of the synergies of design, verification, tools and firmware development within the IBM POWER Systems development organization that would not be realized if any other architecture for PPE 42 had been chosen. The PPE architecture supports the PowerPC Embedded Application Binary Interface (EABI) for 32-bit processors, which is critical to allow the modification of existing PowerPC compiler infrastructures with minimum effort. PPE 42X adds native support for managing the stack frame compatible with EABI Version 4.

The PPE architecture balances area-efficiency with function and performance. PPE 42 implements only 16 of the 32 Power ISA general purpose registers (GPRs) along with a selected set of special-purpose registers (SPRs), comprising less than 1000 bits of architected state. A large subset of the 97 implemented instructions can execute pipelined in a single cycle; other instructions execute in two or three cycles plus any memory or synchronization delays. PPE 42X implements an additional 9 instructions as an extension of the PPE 42 Instruction Set, consisting of 7 single cycle arithmetics and 2 new multi-cycle load and store instructions to assist with stack frame and processor context management.

This user's manual provides an architectural overview, programming model, and detailed functional information about the PPE 42 and 42X embedded processor cores. The information in this user's manual includes details on the instruction set, registers, and the various functions and functional units of the core including interrupts, timers and debugging infrastructure.

This manual only covers the PPE 42 and 42X processor core itself. The PPE 42 core provides interfaces that allow it to be integrated with a variety of memory and peripheral subsystems. Therefore many aspects of a complete embedded processing system architecture are not covered in this manual, and will be documented with the specific instantiations of the core. These aspects include caches, memory management, peripheral units and system-level error handling.

1.1 Audience

This book is for system hardware and software engineers who are engaged in hardware development, firmware and operating system support, and application software development. This manual assumes an understanding of embedded processor design, embedded system design, operating systems and reduced instruction set computing (RISC). The reader may also find it useful to refer to the Power ISA Version 2.07 specification, available from www.power.org.

1.2 Notation

The table below describes the notational conventions used in this manual.

Table 1.1: Notational Conventions

Notation	Meaning
n	A decimal number
x'n … n'	A hexadecimal number
'n … n'	A binary number
=	Assignment
∧	AND logical operator
¬	NOT logical operator
∨	OR logical operator
⊕	Exclusive-OR (XOR) operator
+	Twos complement addition
-	Twos complement subtraction, unary minus
×	Multiplication
÷	Division yielding a quotient
%	Remainder of an integer division
$\Vert$	Concatenation
=, ≠	Equal, not equal relations
<, >	Signed comparison results
$<^{u}$, $>^{u}$	Unsigned comparison results
If … then … else	Conditional execution; if *condition* then *a* else *b*, where *a* and *b* represent one or more pseudocode statements. Indenting indicates the ranges of *a* and *b*. If *b* is null, the else does not appear.
do	Do loop. “to” and “by” clauses specify incrementing an iteration variable; “while” and “until” clauses specify terminating conditions. Indenting indicates the scope of a loop.
leave	Leave innermost do loop or do loop specified in a leave statement.
FLD	An instruction or register field
FLD_{b}	A bit in a named instruction or register field
$FLD_{b:b}$	A range of bits in a named instruction or register field
REG_{b}	A bit in a named register
$REG_{b:b}$	A range of bits in a named register
REG[FLD]	A field in a named register
REG[FLD, FLD ...]	A list of fields in a named register
(Rx)	The contents of a GPR specified as an instruction field, where *x* is A, B, S or T
$(RA \mid 0)$	The contents of the register RA, or 0 if RA is 0
^{n}b	The bit or bit value *b* replicated *n* times
EXTS(x)	The result of extending *x* on the left with sign bits (high-order bits)
CIA	Current instruction address; The 32-bit address of the instruction being described by a sequence of pseudocode.
NIA	Next instruction address; The 32-bit address of the next instruction to be executed. In pseudocode, a successful

Notation	Meaning
	branch is indicated by assigning a value to NIA. For instructions that do not branch, the NIA is CIA +4.
MS(addr, n)	The *n* bytes at the location in main storage represented by *addr*.
EA	Effective address; the 32-bit address, derived by applying indexing or indirect addressing rules to the specified operand, that specifies a location in main storage.
ROTL((RS), n)	Rotate left; the contents of RS are rotated left the number of bits specified by *n*
MASK(MB, ME)	Mask having ones in positions MB through ME (wrapping if MB > ME) and zeros elsewhere.
DW(x..y)	A series of Virtual Doubleword Registers or pairs of concatenated Special Purpose Registers, numbered x to y
DW(n)	Doubleword register n, selected from a series of registers
GPR(r)	General Purpose Register r, where 0 ≤ r ≤ 31
VDR(r)	Virtual Doubleword Register r, where 0 ≤ r ≤ 31
(GPR(r))	The contents of GPR(r)
(VDR(r))	The contents of VDR(r)
(DW(n))	The contents of DW(n)
(LR)	The contents of the Link Register.

2 Programming Model

This sections covers aspects of the PPE 42 programming model that are not discussed elsewhere in this manual.

2.1 Registers

The following two illustrations show the PPE 42X register programming model from either a programmer or an external user's view.

2.1.1 Programmer Visible Registers

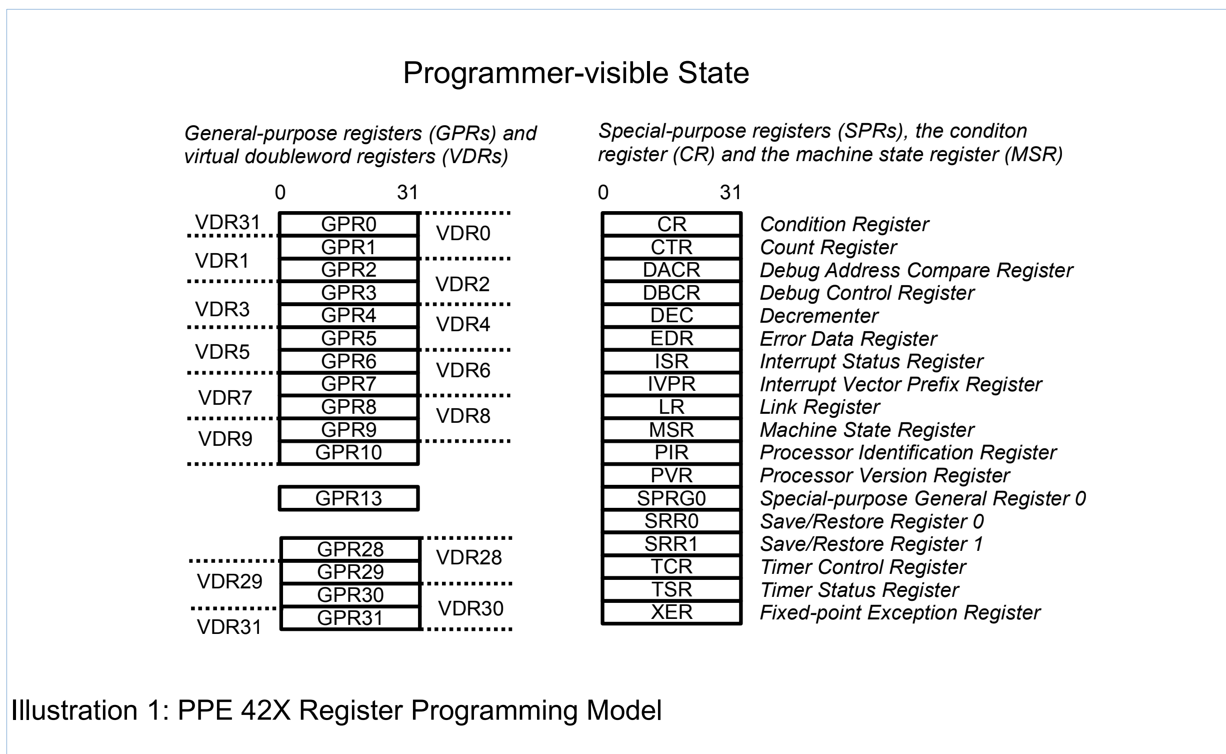


Illustration 1: PPE 42X Register Programming Model

The programmer-visible state, which is the same for both PPE 42 and PPE42X, includes sixteen 32-bit general-purpose registers (GPRs), and eighteen 32-bit special-purpose registers (SPRs). Pairs of consecutively-numbered (modulo 32) GPRs can be used as virtual doubleword registers (VDRs) for 64-bit loads and stores, in addition to selected 64-bit shift and rotate arithmetics in the PPE 42X.

2.1.2 Externally Visible Registers

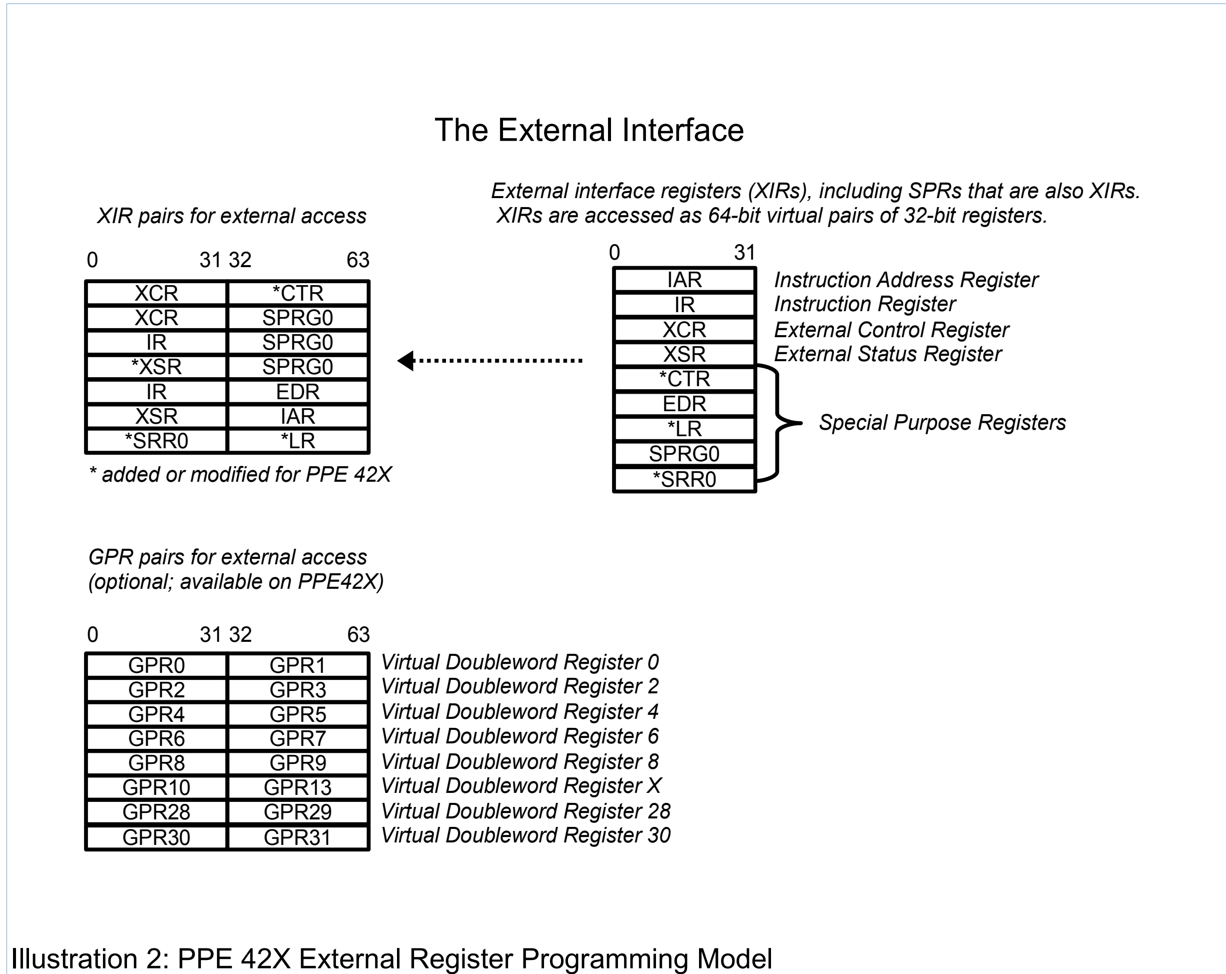


Illustration 2: PPE 42X External Register Programming Model

External control and status is provided by 4 dedicated 32-bit external interface registers (XIRs). Two SPRs (EDR and SPGR0) on PPE 42 are both SPRs and XIRs, with three additional XIR-accessible SPRs (CTR, LR, and SRR0) on PPE 42X. Although architected as 32 bit registers, XIRs are always accessed as 64-bit pairs of 32-bit registers. PPE 42X provides optional access to the even-numbered VDRs (0,2,4,6,8,28, & 30), as well as a psuedo “VDRX” consisting of the GPR10_GPR13 pair, such that all 16 GPRs are readily visible when this feature is enabled.

2.2 Interface Signals

The table below details the major input and output signals of the PPE 42 core that bear on the programming model. The table is not intended as a complete hardware specification, but is only included to help clarify which operations are fully internal to the core vs. those operations controlled by and dependent on the instance-specific environment.

Table 1.2: Selected PPE 42 Core I/O Signals

Dir.	Signal	Bits	Notes
out	`inst_req_addr`	32	A 32-bit, 4-byte aligned instruction address is presented to the memory interface, which returns a

Dir.	Signal	Bits	Notes
in	instruction	32	4-byte instruction.
out	data_req_addr	32	32-bit data addresses are presented to the memory interface. The PPE 42 core supports 8, 16, 32, and 64-bit atomic loads and stores.
in	load_data	64	
out	store_data	64	
in	halt_req	1	Instance-specific logic external to the core can force the core to halt by asserting this signal.
in	hreset_req	1	Instance-specific logic external to the core can force the core to reset by asserting this signal.
in	timer	4	These four events can be programmably selected as the Fixed Interval Timer and Watchdog Timer events.
in	dec_timer	1	This event can be programmably selected as the decrementer event.
in	ext_intr	1	Indication from instance-specific logic external to the core that causes an External Interrupt exception event.
out	arb_high_priority	1	Allows PPE code with time sensitive requirements to choose to run with heightened priority memory accesses, if supported by the instance-specific environment.
out	error	1	Pulses to indicate that the PPE is now halted due to an error condition occurring. May be used by instance-specific logic external to the core to record status or take an action in response to this event.
out	watchdog_timeout	1	Pulses to indicating that a watchdog timeout occurred and caused the PPE to either halt or reset. May be used by instance-specific logic external to the core to record status or take an action in response to this event.
out	halted	1	Signal indicating that the core is in a halted state (for any reason). May be used by instance-specific logic external to the core to record status or take an action in response to this event.

2.3 Privilege Levels

The PPE 42 architecture does not support privilege levels. All processor resources are available to all programs at all times. For this reason the Power ISA **sc** (system call) instruction is neither required nor implemented by PPE 42. It is up to the programming, external hardware and/or operating environments to enforce code and data separation disciplines if required.

2.4 Memory Organization and Addressing

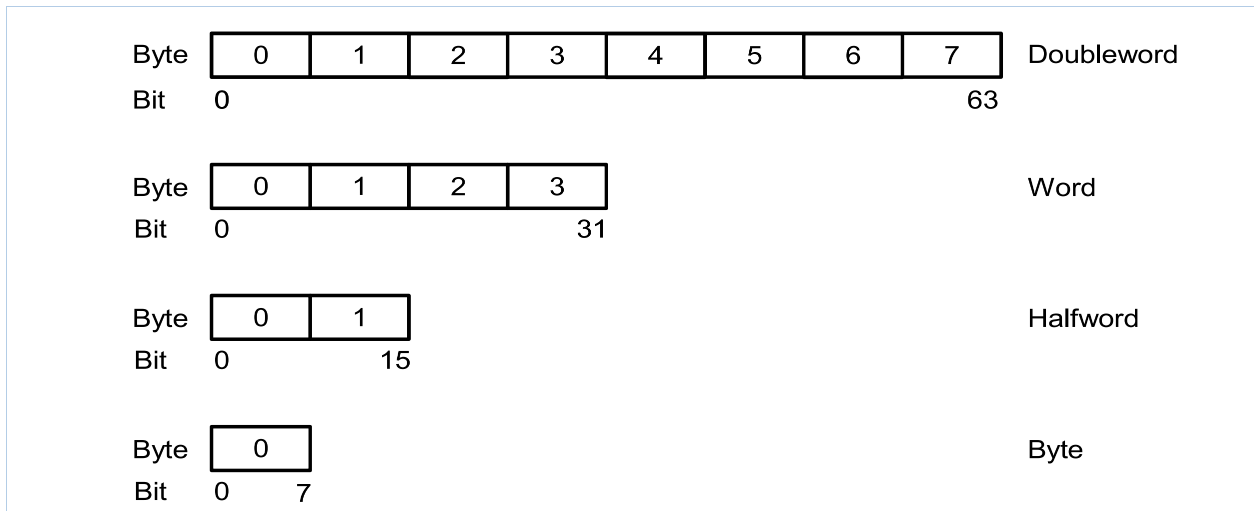
The PPE 42 architecture defines a 32-bit, 4 GB flat address space for instructions and data memory. Almost every feature of the memory subsystem is instance specific however and will be documented with each instance of the PPE 42 core. Instance specific features not defined by this manual include memory maps of defined memory regions, cacheability, access restrictions, speculation, prefetching and addressability and alignment requirements.

Regarding addressability, PPE 42 supports a byte-addressable memory subsystem. Byte addressability is not required however, and the PPE 42 architecture specifically allows accesses to word- and/or doubleword addressable memory spaces to appear as though they were unaligned byte-addressable accesses. The memory subsystem may also signal exceptions for disallowed or ambiguous accesses, for example in response to a byte store to a word-addressable control register space. Again, the interpretation of any memory address is fully instance specific.

2.4.1 Data Types and Byte Ordering

The data types consist of bytes (8 bits), halfwords (2 bytes), words (4 bytes) and doublewords (8 bytes). The figure below illustrates the data types, and also shows their bit and byte definitions for big-endian representation of values. Note that PPE 42 bit numbering is reversed from industry conventions; bit 0

represents the most significant bit of a value. PPE 42 does not support little-endian byte ordering.



PPE 42 only supports the doubleword data type for loads and stores targeting *virtual doubleword* registers. The concept of a virtual doubleword is described in section 8.4, *Virtual Doubleword Registers.* PPE 42 does not implement 64-bit arithmetic or logical instructions, however PPE 42X supports doubleword data types for its five 64-bit shift and rotate instructions.

2.4.2 Alignment

PPE 42 instruction addresses are always presented to the memory interface 4-byte aligned. It is architecturally impossible for the PPE 42 core to request an unaligned instruction address.

Alignment requirements for loads and stores are instance specific and will be documented with each instance. The PPE 42 core presents effective memory addresses to the memory interface as specified with each instruction description, and all alignment exceptions originate from the memory interface, never from the PPE 42 core proper. The only architectural requirement for alignment exceptions is that for the **dcbz** instruction, the memory interface is required to generate an alignment exception if the access is to a non-cacheable memory area, and the memory interface itself does not emulate the **dcbz** instruction.

The data cache control instructions **dcbf**, **dcbi**, **dcbt** and **dcbz** are required to ignore any low-order bits of the effective address that specify bytes within a cache block, and to treat the address of any byte within a cache block as referring to the cache block as a whole. PPE 42 does not specify a cache block size, other than the requirement that the cache block size be an even power of two, greater than or equal to eight.

2.5 Instruction Processing

The PPE 42 core executes instructions in sequential program order. The core supports pipelining of arithmetic, logical and compare instructions only, as these instructions are guaranteed to complete without exception or stalling once they have been successfully decoded. All other instructions are fully fetched, decoded and executed before the next instruction is fetched. Instructions are never fetched speculatively, and the PPE 42 core does not implement branch prediction.

The PPE 42 core executes instructions in two or three cycles, plus memory and synchronization delays. Instructions are fetched and decoded in a single cycle, then most instructions are executed the following cycle assuming no memory delays. Fused compare-branch and synchronizing instructions require a third

cycle. As mentioned previously, execution of arithmetic, logical and compare instructions can be overlapped with the fetch and decode of the following instruction, effectively supporting single-cycle execution for a large class of instructions.

The PPE 42 core supports memory subsystems that provide either single-cycle or delayed responses to instruction fetches and data requests. Although the PPE 42 core implements the *Harvard Architecture,* with separate interfaces for instruction and data accesses, the instruction and data interfaces are never active on the same cycle, thus load, store and cache management instructions require a minimum of two cycles to complete.

The following table details the instruction classes as recognized by the core microarchitecture along with their instruction timings. The table assumes single-cycle instruction access, so does not include any latency in the memory subsystem.

Table 1.3: PPE 42 Core Instruction Classes and Timing

Instruction Class	Instruction(s)	Issue Rate (Cycles)	Notes
A Arithmetic	Arithmetic, logical, compare, **mfspr, mfcr, mtcr0**	1	Since these instructions always complete without exception the cycle after being fetched and decoded, PPE 42 provides single-cycle pipelined execution of Class A instructions.
D Data	Load, store, **dcb***	2 + memory delay	These instructions potentially require blocking while waiting for a response from the memory interface.
B Branch	**b, bc, bcctr, bclr**	2	All branches are fully resolved prior to fetching the next instruction.
F Fused	Fused compare-branch	3	The fused compare-branch instructions execute serialized in 3 cycles. Fused compare-branch instructions provide a code-space advantage but not a direct performance advantage over executing the two underlying instructions.
K Load or Store Stack	**lsku, stsku**	Load = (4 to 24) + (1 to 11) memory delays). Store = (6 to 26) + (3 to 13) memory delays).	These instructions perform multiple operations to memory to push or pop an EABI-compliant stack frame then provide an atomic update of the stack pointer. These instructions potentially require blocking while waiting for responses from the memory interface.
M Move to	**mtspr**, **wrtee**, **wrteei**	2	All **mtspr, wrtee** and **wrteei** execute serialized in 2 cycles. Since many SPRs control asynchronous exceptions, this allows the core to provide a sequential execution model in the presence of asynchronous exceptions. Note that neither the Power ISA nor typical programming applications require **wrtee** and **wrteei** to be context synchronizing.
C Context synchronizing	**mfmsr**, **mtmsr**, **rfi**	3 + context synchronization	**mfmsr, mtmsr** and **rfi** are implemented as context synchronizing instructions by the PPE 42 core. Execution is suspended until all preceding instructions have reported any exceptions they might report.
S Storage synchronizing	**sync**	3 + storage synchronization	**sync** stalls for storage synchronization, which extends context synchronization in certain cases.
T Trap Word	**tw**	1	**trap** form is never executed, but instead always causes an exception or halt. **mark** form is executed as a no-op and causes a trace marker for debug.

2.6 Exception Processing

Exceptions are processed in two machine cycles, plus any delays associated with execution synchronization of the memory interface.

Asynchronous exceptions are implemented as occurring between instructions, specifically before the execution of the next instruction. Synchronous exceptions are implemented as occurring during the execution of an instruction, but before any of the effects of the instruction have been committed to the GPR

state. For further information see section 4, *Interrupts and Exceptions.*

2.7 Branch Processing

The PPE 42 core does not implement branch prediction or any other type of speculative instruction fetching. Whenever a conditional branch instruction is encountered, the next instruction access request to the memory subsystem will always be the correct instruction with respect to the branch condition.

2.7.1 Branch Target Addressing Options

The unconditional branch instruction (**b**) carries the displacement to the branch target address as a signed 26-bit value (the 24-bit LI field right-extended with ‘00’). The displacement enables unconditional branches to cover an address range of ±32 MB.

The conditional branch instruction (**bc**) carries the displacement to the branch target address as a signed 16-bit value (the 14-bit BD field right-extended with ‘00’). The displacement enables conditional branches to cover an address range of ±32 KB.

The PPE 42 fused compare-branch instructions (**bnbw**, **bnbwi**, **clrbwbc**, **clrbwibc**, **cmplwbc**, **cmpwbc**, and **cmpwibc**) carry the displacement to the branch target address as a signed 12-bit value (the 10-bit BDX field right-extended with ‘00’). The displacement enables fused compare-branches to cover an address range of ±2 KB.

For the relative forms of **b** and **bc** (AA is set to '0'), and for all fused compare-branch instructions, the target address is the address of the branch instruction (CIA) plus the signed displacement.

For the absolute (AA is set to ‘1’) forms of **b** and **bc**, the target address is ‘0’ plus the signed displacement. If the sign bit (LI[0] or BD[0]) is ‘0’, the displacement is the target address. If the sign bit is ‘1’, the displacement is a negative value and wraps to the highest memory addresses. For example, if the displacement of an unconditional branch is x‘3FFF FFC’ (the 26-bit representation of -4), the target address is x‘FFFF FFFC’ (0 - 4 bytes, or 4 bytes below the top of memory).

2.7.2 Conditional Branch Operations

Conditional branch instructions can test a CR[CR0] bit. The value of the BI field specifies the bit to be tested. The BO field controls whether the CR bit is tested.

The BO field of the conditional branch instruction specifies the conditions used to control branching, and specifies how the branch affects the CTR. Conditional branch instructions can decrement the CTR by one, and after the decrement, test the CTR value.

For details see the descriptions of the **bc**, **bclr** and **bcctr** instructions.

2.7.3 Fused Compare-Branch Operations

PPE 42 fused compare-branch instructions update CR[CR0] either as the result of a comparison (**cmplwbc**, **cmpwbc**, **cmpwibc**) or as the result of a 32-bit value computation (**bnbw**, **bnbw**, **clrbwbc**, **clrbwibc**).

For the comparison forms, the branch is taken if a bit of the resulting CR[CR0] specified by the BIX field has the same value as the PIX field of the instruction. For the value forms, the branch is controlled by the PIX field and the EQ bit (bit 2) of CR[CR0].

For details see the descriptions of the individual fused compare-branch instructions.

2.8 Precise and Imprecise Memory Accesses

The PPE 42 core implements a hazard-free pipeline. Load-use dependencies are eliminated by requiring all loads to complete in the GPRs before the instruction following the load is executed. This means that the PPE 42 pipeline always stalls for loads, and all errors caused by load instructions are reported precisely.

PPE 42 provides an option to allow the memory subsystem to treat stores imprecisely. Imprecise mode allows the memory interface to process store requests in parallel with continued instruction processing by the core. This may increase performance, at the expense of precise error reporting and recovery.

The imprecise option is selected by setting MSR[IPE] to '1'. Whether the memory subsystem of a PPE 42 instance supports imprecise mode for any interface is instance specific, however if an instance does support imprecise mode then imprecise stores are only allowed when MSR[IPE] = '1'.

The number of outstanding store requests, and the ordering of store completion with respect to a single memory or multiple memories in imprecise mode is instance-specific. Since PPE 42 is an in-order core, it can be guaranteed however that stores are always presented to the memory interface in program order.In precise mode, all errors for loads, stores and all cache management operations are presented precisely. This means that in the event of an error causing an interrupt, SRR0 will contain the address of the erroneous load, store or cache management instruction.

In imprecise mode, errors from stores and all cache management operations may be presented imprecisely. This means that in the event of an error causing a machine check interrupt, SRR0 may not contain the address of the erroneous instruction, but instead may contain the address of a subsequent instruction. However, the EDR will contain the data address associated with the imprecise operation that reported the error. It is also possible that multiple imprecise errors may be reported simultaneously in imprecise mode, and in this case it is instance-specific which of the erroneous memory addresses is reported in the EDR.

Note that even though loads are always precise, an imprecise error for a previous imprecise store or cache management instruction may be reported by a load instruction. In fact, imprecise errors may be reported on *any* type of instruction if the instruction is interrupted by an asynchronous interrupt, and imprecise errors are pending in the memory subsystem. Careful analysis of the EDR, ISR, SRR0 and SRR1 may be required to diagnose and recover from errors in imprecise mode.

The processor always begins execution of interrupt handlers in precise mode. Imprecise mode is enabled and disabled by using the **mtmsr** instruction to modify MSR[IPE]. Since both of these events are context synchronizing, all imprecise errors caused by instructions previous to the entry of the interrupt handler, or previous to the execution of the **mtmsr** instruction will have been reported before execution can continue. Therefore it is never possible for the program to receive a new imprecise error if MSR[IPE] = '0'.

2.9 Synchronization

The following types of synchronization are recognized:

- **Execution Synchronization**: This type of synchronization guarantees that all instructions preceding the synchronizing instruction have completed in the previous context, including the reporting of any and all exceptions the preceding instructions may report.
- **Context Synchronization**: Context synchronization extends execution synchronization with the further requirement that all instructions subsequent to the synchronizing instruction execute in the context established by the synchronizing instruction.
- **Storage Synchronization**: This type of synchronization extends execution synchronization with the further requirement that all storage accesses by the preceding instructions have completed with respect to all mechanisms that access storage.
- **Sequential Execution with Respect to Asynchronous Interrupts**: This type of synchronization guarantees that if any instruction causes or unmasks an asynchronous exception, the associated asynchronous interrupt will be taken before the next sequential instruction is executed.

The PPE 42 core guarantees sequential execution with respect to asynchronous interrupts by serializing any instruction that might cause or unmask an asynchronous interrupt, then taking such an interrupt (or another higher priority interrupt if also pending) prior to fetching the next sequential instruction. The instructions implemented this way are all **mtspr**, **wrtee** and **wrteei.**

PPE 42 implements all synchronizing instructions as context synchronizing. This fact plus sequential execution with respect to asynchronous interrupts means that in general, PPE 42 programs would not require the Power ISA **isync** instruction, therefore PPE 42 does not implement **isync**. The PPE 42 context synchronizing instructions are **mfmsr**, **mtmsr**, **rfi** and **sync**. The **sync** instruction is also storage synchronizing.

2.9.1 Synchronization and Storage Ordering

The PPE 42 core never executes loads out of order, therefore loads are always completed in program order, and any instruction following a load is never executed until the load completes.

Handling of stores is instance-specific. To guarantee store ordering and completion, two methods are available:

1. A series of one or more stores can be followed by a **sync** instruction, to guarantee that all preceding stores are complete before proceeding.
2. The processor can be placed into precise mode (MSR[IPR] = '0') prior to a sequence of stores.

As discussed earlier, in precise mode stores are required to complete up to the point of exception reporting before the core is allowed to continue. For most anticipated memory interfaces, this type of completion is equivalent to storage synchronization. Exceptions would include memory units like write-gathering bus bridges that might report error-free transaction completion immediately, but not flush local buffer contents to the final destination memory (and signal memory synchronization) until later.

2.9.2 Synchronization, Interrupts and Error Reporting

All interrupts are context synchronizing. Whenever an unmasked exception is present, the interrupt taken will always be the highest priority interrupt after execution synchronization. The MSR after execution synchronization is saved in SRR1, and execution continues at the interrupt vector in the context of the new MSR as defined for each interrupt. The MSR saved in SRR1 will record Service Interface Bus (SIB) status of all instructions preceding the interrupt. Similarly, since **mfmsr** is context synchronizing, the value returned by the **mfmsr** instruction includes the SIB status of all instructions preceding the **mfmsr** instruction. See further below for a discussion of the SIB abstraction.

When an interrupt handler returns by executing an **rfi**, if a synchronous imprecise exception is pending in the memory interface it will be reported as an interrupt in lieu of executing the **rfi**. This means that SRR0 and SRR1 will be destroyed, and the **rfi** instruction will not be directly recoverable in this case.

2.10 Non-Maskable Interrupts

The machine check, program, data storage, instruction storage and alignment interrupts are considered *non-maskable*. These interrupts are required to either be processed as soon as they occur, or immediately halt the processor.

The machine check interrupt is taken as an interrupt only if MSR[ME] is '1'. The other unmaskable interrupts are taken as specified only if MSR[UIE] = '1'. However, if a non-maskable interrupt (other than a machine check) occurs when MSR[UIE] = '0' but MSR[ME] = '1', the interrupt is *promoted* and taken as a machine check interrupt. When an interrupt is promoted, all of the ISR updates of the original interrupt take place, however execution continues at the machine check interrupt vector and ISR[MCS] is set to indicate that the machine check interrupt was due to a promoted unmaskable interrupt.

If an unmaskable interrupt can not be taken as an interrupt due to MSR[ME] = '0', then the core processes the interrupt up to the point that the first instruction of the interrupt vector would be fetched, and then halts. Halts due to unmaskable interrupts are reported in the XSR as XSR[HC] = '011'.Service Interface Bus (SIB)

The PPE 42 core provides special architectural resources and error reporting for transactions on a *Service Interface Bus* (SIB). It is instance-specific whether an instance of PPE 42 provides a SIB abstraction, and how it is mapped into the flat 4 GB address space if present.

The SIB abstraction is a memory interface that provides up to 8 unique return codes for memory transactions, where return code 0 indicates “success”. The remaining 7 return codes are not otherwise interpreted by the PPE 42 core, however for data transactions these 7 codes can be individually enabled and disabled from causing a machine check exception. Instruction execution from SIB memory spaces is also supported, however non-0 return codes for SIB instruction accesses are always considered fatal errors and cause a machine check exception.

If a SIB access returns a 0 return code, the assumption is that data loaded into GPRs is correct, and data stored to SIB memories has been stored correctly. In the event of non-0 SIB return codes, the data loaded into a GPR, and the effect on the environment for stores is instance specific.

SIB error control and reporting fields appear in the machine state register (MSR). For further details on SIB support please see the following section 2.11.5, Machine State Register – MSR..

2.11 Special-Purpose Registers

All PPE 42 registers and their fields are documented in section 8, *Register Summary.* This section covers Special-Purpose Registers (SPRs) that are integral to the PPE 42 programming model but not otherwise described in detail in this manual.

2.11.1 Link Register – LR

The LR is written from a GPR using the **mtspr** (**mtlr**) instruction, and read into a GPR using the **mfspr** (**mflr**) instruction. LR is also updated by branch instructions that have the LK bit set to 1. These branch instructions load the LR with the address of the next instruction that follows the branch instruction. Thus, the LR contents can be used as the return address for a subroutine that was called using the branch.

The LR contents can be used as a target address for the branch conditional to link register (**bclr**) instruction. This allows branching to any address.

When the LR contents are used as an instruction address, $LR_{30:31}$ are ignored and substituted with ‘00’ on the instruction interface, as all instruction addressess must be word-aligned. However, when the LR is read or written using an **mfspr** or **mtspr** instruction, all 32 bits are read or written.

With careful programming the LR can also be used as a 32-bit scratch register.

2.11.2 Count Register – CTR

The CTR is written from a GPR using the **mtspr** (**mtctr**) instruction, and read into a GPR using the **mfspr** (**mfctr**) instruction.

Conditional branch instructions (**bc**, **bclr**) can optionally specify the CTR to decrement, and to use a zero/non-zero result of the decremented CTR to control the branch.

The CTR contents can be used as a target address for the branch conditional to counter (**bcctr**) instruction. This allows branching to any address.

When the CTR contents are used as an instruction address, $CTR_{30:31}$ are ignored and substituted with ‘00’ on the instruction interface, as all instruction addresses must be word-aligned. However, when the CTR is read or written using an **mfspr** or **mtspr** instruction, all 32 bits are read or written.

With careful programming the CTR can also be used as a 32-bit scratch register.

2.11.3 Condition Register – CR

The Power ISA Condition Register (CR) contains eight 4-bit fields, CR0 – CR7. PPE 42 only implements the CR0 field, but reserves ISA features for other CR fields to ensure future compatibility.

The CR fields contain conditions detected during the execution of arithmetic and logical instructions, and integer or logical comparison instructions, as indicated in the instruction descriptions. The CR[CR0] contents can be used in conditional branch instructions. The CR[CR0] contents are always used by PPE 42 fused compare-branch instructions.

The CR[CR0] field can be modified in any of the following ways:

- The **mtcr0** instruction sets CR[CR0] by writing to CR[CR0] from a GPR.
- The “with update” forms of arithmetic and logical instructions implicitly update CR[CR0].
- Integer comparison instructions update a specified CR field, which PPE 42 requires to be CR[CR0].
- PPE 42 fused compare-branch instructions implicitly update CR[CR0]

2.11.3.1 CR[CR0] Fields After Comparison Instructions

Comparison instructions compare two 32-bit values. The Power ISA instructions

cmplw, cmplwi, cmpw and **cmpwi,**

and the PPE-specific fused compare-branch instructions

cmplwbc, **cmpwbc** and **cmpwibc**

are considered comparison instructions for the purposes of the following discussion.

The two types of comparison instructions are *arithmetic* instructions and *logical* instructions. These instructions are distinguished by the interpretation given to the 32-bit values compared. For arithmetic comparisons, the values are considered to be signed, 2's complement integers. For logical comparisons, the values are considered to be unsigned.

As an example, consider the comparison of 0 with x'FFFF FFFF'. In an arithmetic comparison, 0 is larger, because x'FFFFFFFF' represents -1; in a logical comparison, x'FFFFFFFF' is larger.

PPE 42 comparison instructions always specify an update of CR[CR0]. This is an encoding restriction for the Power ISA comparison instructions, and implied for the PPE 42 fused compare-branch instructions. The first data operand of a comparison instruction specifies a GPR. The second data operand specifies another GPR, 16-bit signed immediate data derived from the IM field of the immediate instruction form, or 5-bit unsigned (positive) immediate data derived from the UIX field of fused compare-branch immediate-form instructions. The contents of the GPR specified by the first data operand are compared with the contents of the GPR specified by the second data operand (or with the immediate data). See descriptions of the comparison instructions for precise details.

The table below details how CR[CR0] is interpreted after a comparison.

Table 1.4: CR[CR0] After Comparison Instructions

CR[CR0] Bit	Mnemonic	Interpretation
0	LT	The first operand is less than the second operand.
1	GT	The first operand is greater than the second operand.
2	EQ	The first operand is equal to the second operand.
3	SO	Summary overflow; a copy of XER[SO]

Note that comparison instructions do not update XER[SO]. XER[SO] is copied to $CR[CR0_3]$ during comparison instructions for consistency with update-form instructions which may also update XER[SO].

2.11.3.2 CR[CR0] Fields After Update-Form Instructions

Many PPE 42 instructions optionally, explicitly or implicitly update CR[CR0] as described here. These instructions include the optional or explict “dot” forms of arithmetic and logical instructions. The PPE 42 fused compare-branch instructions

bnbw, **bnbwi**, **clrbwbc** and **clrbwibc**

also behave as implicit update-form instructions with respect to CR[CR0].

Update-form instructions compute a 32-bit result. The table below details how the bits of CR[CR0] are set by update-form instructions.

Table 1.5: CR[CR0] After Update-Form Instructions

CR[CR0] Bit	Mnemonic	Interpretation
0	LT	Less than 0; set if the most significant bit of the 32-bit result is '1'.
1	GT	Greater than 0; set if the 32-bit result is non-0 and the most significant bit of the 32-bit result is '0'.
2	EQ	Equal to 0; set if the 32 bit result is 0.
3	SO	Summary overflow; a copy of XER[SO] at instruction completion

The $CR[CR0]_{LT, GT, EQ}$ subfields are set as the result of an algebraic comparison of the instruction result to 0, regardless of the type of instruction that sets CR[CR0]. If the instruction result is ‘0’, the EQ subfield is set to ‘1’. If the result is not 0, either LT or GT is set, depending on the value of the most-significant bit of the result.

When updating CR[CR0], the most significant bit of an instruction result is considered a sign bit, even for instructions that produce results that are not usually thought of as signed. For example, logical instructions such as **and.**, **or.**, and **xor.** update $CR[CR0]_{LT, GT, EQ}$ using such an arithmetic comparison to 0, although the result of such a logical operation is not actually an arithmetic result.

If an arithmetic overflow occurs, the sign of an instruction result indicated in $CR[CR0]_{LT, GT, EQ}$ might not represent the true (infinitely precise) algebraic result of the instruction that set CR[CR0]. For example, if an **addc.** instruction adds two large positive numbers and the magnitude of the result cannot be represented as a 2's complement number in a 32-bit register, an overflow occurs and $CR[CR0]_{LT, SO}$ are set, although the infinitely precise result of the add operation is positive.

Adding the largest 32-bit 2's-complement negative number, x‘80000000’, to itself results in an arithmetic overflow and x‘00000000’ is recorded in the target register. $CR[CR0]_{EQ, SO}$ is set, indicating a result of 0, but the infinitely precise result is negative.

The $CR[CR0]_{SO}$ subfield is a copy of XER[SO]. Instructions that do not alter the XER[SO] bit cannot cause an overflow, but even for these instructions $CR[CR0]_{SO}$ is a copy of XER[SO].

2.11.3.3 mtcr0 and mfcr

The **mtcr0** instruction sets CR[CR0] from the 4 high-order bits of a GPR. The **mfcr** instruction sets the 4 high-order bits of a GPR with the contents of CR[CR0], and clears the 28 low-order bits of the GPR.

2.11.4 Fixed-Point Exception Register – XER

The XER records overflow and carry conditions generated by integer arithmetic instructions. The Summary Overflow (SO) field is set to ‘1’ when instructions cause the Overflow (OV) field to be set to ‘1’. The SO field does not necessarily indicate that an overflow occurred on the most recent arithmetic operation, but that an overflow occurred since the last clearing of XER[SO].

XER[OV] is set to indicate whether an instruction that updates XER[OV] produces a result that overflows the 32-bit target register. Setting the XER[OV] bit to ‘1’ indicates an overflow. For arithmetic operations, an overflow occurs when an operation has a carry-in to the most-significant bit of the result that does not equal the carry-out of the most-significant bit (that is, the exclusive-or of the carry-in and the carry-out is ‘1’).

The **mtspr**(XER) instruction sets XER[SO] to the value of bit position 0 and XER[OV] to the value of bit positions 1 in the source register. When set, XER[SO] is not reset until an **mtspr**(XER) instruction is executed with data that explicitly puts a ‘0’ in the SO bit.

Table 1.6: Instructions that Update XER[SO, OV]

Add	Subtract	Negate	Special
addco[.] **addeo[.]** **addmeo[.]** **addo[.]** **addzeo[.]**	**subfco[.]** **subfeo[.]** **subfmeo[.]** **subfo[.]** **subfzeo[.]**	**nego[.]**	**mtspr** (XER)

The Carry (CA) field is set to indicate whether an instruction that updates the XER[CA] bit produces a result that has a carry-out of the most-significant bit. Setting the XER[CA] bit to ‘1’ indicates a carry. The **mtspr**(XER) instruction sets XER[CA] to the value of bit position 2 in the source register.

The **sraw[.]** and **srawi[.]** instructions set XER[CA] differently. These instructions set XER[CA] to '1' if any 1-bits are shifted out of the least-significant bit position, otherwise '0'. For more information see the instruction description of each instruction.

Table 1.7: Instructions that Update XER[CA]

Add	Subtract	Shift	Special
addco[.] **addeo[.]** **addic** **addic.** **addmeo[.]** **addzeo[.]**	**subfco[.]** **subfeo[.]** **subfic** **subfzeo[.]**	**sraw[.]** **srawi[.]**	**mtspr** (XER)

2.11.5 Machine State Register – MSR

The MSR controls processor core functions, such as the enabling or disabling of interrupts. The MSR also includes error status for SIB accesses, as well as “hint” bits that have no effect on the operation of the core, but may affect the programming model and memory operations outside of the core. MSR fields and layout are summarized in section 8, *Register Summary*.

The MSR is written from a GPR using the **mtmsr** instruction. The contents of the MSR can be read into a GPR using the **mfmsr** instruction. MSR[EE] may also be set or cleared using the write external enable (**wrtee**) or the write external enable immediate **(wrteei)** instructions.

2.11.5.1 Interrupt Processing and Control

The system reset interrupt and the machine check interrupt clear all fields of the MSR except MSR[SIBRC]. All other interrupts clear all fields of the MSR except for MSR[ME] (the machine check interrupt enable) and MSR[SIBRC]. Prior to modifying the MSR during interrupt processing, the contents of the MSR are copied to SRR1. The value of the MSR copied to SRR1 includes the reporting of any error status for all SIB transactions outstanding prior to the interrupt.

MSR[ME] controls whether machine check interrupts are processed (MSR[ME] = '1') or cause the processor to halt (MSR[ME] = '0'). MSR[UIE] controls whether program, alignment, data storage and instruction storage interrupts are processed (MSR[UIE] = '1'), or cause the processor to halt (MSR[UIE] = '0' and MSR[ME] = '0'), or are promoted to machine check interrupts (MSR[UIE] = '0' and MSR[ME] = '1'). MSR[EE] controls whether the external interrupt, decrementer, fixed-interval timer and watchdog interrupts are processed (MSR[EE] = '1') or masked (MSR[ME] = '0').

2.11.5.2 WAIT mode

Setting MSR[WE] = '1' causes the processor to enter the wait state, in which instructions are no longer fetched and executed. The processor remains in the wait state until an asynchronous exception occurs. The processor also leaves the wait state in the event of a system reset, or if the processor is halted by an external agent. Note that since the **mtmsr** instruction that sets MSR[WE] = '1' is context synchronizing, any pending imprecise data machine checks will be reported prior to the processor being allowed to enter the wait state.

2.11.5.3 Imprecise Mode Enable

MSR[IPE] controls whether memory and cache instructions execute precisely or imprecisely. Precise mode is discussed above in section 2.8, *Precise and Imprecise Memory Accesses.*

2.11.5.4 SIB Error Reporting and Accumulation

The MSR[SEM] (SIB error mask) field controls whether SIB return codes 1 through 7 for data loads and stores to SIB memory spaces are either treated as machine check exceptions or ignored for the purposes of error reporting. A SIB return code of 0 for a SIB transaction never generates an error. Non-0 SIB return codes for instruction fetches from SIB memory spaces always cause a machine check exception.

The return code for the last completed SIB data access can always be read from MSR[SIBRC]. MSR[SIBRC] is also updated in the event that a SIB instruction fetch causes an error. Whenever unmasked SIB data errors or SIB instruction errors occur, MSR[SIBRC] is updated with the return code causing the error, and the updated MSR is copied to SRR1 during error interrupt processing, which then clears other fields of the MSR.

MSR[SIBRCA] is the SIB return code accumulator. These status bits record whether any of the SIB return codes have been observed since the last time the bit was cleared. MSR[SIBRCA] includes a bit for SIB return code 0, so that software can observe whether any of a set of SIB accesses succeeded, as well as observing if any non-0 return code was returned by a set of accesses.

MSR[SIBRCA] fields are cleared by a direct write of the bit with 0, so normally they will be cleared by a read-modify-write of the MSR. MSR[SIBRCA] is only updated for erroneous SIB instruction accesses.

2.11.5.5 Low-Priority Mode

The PPE 42 core asserts a priority signal to the environment whenever MSR[LP] = '0' or ISR[EP] = '1'. The value of MSR[LP] and the state of the priority signal have no effect on the operation of the core, and the priority signal is not required to have any effect in the environment.

This mechanism is designed to support resource-constrained environments. The assumption is that PPE 42 will be used in real-time systems where the majority of tasks have fixed deadlines, and thus demand priority access to resources. For this reason the default value of MSR[LP] = '0'. Some tasks however may be viewed as lower in priority, and can elect to set MSR[LP] = '1' to signal lower priority to the environment.

ISR[EP] = '1' also asserts priority to the external environment to help avoid priority inversion. If a task is running with MSR[LP] = '1' and an asynchronous interrupt is also pending, the assumption is that the interrupt needs to be handled with priority, but the low-priority task has interrupts temporarily disabled. Therefore priority is asserted to help enable the low-priority task to complete its critical section as quickly as possible so that the interrupt can be taken. Taking the interrupt re-establishes priority when MSR[LP] is set to '0' during interrupt processing.

2.11.5.6 Instance-Specific Control

The PPE 42 MSR provides 4 uninterpreted control fields, MSR[IS0, IS1, IS2, IS3]. The fields have no effect

on the operation of the core itself, and the behavior they control (if any) is fully defined by the instance-specific environment. Like most other MSR fields, these Instance Specific fields are cleared by all interrupts, restored from SRR1 during execution of **rfi** instructions, and may also be written directly using the **mtmsr** instruction.

3 Initialization,Reset, and Starting Execution

This section covers the initial (power-on, scan-flush) state of the core,state transitions that occur during internal and external resets, as well as how to start executing instructions.

3.1 Initial State

The initial power-on (scan-flush) state of all bits of all architected general-purpose registers (GPRs), special-purpose registers (SPRs) and external interface registers (XIRs) is '0', except for XSR[HCP] which initializes to '1', and the processor control state (observable as XSR[SMS]) which initializes consistent with the core being halted. In other words, the PPE 42 core initializes to the halted state. An instantiation of the PPE 42 core may provide instance-specific means to override these defaults, which will be documented with each instance.

3.2 Reset Operations

The PPE 42 core can be reset in one of four ways:

- By the environment asserting the external `hreset_req` signal for one or more cycles;
- By a second watchdog timer (WDT) event when TCR[WRC] is either '01' or '10';
- By writing DBCR[RST] with either '01' or '10';
- By writing XCR[CMD] with either '101' or '110'.

Two types of reset are supported. A *soft reset* is initiated by a second WDT event when TCR[WRC] = '01' or by writing DBCR[RST] with '01' or by writing XCR[CMD] with '101'. A *hard reset* is initiated by the active level of the external hreset_req signal, a second WDT event when TCR[WRC] = '10',by writing DBCR[RST] with '01' or by writing XCR[CMD] with '110'. The differences between soft and hard resets are instance specific in the memory subsystem and will be documented with each instance of the PPE 42 core. Resets initiated by the `hreset_req` signal also have unique behavior as documented below. In the event that both a hard and soft reset are asserted simultaneously, the hard reset takes priority.

All resets are implemented as interrupts to the processor and cause it to take a system reset interrupt. As a by-product of taking the system reset interrupt, SRR0 and SRR1 will contain the IAR and MSR current at the time of the reset when the system reset vector begins execution. Resets that occur when the processor is halted do not cause instructions to start executing, except for the external `hreset_req` which also clears the halted state.

Note that if the processor is not halted when the reset occurs, the instruction addressed by SRR0 may be unexecuted, partially executed or fully executed with respect to the processor state at the time of the reset. Resets are truly asynchronous events that are handled immediately, aborting any in-flight instruction in execution at the time of the reset event. However a core reset can not complete until the memory subsystem has also acknowledged the reset, which means that reset operations may be delayed or hang. If a reset operation is hung, the only debugging option may be to *force-halt* the processor by writing XCR[CMD] = '111' as described in the *Debugging Procedures* chapter.

Also note that if the reset is initiated externally by the `hreset_req` signal, the processor will effectively remain in the reset state (neither running nor halted) until the `hreset_req` signal is deasserted, at which time the reset-specific state changes are made and instructions start executing at the system reset interrupt vector as soon as the memory subsystem acknowledges the hard reset. How the `hreset_req` is handled by the memory subsystem is instance specific but must always cause it to acknowledge the resultant reset request from the core.

3.3 Core State Subsequent to a Reset Event

Reset operations only modify the minimum amount of state required to ensure deterministic operation after a reset operation completes. This can aid debugging, particularly if the reset was caused by a watchdog timeout. The DBCR is cleared to ensure that no unexpected debug halt events occur, TCR[WRC] is cleared to ensure that no unexpected watchdog events occur, and the system reset interrupt handler can check for a non-zero value in TSR[WRS] to determine is the reset was due to a watchdog timeout.

The table below details the register state after each type of reset. Any part of the architected state not specifically mentioned below does not change as a result of the reset. As noted above, the GPR state and other SPR state may or may not reflect execution of the instruction addressed by SRR0 at entry to the system reset interrupt handler.

Table 1.8: PPE 42 Core State Subsequent to a Reset Event

Register	Field	Field or Register Value by Reset Type			
		hreset_req	**WDT reset**	**DBCR[RST]**	**XCR[CMD]**
DBCR		DBCR ← x'0000 0000'			
IAR		IAR ← IVPR || 0x040 (System Reset Vector)			
ISR	SRSMS	Set to the core state machine state, current on the first cycle reset was asserted			
MSR		MSR[SIBRC] is unchanged; All other MSR fields are reset to 0			
SRR0		SRR0 ← IAR at the time of the reset			
SRR1		SRR1 ← MSR at the time of the reset			
TCR	WRC	TCR[WRC] ← '00'			
TSR	WRS	Unchanged	TSR[WRS] ← TCR[WRC]	Unchanged	Unchanged
XSR	HCP	XSR[HCP] ← '0'	Unchanged	Unchanged	Unchanged
	HC	XSR[HC] ← '000'			
	RIP	XSR[RIP] ← '0'			
	SIP	XSR[SIP] ← '0'			

The programmer will observe the reset state described in the above table at the entry to the system reset interrupt handler. Technically, the state changes described above occur as follows:

- Whenever the processor enters the RESET state from a non-RESET state, the current state machine state is copied to ISR[SRSMS].
- Whenever a second watchdog timeout occurs with TCR[WRC] not equal to '00', TSR[WRS] takes the value of TCR[WRC].
- The processor enters the RESET state for at least one cycle while waiting for the memory subsystem to acknowledged the reset. The state changes to the DBCR, TCR and XSR occur during this time.
- The changes to the MSR, SRR0 and SRR1 occur during processing of the system reset interrupt, which occurs immediately after the memory subsystem has acknowledged the reset.

Note that only the assertion of the external `hreset_req` input clears XSR[HCP], which means that the processor will always begin execution after this event. As a debugging aid, the other reset conditions do not modify XSR[HCP]. This allows the processor to be reset but remain halted, which supports debugging applications from their initial state.

3.4 Starting Instructions

The processor will start executing instructions from a Halted state at the location stored in the IAR from an initial or reset state, unless an external interrupt is pending which is only possible if the PPE has been halted while running. Prior to starting instructions, the IAR can be written directly by the XIR interface. The IAR can also be set to the system reset interrupt vector (IVPR || 0x040) by requesting a reset using one of the above indicated methods.

If the processor is initially powered-on or was reset from a bad state, the IVPR should also be initialized such that any subsequent interrupts fetch from the correct memory location prior to starting instructions. In addition, the XIR interface can optionally be used to RAM registers necessary to define the desired starting machine state, unless relying on software to set those registers on a System Reset as described in the next section.

Otherwise, if the processor experienced a debug halt condition while running, XCR[CMD] should be written with “000” to clear the debug-related XSR fields prior to starting instructions.

Once the desired starting processor state has been established, the XIR interface can be used to to start instructions as follows (using the XIR interface):

1. Write XCR[CMD] with '010'.
2. Optionally check that XSR[HS] = '0', which should be true immediately after step 1 completes.

Instructions can also be started by asserting then deasserting the `hreset_req`, for instances that connect that input, as described in the previous section.

3.5 System Reset Interrupt Handler

Upon the System Reset interrupt taken after being reset, software may want to first save away any debug state of the processor before it gets modified by executing future instructions. This includes a subset of the SPRs such as the TSR, SRR0, SRR1, SPRG0, ISR, LR, and EDR registers which may hold clues of what was happening at the time of the reset.

In the System Reset interrupt handler, software must establish the desired execution behavior, starting from the core state described in the previous section. This involves setting the TCR, DBCR, and MSR to their desired values, then clearing the TSR (since the TSR[WSR] does not get reset and so must rely on software to clear it).

Although not required, it is recommended for cleanliness and subsequent debug for software to also clear the SRR0, SRR1, SPRG0, LR, and EDR registers after receiving a System Reset.

4 Interrupts and Exceptions

An *interrupt* is the action in which the processor saves its old context (MSR and instruction pointer) and begins execution at a predetermined interrupt-handler address, with a modified MSR. *Exceptions* are events that, if enabled, cause the processor to take an interrupt. Exceptions are generated by signals from external peripherals, instructions, internal timer facilities, debug events, or error conditions.

Section 4.2, *Interrupt Vector Offsets* lists the interrupts that the PPE 42 core handles in the order of interrupt vector offsets. Detailed descriptions of each interrupt follow, in the same order.

Several registers support interrupt handling and control. Section 4.4, *General Interrupt Handling Registers* describes the general interrupt handling registers:

- Error Data Register (EDR)
- Interrupt Status Register (ISR)
- Interrupt Vector Prefix Register (IVPR)
- Machine State Register (MSR)
- Save/Restore Registers (SRR0, SRR1)

4.1 Architectural Definitions and PPE 42 Behavior

The default behavior of the PPE 42 core is to handle interrupts *precisely*. An *imprecise* mode is available by setting MSR[IPE] = '1'. In imprecise mode, data machine checks for store operations may be imprecise. Note that the terminology used to describe interrupts in the PPE 42 architecture differs slightly from the Power ISA specification.

4.1.1 Interrupt Precision

In the PPE 42 architecture, an interrupt is said to be *precise* if the following conditions are met:

- The saved instruction pointer must be either the address of the excepting instruction or the address of the next sequential instruction.
- All instructions before the one whose address is reported to the interrupt handling routine in the save/restore register have completed execution. All storage accesses generated by these preceding instructions have also completed to the point that any and all exceptions they might report have been reported.
- All load operations generated by these preceding instructions have completed with respect to the PPE 42 GPRs, including the address updates for update-form addressing.
- All store operations generated by these preceding operations have completed from the point of view of the PPE 42 core. This does not guarantee that these stores have completed with respect to all mechanisms that access storage.
- No subsequent instruction has begun execution, including the instruction whose address is reported to the interrupt handling routine.
- The instruction having its address reported to the interrupt handler has not executed or partially executed with respect to the PPE 42 core architected state, other than updates to error reporting

SPRs. This does not guarantee that state outside of the PPE 42 core has not been modified by any attempted or partial execution of the instruction.

In the PPE 42 architecture, the following conditions are true for *imprecise* interrupts:

- The saved instruction pointer holds the address of an instruction that has not been executed. This instruction may or may not be the cause of the interrupt.
- All instructions before the one whose address is reported to the interrupt handling routine in the save/restore register have completed execution with respect to the PPE 42 core architected state.
- Storage accesses generated by these preceding instructions may or may not have completed, however any exceptions they might report have been reported.
- All load operations generated by these preceding instructions have completed with respect to the PPE 42 GPRs, including the address updates for update-form addressing.
- No subsequent instruction has begun execution, including the instruction whose address is reported to the interrupt handling routine.
- The instruction having its address reported to the interrupt handler has not executed or partially executed with respect to the PPE 42 core architected state, other than updates to error reporting SPRs. This does not guarantee that state outside of the PPE 42 core has not been modified by any attempted or partial execution of the instruction.

The system reset interrupt is a special form of unmaskable interrupt that is imprecise, but can not be guaranteed to satisfy all of the above conditions. All that can be guaranteed about the system reset interrupt is that the saved instruction address and saved MSR were current at the time of the system reset event, and no state changes or exceptions from instructions previous to or contemporary with the saved instruction address will take place or be reported respectively after the system reset interrupt is taken.

4.1.2 Asynchronous, Synchronous, and Machine Check Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. In the PPE 42 architecture the External interrupt and all timer interrupts are asynchronous, precise interrupts. Asynchronous interrupts effectively occur between instructions, and the address reported in SRR0 is always the address of the next sequential (unexecuted) instruction. Technically, asynchronous interrupts are recognized before the execution of the next instruction, rather than after the execution of the current instruction. This distinction is only significant when single-stepping instructions during debugging.

Synchronous interrupts are caused directly by the execution or attempted execution of instructions. Synchronous interrupts can be either precise or imprecise. In the PPE 42 architecture instruction storage, data storage, alignment and program interrupts are synchronous, precise interrupts. Instruction machine check interrupts, all data machine check interrupts for loads, and data machine check interrupts for stores when MSR[IPE] = '0' are also synchronous, precise interrupts. Data machine check interrupts for stores when MSR[IPE] = '1' are synchronous, imprecise interrupts.

Instruction-side machine check interrupts are errors that occur during an attempt to fetch an instruction from an instruction cache (if present) or from external memory. If any fetched instruction is associated with an exception, an interrupt occurs upon attempted execution of the instruction, not when the instruction is brought into the instruction cache (if any). Although such an interrupt is asynchronous to the erroneous memory access, it is handled synchronously with respect to the attempted execution from the erroneous address.

Data-side machine checks for stores may be either precise (MSR[IPE] = '0') or imprecise (MSR[IPE] = '1'). Because the PPE 42 core pipeline is hazard-free, all data-side machine checks for loads are precise.

4.1.3 Interrupt Address Reporting

Synchronous, precise interrupts always report the address of the instruction causing the exception in SRR0. Synchronous, imprecise interrupts typically report the address of an instruction subsequent to the instruction causing the exception. In either case, all instructions preceding the instruction at the reported address have been executed, and the instruction at the reported address has not been executed.

4.2 Interrupt Vector Offsets

The Interrupt Vector Prefix Register (IVPR) identifies a 512-byte aligned memory address where the initial code sequences of interrupt handlers are stored. The IVPR is read-only to the PPE 42 core, however an instance of the PPE 42 core may provide a means to modify the IVPR as a memory-mapped register. The PPE 42 IVPR and interrupt vector offsets are defined slightly differently than the Power ISA Book III-E architecture.

Each interrupt vector is 32 bytes (8 instructions). Interrupt vector offsets are 9 bits. When an interrupt is taken, the first instruction address executed by the interrupt handler is

$(IVPR)_{0:22}$ || <Interrupt Vector Offset>

The interrupt vector offsets are defined in the table below. The table also contains a reference to a detailed description of the interrupt.

Table 1.9: PPE 42 Interrupt Vector Offsets

Offset	Interrupt	Page
x'000'	PPE42: Machine check (all) PPE42X: *Reserved*	44
x'020'	PPE42: *Reserved* PPE42X: Machine check (all)	
x'040'	System Reset	49
x'060'	Data Storage	50
x'080'	Instruction Storage	51
x'0A0'	External	52
x'0C0'	Alignment	53
x'0E0'	Program	54
x'100'	Decrementer	55
x'120'	Fixed Interval Timer	56
x'140'	Watchdog Timer	57
X'160' - x'1E0'	*Reserved*	

4.3 Interrupt Handling

The PPE 42 core handles only one interrupt at a time. Assuming the interrupt types are enabled, multiple simultaneous interrupts are handled in the priority order shown in *Table 1.10: PPE 42 Interrupt and Exception Priorities and Conditions.*

Multiple exceptions can exist simultaneously, each of which requires the generation of an interrupt. The PPE 42 architecture does not provide for simultaneously reporting more than one interrupt of the same class, except that multiple data machine checks of different types may be reported concurrently as a single machine check interrupt. Therefore, interrupts are ordered with respect to each other.

4.3.1 Interrupt Masking

A masking mechanism is available for certain persistent interrupt types. When an interrupt type is masked, and an event causes an exception which would normally generate an interrupt of that type, the exception *persists* as a *status* bit in a register. However, no interrupt is generated. Later, if the interrupt type is enabled (unmasked), and the exception status has not been cleared by software, the interrupt that results from the original exception event is finally generated.

The PPE 42 core provides an external interrupt input signal. The external interrupt is maskable by way of MSR[EE], however masking the external interrupt also masks the internal timer interrupts.

The machine check, alignment, instruction storage, data storage and program interrupts can not be masked. If these interrupts are the highest priority pending interrupt, then these interrupts must either be taken immediately as a branch to an interrupt handler, or the processor halts. Machine check interrupts are taken when MSR[ME] = '1', and the other unmaskable interrupts are taken when MSR[UIE] = '1'. If an interrupt controlled by MSR[UIE] would have otherwise been taken, but MSR[UIE] = '0' and MSR[ME] = '1', the interrupt is promoted to a machine check interrupt and taken as a machine check interrupt. Interrupt promotion is discussed below in section 4.3.5, *Unmaskable Interrupt Promotion.*

The system reset action is also unmaskable, and processed as an interrupt. If a system reset interrupt resets a running processor, then at the entry of the interrupt vector the save/restore registers will contain the contents of the IAR and the MSR current at the time of the system reset event.

4.3.2 Interrupt Priority

PPE 42 interrupt priorities do not strictly follow the Power ISA Book III-E specification. In particular, all asynchronous interrupts are prioritized below all synchronous interrupts. Since PPE 42 maintains a minimum of state, and only implements the single SRR0/SRR1 register pair, it is necessary to order interrupts and define MSR updates by interrupts in a way that maintains a reasonable priority but precludes back-to-back interrupts, which are unrecoverable if SRR0/SRR1 are destroyed by a subsequent interrupt before a first interrupt handler can save SRR0 and SRR1.

Section 4.3.4, *Interrupt Halt Semantics* details conditions under which the interrupt condition will actually halt the processor. Note that processor halts due to debug events and halts due to writing XCR[CMD] are *not* treated as interrupts and halt the processor immediately. These debug halts effectively have the highest priority of any exceptional condition other than system reset, including unmaskable interrupts that may also halt the processor.

Table 1.10: PPE 42 Interrupt and Exception Priorities and Conditions

Priority	Interrupt /Exception Type	Cause and/or Interrupt Conditions	Halt, Reset and Promotion Conditions
1	System reset	External reset signal; Writing DBCR[RST] to '01' or '10'; Writing XCR[RST] to '01' or '10'.	Always associated with reset
2	Machine check – data	Attempted load, store or cache operation which an external memory interface reported as an error.	Halt if MSR[ME] = '0'.
3	Machine check – instruction	Attempted fetch of an instruction which an external memory interface reported as an error	Halt if MSR[ME] = '0'.
4	Instruction storage	An instance-specific error associated with an instruction address, determined before the address is presented on an external memory interface.	Halt if MSR[UIE] = '0' and MSR[ME] = '0'. Promoted to machine check if MSR[UIE] = '0' and MSR[ME] = '1'.
5	Program	Illegal instruction encoding; **trap** instruction when DBCR[TRAP] = '0'	Halt if MSR[UIE] = '0'. Promoted to machine check if MSR[UIE] = '0' and MSR[ME] = '1'. **trap** halts the processor if DBCR[TRAP] = '1'.
6	Data Storage	An instance-specific error associated with a data address, determined before the address is presented on an external memory interface.	Halt if MSR[UIE] = '0'. Promoted to machine check if MSR[UIE] = '0' and MSR[ME] = '1'.
7	Alignment	An instance-specific error associated with the data address of a load, store or cache management instruction, determined before the address is presented on an external memory interface.	Halt if MSR[UIE] = '0'. Promoted to machine check if MSR[UIE] = '0' and MSR[ME] = '1'.
8	Watchdog Timer (WDT)	TCR[WIE] = '1' and TSR[WIS] = '1' for the first timeout.	The following actions are taken when an event that would otherwise set TSR[WIS] occurs, and TSR[WIS] is already '1': • If TCR[WRC] = '11' the processor is force-halted. • If TCR[WRC] = '01' or '10' the watchdog reset action is taken.
9	External interrupt input	Interrupts external to the PPE 42 core	N/A
10	Fixed Interval Timer (FIT)	TCR[FIE] = '1' and TSR[FIS] = '1'	N/A
11	Decrementer (DEC)	TCR[DIE] = '1' and TSR[DIS] = '1'	N/A

4.3.3 Interrupt Processing

The following high-level description of instruction processing with respect to interrupts provides a practical view of interrupt signaling and priorities.

1. Before beginning processing of the next instruction, if an asynchronous exception is pending and not masked the processor will take an interrupt.
2. If the address of the next instruction to be executed causes an instruction address compare debug event, the processor will immediately halt before presenting the instruction address to the storage subsystem.
3. Fetching the instruction may lead to an instruction machine check exception, or an instruction storage exception.
4. Decoding the instruction may cause a program exception, which may cause an interrupt or immediately halt the processor for **trap** debug events.
5. If the instruction is either a load, store or cache management instruction, the load or store address may trigger a debug data address compare exception, which may cause the processor to halt immediately before the address is presented to the memory subsystem.
6. If the instruction is either a load, store or cache management instruction, the load or store address may trigger a data storage or alignment exception which is taken as an interrupt. Note that both errors may present simultaneously, and if so the data storage interrupt takes priority.
7. Executing a load, store or cache management instruction may cause a precise data machine check exception, or may uncover pending imprecise machine check exceptions.
8. Having reached this point without exception, the result of executing the instruction is committed to the architected state of the core, and the sequence repeats with the next instruction.

Whenever an exception is taken as an interrupt (other than the system reset interrupt), the core must synchronize execution before proceeding. This specifically means that the core must wait until all memory operations by preceding instructions have completed to the point that any imprecise data machine checks that *may* be reported by the preceding instructions *have* been reported. This execution synchronization phase is where interrupt priorities are actually resolved. For example:

- Imprecise data machine checks uncovered here may take priority over the interrupt causing execution synchronization.
- An asynchronous interrupt that becomes newly pending here may take priority over the interrupt causing execution synchronization.

The system reset interrupt is a special case with regard to synchronization. Any memory synchronization is performed during the reset operation, not during interrupt handling. For further details see section 3, *Initialization,Reset, and Starting Execution.*

Finally note that if an instruction is interrupted, none of the effects of executing the instruction have been committed to the architected GPR and SPR state of the core, with the exception of MSR and ISR status bit changes, EDR updates for certain interrupts, and the GPRs and SPRs whose values had already been successfully read from memory for the **lcxu** and **lsku** instructions. It is also possible that temporary or permanent error status, or other state changes may be logged in the storage subsystem even if an interrupted instruction is never executed by the core. From the point of view of the PPE 42 GPR programming model, however, the interrupted instruction may always be re-fetched and re-executed.

4.3.4 Interrupt Halt Semantics

If a machine check interrupt occurs with MSR[ME] = '0' the processor will halt. Similarly, if any of the other unmaskable interrupts occurs (program, alignment, instruction storage or data storage interrupts) with MSR[UIE] = '0' and MSR[ME] = '0', the processor will halt. These events are treated as normal interrupt events, except that the processor halts at the interrupt vector address instead of continuing execution at the interrupt vector address. This means:

- The processor synchronizes execution as normal before taking the highest priority interrupt.
- MSR, ISR, EDR, SRR0 and SRR1 are updated as specified for the interrupt.
- In the case of machine check interrupts, all ISR machine check status will be updated to record the cause of the most recent machine check.
- In the case of other unmaskable interrupts promoted to machine check interrupts, ISR status for both the original interrupt and the promoted machine check interrupt will be updated.
- The processor will halt with the interrupt vector address in the IAR.
- XSR[HC] is set to '011' to indicate a halt due to an unmaskable interrupt.

Note that debug events are *not* treated as interrupts and halt the processor immediately. These debug halts effectively have the highest priority of any exceptional condition other than system reset, including unmaskable interrupts that may also halt the processor. Debug halt semantics are discussed in section 7, *Debugging.*

The presence of a halt condition, XSR[HCP] = '1', is treated as the lowest-priority exceptional condition, and causes the processor to halt after completing the current instruction. If the instruction in-flight when XSR[HCP] becomes '1' causes or unmasks an asynchronous interrupt, the interrupt is not taken but remains pending. If the instruction in-flight when XSR[HCP] becomes '1' causes a synchronous interrupt, the interrupt is taken and the processor halts with the IAR addressing the interrupt vector.

4.3.5 Unmaskable Interrupt Promotion

The PPE 42 architecture allows unmaskable interrupts (other than the machine check interrupt) to be promoted to machine check interrupts if they occur when MSR[UIE] = '0' and MSR[ME] = '1'. This feature supports the ability for software to attempt to recover or fail gracefully if unmaskable exceptions occur in contexts where MSR[UIE] = '0', for example, in the interrupt handlers of unmaskable exceptions. When the machine check interrupt handler is entered, both MSR[UIE] and MSR[ME] will be '0', and any subsequent unmaskable interrupt will immediately halt the processor.

During interrupt prioritization, the highest priority pending interrupt is selected for execution. If this is either a program, instruction storage, alignment or data storage interrupt, and MSR[UIE]= '0' and MSR[ME] = '1' the following actions occur:

- ISR[PTR] is set or cleared (for program interrupts)
- ISR[ST] is set or cleared (for data storage and alignment interrupts)
- The EDR is updated (for data storage and alignment interrupts)
- ISR[MFE] is cleared
- ISR[MCS] is set to indicate that an unmaskable interrupt has been promoted to a machine check
- SRR0/SRR1 are updated as for a machine check interrupt
- Execution continues at the machine check interrupt vector

The machine check interrupt handler can use ISR[MCS] to determine the cause of the machine check and

take appropriate action.

4.4 General Interrupt Handling Registers

The general interrupt handling registers are the MSR, SRR0, SRR1, IVPR, ISR and EDR. Full register field descriptions of these registers appear in section 8, *Register Summary.*

4.4.1 Machine State Register – MSR

The MSR is a 32-bit register that holds the current context of the PPE 42 core. When an interrupt is taken, the MSR contents are first synchronized. In particular this means that the contents of MSR[SIBRC] and MSR[SIBRCA] are both up-to-date with respect to all previous SIB accesses, including a SIB accesses causing an interrupt, before the MSR is written into SRR1.

Once synchronized, the MSR is written to SRR1, and the MSR is modified in an interrupt-specific way. When an **rfi** instruction is executed, the contents of the MSR are loaded from SRR1.

The MSR contents can be read into a General Purpose Register (GPR) using an **mfmsr** instruction. The contents of a GPR can be written to the MSR using an **mtmsr** instruction. The MSR[EE] bit can be set or cleared atomically using the write to external enable (**wrtee**) or the write to external enable immediate (**wrteei**) instructions.

4.4.2 Save/Restore Registers 0 and 1 – SRR0/1

SRR0 and SRR1 are 32-bit registers that hold the interrupted machine context when an interrupt is processed. On interrupt, SRR0 is set to the current or next instruction address, depending on the interrupt type, and the contents of the MSR are written to SRR1. When an **rfi** instruction is executed at the end of the interrupt handler, the program counter is restored from SRR0. Likewise, the contents of the MSR are restored from SRR1.

The contents of SRR0 and SRR1 can be read from and written to GPRs with the **mfspr** and **mtspr** instructions respectively.

Unlike the Power ISA Book III-E architecture, PPE 42 only implements a single pair of save/restore registers, SRR0 and SRR1, that are used for all interrupts. This restriction means that if an interrupt handler is interrupted before saving SRR0 and SRR1, the original interrupted context is unrecoverable. The PPE 42 architecture generally considers this as an unrecoverable error event.

Every taken interrupt clears MSR[EE] and MSR[UIE], and machine check interrupts also clear MSR[ME] to protect against such back-to-back interrupts. External interrupts can not be taken in these modes, and any unmaskable interrupts either halt the processor or are promoted to machine check interrupts rather than continuing with a silently corrupted state. If an application enables both MSR[UIE] = '1' and MSR[ME] = '1', then the machine check interrupt handler may need to analyze ISR[MCS] and the MSR saved in SRR1 to determine if an unmaskable interrupt handler has been corrupted by taking the machine check interrupt.

Several strategies for managing the single SRR0/SRR1 pair are possible:

- Applications running with MSR[EE, ME, UIE] all set to '0' do not need to define interrupt vectors at all (other than for system reset) and may simply poll for timer and external exceptions. In this mode all unmaskable interrupts will halt the processor.
- Applications can chose to halt on all unmaskable interrupts by leaving MSR[ME, UIE] = '0' throughout, while handling maskable interrupts.
- Applications can leave MSR[EE, UIE] = '0' (and MSR[ME] = '0') in all interrupt handlers, but with other settings in normal application code, especially if unmaskable exceptions should never occur in interrupt handlers.
- Interrupt handlers can save SRR0/SRR1 to memory and then re-enable MSR[EE] = '1' and/or MSR[UIE] = '1' and/or MSR[ME] = '1' as appropriate.

4.4.3 Interrupt Vector Prefix Register – IVPR

The Interrupt Vector Prefix Register (IVPR) identifies a 512-byte aligned memory address where the initial code sequences of interrupt handlers are stored. The IVPR is read-only to the PPE 42 core using the **mfspr** instruction. Instances of the PPE 42 core may provide a means to modify the IVPR as a memory-mapped register.

When an interrupt is taken, the first instruction address executed by the interrupt handler is

$(IVPR)_{0:22}$ || <Interrupt Vector Offset>

The interrupt vector offsets are defined above in *Table 1.9: PPE 42 Interrupt Vector Offsets*.

4.4.4 Interrupt Status Register – ISR

The ISR contains several status bits that record information associated with unmaskable interrupts. The ISR can be read and written using **mfspr** and **mtspr** respectively. Note that ISR status bit are guaranteed to be up-to-date and unambiguous at the entry points of interrupt handlers. Therefore in general there is no need for interrupt handlers to clear the ISR, although doing so may help avoid confusion when debugging.

4.4.5 Error Data Register – EDR

The EDR is updated when program, data storage, alignment and data machine check exceptions are taken as interrupts, as noted in the detailed description of the register and the detailed descriptions of the interrupts. The EDR can be read and written using **mfspr** and **mtspr** respectively. The EDR is also read-only as an XIR, to support external error diagnosis.

4.5 Detailed Interrupt Descriptions

The following sections describe the PPE 42 interrupts in the order of their interrupt vector offsets.

4.5.1 Machine Check Interrupt – PPE 42 Vector x'000'; PPE 42X Vector x'020'

The Power ISA Book III-E architecture leaves the details of the causes of, and actions taken in response to machine check interrupts as implementation-dependent. This section describes the PPE 42 implementation of machine checks.

PPE 42 recognizes instruction and data machine check exceptions. Both types are reported by the machine check interrupt. A true machine check normally indicates an unrecoverable error in the memory subsystem, such as an illegal address or uncorrectable memory error. Machine checks may be precise or imprecise. The PPE 42 also provides for other unmaskable interrupts to be promoted to machine check interrupts when MSR[UIE] = '0'.

The machine check interrupt is an unmaskable interrupt. If a machine check interrupt is taken with MSR[ME] = '0', the machine check interrupt makes all of the register updates specified for the machine check type and then halts the processor with the IAR addressing the first instruction of the machine check interrupt vector.

Status bits in the ISR are set based on the type of the most recent machine check(s). The type may be plural because multiple imprecise machine checks may be reported simultaneously.

To aid in program debug, PPE 42X relocates the Machine Check Interrupt Vector from offset x'000' to previously reserved vector offset x'020', such that instruction fetches to an address offset of zero are distinguishable from a Machine Check. Programs intended to run on both PPE 42 and 42X should place a "b 0x20" instruction at the address given by $(IVPR)_{0:22}$ || x'000' with the first instruction of the Machine Check interrupt handler at $(IVPR)_{0:22}$ || x'020'.

4.5.1.1 Service Interface Bus (SIB) Error Reporting and Handling

In the PPE 42 architecture, non-0 Service Interface Bus (SIB) return codes for data accesses will cause machine check exceptions unless masked by MSR[SEM]. Non-zero SIB return codes may or may not be recoverable, depending on their instance-specific interpretation. Both masked and unmasked SIB return codes for SIB data accesses are reported in MSR[SIBRC] and accumulated in MSR[SIBRCA].

Instruction machine checks for instruction fetches from SIB memory areas are not maskable by MSR[SEM]. Any SIB instruction fetch returning a non-0 return code causes an instruction machine check. SIB return codes for SIB instruction fetches are only reported in MSR[SIBRC] when they cause an instruction machine check. SIB return codes for SIB instruction accesses are never accumulated in MSR[SIBRCA].

Whenever any interrupt is taken, the SIB return code and SIB return code accumulation in the MSR are considered part of the interrupted context, not the context of the interrupt handler. The MSR is updated with SIB return code status before being saved in SRR1, and MSR[SIBRCA] is always set to 0 when an interrupt is taken.

To support external debugging, MSR[SIBRC] is mirrored in the XSR, and is not cleared by any interrupt. In general, programmers can disambiguate whether MSR[SIBRC] is from an access in the current context or the previous context by nothing whether MSR[SIBRCA] contains any set bits. The exception to this rule is at entry to the machine check interrupt handler when ISR[MCS] = '000' (instruction machine check). In this case, if the instruction address held in SRR0 is a SIB address, then MSR[SIBRC] is the SIB return code of the failed SIB instruction fetch.

4.5.1.2 Instruction Machine Check Handling

Table 1.11: Register Settings During Machine Check-Instruction Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, ME, IS2, IS3, IPE, SIBRCA ← 0 SIBRC unchanged
SRR0	Written with the address that caused the machine check – instruction interrupt
SRR1	Written with the contents of the MSR, noting that SIBRC and SIBRCA are unchanged for non-SIB instruction machine checks SIBRC ← SIB return code, and SIBRCA accumulates status for SIB instruction machine checks
ISR	MFE ← '0' MCS ← '000' Other fields unchanged
EDR	Unchanged
PC	PPE 42X: $(IVPR)_{0:22}$ || x'020' PPE 42: $(IVPR)_{0:22}$ || x'000'

Instruction machine checks occur when an error is associated with an instruction fetch, and are always synchronous and precise. This exception is only signaled when the erroneous instruction is fetched by the PPE 42 core. This exception is *not* signaled when an error occurs on an instruction cache line fill. Any error status on a cache line fill must be held by the instruction cache and only reported to the core on an attempted fetch from that line. An instruction cache is required to invalidate any cache line after returning an error for a fetch from that line.

4.5.1.3 Data Machine Check Handling for Load-Type Operations

Table 1.12: Register Settings During Machine Check-Data, Load-Type Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, ME, IS2, IS3, IPE, SIBRCA ← 0 SIBRC unchanged
SRR0	Written with the address that caused the machine check – data interrupt
SRR1	Written with the contents of the MSR
ISR	MFE ← '0', MCS ← '001' Other fields unchanged
EDR	Written with the (a) data address causing the error.
PC	PPE 42X: $(IVPR)_{0:22}$ || x'020' PPE 42: $(IVPR)_{0:22}$ || x'000'

Data machine checks for loads are always synchronous and precise. The PPE 42 core always waits for loads to compete before continuing with the next instruction. A data machine check exception for loads is only signaled when the erroneous data is loaded by the PPE 42 core. This exception is *not* signaled when an error occurs on a data cache line fill. Any error status on a cache line fill must be held by the data cache and only reported to the core on an attempted load from that line. The data cache is required to invalidate any cache line that returns an error for a load from that line.

Note that the **dcbt** instruction explicitly hints at a data cache line fill. As per the preceding paragraph, any errors associated with the data cache line fill are not reported until the core attempts to load from the line. PPC 42 storage subsystems should not require the core to wait for **dcbt** to complete in the data cache before continuing with the next instruction, however if MSR[IPE] = '0', the storage subsystem must report any errors from a cache line flush caused by **dcbt** before allowing the core to proceed.

4.5.1.4 Data Machine Check Handling for Store-type Operations

Table 1.13: Register Settings During Machine Check-Data, Store-Type Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, ME, IS2, IS3, IPE, SIBRCA ← 0 SIBRC unchanged
SRR0	Written with the address of the instruction causing (precise) or associated with (imprecise) the machine check interrupt
SRR1	Written with the contents of the MSR
ISR	MFE ← '0', MCS ← '010' if a single precise machine check is reported MFE ← '0', MCS ← '011' if a single imprecise machine check is reported MFE ← '1', MCS ← '011' if multiple imprecise machine checks are reported Other fields unchanged
EDR	Written with the (a) data address causing the error
PC	PPE 42X: $(IVPR)_{0:22}$ || x'020' PPE 42: $(IVPR)_{0:22}$ || x'000'

Characteristics of data machine checks for stores are controlled by MSR[IPE]. When MSR[IPE] = '0' all data machine checks for stores are synchronous and precise. In this mode the PPE 42 core waits for stores to complete in the memory subsystem, at least up to the point at which any possible errors associated with the store have been or would have been reported.

If MSR[IPE] = '1' the storage subsystem is allowed to process stores asynchronously with respect to program execution. In this mode data machine checks for stores may be imprecise, and machine checks for multiple stores may be reported concurrently. Imprecise store handling is not a requirement. An instance of PPE 42 may elect to handle all stores precisely, as documented with each instance of PPE 42.

If a machine check is imprecise, the instruction whose address is written to SRR0 is not necessarily the instruction causing the machine check. Although architecturally this instruction has not been executed and can be re-executed against the current state, it may be difficult or impossible for a machine check handler to determine whether the current instruction, or which previous instruction or instructions failed in order to correctly recover the application.

Note that loads from cacheable addresses, as well as the **dcbt** and **dcbf** instructions may indirectly cause the store of a modified cache line from the data cache. If MSR[IPE] = '0', execution of loads from cacheable addresses, **dcbt** and **dcbf** stall execution until any errors have been or would have been reported from any associated data cache writeback. Thus it is possible that the error address loaded into the EDR in the event of a data machine check for a load may be the address of a flushed cache line, not the load, **dcbt** or **dcbf** address. This type of machine check will be reported as a store error.

The multiple-fault error (ISR[MFE] = '1') can only be reported for imprecise store operations. This error indicates that error recovery information may have been irrevocably lost from the hardware state. In particular, the EDR will only report one of the data addresses responsible for the multiple machine checks.

4.5.1.5 Machine Checks Promoted from Other Unmaskable Interrupts

Table 1.14: Register Settings During Promoted Machine Check Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, ME, IS2, IS3, IPE, SIBRCA ← 0 SIBRC unchanged
SRR0	Written with the address of the instruction causing the promoted machine check interrupt
SRR1	Written with the contents of the MSR
ISR	The contents or ISR[PTR, ST] are updated (or not) precisely as if the underlying unmaskable interrupt had been taken ISR[MFE] ← '0' ISR[MCS] ← '100' for program interrupts ISR[MCS] ← '101' for instruction storage interrupts ISR[MCS] ← '110' for alignment interrupts ISR[MCS] ← '111' for data storage interrupts Other fields unchanged
EDR	Updated precisely as if the underlying unmaskable interrupt had been taken
PC	PPE 42X: $(IVPR)_{0:22}$ || x'020' PPE 42: $(IVPR)_{0:22}$ || x'000'

Program, instruction storage, alignment and data storage interrupts are taken as machine check interrupts if the interrupt would have otherwise been taken as the highest priority interrupt, but MSR[UIE] = '0' and MSR[ME] = '1'. All of the state changes take place as for the original interrupt, and are then extended by the changes specified for machine check interrupts. Execution continues at the machine check interrupt vector.

4.5.2 System Reset Interrupt – Vector x'040'

Table 1.15: Register Settings During System Reset Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, ME, IS2, IS3, IPE, SIBRCA ← 0 SIBRC unchanged
SRR0	Written with the instruction address interrupted by the system reset interrupt
SRR1	Written with the contents of the MSR at the time of the system reset interrupt
ISR	SRSMS ← The PPE 42 core state machine state at the time of the system reset Other fields unchanged
EDR	Unchanged
PC	$(IVPR)_{0:22}$ || x'040'

All reset actions are treated as the highest priority, unmaskable event that is immediately taken as an interrupt.

Treating system reset as an interrupt may aid debugging. For example, if a watchdog reset action resets the processor, the contents of SRR0 and SRR1 after the reset may be helpful in determining why the application missed the watchdog deadline. The system reset forces an immediate reset however, and there is no indication as to whether the instruction whose address is written to SRR0 has executed or partially executed, and SRR1 is not necessarily up to date with respect to previous instructions that may have caused exceptions. Programmers with access to PPE 42 core hardware design information may be able to glean more information from the contents of ISR[SRSMS], which is updated with the PPE 42 core state machine state at the time of the system reset.

4.5.3 Data Storage Interrupt – Vector x'060'

Table 1.16: Register Settings During Data Storage Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the instruction responsible for the data storage interrupt
SRR1	Written with the contents of the MSR
ISR	ST ← 1 for store-type operations ST ← 0 otherwise Other fields unchanged
EDR	Written with the data address responsible for the data storage interrupt
PC	$(IVPR)_{0:22}$ || x'060'

The data storage interrupt is an unmaskable interrupt. If a data storage interrupt is taken with MSR[UIE] = '1', all of the actions in the above table occur.

If a data storage interrupt is taken with MSR[UIE] = '0', then the processor first makes the specified ISR updates. If MSR[ME] is also '0', the processor then makes the remainder of the register updates and halts with the IAR addressing the first instruction of the data storage interrupt vector. If MSR[ME] = '1' however, the interrupt is taken as a machine check interrupt as specified in section 4.3.5, *Unmaskable Interrupt Promotion*.

The conditions causing the data storage interrupt are instance-specific, and will be documented with each instance of the PPE 42 core. The intention is that a data storage interrupt indicates an attempted data memory access that has been determined to be invalid or illegal before the (translated) address is presented on an external memory bus. For example:

- Attempt to access an unimplemented memory space
- Attempt to access an address without a valid memory translation
- Access mode violations, such as writing to read-only memory spaces

The memory subsystem must signal data storage exceptions with higher priority than alignment exceptions.

ISR[ST] is set to '1' if a data storage interrupt is caused by a store, or a store-type data cache operation. The following table details how instructions are classified for the purposes of the data storage interrupt.

Table 1.17: Instruction Treatment by Data Storage Interrupts

Instruction Type	Treated As
All loads	Load
All Stores	Store
dcbf	Load
dcbi	Store
dcbq	Load
dcbt	Load
dcbz	Store

4.5.4 Instruction Storage Interrupt – Vector x'080'

Table 1.18: Register Settings During Instruction Storage Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the instruction responsible for the instruction storage interrupt
SRR1	Written with the contents of the MSR
ISR	Unchanged
EDR	Unchanged
PC	$(IVPR)_{0:22}$ || x'080'

The instruction storage interrupt is an unmaskable interrupt. If an instruction storage interrupt is taken with MSR[UIE] = '1', all of the actions in the above table occur.

If a data storage interrupt is taken with MSR[UIE] = '0', then the processor first makes the specified ISR updates. If MSR[ME] is also '0', the processor then makes the remainder of the register updates and halts with the IAR addressing the first instruction of the instruction storage interrupt vector. If MSR[ME] = '1' however, the interrupt is taken as a machine check interrupt as specified in section 4.3.5, *Unmaskable Interrupt Promotion.*

The conditions causing the instruction storage interrupt are instance-specific, and will be documented with each instance of the PPE 42 core. The intention is that an instruction storage interrupt indicates an attempted instruction fetch that has been determined to be invalid or illegal before the (translated) address is presented on an external memory bus. For example:

- Attempt to fetch from an unimplemented memory space
- Attempt to fetch from an address without a valid memory translation
- Access mode violations, such as an attempted fetch from a non-executable memory space

4.5.5 External Interrupt – Vector x'0A0'

Table 1.19: Register Settings During External Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
ISR	Unchanged
EDR	Unchanged
PC	$(IVPR)_{0:22}$ || x'0A0'

External interrupts are triggered by active levels on the external interrupt input. All external interrupting events are presented to the processor as a single external interrupt.

External interrupts are enabled or disabled by MSR[EE]. MSR[EE] also enables Watchdog Timer (WDT), Fixed Interval Timer (FIT) and Decrementer (DEC) interrupts. However for timer interrupts, control passes to different interrupt vectors than for the interrupts discussed in the preceding paragraph. Timer interrupt handling is described further below for each particular timer interrupt.

4.5.5.1 External Interrupt Recognition; Phantom Interrupt Avoidance

Conceptually, whenever an instruction is about to be fetched and executed, or whenever an instruction causes an exception, or whenever a timer exception is outstanding, the PPE core observes the value of the external interrupt signal, and if MSR[EE] = '1', includes the external interrupt in the set of interrupts to prioritize after execution synchronization. If the external interrupt is the highest priority interrupt outstanding then the external interrupt is taken. If the external interrupt signal is not active or MSR[EE] = '0' in the cases described above, the external interrupt is not considered for prioritization.

The above description of external interrupt processing means that it is acceptable for the external interrupt signal to the PPE 42 core to arbitrarily change value while MSR[EE] = '0'. It is typically not acceptable for the external interrupt signal to change from an active value to an inactive value when MSR[EE] = '1', as the processor may erroneously begin processing an external interrupt that has subsequently been masked or is no longer active.

Software must take care to avoid this type of “phantom” interrupt, for example by only manipulating hardware related to external interrupt presentation while MSR[EE] = '0', and then issuing a **sync** instruction to insure that all previous operations are complete, prior to setting MSR[EE] = '1'. Depending on the hardware design it may also be necessary to further delay setting MSR[EE] = '1' to allow time for the external interrupt input to settle.

4.5.6 Alignment Interrupt – Vector x'0C0'

Table 1.20: Register Settings During Alignment Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the instruction causing the alignment interrupt
SRR1	Written with the contents of the MSR
ISR	ST ← 1 for stores and **dcbz** ST ← 0 otherwise Other fields unchanged
EDR	Written with the data address responsible for the alignment interrupt
PC	$(IVPR)_{0:22}$ || x'0C0'

The alignment interrupt is an unmaskable interrupt. If an alignment interrupt is taken with MSR[UIE] = '1', all of the actions in the above table occur.

If an alignment interrupt is taken with MSR[UIE] = '0', then the processor first makes the specified ISR updates. If MSR[ME] is also '0', the processor then makes the remainder of the register updates and halts with the IAR addressing the first instruction of the alignment interrupt vector. If MSR[ME] = '1' however, the interrupt is taken as a machine check interrupt as specified in section 4.3.5, *Unmaskable Interrupt Promotion.*

The conditions causing the alignment interrupt are instance-specific, and will be documented with each instance of the PPE 42 core. The intention is that an alignment interrupt indicates an attempted data access that the memory subsystem can not perform as specified, but may be emulatable by the alignment interrupt handler. For example:

- An attempt to load or store an unaligned address might be emulated with multiple loads and/or stores.
- A **dcbz** instruction targeting a noncaheable memory area might be emulated by storing 0 bytes to that location.

ISR[ST] is set to '1' if an alignment interrupt is caused by a store or a **dcbz** instruction.

4.5.7 Program Interrupt – Vector x'0E0'

Table 1.21: Register Settings During Program Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the instruction responsible for the program interrupt
SRR1	Written with the contents of the MSR
ISR	PTR ← 0 if the program interrupt is caused by an illegal instruction encoding PTR ← 1 if the program interrupt is caused by a **trap** instruction Other fields unchanged
EDR	Written with the putative instruction responsible for the program interrupt
PC	$(IVPR)_{0:22}$ || x'0E0'

The program interrupt is an unmaskable interrupt. If a program interrupt is taken with MSR[UIE] = '1', all of the actions in the above table occur.

If a program interrupt is taken with MSR[UIE] = '0', then the processor first makes the specified ISR updates. If MSR[ME] is also '0', the processor then makes the remainder of the register updates and halts with the IAR addressing the first instruction of the program interrupt vector. If MSR[ME] = '1' however, the interrupt is taken as a machine check interrupt as specified in section 4.3.5, *Unmaskable Interrupt Promotion.*

Program interrupts are caused by attempting to execute any of the following instructions:

- An illegal instruction encoding for the PPE 42 core
- A **trap** instruction when DBCR[TRAP] = '0'

When the program interrupt is taken, the putative instruction causing the program interrupt is written to the EDR. If the program interrupt is caused by a **trap,** this will be a **trap** instruction (special form of **tw**). The EDR value can be used to to aid emulation of unimplemented instruction encodings, or as a debugging aid to validate that the putative instruction in the EDR is the actual data held at the address written to SRR0.

4.5.8 Decrementer (DEC) Interrupt – Vector x'100'

Table 1.22: Register Settings During Decrementer Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
ISR	Unchanged
EDR	Unchanged
PC	$(IVPR)_{0:22}$ || x'100'

The decrementer (DEC) interrupt occurs when TSR[DIS] = '1', TCR[DIE] = '1', MSR[EE] = '1' and the decrementer interrupt is the highest priority pending interrupt. For a description of PPE 42 timers, including interrupt causing conditions and interrupt handling procedures, see section 5, *Timer Facilities.*

4.5.9 Fixed Interval Timer (FIT) Interrupt – Vector x'120'

Table 1.23: Register Settings During Fixed Interval Timer (FIT) Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
ISR	Unchanged
EDR	Unchanged
PC	$(IVPR)_{0:22}$ || x'120'

The Fixed Interval Timer (FIT) interrupt occurs when TSR[FIS] = '1', TCR[FIE] = '1', MSR[EE] = '1' and the FIT interrupt is the highest priority pending interrupt. For a description of PPE 42 timers, including interrupt causing conditions and interrupt handling procedures, see section 5, *Timer Facilities.*`

4.5.10 Watchdog Timer (WDT) Interrupt – Vector x'140'

Table 1.24: Register Settings During Watchdog Timer (WDT) Interrupts

Register	Setting
MSR	SEM, IS0, LP, WE, IS1, UIE, EE, IS2, IS3, IPE, SIBRCA ← 0 SIBRC, ME unchanged
SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
ISR	Unchanged
EDR	Unchanged
PC	$(IVPR)_{0:22}$ || x'140'

The watchdog timer (WDT) exception occurs when TSR[WIS] = '1' and TCR[WIE] = '1'. If TCR[WRC] = '00' and MSR[EE] = '1', this exception causes a watchdog interrupt if the watchdog interrupt is the highest priority pending interrupt.

If TCR[WRC] is not '00', then a watchdog timer timeout when TSR[WIS] = '1' causes a watchdog reset or halt action, and pulses the Watchdog Timeout output. For a description of PPE 42 timers, including interrupt causing conditions, interrupt handling procedures and watchdog timer protocols and actions, see section 5, *Timer Facilities.*

5 Timer Facilities

The PPE 42 core provides 3 timer facilities: A 32-bit decrementer (DEC), a fixed interval timer (FIT) and a watchdog timer (WDT). These facilities are modeled after the Power ISA Book III-E timer specification, with several modifications specific to the PPE 42 architecture.

PPE 42 core timer facilities include the following registers:

- The decrementer (DEC)
- The Timer Control Register (TCR)
- The Timer Status Register (TSR)

The DEC is implemented as a special-purpose register (SPR). The FIT and WDT are not actually timers in the PPE 42 core, but instead simply record the occurrence of events that occur outside of the core. The FIT and WDT events are represented by the TSR[FIS] and TSR[WIS] fields respectively. The sources of the DEC, FIT and WDT are selected by TCR[DS], TCR[FP] and TCR[WP] fields respectively.

The diagram below shows the operation of PPE 42 timer facilities schematically. The time sources `dec_timer` and `timer[0:3]` originate outside of the PPE 42 core.

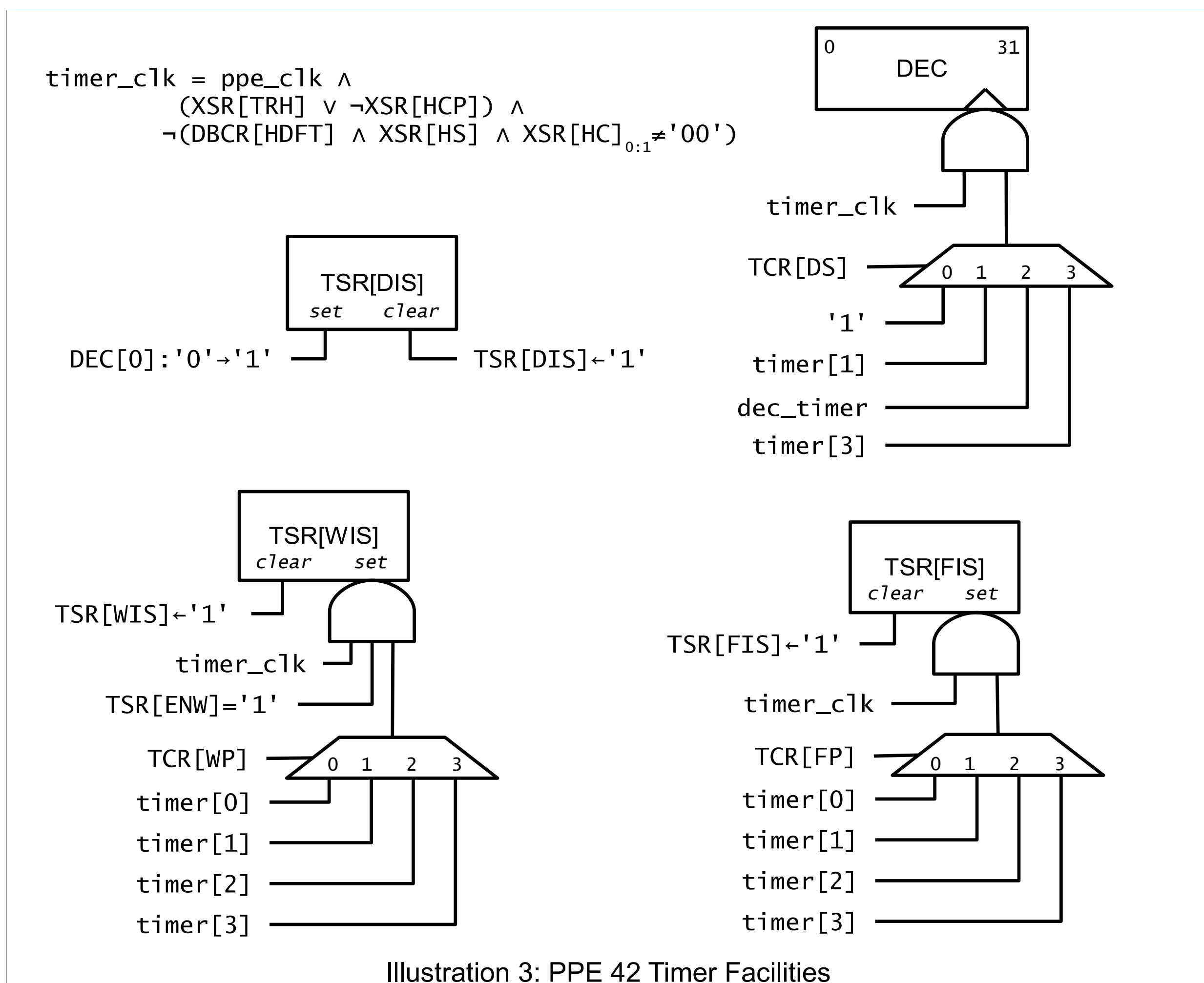


Illustration 3: PPE 42 Timer Facilities

The availability, frequency and frequency stability of any external time source is instance specific, and will be specified with each instance of PPE 42. An instantiation of the PPE 42 core may also define the DEC, FIT and/or WDT events as general-purpose events rather than timed events. There is no requirement that any of the time sources be provided in order for the PPE 42 core to correctly execute instructions, or to correctly perform any other function not specifically related to the timer facilities.

Timer interrupts are individually enabled by TCR[DIE, FIE, WIE] for the DEC, FIT and WDT interrupts respectively. Timer interrupts are masked by MSR[EE]. If MSR[EE] = '1' and any timer interrupt or external interrupt is pending, these events will be handled in the following priority order:

- WDT interrupts
- External interrupts
- FIT interrupts
- DEC interrupts

The interrupt handler for timer interrupts must either clear the timer interrupt status by writing the appropriate TSR timer interrupt status bit with '1', or clearing the appropriate TCR interrupt enable bit before re-enabling external interrupts.

5.1 The Decrementer (DEC)

The DEC register is a 32-bit SPR that decrements on any cycle that the decrement condition is true. PPE 42X increases the TCR[DS] field to two bits, where DS[1] was previously reserved on PPE 42. If TCR[DS] = '00', then the DEC decrements every cycle. If TCR[DS] = '10', then the DEC decrements on any cycle that the external `dec_timer` input is active. Otherwise, the DEC decrements on any cycle the `timer[1]` or `timer[3]` input is active, as selected by TCR[DS] = '01' or '11' respectively.

If the DEC contains 0 when the decrementer event occurs the DEC underflows to x'FFFF FFFF'. The DEC is read using the **mfspr** (**mfdec**) instruction, and written using the **mtspr** (**mtdec**) instruction.

Decrementer interrupt status, TSR[DIS], is set on any cycle that DEC_0 transitions from 0 to 1, including transitions due to **mtspr** instructions that update DEC. If TSR[DIS] = '1', TCR[DIE] = '1' and MSR[EE] = '1', then a decrementer interrupt will be taken if the decrementer interrupt is the highest priority pending interrupt.

Decrementer interrupt status is cleared by writing TSR[DIS] with '1'. Clearing TSR[DIS] this way takes precedence over a simultaneous set of TSR[DIS] from the hardware. Note that the DEC is implemented in lieu of the Programmable Interval Timer (PIT) register that can be found in other embedded Power processors.

5.1.1 Using DEC as a Programmable Interval Timer

The DEC can be used as a programmable interval timer. The following procedure details how to program the DEC for this application. This procedure must be executed with external interrupts disabled (MSR[EE] = '0'), and assumes that any preexisting or soon-to-be-existing decrementer interrupt status may be ignored.

1. Write DEC with x'FFFFFFFF'.
2. Write TSR[DIS] with '1' to clear any pending DEC interrupt status.
3. Write DEC with the desired timeout.
4. Write TCR[DIE] with '1'

This procedure guarantees that the next DEC interrupt will be the result of the programmed timeout.

5.1.2 Using DEC to Emulate a Timebase

PPE 42 does not implement the Power ISA 64-bit *timebase* facility. However the semantics of the PPE 42 DEC allow it to be used with simple software procedures to emulate a timebase. The following procedures detail how to use the DEC for this application. These procedures must be executed with external interrupts disabled (MSR[EE] = '0').

To initialize timebase emulation:

- Initialize a 64-bit global variable to 0 or another appropriate value. This global variable will be referred to as *the timebase*.
- Write the DEC with x'FFFFFFFF', clear TSR[DIS] and set TCR[DIE].

In the event of a DEC interrupt:

- Add x'100000000' to the timebase and clear TSR[DIS].

To compute the current time:

- Return the sum of the timebase plus the difference ((DEC) – x'FFFFFFFF'). If TSR[DIS] = '1', also add x'100000000' to the returned value.

Assuming that all DEC interrupts are processed in a timely manner, the current times computed by the procedure should always reflect an absolute time as measured by the DEC time source, modulo the small amount of time required to perform the calculation.

If the DEC is being used both as a programmable interval timer and for timebase emulation, the above procedures can be generalized as follows:

- Whenever the DEC is written, add one plus the difference of the current contents of DEC minus the last value written to DEC to the timebase.
- Whenever a DEC interrupt occurs, add one plus the last value written to DEC to the timebase.
- To compute the current time, return the sum of the timebase plus the difference of the current contents of DEC minus the last value written to DEC to the timebase, performing the analogous compensation if TSR[DIS] = '1'.

If the DEC is being used as a programmable interval timer, then the emulated timebase will likely drift with respect to an absolute time reference. Part of this drift may be compensated by modifying the timebase updates for the time required to execute the updates, however if high fidelity is required it may be necessary to periodically synchronize the emulated timebase with an external source.

5.2 The Fixed Interval Timer (FIT)

The FIT simply records the occurrence of one of four external events represented by active levels of the `timer[0:3]` inputs of the PPE 42 core. The timer event selected as the FIT is programmed by setting TCR[FP].

FIT interrupt status, TSR[FIS], is set on any cycle that the selected time source is active. If TSR[FIS] = '1', TCR[FIE] = '1' and MSR[EE] = '1', then a FIT interrupt will be taken if the FIT interrupt is the highest priority pending interrupt.

FIT interrupt status is cleared by writing TSR[FIS] with '1'. Clearing TSR[FIS] this way takes precedence over a simultaneous set of TSR[FIS] from the hardware.

5.3 The Watchdog Timer (WDT)

The WDT simply records the occurrence of one of four external events represented by active levels of the `timer[0:3]` inputs of the PPE 42 core. The timer event selected as the WDT is programmed by setting TCR[WP].

If a watchdog event occurs when TSR[ENW] = '0', then the only action is to set TSR[ENW] = '1'.

WDT interrupt status, TSR[WIS], is set when a watchdog event occurs and TSR[ENW] (enable next watchdog) is '1'. If TSR[WIS] = '1', TCR[WIE] = '1' and MSR[EE] = '1', then a WDT interrupt will be taken if the WDT interrupt is the highest priority pending interrupt.

WDT interrupt status is cleared by writing TSR[WIS] with '1'. Clearing TSR[WIS] this way takes precedence over a simultaneous set of TSR[WIS] from the hardware.

If a watchdog event occurs while TSR[WIS] is set to ‘1’ and TSR[ENW] is set to ‘1’, a hardware reset or halt occurs if enabled by a nonzero value of TCR[WRC]. In other words, a reset or halt can occur if a watchdog event occurs while a previous watchdog interrupt is pending. The assumption is that TSR[WIS] has not been cleared because the processor cannot execute the watchdog handler, leaving reset or halt as the only way to restart or debug the system. Whenever a WDT reset or halt occurs, the contents of TCR[WRC] are copied to TSR[WRS], and TCR[WRC] is cleared.

The following table summarizes WDT behavior.

Table 1.25: Action in Response to the WDT Event

Current State		Action in response to WDT event
TSR[ENW]	**TSR[WIS]**	
0	0	TSR[ENW] ← '1'
0	1	TSR[ENW] ← '1'
1	0	TSR[WIS] ← '1', which will cause a WDT interrupt if TCR[WIE] = '1', MSR[EE] = '1' and the WDT interrupt is the highest priority pending interrupt.
1	1	The action specified by TCR[WRC] will occur: 00 No action 01 Soft reset 10 Hard reset 11 Force-halt the core In the event of a reset or halt action, TCR[WRC] is copied to TSR[WRS], and TCR[WRC] is cleared.

The controls described in the above table imply three different modes of operation that a programmer might select for the WDT. Each of these modes assumes that TCR[WRC] has been set to allow reset or halt by the WDT facility:

1. Always take the WDT interrupt when pending, and never attempt to prevent its occurrence. In this mode, the WDT interrupt caused by a first event is used to clear TSR[WIS] so a second event does not cause a reset or halt. TSR[ENW] is not cleared, thereby allowing the next event to cause another interrupt.

2. Always take the WDT interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the FIT interrupt handler) is used to repeatedly clear TSR[ENW] such that a first event exception is avoided, and thus no WDT interrupt occurs. Once TSR[ENW] has been cleared, software has between one and two full WDT periods before a WDT event will be posted in TSR[WIS]. If this occurs before the software is able to clear TSR[ENW] again, a WDT interrupt will occur. In this case, the WDT interrupt handler will then clear both TSR[ENW] and

TSR[WIS], in order to (hopefully) avoid the next WDT interrupt.

3. Never take the WDT interrupt. In this mode, WDT interrupts are disabled (via TCR[WIE] = '0'), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the FIT interrupt handler) to repeatedly clear TSR[WIS] such that a second event is avoided, and thus no reset or halt occurs. TSR[ENW] is not cleared, thereby allowing the next event to set TSR[WIS] again. The recurring code loop must have a period which is less than one WDT period in order to guarantee that a WDT reset or halt will not occur.

5.3.1 Implications of TSR[ENW]

The TSR[ENW] field allows an operating environment to set up the WDT actions in TCR[WRC] very early in the initialization process. If TSR[ENW, WIS] are cleared before setting TCR[WRC], TSR[ENW] gives initialization code between one and two full WDT periods to perform system setup before the first possible WDT interrupt. On the other hand, if the WDT interrupt is also used as a generic timer event, then the first event will be similarly delayed by between one and two full WDT periods.

5.4 Debug Behavior

The XCR[TRH] (timers run while halted) field controls the behavior of timer facilities whenever the PPE 42 core reports a halt condition is present, that is whenever XSR[HCP] = '1'. If XCR[TRH] = '0', then the timer facilities are “frozen” while the halt condition is present. This means that the decrementer does not decrement, no new timer events will be registered in TSR[FIS, WIS, ENW], and no new WDT reset or halt actions will occur. This does not imply that external timer events do not occur, only that if they occur they do not cause any changes to the architected state of the core. This mode of operation is designed for general debugging, with the caveat that discontinuities in the passage of time (as recorded by timer facilities) may lead to unexpected behaviors if the core is restarted after debugging.

If XCR[TRH] = '1', then the decrementer continues to decrement, new timer events are registered in TSR[FIS, WIS, ENW], and new WDT reset or halt actions may occur, even though the core is otherwise halted. This mode of operation gives high-speed external agents the ability to halt, manipulate, and restart the core in a way that minimally perturbs the running application.

All of the fields of the TSR are specified with write-1-to-clear semantics. However, if the processor is halted and an **mttsr Rx** instruction is rammed, then all defined fields of the TSR are updated directly from the contents of the GPR **Rx**. This allows debugging code to directly set timer interrupt status, and/or restore the TSR to an original state after debugging.

5.5 Reset Behavior

PPE 42 core reset behavior is documented in section 3, *Initialization,Reset, and Starting Execution*. To summarize the changes to the timer subsystem, every reset clears TCR[WRC], to insure that unintended WDT events do not occur. If a second watchdog timeout resets or halts the processor, TCR[WRC] is copied to TSR[WRS]. The TCR and TSR are not otherwise modified by any other reset. The DEC is not modified by any reset and continues to decrement, potentially allowing the application to timestamp the occurrence of a WDT reset, and/or maintain a consistent emulated timebase across WDT resets.

6 External Interface Registers

The PPE 42 core defines six External Interface Registers, or XIRs. PPE 42X adds three additional XIRs. Four of the XIRs are outside of the programmer-visible architected state, and are only accessible by system elements external to the core. Two of the XIRs (five on PPE 42X) are also SPRs, and are visible both internally and externally. The table below summarizes the PPE 42 XIRs.

Table 1.26: PPE 42X External Interface Registers

Mnemonic	Register Name	Programmer-Visible? (Access)	XIR Access
EDR	Error Data Register	Yes (Read/Write)	Read-only
IAR	Instruction Address Register	No	Read/Write
IR	Instruction Register	No	Read/Write
SPRG0	SPR General 0	Yes (Read/Write)	Read/Write
XCR	External Control Register	No	Write-only
XSR	External Status Register	No	PPE42: Read-only PPE42X: Read/Write
CTR	Link Register	Yes (Read/Write)	PPE42X: Read-only
SRR0	Link Register	Yes (Read/Write)	PPE42X: Read-only
LR	Link Register	Yes (Read/Write)	PPE42X: Read-only

Although the XIRs are architected as 32-bit registers, the XIR interface implemented by the PPE 42 core is a 64-bit data interface. Therefore, XIRs are always accessed as 64-bit pairs of 32-bit registers. Six XIR pairs are defined as documented in the table on the following page. The XIR pairings are defined in a way that allows accelerated debugging by reducing the number of XIR accesses required by certain debugging procedures.

The XIR pairing scheme also supports 32-bit hardware environments. External agents that can only generate 32-bit accesses can be easily accommodated, as each writable XIR is paired with a read-only XIR, and none of the XIRs have read side-effects. Pairings are supported such that when the hardware converts a 32-bit external access to a 64-bit XIR access, the other 32 bits of write data are irrelevant, and the other 32-bits of read data can be discarded. Pairings suitable for 32-bit access are noted in the following table.

The methods and addressing conventions for accessing XIR pairs is instance specific. The XIR pair numbers in the table are for reference only, and may not correspond to register addresses in any PPE instance.

To guarantee robust and predictable operation, many types of XIR access are considered illegal, and the PPE 42 core discards illegal XIR accesses and returns an Invalid Access error return code on the XIR bus. In general, attempts to write registers, other than certain encodes to XCR, via XIRs are only accepted by the PPE when it is in a Halted State and the XIR is not read-only. Reads to XIRs are accepted irregardless of the Halted State, unless it is write-only XIR.

Access restrictions for XIR accesses are also documented in the following table.

Table 1.27: PPE 42 XIR Pairs and Access Restrictions

XIR Pair		Access	Word 0 (Bits 0:31)	Word 1 (Bits 32:63)	Notes	Usage
0	XCR_CTR	PPE42: Write-only PPE42X: Read/Write	XCR	PPE42: N/A PPE42X: CTR	a, g, h, i, j, f ,n	Write to processor control with no other side effects. On PPE42X, also allows reading of CTR for debug. This register allows writes to the XCR when PPE is not in a Halted State.
1	XCR_SPRG0	Write-only	XCR	SPRG0	b, g, h, i, j, k, l	A single write of XCR_SPRG0 can be used to both restore SPRG0 and continue execution after single-stepping or ramming.
2	IR_SPRG0	Write-only	IR	SPRG0	b, g, k	A single write of IR_SPRG0 can be used to update SPRG0, and also ram a **mfsprg0** instruction to set the value of a GPR to the new value of SPRG0.
3	XSR_SPRG0	Read/Write	XSR	SPRG0	c, f, k, m	After ramming or single stepping, reading XSR_SPRG0 returns both the processor status and the current value of SPRG0, for example after ramming a **mtsprg0** instruction to read the value of a GPR.
4	IR_EDR	Read/Write	IR	EDR	d, f, k	Allows simultaneous reading of IR and EDR for diagnosing error and debug halts.
5	XSR_IAR	Read/Write	XSR	IAR	e, k, m	Allows simultaneous reading of XSR and IAR for diagnosing error and debug halts, as well as for profiling the PPE state during runtime. Allows writes to modify the IAR and on PPE42X to restore the XSR after ramming.
6	SRR0_LR	PPE42X: Read-only	SRR0	LR	g	On PPE42X, allows reading of the most recent procedure call return address from LR and the instruction address at the time of the most recent interrupt from SRR0.

Notes for 32-bit Environments

a. This register provides 32-bit, write-only access to the XCR, and on PPE42X 32-bit read access to the CTR via this register.
b. This register must not be used for 32-bit writes due to the unspecified, simultaneous update of both 32-bit registers. Note that there is no error indication from the PPE core hardware itself if this register is written in a 32-bit environment.
c. This register provides 32-bit read access to the XSR, and 32-bit read/write access to SPRG0.
d. This register provides 32-bit read access to the EDR, and 32-bit read/write access to the IR.
e. This register provides 32-bit read/write access to the IAR, and on PPE42X 32-bit read/write access to the XSR.

Notes for all Environments

f. Any data written to the read-only CTR, LR, SRR0, and EDR is discarded. This includes the read-only XSR on PPE42.

Access Restrictions

g. Attempted reads of write-only XIR pairs, or writes of read-only XIR pairs, are not accepted by the PPE core. On PPE42, writes to SRR0_LR and reads to XCR_CTR are not accepted.

h. Any write of XCR[CMD] = '000' (clear debug status) is only accepted by the PPE core if the processor is in the halted state (XSR[HS] = '1').

i. Any write of XCR[CMD] = '010' (resume from halted state) is only accepted by the PPE core if the processor is either in the halted state (XSR[HS] = '1'), or if no halt condition is present (XSR[HCP] = '0').

j. Any write of XCR[CMD] = '011' (single-step) is only accepted by the PPE core if the processor is in the halted state (XSR[HS] = '1').

k. Writes to the IR, IAR, and SPRG0 are only accepted by the PPE core if the processor is in the halted state (XSR[HS] = '1').

l. Writes to XCR_SPRG0 must satisfy the constraints for both registers, otherwise the write is not accepted and neither register is updated.

m. On PPE42, any data written to the read-only XSR is discarded. On PPE42X, writes will modify XSR bits 1:3, 7:8, and 12:13, but only in the Halted State (XSR[HS] = '1') otherwise thewrite is not accepted.

n. Reads to the XCR return all zeroes.

XIR Pair	Access	Word 0 (Bits 0:31)	Word 1 (Bits 32:63)	Notes	Usage

7 Debugging

The debug facilities of the PPE 42 core include support for debugging during hardware and software development, and debug events that allow developers to control the debug process. Debug registers control debug events and report debug status. The debug registers are accessed through software running on the processor, or through a set of external interface registers (XIRs). Methods to access the XIRs are instance-specific.

PPE 42 debugging facilities are modeled after the capabilities architected in the Power ISA Book III-E specification, but are only compliant with the Power ISA architecture to the extent practical, in keeping with the overall goals of the PPE 42 core.

7.1 External Debug Mode

The PPE 42 core supports only a single debug mode. This mode is similar to the *External Debug Mode* capabilities specified by the Power ISA Book III-E. External debug mode provides access to architected processor resources and supports stopping, starting, and stepping the processor. It also supports setting hardware and software breakpoints and monitoring processor status. In this mode, debug events cause the processor to halt, that is to become architecturally frozen. While the processor is halted, normal instruction execution stops and architected processor resources can be accessed and altered. External bus activity continues in external debug mode.

By using XIRs, external debugging tools and procedures can pass instructions to the processor for execution (colloquially known as *ramming* instructions), allowing a debugger to display and alter processor resources, including memory. The processor can also be single-stepped while halted.

7.2 Processor Control

The following table details the debug functions for processor control supported by the PPE 42 core.

Function	Notes
Single-step	When halted, the processor can be stepped one instruction at a time via commands issued through the XCR.
Ramming	When halted, any instruction can be executed by writing the instruction to the IR.
Halt	The processor can be halted by way of an external signal, by an explicit write of DBCR[RST] to '11' from the program, by the occurrence of a debug event, by a watchdog timeout, by an unmaskable interrupt or by writing XCR[CMD] = '001'.
Reset	The processor can be reset by way of an external signal, by an explicit write of DBCR[RST] to '01' or '10' from the program, or by writing XCR[CMD] to '101' or '110'. Both soft and hard resets are provided.
Debug Events	Debug events are enabled in the DBCR, set debug status bits in the XSR, and halt the processor when they occur.
Freeze Timers	Normally, no new timer events are allowed to occur while the processor is halted. A mode is provided to allow timer events to continue while the processor is halted (XSR[TRH] = '1'). XSR[TRH] is toggled (inverted) by writing '100' to XCR[CMD]. Similarly, DBCR[HDFT] prevents new timer events when the processor is halted by a debug event, allowing visibility to timer values present at the time of the halt condition.
Freeze Asynchronous Interrupts	DBCR[HDFI] prevents change in the external interrupt event state when the processor is halted by a debug event, allowing visibility to asynchronous interrupts present at the time of the halt condition.
trap Instruction	The **trap** instruction can be enabled as a debug event to implement breakpoints.

7.3 Processor Status

The following table details the debug functions for processor status supported by the PPE 42 core.

Function	Notes
Execution Status	All interrupt status fields from the ISR are mirrored externally via the XSR to simplify external debugging and diagnosis. XSR[SMS] indicates the current microarchitectural state of the PPE 42 core, and MSR[LP, EP, WE, SIBRC] are also mirrored in the XSR. The XCR allows external debugging tools to control the run/halt/reset state of the processor.
Exception Status	ISR[EP] (also mirrored as XSR[EP]) indicates the presence of a pending asynchronous interrupt. The EDR is also visible externally as an XIR.

7.3.1 Status outputs

The PPE 42 core provides seven outputs that summarize the state of the PPE and whether error conditions have occurred.

7.3.1.1 Halted indication

The Halted output follows the value of XSR[HS]. This indication may be used as a debug aid, where the user may then check XSR[HC] to determine the reason.

7.3.1.2 Watchdog Timeout indication

The Watchdog Timeout output pulses if the watchdog timer is enabled and a watchdog timeout occurs that causes the processor to halt or reset (meaning that the WCR != “00”). This indication may be used as a debug aid.

7.3.1.3 Error indications

Unanticipated events that occur during runtime are considered errors and must be serviced externally to the PPE. These events cause the PPE to immediately enter the Halted state and will cause an Error output to pulse indicating the type of error that occurred. In addition, the PPE can be configured to intentionally halt for certain scenarios as a breakpoint for aiding hardware or code debug.

There are 4 types of events that will halt the PPE and assert a unique error output:

Output	Type	Description
0	Internal State Error	PPE detected an invalid logic event inside the processor not caused by an external interface. Specifically: 1. State machine entered into invalid (other than defined) state 2. Internal interrupt state latches having non zero value in other than "reset, exception, or vector" states 3. More than one write port active for a particular GPR on the same cycle 4. More than one write path active for an SPR on the same cycle (including IAR updates) 5. PPE fetching instruction from MIB when IR is being used as the decode source (e.g. for RAM) instead of the MIB interface (mib_ppe_instruction) 6. Both Instruction and data request valid are simultaneously being driven active by the PPE (ppe_mib_data_req_v_int and ppe_mib_inst_req_v_int)
1	External Interface Error	PPE detected an invalid sequence or state on it inputs that is not allowed by an interface protocol. Specifically 1. Instruction ack without a PPE instruction request valid (mib_ppe_inst_ack and not(ppe_mib_inst_req_v_int)) 2. Data ack without PPE data request (mib_ppe_data_ack and not(ppe_mib_data_req_v_int)) 3. Instruction Error without instruction ACK (mib_ppe_inst_err and not(mib_ppe_inst_ack)) 4. Data Error without Data ACK (mib_ppe_data_err and not(mib_ppe_data_ack) 5. reset ack without PPE reset valid request (mib_ppe_reset_mem_ack and not(ppe_mib_reset_mem); 6. XIR request with more than one xir_addr_valid asserted (mib_ppe_xir_req and popcount(mib_ppe_xir_addr_v) > "001") 7. Memory interface driving data_multi_err along with non imprecise error type
2	Forward Progress Error	Either: • Watchdog Timeout occurred and was configured to halt (instead of interrupt or reset), indicating that the PPE is no longer making forward progress. or: • a a Machine Check occurred, or another interrupt was promoted to a machine check by MSR[UIE], when MSR[ME]=0 Note: XSR[HC] will indicate which occurred.
3	Debug/Code Breakpoint	Either: • code running on the processor requested a halt by writing DBCR[RST] = “11”, e.g. software detected a a debug scenario, an unexpected state, or erroneous condition or: • a Debug Breakpoint event halted the processor, i.e. a trap instruction or an address match when enabled in the DBCR as halt conditions Note: XSR[HC] will indicate which ocurred, and in the case of a Debug breakpoint, XSR[7:13] will contain the exact event that caused the hardware breakpoint.

Table 1.28: Error Output Definitions

7.4 Debug Registers

Several registers control debugging modes and behaviors, and allow the internal and external observation of debug status. Full descriptions of all PPE 42 registers and their fields appear in section 8, *Register Summary.* The debugging registers are only described briefly here.

7.4.1 DACR – Debug Address Compare Register

PPE 42 implements a single address comparison register, the DACR, used for both instruction- and data-address comparison. If DBCR[IACE] = '1', the DACR is used for instruction address comparison. The DACR is used for data-address comparison if DBCR[DACE] is non-0. Note that the DACR may be used on PPE42X simultaneously by IACE and DACE, and also at the same time as a compare against an address of all zeroes if DBCR[ZACE] = '1'.

7.4.2 DBCR – Debug Control Register

The DBCR controls debugging events. Debug events are enabled by writing non-0 values to DBCR[TRAP], DBCR[ZACE], DBCR[IACE] and/or DBCR[DACE].

Writing a non-0 value to DBCR[RST] causes either a soft or hard reset, or causes the processor to halt, as described with the detailed register field descriptions.

7.4.3 EDR – Error Data Register

In the event of data machine check, data storage or alignment interrupts, the EDR contains the data address associated with the exception. In the event of a program interrupt, the EDR contains the putative instruction responsible for the program interrupt.

7.4.4 ISR – Interrupt Status Register

The PPE 42 ISR consolidates analogous functions from two Power ISA Book III-E SPRs:

- The ISR includes exception status for program, alignment and data storage exceptions, similar to the Power ISA ESR (Exception Syndrome Register), in the PTR and ST fields.
- The ISR includes machine-check exception status, similar to the Power ISA MCSR (Machine Check Syndrome Register), in the MFE and MCS fields.

The ISR[EP, SRSMS] fields are specific to the PPE 42 architecture.

Strictly speaking it is not necessary to clear interrupt status from the ISR, however debugging may be easier in some cases if the ISR is cleared during interrupt processing.

7.4.5 XCR – External Control Register

The XCR gives external debugging tools and procedures the ability to start, stop, single-step, ram and reset the processor. Procedures involving XCR for debugging operations are documented later in this section.

7.4.6 XSR – External Status Register

Debug events are recorded in the XSR[TRAP, IAC, DACR, DACW] fields. These fields only *record* debug events that halt the processor, they do not *cause* the processor to halt. Strictly speaking it is not necessary to clear debug event status from the XSR, however debugging may be easier in some cases if debugging event status is cleared by debugging procedures. Debug event status is cleared by writing XCR[CMD] =

'000' on a halted processor.

The XSR also mirrors the contents of the ISR externally, in order to simplify debugging and diagnosis. The XSR also includes several other status fields that allow an external debugging agent to determine the run/wait/halt state of the processor, and determine in many cases exactly why a halted processor halted.

7.5 Debug Events

The PPE 42 core implements debug events for **trap** instructions, and instruction- and data-address comparisons. The trap and instruction address compare events can be used for software and hardware breakpoints respectively. Data address compare events are used to capture loads and/or stores to a specific address.

If an enabled debug event occurs, the processor will halt. Debug halt events are the highest priority exceptional conditions, higher than all interrupt types other than system reset events.

7.5.1 Trap Events

Trap debug events are associated with the execution of a **trap** instruction, and are enabled by setting DBCR[TRAP] = '1'. The following table details the behavior of the PPE 42 core in response to the execution of a **trap** instruction.

Table 1.29: PPE 42 **trap** Instruction Handling

DBCR[TRAP]	Action
0	A program exception is signaled, and a program interrupt will be taken if the program interrupt is the highest priority pending interrupt. If a program interrupt is taken, ISR[PTR] ← '1'.
1	Immediate Halt; XSR[TRAP] ← '1'

If the execution of a **trap** instruction causes a program interrupt, then ISR[PTR] is set to '1'. The execution of a **trap** instruction is disambiguated from an illegal instruction in the program interrupt handler by observing that ISR[PTR] = '1' for **trap**, and ISR[PTR] = '0' for an illegal instruction.

Note that architecturally, **trap** instructions are never executed, but always treated as exceptional conditions. In other words, there is no way to disable the **trap** instruction from causing an exceptional event.

Several strategies for dealing with **trap** instructions and their non-executable nature are possible, including:

- An operating environment may elect to ignore any or all **trap** instructions unless they are enabled to cause a debug halt, or unless they are enabled to perform a debug action by a software mode. In this mode the program interrupt handler for **trap** could simply increment SRR0 by 4 (to bypass the **trap** instruction), and then execute an **rfi** instruction.
- External debuggers can bypass **trap** instructions by rewriting the IAR with IAR + 4 before continuing execution from a debug halt due to a **trap**, in order to bypass the **trap** instruction.
- In some debugging schemes the **trap** instruction replaces an original program instruction, and upon the occurrence of a **trap**, the debugger re-installs the original instruction at the IAR before continuing.

7.5.2 Instruction-Address Comparison Events

Every time a new instruction is about to be fetched, the instruction address is compared with the contents

of the DACR. If the instruction address is equal to the DACR and DBCR[IACE] = '1', or is equal to all zeroes and DBCR[ZACE] = '1', an instruction-address comparison (IAC) event occurs, XSR[IAC] is set to '1' and the processor immediately halts prior to executing the instruction. Architecturally, the instruction has neither been fetched nor decoded in this case, so no other exceptions that might have otherwise been reported by the instruction will be reported. The processor will halt with the IAR containing the instruction address.

Executing through the instruction causing an IAC debug halt will require a procedure similar to the following:

1. Read-modify write the DBCR to set DBCR[IACE] = '0'
2. Single step the instruction at the IAR
3. Restore the original DBCR
4. Write XCR[CMD] = '000' to clear debug status from the XSR (optional)

Note that the DACR is a full 32-bit register, and PPE 42 instruction addresses are always 4-byte aligned. Therefore IAC debug events can only occur if $DACR_{30:31}$ = '00'.

7.5.3 Data-Address Comparison Events

Every time a data address is ready to be presented to the memory interface, the data address (for loads, store and most data cache management instructions), or the query address (for **dcbq**) is compared with the contents of the DACR. If the data address is equal to the DACR and DBCR[DACE] is not '00', or is equal to all zeroes and DBCR[ZACE] = '1', a data-address comparison (DAC) event occurs. DBCR[DACE] allows debug event selection for loads and stores either individually or together. The XSR[RDAC] and XSR[WDAC] fields report DAC debug events for loads (reads) and stores (writes) respectively. The table below details how the different instruction forms that can cause DAC debug events are treated.

Table 1.30: Instruction Treatment With Respect to DAC Debug Events

Instruction Form	All Loads	All Stores	dcbf	dcbi	dcbq	dcbt	dcbz
Treated as	Load	Store	Store	Store	Load	Load	Store

Whenever a DAC debug event occurs, the processor immediately halts prior to performing the load, store or data cache management phase of the current instruction address (CIA). Architecturally, the instruction at the CIA has been fetched and decoded in this case, and any fetch or decode phase exceptions caused by the instruction would have already been reported. No data phase (data storage or machine check) exceptions will have been reported, and no GPR updates for update-form addressing will have taken place. In other words, the instruction has effectively not been executed. The processor will halt with the IAR containing the CIA.

Executing through the instruction causing a DAC debug halt will require a procedure similar to the following:

1. Read-modify write the DBCR to set DBCR[DACE] = '00'
2. Single step the instruction at the IAR
3. Restore the original DBCR
4. Write XCR[CMD] = '000' to clear debug status from the XSR (optional)

7.5.4 Zero Address Comparison

PPE42X adds the capability to trigger a debug event (and therefore halt) on an all zero instruction or data

address via DBCR[ZACE], which allows the DACR facility to simultaneously compare on a non-zero instruction and/or data address. An example usage would be to always trap on a non-zero address since those are typically only generated by a bug in the code or data being processed by that code, e.g. an all-zero pointer. The DACR could then be used with DBCR[DACE] to simultaneously detect stack overflow conditions, or with the DBCR[IACE] to detect instruction fetches from a defined data region of memory. Both checks can now be enabled during runtime by default to expedite debug of these scenarios, when the DACR is not being used for another more specific debug situation.

7.5.5 Data Address Comparison and Alignment

The PPE42 core neither recognizes nor enforces alignment constraints on any data or data cache access. The memory subsystem attached to the PPE core returns alignment errors for any unsupported accesses. Therefore, it is possible for the debug facilities to “miss” the access of a 4-byte aligned data address if that address is targeted indirectly by an 8-byte virtual doubleword load or store. Also, since by specification the low-order bits of addresses for the data cache management instruction are ignored, it is also possible to miss debug events for these instructions if the generated address is not equal to the DACR in every bit position, even though the generated address and the contents of the DACR effectively address the same data cache block.

7.6 Halt Processing

7.6.1 Definition of Halted

The PPE 42 core is said to be *in the halted state* when the core is halted and all instruction processing has completed. The core is in the halted state if and only if XSR[HS] = '1'. At the implementation level, the halted state is identified with a particular state of the processor core. Whenever the processor is in the halted state, the IAR addresses the instruction that will be executed when leaving the halted state, or if an asynchronous interrupt is pending when leaving the halted state, the value that will be written to SRR0 when the interrupt is taken on exit from the halted state.

The XSR[HC] field details the last condition or event that caused the processor to halt. This field is only valid if XSR[HS] is '1', that is, when the processor is in the halted state.

7.6.2 Entering the Halted state

The PPE enters the halted state after there is a Halt Condition Pending, as reflected in the XSR[HCP]. The processor will halt as soon as possible, subject to the completion of any in-flight instructions. When the processor halts, the state of the core will be the precise state produced by executing all instructions preceding the instruction addressed by the IAR.

XCR[CMD] can be written with '001' to halt the processor at any time. Writing XCR[CMD] with '001' sets XSR[HCP] (halt condition pending). XSR[HCP] will also change autonomously to '1' if the processor halts for any other reason:

- An enabled debug event;
- An unmaskable interrupt with MSR[UIE] = '0' ;
- A second watchdog timeout when TCR[WRC] = '11';
- The program writing DBCR[RST] with '11';
- An unrecoverable hardware failure;
- An active level on the `halt_req` input to the core. The conditions that cause the `halt_req` signal to be active are instance specific, and will be documented with each instantiation of the PPE 42

core.

The PPE core treats XSR[HCP] similar to an exception condition that causes it to halt instead of taking an interrupt. A processor reporting XSR[HCP] = '1' may also be executing an instruction:

- If an instruction is in flight when XSR[HCP] becomes '1'
- If a halted processor is in the process of single-stepping an instruction
- If a halted processor is in the process of ramming an instruction

The processor can be *force-halted* by writing XCR[CMD] to '111'. A second watchdog timeout with TCR[WRC] = '11' also force-halts the processor. A force-halted core also reports XSR[HCP] = '1'.

Force-halting bypasses the completion of any in-flight instructions or synchronization or reset operations, and immediately puts the processor into the halted state (see below). Force-halting is designed to be used in cases where the processor will not or may not halt normally due to errors (hangs) in the memory subsystem. A system that had been force-halted may not be recoverable without a hard reset.

7.6.3 Halt Conditions and Error indication

Certain Halt conditions are considered errors and will cause the Error output to be asserted. The below table describes the seven Halt conditions and which four also cause the Error output to be asserted.

HCR[HC]	Meaning	Asserts Error Output?	Reason
000	None	N	N/A
001	XCR[CMD] written '111'	N	Intended for external code (IPL or lab tools) to control the PPE, access its architected state, and instruction step for code debug.
010	Watchdog Timeout	Y	Watchdog Halt condition. PPE can be configured to halt if a second watchdog event occurs while the previous interrupt is still pending, meaning it is not making forward progress and is therefore in a hang situation.
011	Machine Check Error	Y	Machine Check occurred when MSR[ME] = '0". If a second machine check happens while trying to service a previous machine check error. Note this can also occur if an unmaskable interrupt occurs when both MSR[UIE] = '0' and MSR[ME]='0', since this case is promoted to a machine check.
100	Debug halt	N	Conditions listed in the XSR, e.g. either trap instruction or an instruction address match or data address match were configured to halt.
101	DBCR induced halt	Y	Code running on PPE requested a halt condition. By convention this only used for error cases where code is unable to recover or make forward progress.
110	Halt Request input active	N	Halt on trigger event was enabled, halt on system checkstop was enabled, or the halt_req input activated. Used for lab debug only.
111	Hardware failure (invalid state)	Y	Logic detected that the PPE core state machine entered an invalid state, which should never happen with good hardware.

7.6.4 Exiting the Halted state

The processor exits the halted state whenever.

1. The core is reset
2. XCR[CMD] is written with '010'
3. XSR[HS] = '1' and XCR[CMD] is written with '011' to initiate single-stepping
4. XSR[HS] = '1' and the IR is written with an instruction encoding to initiate ramming

In case 1 above, the core re-enters the halted state after the reset unless the reset is caused by the `hreset_req` input. If the PPE 42 core is reset using the external `hreset_req` signal, then XSR[HCP] is reset to 0. This means that a halted core will no longer be halted if the core is reset externally, but instead will begin execution at the system reset vector. If the core is reset by an external write to XCR[CMD], XSR[HCP] is not reset, and a halted core will remain halted after the reset. A program writing to DBCR[RST], or externally ramming DBCR[RST] do not modify the XSR[HCP].

In cases 3 and 4 above the core will re-enter the halted state once the single-stepped or rammed instruction completes.

Specific procedures are provided to halt the processor, single-step and ram instructions, and exit the halted state.

7.6.5 Halting and Synchronization

The PPE 42 core does not halt with “hidden state” related to synchronous exceptions. Any imprecise machine check exceptions pending at the time of any halt remain pending in the memory interface (not the PPE 42 core). If the in-flight instruction at the time of a halt, or a single-stepped or rammed instruction causes or uncovers a synchronous exception, the interrupt will be processed and the core will then halt at the first instruction of the interrupt vector. If the processor halts due to an unmaskable interrupt halt, the normal interrupt prioritization and synchronization takes place and the core will also halt at the first instruction of the interrupt vector.

7.7 Single-Stepping and Ramming

Single-stepping and ramming first require halting the processor. Specific procedures for entering into, exiting from and operating in single-stepping and ramming modes are provided. Single-stepping and ramming are intended for different but complementary purposes, and therefore have very different behaviors.

7.7.1 Single-Stepping

Single-stepping is designed for software debugging. The processor is halted, and the application is then stepped instruction-by-instruction to observe its behavior. Register state can be observed and modified during single-stepping by ramming.

During single-stepping, instructions are fetched and executed normally, except that the processor returns to the halted state after each instruction is executed. If a single-stepped instruction triggers a debug halt event, the processor will return to the halted state before executing the instruction. Because all PPE 42 debug halts are specified to halt before executing the instruction causing the event, in general it will be necessary to modify the DBCR before a single-stepped program can make progress through the instruction causing a debug halt.

7.7.1.1 Single-stepping and Exceptions

If a single-stepped instruction causes a precise synchronous exception, or uncovers an imprecise synchronous exception, the processor will take the interrupt, all state changes associated with the interrupt will occur, and the processor will halt at the interrupt vector address formed by the IVPR and the associated interrupt vector offset. Further single-stepping will then step through the interrupt handling code.

If an asynchronous interrupt is pending prior to single-stepping, the processor will take the interrupt and halt at the associated interrupt vector offset. In this case no instruction is actually executed, and the single-stepping operation simply causes the state transitions specified for taking the asynchronous interrupt.

If a single-stepped instruction causes or unmasks an asynchronous interrupt, no action is taken

immediately. As specified above, a subsequent single-step operation will cause the asynchronous interrupt to be taken, assuming that no intervening action (ramming, changes in the environment) has removed or masked the asynchronous interrupt.

7.7.2 Ramming

Ramming will also be used during software debugging since ramming procedures allow the GPR and SPR values to be observed and modified externally. Ramming loads and stores also provides the programmer a view of memory as seen by the PPE 42 core, which may be different from the contents of external memories if caches are present, and may also involve address translation if supported by the instantiation.

However ramming is also used for hardware debugging, and for this application it is critical to be able to gather as much information as possible from the core, even if the external environment is faulty or a memory interface is hung.

The table below details various instruction forms and conditions, and the behavior of ramming in these cases.

Table 1.31: PPE 42 Ramming Behavior

Instruction(s), Condition(s) or Side-effects	Notes
IAR updates	IAR updates associated with instruction fetch are suppressed while ramming. Ramming branches will modify the IAR if the branch is taken, and ramming **rfi** will update the IAR from SRR0. However, the simplest way to modify the IAR while halted is via a direct write, not by ramming branches. Note that if a relative branch instruction is rammed and the branch is taken, the IAR serves as the CIA for the relative branch target computation.
mfsprg0, **mtsprg0**	SPRG0 is an XIR used indirectly to read (**mtsprg0**) and write (**mfsprg0**) GPRs by ramming the respective instruction. These instructions change no architected state other than the GPR targeted by **mfsprg0**.
mfspr, mfcr, mtcr0, wrtee, wrteei, arithmetic and logical instructions	Ramming these instructions do not cause state changes other than the GPR updates, CR updates for **mtcr0**, MSR[EE] updates for **wrtee** and **wrteei**, and side-effects to CR and XER from certain arithmetic and logical instructions.
mfmsr, mtmsr, rfi	In ramming mode, **mfmsr, mtmsr** and **rfi** *do not* synchronize the memory interface for outstanding exceptions, as this could hang the processor if the memory interface is hung. This allows the MSR to be read and written externally regardless of the state of the memory interface. If synchronization is required before **mfmsr, mtmsr** or **rfi** an explicit **sync** should be rammed.
sync	**sync** behaves normally, establishing execution and storage synchronization before completing. This operation may hang if the memory interface is hung.
mttsr	In ramming mode, all write-1-to-clear fields of the TSR are directly writable. This behavior supports ramming timer status bits for test and debug purposes, and also to restore the original states of the TSR, since the register contents may have been modified by side effects of rammed instructions, single-stepped instructions or external or internal events.
mtspr other than **mttsr**	These SPR updating instructions behave as normal. However, as noted below, any asynchronous interrupts they cause or uncover are completely ignored while ramming.
trap	Ramming a **trap** instruction when DBCR[TRAP] = '0' causes a program interrupt, which will be handled as described below under *interrupts*. If DBCR[TRAP] = '1', then the only effects of ramming a **trap** will be to set XSR[TRAP] = '1' and XSR[HC] = '100' since the processor is already always halted while ramming.
Interrupts	Asynchronous interrupts are completely ignored while ramming, even if the asynchronous interrupt is caused or unmasked by a rammed instruction. However pending asynchronous interrupts *will* be taken if the processor is later single-stepped or restarted. Since no instruction is fetched while ramming, ramming instructions can never cause instruction storage or instruction machine check exceptions and interrupts. Other synchronous interrupts *are* processed while ramming. For example, if a rammed load or store causes a precise data machine check or uncovers an imprecise data machine check, the core will process the interrupt, including all EDR and ISR state changes, then halt with the IAR addressing the first instruction of the interrupt vector. Similarly, ramming an illegal instruction encoding or **trap** will effectively cause a program interrupt to be handled. Note that interrupt prioritization for synchronous interrupts works as normal while ramming. For example, if a processor is halted with pending imprecise machine checks, and an otherwise legal load or store is rammed, the core will not execute the rammed instruction, but will instead take a machine check interrupt. In general it may be best to always ram a **sync** instruction prior to ramming loads and stores, in order to clear up any pending imprecise exceptions.
Debug events	In ramming mode, IAC and DAC debug events are suppressed. This means it is possible to ram instructions even if the IAR or data address matches the DACR, and the XSR will not record these events even if they are otherwise enabled in the DBCR. Ramming a **trap** will cause the normal behavior, modulo ramming-mode interrupt handling.

7.8 Debugging Procedures

Several important debugging procedures are described here at a high level. Debugging procedure require manipulation of the PPE 42 XIRs. As described in section 6, *External Interface Registers*, XIRs are always accessed as 64-bit pairs of 32-bit registers, however pairings are provided that always provide safe, 32-bit access to each XIR, as some instances of the PPE 42 core may not support 64-bit access for all external agents. For simplicity, basic debugging procedures are first described in terms of 32-bit XIR operations. Next, some accelerated debugging procedures are described that take advantage of 64-bit XIR pairs, for those instances that support 64-bit access to XIRs.

7.8.1 Basic Debugging Procedures

7.8.1.1 Halting the Processor

To safely halt the processor from any state:

1. Write XCR[CMD] with '001'.
2. Poll the XSR until XSR[HS] = '1'.

7.8.1.2 Force-Halting the Processor

If the previous procedure for halting the processor does not terminate in a reasonable amount of time, it may be necessary to force-halt the processor. Note that force-halting the processor may leave the system in an unrecoverable or inconsistent state. Prior to force halting the processor it may be useful to make note of the contents of XSR[SMS], as this may help diagnose why force-halting is required.

To force-halt the processor from any state:

1. Write XCR[CMD] with '111'.

After step 1 above, the processor will report halted status (XSR[HS] = '1') in at most two machine cycles.

7.8.1.3 Clearing Debug Halt Status

To avoid confusion while debugging, it may be helpful for debugging procedures to clear debug halt status fields from the XSR prior to restarting a halted processor. To clear debug status from the halted state:

1. Write XCR[CMD] = '000'.

The PPE 42 core neither implements nor acknowledges a request to clear debug status bits unless the processor is halted (XSR[HS] = '1').

7.8.1.4 Resetting the Processor

The processor can be reset from any state. Two types of reset are supported, the so-called *soft* and *hard* resets. The differences and effects of the hard and soft reset are instance specific. To reset the processor:

1. Write XCR[CMD] = '101' (for a soft reset) or '110' (for a hard reset).

Note that resetting the processor via the XCR does not clear XSR[HCP], that is, a halted processor remains halted after an XCR-controlled reset. A subsequent processor restart will then fetch the System Reset interrupt vector with the previous IAR placed in SRR1.

7.8.1.5 Restarting the Processor

The processor can be restarted after halting, regardless of whether the halt was due to an internal event or

an external command. Restarting a processor that was force halted, however, may not produce meaningful results. Also, if the processor is halted due to an active level on the `halt_req` input to the core, instance-specific procedures may also be necessary to deassert `halt_req` prior to restarting the processor.

When the processor is restarted from the halted state, the next instruction fetched and executed is the instruction addressed by the IAR, unless an asynchronous interrupt is pending when the processor is restarted, in which case an interrupt will be taken immediately and the IAR will be copied to SRR0.

To restart the processor

1. Write XCR[CMD] with '010'.
2. Optionally check that XSR[HS] = '0', which should be true immediately after step 1 completes.

The PPE 42 core neither implements nor acknowledges a restart request unless the processor is either halted (XSR[HS] = '1') or already running (XSR[HCP] = '0').

7.8.1.6 Single-Stepping an Instruction

The processor can single-step instructions from the halted state. Single-stepping may not produce meaningful results after force-halting the processor.

The instruction that will be single-stepped is the instruction addressed by the IAR whenever the processor is in the halted state. This is normally the next sequential instruction. If the debugging task requires single stepping another section of code, the IAR can be modified to address the other section of code prior to single-stepping.

Note that the PPE 42 core does not require an instruction cache, nor does it provide any modes to control instruction cacheability for instances that include caches. Therefore, whether or not a single-stepped instruction represents the contents of an instruction cache or the contents of memory is instance-specific.

To single-step an instruction:

1. Halt the processor using the procedure provided.
2. Modify the IAR if required.
3. Write XCR[CMD] = '011'.
4. Poll the XSR until XSR[HS] = '1'.

Repeat steps 2 through 4 to single-step multiple instructions. Note that the PPE 42 core neither implements nor acknowledges writes to the IAR nor the single-step command unless the processor is halted (XSR[HS] = '1').

7.8.1.7 Ramming an Instruction

The processor can ram instructions from the halted state. Ramming is also possible after force-halting the processor, subject to the previous caveats with respect to ramming various types of instructions.

Ramming branches and **rfi** can be used to to modify the IAR, however if the debugging task requires restarting or single stepping another section of code, it is simpler to directly write the IAR to address the other section of code prior to restarting or single-stepping.

The GPRs and may SPRs in the PPE core are only directly accessible external to the PPE core by ramming a particular sequence of instructions. The SPRG0 register may be used as an externally-accessible intermediate location in which to read data from, or write data into, other GPRs or SPRs in the PPE core via the ram mechanism.

To ram an instruction:

1. Halt the processor using the procedure provided, unless it is already halted as indicated in the XSR.(Optional) Selectively read and save any state, such as SPRG0 and any other SPRs or GPRs, that may be overwritten during the RAM operation. Beginning with PPE 42X, this includes the XSR.
2. (If needed) Write SPRG0 with any data needed by the instruction being rammed, e.g. when altering the contents of other SPRs or the GPRs.
3. Write the IR with the instruction to be executed.
4. Poll the XSR until XSR[HS] = '1'.
5. (If needed) Read SPRG0 to obtain the data result of the rammed instruction, e.g. when extracting the contents of other SPRs or the GPRs.
6. (Optional) Restore any state, such as SPRG0 and XSR, that may have been modified. Additional ram operations may be needed to restore the state of selected SPRs and GPRs.

Repeat steps 2 and 3 to ram multiple instructions. Note that the PPE 42 core neither implements nor acknowledges writes to the IR unless the processor is halted (XSR[HS] = '1').

PPE 42X adds the ability to restore the debug halt information in XSR that is cleared by the act of ramming by making the XSR writable.

7.8.1.8 Low-overhead Ramming

When ramming, all non-synchronizing, non-memory and non-cache management instructions are guaranteed to complete in at most 3 cycles after being rammed by writing the instruction to the IR. Debugging procedures may elect to reduce overhead by forgoing the polling of XSR[HS] after ramming arithmetic, compare and move from/to instructions, assuming that writes of the IR can be guaranteed to be paced at least 4 cycles apart. If maintaining the state of the processor is not important, the optional save and restore steps listed may also be skipped.

7.8.1.9 Toggling XSR[TRH]

The field XSR[TRH] controls whether timer facilities continue to run while the processor is halted. XSR[TRH] is not modified directly, instead an XCR command is provided to toggle (invert) the value of XSR[TRH].

To set the value of XSR[TRH]:

1. Observe the current value of XSR[TRH].
2. If the value of XSR[TRH] is not the desired value, write XCR[CMD] = '100' to toggle (invert) the current value.

7.8.2 Advanced Debugging Procedures

The following take advantage of simultaneous 64-bit (doubleword) XIR accesses, requiring fewer operations to be performed for debug state extraction. This is especially useful for more efficient ramming operations.

7.8.2.1 Reading Status and IAR Contents Simultaneously

Reading the XSR_IAR pair allows debug status and the current value of the IAR to be read from the PPE 42 core with a single transaction. This pairing is useful because for most types of debug halts, the value of the IAR at the time of the halt is a critical piece of information required to diagnose the halt.

7.8.2.2 Reading Status and SPRG0 Simultaneously

Reading the XSR_SPRG0 pair allows debug status and the current value of SPRG0 to be read from the PPE 42 core with a single transaction. This pairing is useful because ramming procedures will always use SPRG0 as a conduit for reading GPR and SPR data out of the core, and XSR[HS] indicates or guarantees that a previous ramming operation is complete.

7.8.2.3 Writing IR and SPRG0 Simultaneously

Writing the IR_SPRG0 pair allows a value to be written to SPRG0 while simultaneously ramming **mfsprg0 Rx** to cause the new SPRG0 value to be placed into a GPR **Rx**.

7.8.2.4 Writing XCR and SPRG0 Simultaneously

Writing the XCR_SPRG0 pair allows a value to be deposited into SPRG0 (to restore its value after a ramming sequence) and simultaneously restart the processor.

7.8.2.5 Writing XSR and IAR Simultaneously

Beginning with PPE 42X, the XSR_IAR pair is writeable, allowing the XSR to be restored to the debug halt state present in the XSR prior to the ramming operation. Of course, the XSR and IAR must first be saved prior to the ram operation.

7.8.2.6 Reading CTR

Beginning with PPE 42X, the CTR is readable via the lower word of XIR0, which previously only provided write-only access to the XCR. The CTR value may be useful in certain debug scenarios, especially when the CTR contains a branch target address.

7.8.2.7 Reading SRR0 and LR Simultaneously

Beginning with PPE 42X, XIR6 allows reading the SRR0_LR pair with a single transaction. This is useful for debug, since the most recent procedure call return address is in the LR and the instruction address at the time of the most recent interrupt is in the SRR0. Both are critical pieces of information typically needed to diagnose a halt or misbehaving code.

7.8.2.8 Reading GPR pairs (VDRs) Simultaneously

Beginning with PPE 42X, the GPRs are available to be read via XIR. Enablement of this capability is optional since this visibility may result in a security exposure for certain PPE processor applications. GPRs are otherwise only available via the ramming procedure which requires a series of operations including an XIR write, which is also a security exposure in some PPE processor applications.

When this feature is enabled, the even-numbered VDRs (0,2,4,6,8,28, & 30) are available, as well as a psuedo “VDRX” consisting of the GPR10_GPR13 pair, such that all 16 GPRs are readily visible.

8 Register Summary

All registers contained in the PPE 42 core are architected as 32-bits. PPE 42 also supports 64 bit operations targeting a pair of consecutive General Purpose Registers referred to as a Virtual Doublewords. This section summarizes the registers and the bit-field usage within the registers.

The registers are grouped into categories, based on access mode: General Purpose Registers (GPRs), Virtual Doubleword Registers (VDRs), Special Purpose Registers (SPRs), External Interface Registers (XIRs), the Machine State Register (MSR) and the Condition Register (CR).

Each instantiation of the PPE 42 core will also likely define and document a number of memory-mapped I/O registers (MMIO) and/or memory-mapped control registers (MMCR). Instance-specific MMIO and MMCR will be documented with each instantiation of the PPE 42 core.

8.1 Reserved Registers

Any register numbers not listed in the tables which follow are reserved, and should be neither read nor written. These reserved register numbers may be used for additional functions in future processors. Any attempt to execute an instruction referring to a reserved register will cause an illegal instruction exception. XIRs do not have register numbers as no specific instructions are provided to access XIRs.

8.2 Reserved Fields

For all registers having fields marked as reserved, the reserved fields should be written as zero and read as undefined. That is, when writing to a reserved field, write a 0 to the field. When reading from a reserved field, ignore the field.

As a matter of completeness however all reserved fields of PPE 42 architected registers are defined to return 0 when read.

It is good coding practice to perform the initial write to a register with reserved fields as described in the preceding paragraph, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register, alter desired fields with logical instructions, and then write the register.

8.3 General Purpose Registers

The PPE 42 core provides 16 General Purpose Registers (GPRs), selected from the 32 GPRs architected in the Power ISA. The contents of these registers can be loaded from memory using load instructions and stored to memory using store instructions. GPRs are also addressed by all integer instructions.

The table below documents the PPE 42 GPRs. The ABI notes refer to register usage as defined by the PowerPC System V Application Binary Interface (ABI), and its extension the PowerPC Embedded Application Binary Interface (EABI) for 32-bit processors. The assumption in selecting the Power ISA GPR subset to implement in PPE 42 was that the majority of PPE applications would be developed consistent with these ABIs. In particular, GPRs 0:1, 3:10 and 30:31 are critical to allow PPE 42 assembly code to be generated from current PowerPC compiler infrastructures with minimum effort. The PPC 42 GPR set is sufficient to support all ABI parameter passing mechanisms including variadic functions, allowing code libraries compiled for PPE 42 (without PPE 42 specific instructions) to execute correctly on 32-bit Power ISA platforms. Internal studies also show that the volatile/nonvolatile GPR ratio implemented by PPE 42 represents a good tradeoff for a variety of embedded application routines compiled with a modern optimizing compiler.

Table 1.32: PPE 42 General Purpose Registers

Mnemonic	Register Name	GPR Number		ABI Notes
		Decimal	Hexadecimal	
R0	GPR 0	0	x'0'	Volatile register which may be modified during subroutine linkage. R0 is also treated as containing 0 by many instructions.
R1	GPR 1	1	x'1'	The stack frame pointer
R2	GPR 2	2	X'2'	Defined as *system-reserved* by the ABI. Defined to contain the base address of the ELF sections named **.sdata2** and **.sbss2** by the EABI.
R3	GPR 3	3	x'3'	Volatile register used for parameter passing and return values
R4	GPR 4	4	x'4'	Volatile register used for parameter passing and return values
R5	GPR 5	5	x'5'	Volatile register used for parameter passing
R6	GPR 6	6	x'6'	Volatile register used for parameter passing
R7	GPR 7	7	x'7'	Volatile register used for parameter passing
R8	GPR 8	8	x'8'	Volatile register used for parameter passing
R9	GPR 9	9	x'9'	Volatile register used for parameter passing
R10	GPR 10	10	x'a'	Volatile register used for parameter passing
R13	GPR 13	13	x'd'	Defined to contain the base address of the ELF sections named **.sdata** and **.sbss** by the ABI and EABI.
R28	GPR 28	28	x'1c'	Non-volatile register used for local variables
R29	GPR 29	29	x'1d'	Non-volatile register used for local variables
R30	GPR 30	30	x'1e'	Non-volatile register used for local variables
R31	GPR 31	31	x'1f'	Non-volatile register used for local variables

8.4 Virtual Doubleword Registers

PPE 42 supports 64-bit operations to pairs of consecutive GPRs known as Virtual Doubleword Registers (VDRs). To the PPE 42 core, instructions targeting virtual doublewords are atomic operations; however the handling of virtual doubleword load and store operations by the memory interface is instance-specific.

A VDR is designated in the instruction encodings by the GPR number of the GPR containing the high-order 32 bits of the VDR. The next consecutive GPR (modulo 32) contains the low-order 32-bits of the VDR. Loading and storing a VDR always operates on both GPRs atomically, and in no way alters the ability to independently operate on either GPR either before, during or after its use as part of a VDR.

The sixteen PPE 42 GPRs can also be used as fourteen VDRs as described in the table below. GPRs 10 and 13 can not be used to designate a VDR as PPE 42 does not define the next consecutive GPR (modulo 32) for these GPRs. Note that use of several of the VDRs (for example, D0, D1, D2 and D31) will be precluded in most cases by ABI considerations.

Table 1.33: PPE 42 Virtual Doubleword Registers

Mnemonic	Register Name	GPRs		VDR Number	
		High Order	Low Order	Decimal	Hexadecimal
D0	VDR 0	GPR 0	GPR 1	0	x'0'
D1	VDR 1	GPR 1	GPR 2	1	x'1'
D2	VDR 2	GPR 2	GPR 3	2	x'2'
D3	VDR 3	GPR 3	GPR 4	3	x'3'
D4	VDR 4	GPR 4	GPR 5	4	x'4'
D5	VDR 5	GPR 5	GPR 6	5	x'5'
D6	VDR 6	GPR 6	GPR 7	6	x'6'
D7	VDR 7	GPR 7	GPR 8	7	x'7'
D8	VDR 8	GPR 8	GPR 9	8	x'8'
D9	VDR 9	GPR 9	GPR 10	9	x'9'
D28	VDR 28	GPR 28	GPR 29	28	x'1c'
D29	VDR 29	GPR 29	GPR 30	29	x'1d'
D30	VDR 30	GPR 30	GPR 31	30	x'1e'
D31	VDR 31	GPR 31	GPR 0	31	x'1f'

8.5 Machine State Register and Condition Register

Because these registers are accessed using special instructions, they do not require addressing. The **mfmsr**/**mtmsr** and **mfcr**/**mtcr0** instructions are used to access the MSR and CR[CR0] respectively.

8.6 Special Purpose Registers

Special Purpose Registers (SPRs) are used for subroutine linkage and iteration, and control the use of the debug facilities, timers, interrupts, and other architected processor resources. SPRs are accessed using the move to special purpose register (**mtspr**) and move from special purpose register (**mfspr**) instructions. The CTR and LR are also implicitly read and written by several instructions. All SPRs may be read by any PPE 42 program, and most may be written by any PPE 42 program. A subset of SPRs are read-only to all programs.

PPE 42 SPRs are modeled after Power ISA BOOK I and Book III-E architected SPRs that serve an identical or analogous function, and PPE 42 borrows Power ISA Book I and Book III-E SPR names for many SPRs, and Power ISA Book I and Book III-E SPR numbers for all SPRs. This convention does not mean that PPE 42 implements the Power ISA specification for like-named or numbered SPRs; This manual provides the full specification of PPE 42 SPRs.

The table below shows the mnemonics, names, and numbers of the PPE 42 SPRs. All SPR numbers that are not listed are reserved and must be neither read nor written. The SPRN columns list the register numbers used as operands in assembler language coding of the **mfspr** and **mtspr** instructions. The translation of the SPRN into the SPRF field of the **mfspr** and **mtspr** instructions is documented with those instructions. The analogous Power ISA Book I or Book III-E SPR is also indicated.

Table 1.34: PPE 42 Special Purpose Registers

Mnemonic	Register Name	SPRN		Access	Power ISA Book I or Book III-E Analog
		Decimal	Hexadecimal		
CTR	Count Register	9	x'009'	Read/Write	CTR
DACR	Debug Address Compare Register	316	x'13c'	Read/Write	DAC1
DBCR	Debug Control Register	308	x'134'	Read/Write	DBCR0
DEC	Decrementer	22	x'016'	Read/Write	DEC
EDR	Error Data Register	61	x'03d'	Read/Write	DEAR
ISR	Interrupt Status Register	62	x'03e'	Read/Write	ESR/MCSR
IVPR	Interrupt Vector Prefix Register	63	x'03f'	Read-only	IVPR
LR	Link Register	8	X'008'	Read/Write	LR
PIR	Processor Identification Register	286	x'11e'	Read-only	PIR
PVR	Processor Version Register	287	x'11f'	Read-only	PVR
SPRG0	SPR General 0	272	x'110'	Read/Write	SPRG0
SRR0	Save/Restore Register 0	26	x'01a'	Read/Write	SRR0
SRR1	Save/Restore Register 1	27	x'01b'	Read/Write	SRR1
TCR	Timer Control Register	340	x'154'	Read/Write	TCR
TSR	Timer Status Register	336	x'150'	Read/Write	TSR
XER	Fixed-Point Exception Register	1	x'001'	Read/Write	XER

8.6.1 Using SPRs as Scratch Registers

Due to the limited number of GPRs provided by PPE 42, there may be cases where it is expedient to use an SPR as a scratch register in application code, rather than saving and restoring temporary values to memory. Several SPRs implement all 32 bits and can potentially be used this way, as detailed in the following table.

Table 1.35: SPRs Implementing All 32 Bits, Potentially Usable as Scratch Registers

SPR	Notes and Caveats
CTR LR	These registers are considered part of the programmer state and should always be saved and restored across interrupts. The CTR would be the first choice for use as a scratch register, since the CTR is not required to be saved and restored across subroutine calls in standard Power ISA ABI specifications, whereas the LR must be.
SPRG0 EDR	An operating environment may or may not save these registers across interrupts, meaning they could only potentially be used when interrupts are disabled. However it is also possible that SPRG0 has a specific use and will not be allowed to be modified. The EDR could only be used if interrupts that set the EDR are considered unrecoverable to the code using the EDR as scratch.
DACR	It is possible, but unlikely that the DACR could be used as a scratch register, since this would require limiting the types of debugging that could be performed on the application.

8.7 External Interface Registers

External interface registers (XIRs) are used to control and observe the state of the PPE 42 core from outside of the core. XIRs are architected 32-bit registers that are not necessarily part of the PPE 42 programming model, but some registers may be both XIRs and SPRs.

The way that XIRs are accessed will vary with each instantiation of the PPE 42 core. Each PPE 42 instance will document register addressing and access conventions for PPE 42 XIRs, for example an instance of PPE 42 may allow programs to access XIRs as MMCRs.

Although the XIRs are architected as 32-bit registers, XIRs are always read and written externally as 64-bit pairs of 32-bit registers. XIR access and pairing is fully documented in section 6, *External Interface Registers.*

8.8 Simultaneous Update

Certain SPRs contain writable fields that also change asynchronously with respect to program execution. Unless otherwise explicitly specified, if the modification of a field of an SPR from a PPE program is simultaneous with an update from the underlying hardware, the update from the PPE program always takes precedence.

Note that the preceding paragraph refers to *fields* of registers. For register fields that are specified with the *write-1-to-clear* semantics, writing a 0 to the field does not inhibit a simultaneous asynchronous update to the field.

Certain XIRs also contain writable fields that similarly change asynchronously with respect to the external environment. The PPE 42 only core only allows updates to XIRs in processor states where there is no possibility of simultaneous access.

8.9 Initialization and Reset

Initialization and reset of the PPE 42 core register state is fully covered in section 3, *Initialization,Reset, and Starting Execution.*

8.10 Alphabetical Listing of PPE 42 Registers

The following pages list the registers available in the PPE 42 core. For each register, the following information is supplied:

- Register mnemonic and name
- Register access information, including the register type and number if appropriate and allowed access modes
- A table describing the register fields giving each field mnemonic, field bit location and a brief description of the function of the field

Unless an entire register is read/write or read-only, the register description tables include specific access modes for individual register fields. Access mode abbreviations are described in the table below.

Table 1.36: Register Field Access Modes

Mode	Description
RW	Read/Write
RWX	Read/Write, with updates also possible from the underlying hardware
RO	Read-only; Any write to the register field is ignored.
W1TC	Write-1-to-clear; These SPR fields represent status set by the underlying hardware. Writing a '1' to the field clears the field to '0'. Writing a '0 to the field does not change the value of the field. However, note that ramming an **mtspr** instruction targeting the SPR allows the fields to be directly written from the contents of a GPR.
WO	Write-only; These fields exist for side-effect only. Writing values to the field may cause a side effect, however reading the field always returns 0.

8.10.1 CR – Condition Register

Register Access: mtcr0 and mfcr instructions

Table 1.37: CR – Condition Register

Bits	Field	Description	Notes
0:3	CR0	Bit 0: Less Than (LT) Bit 1: Greater Than (GT) Bit 2: Equal (EQ) Bit 3: Summary Overflow (SO)	PPE 42 only implements CR[CR0] of the Power ISA specification.
4:31	Reserved	Ignored by **mtcr0**; Always read as 0 by **mfcr**	

8.10.2 CTR – Count Register

Register Access: SPR 9, Read/Write; PPE42X: XIR, Read-only

Table 1.38: CTR – Count Register

Bits	Field	Description
0:31	CTR	The CTR simplifies iteration by way of conditional branch instruction forms predicated on the value of the decremented CTR. The CTR holds the targets of indirect branches for the **bcctr[l]** instruction. Note that the branch target for **bcctr[l]** is specified as $(CTR)_{0:29} \parallel {}^{2}0$. With careful programming the CTR may also be used as a 32-bit scratch register.

8.10.3 DACR – Debug Address Compare Register

Register Access: SPR 316, Read/Write

Table 1.39: DACR – Debug Address Compare Register

Bits	Field	Description
0:31	DACR	DACR holds the 32-bit address used to generate debug events for both instruction address comparison (DBCR[IACE] = '1') and data address comparison (DBCR[DACE] ≠ '00'). Note: For PPE42, all 32 bits of DACR are implemented and compared. When comparing instruction addresses, bits 30:31 of the instruction address must be configured as '00' for the comparison to succeed. For PPE42X, DBCR[ACS] chooses which 32 bits of DACR to use in the comparison, to select a byte, word, doubleword, or octword region. Note that the byte compare selection instead does word compare for instruction addresses.

8.10.4 DBCR – Debug Control Register

Register Access: SPR 308, Read/Write

Table 1.40: DBCR – Debug Control Register

Bits	Field	Description	Mode	Notes
0:1	Reserved			
2:3	RST	Reset 00 No Action 01 Soft Reset 10 Hard Reset 11 Halt	WO	Writing this field to a non-0 value immediately causes the specified action. The effect of reset on the PPE 42 core is define in section 3, *Initialization,Reset, and Starting Execution.* The effect of resetting the environment is specific to each instantiation of PPE 42. Writing the '11' (halt) command sets XSR[HCP] to '1'. This event will halt the processor with the IAR containing the address of the instruction following the **mtdbcr** instruction that wrote this field to '11'. Reading this field always returns the value '00'.
4:6	Reserved			
7	TRAP	Trap Instruction Enable 0 Disabled 1 Enabled	RW	If disabled, the **trap** instruction causes a program exception. If enabled, the **trap** instruction sets XSR[TRAP] = '1' and causes the processor to halt.
8	IACE	Instruction Address Compare Enable 0 Disabled 1 Enabled	RW	If enabled, then if the instruction address matches the DACR, XSR[IAC] is set to '1' and the processor halts prior to fetching the instruction.
9:11	Reserved			
12:13	DACE	Data Address Compare Enable 00 Disabled 01 Compare store addresses 10 Compare load addresses 11 Compare store and load addresses	RW	If DBCR[DACE] ≠ '00',and a load and/or store address matches the DACR as selected by DBCR[DACE], then the processor halts before executing the load, store or cache control instruction. See the notes for XSR[RDAC] and XSR[WDAC] for a description of how XSR is updated by this debug event. PPE42: also requires DBCR[IACE] = '0', a restriction removed by PPE42X.
14:23	Reserved			
24:25	ACS	Address Compare Size 00 Data: Compare all bits (byte address) Instruction: ignore two lsb (word address) 01 Ignore two lsb (word address) 10 Ignore three lsb (doubleword address) 11 Ignore five lsb (octword address)	RW	PPE42: Reserved. PPE42x: Chooses which bits of the DACR to include in the address comparison when IACE or DACE are non-zero. Omits selected least significant bits of the address from the comparison, to allow for any instruction or data accesses to a word (4 byte) doubleword (8 byte) or octword (32 byte cache line) regions. Note that encode 00 actually performs word, not byte, address comparisons for instruction addresses, same as the 01 encode.
26:28	Reserved			
29	ZACE	Zero Address Compare Enable	RW	PPE42: Reserved. PPE42x: when set and either the instruction or data address to be accessed all zero, the processor halts prior to fetching or executing the instruction. Setting this bit is equivalent to enabling both DBCR[IACE]='1' and DBCR[DACE]='11' and DBCR[ACS]='00' and DACR to all zeroes. However, when

Bits	Field	Description	Mode	Notes
				this bit is set, the DACR can simultaneously compare on a non-zero instruction and/or data address.
30	HDFI	Halt Debug Freeze Interrupts	RW	PPE42: Reserved. PPE42x: when set, Halt Conditions other than XCR[CMD] will freeze the state of External Interrupts for debug. Specifically, whenever this bit is set and XSR[HC] is greater than 0x1, the XSR[EP], the TSR, and the state of any interrupts feeding the external interrupt pin of the PPE core are frozen for debug.
31	HDFT	Halt Debug Freeze Timers	RW	PPE42: Reserved. PPE42x: when set, Halt Conditions other than XCR[CMD] will freeze the state of the PPE Timers for debug. Specifically, whenever this bit is set and XSR[HC] is greater than 0x1, the Decrementer, Fixed Interval Timer, and Watchdog Timer are frozen. Note that is mode is similar to XSR[TRH] except that mode includes XSR[HC]==0x1, and is intended for instruction stepping.

8.10.5 DEC – Decrementer

Register Access: SPR 22, Read/Write

Table 1.41: DEC – Decrementer

Bits	Field	Description
0:31	DEC	The decrementer is a free-running 32-bit decrementer. The decrementer counts down by 1 on each clock cycle that the decrement condition is true (as controlled by TCR[DS]), underflowing to x'FFFF FFFF' if the current contents of DEC are 0 when this is true. Decrementer interrupt status (TSR[DIS]) is set to '1' on any cycle that $(DEC)_0$ transitions from '0' to '1', including transitions caused by **mtspr** instructions targeting DEC. The decrementer does not decrement if a halt condition is present (XSR[HCP] = '1') unless XSR[TRH] = '1' (timers run while halted).

8.10.6 EDR – Error Data Register

Register Access: SPR 61, Read/Write; XIR, Read-only

Table 1.42: EDR – Data Error Register

<table>
<tr><th>Bits</th><th>Field</th><th>Description</th></tr>
<tr><td>0:31</td><td>EDR</td><td>The EDR holds data associated with certain types of exceptions. The EDR is updated when the exception is taken as an interrupt, and remains unchanged until the next EDR-updating interrupt occurs or until the EDR is explicitly overwritten. The following table details the information placed in the EDR based on the type of interrupt.

Table 1.43: EDR Contents by Interrupt
<table>
<tr><th>Interrupt</th><th>EDR Contents</th></tr>
<tr><td>Data storage</td><td rowspan="3">The data address responsible for the exception.</td></tr>
<tr><td>Alignment</td></tr>
<tr><td>Data Machine Check</td></tr>
<tr><td>Program</td><td>The 32-bit value being decoded as an instruction at the time of the exception. For Program Exceptions caused by trap instructions, this will be a trap instruction.</td></tr>
</table>
The EDR is also accessible read-only as an XIR.</td></tr>
</table>

8.10.7 IAR – Instruction Address Register

Register Access: XIR, Read/Write

Table 1.44: IAR – Instruction Address Register

Bits	Field	Description
0:29	IAR	When the processor is fully halted (XSR[HS] = '1'), reading the IAR returns the address of the next instruction that will be executed when the processor is returned to the running state, or single-stepped. Architecturally, the instruction addressed by the IAR has not been executed. If the processor is halted in the wait-enable state (XSR[HS] = '1' and XSR[WS] = '1'), the IAR contains the address of the instruction following the **mtmsr** instruction that set MSR[WE], however this instruction is not executed upon resumption as a core in the wait-enable state remains in that state until an asynchronous exception occurs. If the processor halts due to a machine check occurring with MSR[ME] = '0', or due to an unmaskable interrupt when MSR[UIE] = '0' and MSR[ME] = '0', the IAR will contain the address of the interrupt vector associated with the interrupt. If the processor is halted due to a debug event then the IAR contains the address of the instruction associated with the event. For **trap** debug halts this is the address of the **trap** instruction. For instruction-address compare debug halts this is the address contained in DACR. For data-address compare debug halts this is the address of the instruction that would perform the load or store to the address contained in DACR. When the processor is running, single-stepping or ramming(XSR[HS] = '0'), reading the IAR can be used to gauge program progress, however the interpretation of the IAR with respect to other observable state is not specified here. Although the values observed in the IAR during execution are always instruction addresses generated by the program flow, there is no guarantee that the instruction addressed by the IAR has been or will be executed. The IAR is read/write as an XIR. When single-stepping, the IAR is updated by the effects of each instruction as it executes, and by asynchronous interrupts processed during single-stepping. When ramming, the IAR is not modified unless either a taken branch or **rfi** is explicitly rammed, or ramming causes or uncovers a precise or imprecise synchronous exception. Asynchronous exceptions are ignored while ramming. The IAR can only be written when the processor is halted (XSR[HS] = '1').
30:31	Reserved	Always read as '00'.

8.10.8 IR – Instruction Register

Register Access: XIR, Read/Write

Table 1.45: IR – Instruction Register

Bits	Field	Description
0:31	IR	The IR can be read at any time, but can only be written when the processor is halted. When the processor is halted (XSR[HS] = '1'), writing the IR initiates *ramming*, that is, the processor executes the instruction written to the IR and then returns to the halted state. During normal execution the IR is only updated when the memory interface acknowledges an instruction fetch. In other words, whenever the processor halts the IR contains the last instruction fetched (but not necessarily executed). This fact may be useful for debugging. The IR is also copied to the EDR whenever a program interrupt is taken.

8.10.9 ISR – Interrupt Status Register

Register Access: SPR 62, Read/Write and Read-only

Table 1.46: ISR – Interrupt Status Register

Bits	Field	Description	Mode	Notes
0:15	Reserved			
16:19	SRSMS	System Reset State Machine State.	RW	This field records the value of the PPE 42 core state machine state at the time of the most recent system reset. The interpretation of this field is documented with the PPE 42 core hardware design. This field is implemented to assist debugging of watchdog timer resets. Should a reset event occur simultaneously with the execution of an **mtisr** instruction, the field will be updated according to the reset semantics.
20	Reserved			
21	EP	MSR[EE] Maskable Event Pending 0 No event maskable by MSR[EE] is pending 1 An event maskable by MSR[EE] is pending	RO	This field reads as '1' whenever any of the following conditions are satisfied, otherwise '0': • The external interrupt input of the PPE 42 core is active • TCR[DIE] = '1' and TSR[DIS] = '1' • TCR[FIE] = '1' and TSR[FIS] = '1' • TCR[WIE] = '1' and TSR[WIS] = '1' Note that the value of XSR[EP] is not predicated on the value of MSR[EE] or any other architected state not mentioned above.
22:23	Reserved			
24	PTR	Program Interrupt from **trap** 0 Program interrupt caused by illegal instruction 1 Program interrupt caused by **trap** instruction	RW	This field is updated by the hardware whenever a program interrupt is either taken directly, taken as a machine check interrupt, or causes a processor halt. If '0', then the program interrupt was caused by an illegal instruction. If '1', then the program interrupt was caused by execution of a **trap** instruction.
25	ST	Data Interrupt Caused by a Store 0 Data interrupt caused by a load 1 Data interrupt caused by a store	RW	This field is updated by the hardware whenever a data storage or alignment interrupt is either taken directly, taken as a machine check interrupt, or causes a processor halt. Under these conditions the field will be set if the operation causing the exception was either a store, or a store-type data cache operation (**dcbf**, **dcbi** or **dcbz**), otherwise the field will be cleared.
26:27	Reserved			

Continued next page

Bits	Field	Description	Mode	Notes
28	MFE	Multiple Fault Error 0 A single machine check is reported 1 Multiple imprecise store-type machine checks are reported as one	RW	This field is valid whenever a machine check interrupt is either taken or causes a processor halt, including the case of other unmaskable interrupts promoted to machine checks. If set, it indicates either that the memory interface reported multiple imprecise store-type machine checks simultaneously, or that a second (or multiple) imprecise store machine check(s) was (were) reported during synchronization and prioritization following the occurrence of a first imprecise store machine check. If ISR[MFE] = '1', ISR[MCS] will always read as '011' indicating that multiple imprecise store errors are being reported. To clarify further, this field is read as '1' only if multiple imprecise store-type machine checks are reported simultaneously; it does not indicate that machine checks of different types are being reported simultaneously. For example, if an instruction machine check or data machine check for a load occurs, and synchronization and prioritization then uncovers (an) imprecise store machine check(s) pending, only the imprecise store machine check(s) will be reported. The instruction or load data machine checks will then be rediscovered in the event that software recovers and resumes execution at the excepting instruction. The multiple-fault error indicates that error recovery information may have been irrevocably lost from the hardware state. In particular, the EDR will only report one of the data addresses responsible for the multiple machine checks.
29:31	MCS	Machine Check Status 000 – Instruction machine check 001 – Data machine check – load 010 – Data machine check – precise store 011 – Data machine check – imprecise store 100 – Program interrupt , promoted 101 – Instruction storage interrupt, promoted 110 – Alignment interrupt, promoted 111 – Data storage interrupt, promoted	RW	This field is valid whenever a machine check interrupt is either taken, or causes a processor halt. This field details the cause of the most recent machine check. Unmaskable interrupts are taken and reported as machine check interrupts whenever MSR[UIE] = '0' and MSR[ME] = '1'. Note that a program interrupt taken as a machine check will still update ISR[PTR], and data storage and alignment interrupts taken as a machine check will still update ISR[ST].

8.10.10 IVPR – Interrupt Vector Prefix Register

Register Access: SPR 63, Read-only

Table 1.47: IVPR – Interrupt Vector Prefix Register

Bits	Field	Description
0:22	IVPR	The IVPR holds the base address of the 512-byte aligned interrupt vector area in memory. Although the IVPR is read-only as an SPR, an instantiation of the PPE 42 core may provide write access to IVPR as an MMCR.
23:31	Reserved	Always read as '0000000000'

8.10.11 LR – Link Register

Register Access: SPR 8, Read/Write; Also updated by branch with link instruction forms;
PPE42X: XIR, Read-only

Table 1.48: LR – Link Register

Bits	Field	Description
0:31	LR	The LR is updated with the address of the next sequential instruction whenever a Power ISA branch or PPE fused compare-branch instruction specifying LK = '1' is executed, regardless of whether the branch is taken or not taken. The LR also holds the targets of indirect branches for the **bclr[l]** instruction. Note that the branch target for **bclr[l]** is specified as $(LR)_{0:29} \parallel {}^{2}0$. With careful programming the LR may also be used as a 32-bit scratch register.

8.10.12 MSR – Machine State Register

Register Access: Written by **mtmsr, wrtee and wrteei**; Read by **mfmsr**

Table 1.49: MSR – Machine State Register

Bits	Field	Description	Notes
0	Reserved		
1:7	SEM	Service Interface Bus (SIB) Error Mask Bit 0: Mask for SIB return code 1 Bit 1: Mask for SIB return code 2 Bit 2: Mask for SIB return code 3 Bit 3: Mask for SIB return code 4 Bit 4: Mask for SIB return code 5 Bit 5: Mask for SIB return code 6 Bit 6: Mask for SIB return code 7	The bits of MSR[SEM] individually mask Service Interface Bus (SIB) return codes 1 through 7 respectively for data accesses. Return code 0 is always interpreted to indicate successful completion, and never generates an error. Return codes of 0 are accumulated in MSR[SIBRCA], however. If a bit in the mask is '0' and a SIB data access returns the associated code, a data machine check exception is generated. If a bit in the mask is '1' and a SIB data access returns the associated code then the contents of the GPR or VDR target of a load and the effect of a store on the targeted component are implementation dependent. Regardless, the contents of MSR[SIBRC] are always updated with the return code of the most recently completed SIB data access, and the corresponding bit of MSR[SIBRCA] is set. Note that non-0 return codes on instruction fetches can not be masked from SIB memory spaces, and always generate an instruction machine check.
8	IS0	Instance-specific Field 0	The state of this field has no effect on the operation of the PPE 42 core. The contents of the field are simply presented to the environment as context-specific information for instance-specific control purposes.
9:11	SIBRC	Last SIB Return Code	This field contains the Service Interface Bus (SIB) return code for the most recently completed SIB data access, regardless of whether the access was successful or caused an error. Note that in imprecise mode (MSR[IPE] = '1'), the value observed is the value of the last SIB access that completed, which is not necessarily the value of the last SIB access that was executed. This field is also updated for instruction fetches from SIB memory areas but only if the instruction fetch causes a machine check due to a non-zero SIB return code. Note that this is the only field of the MSR that is not cleared by any interrupt. MSR[SIBRC] is visible externally as XSR[SIBRC].
12	LP	Low Priority 0 Context requests high priority 1 Context requests low priority	The PPE 42 core asserts a *priority* signal to the environment whenever MSR[LP] = '0' or ISR[EP] = '1'. The value of MSR[LP] and the state of the *priority* signal have no effect on the operation of the core, and the *priority* signal is not required to have any effect in the environment. MSR[LP] is visible externally as XSR[LP].
13	WE	Wait Enable 0 Processor is not in WAIT state 1 Processor is in WAIT state	If MSR[WE] = '1' and the processor is not halted, then the processor is in the wait state, and no instructions are fetched and/or executed. The processor exits the wait state on any enabled asynchronous interrupt. MSR[WE] is visible externally as XSR[WS].
14	IS1	Instance-specific Field 1	The state of this field has no effect on the operation of the PPE 42 core. The contents of the field are simply presented to the environment as context-specific information for instance-specific control purposes.
Continued next page			

<table>
<tr><th colspan="4">MSR - Machine State Register – Continued from preceding page</th></tr>
<tr><th>Bits</th><th>Field</th><th>Description</th><th>Notes</th></tr>
<tr><td>15</td><td>UIE</td><td>Unmaskable Interrupt Enable
0 Unmaskable interrupts are handled specially
1 Unmaskable interrupts are taken normally</td><td>This field controls whether the unmaskable interrupts (other than the machine check interrupt) are allowed to be taken. The interrupts controlled by this field are the program interrupt, alignment interrupt, data storage interrupt and instruction storage interrupt.

If this field is '1', then the controlled interrupts are taken and processed normally. If this field is '0' and a controlled interrupt is pending and the highest priority interrupt, then there are two cases:

1. If MSR[ME] = '1', then the controlled interrupt is taken as a machine check interrupt. Any ISR updates associated with the controlled interrupt take place, however ISR[MCS] is also set to indicate a controlled interrupt has been promoted to a machine check, and control transfers to the machine check interrupt vector.

2. If MSR[ME] = '0', then the controlled interrupt immediately halts the processor. Once halted (XSR[HS] = '1'), the processor state is consistent with the processor having taken the controlled interrupt but halted before executing the first instruction at the controlled interrupt vector address.</td></tr>
<tr><td>16</td><td>EE</td><td>External Enable
0 External exceptions are disabled
1 External exceptions are enabled</td><td>The following conditions will cause the associated interrupt to occur if and only if MSR[EE] = '1', and the associated interrupt is the highest priority interrupt pending.

Table 1.50: Interrupts Controlled by MSR[EE]
<table>
<tr><th>Condition</th><th>Interrupt</th></tr>
<tr><td>The environment drives an active value on the external interrupt input</td><td>External Interrupt</td></tr>
<tr><td>TSR[WIS] = '1' and TCR[WIE] = '1'</td><td>Watchdog Timer Interrupt</td></tr>
<tr><td>TSR[FIS] = '1' and TCR[FIE] = '1'</td><td>Fixed Interval Timer Interrupt</td></tr>
<tr><td>TSR[DIS] = '1' and TSR[DIE] = '1'</td><td>Decrementer Interrupt</td></tr>
</table></td></tr>
<tr><td>17:18</td><td colspan="3">Reserved</td></tr>
<tr><td>19</td><td>ME</td><td>Machine Check Enable
0 Machine Check interrupts halt the processor
1 Machine Check interrupts are taken</td><td>If MSR[ME] = '0' then any machine check interrupt immediately halts the processor. Once halted (XSR[HS] = '1'), the processor state is consistent with the processor having taken the machine check interrupt but halted before executing the first instruction at the machine check interrupt vector address.

If MSR[ME] = '1' then any machine check immediately generates a machine check interrupt, as the machine check interrupt is the highest priority interrupt. The ISR fields recording the cause of the machine check will be valid at the entry point of the machine check interrupt handler.

MSR[ME] also controls the behavior of MSR[UIE] (which see).</td></tr>
<tr><td>20</td><td>IS2</td><td>Instance-specific Field 2</td><td>The state of this field has no effect on the operation of the PPE 42 core. The contents of the field are simply presented to the environment as context-specific information for instance-specific control purposes.</td></tr>
<tr><td>21</td><td>IS3</td><td>Instance-specific Field 3</td><td>The state of this field has no effect on the operation of the PPE 42 core. The contents of the field are simply presented to the environment as context-specific information for instance-specific control purposes.</td></tr>
<tr><td>22</td><td colspan="3">Reserved</td></tr>
<tr><td>23</td><td>IPE</td><td>Imprecise Mode Enable
0 Imprecise mode is disabled</td><td>This field controls whether the storage subsystem must report store errors to the PPE 42 core precisely (IPE = '0'), or may report store errors imprecisely (IPE = '1').</td></tr>
</table>

MSR - Machine State Register – *Continued from preceding page*			
		1 Imprecise mode is enabled	Imprecise mode provides a potential performance improvement as the PPE 42 core may continue execution with one or more stores pending in the storage subsystem.
24:31	SIBRCA	SIB Return Code Accumulator Bit 0: SIB code 0 observed Bit 1: SIB code 1 observed Bit 2: SIB code 2 observed Bit 3: SIB code 3 observed Bit 4: SIB code 4 observed Bit 5: SIB code 5 observed Bit 6: SIB code 6 observed Bit 7: SIB code 7 observed	The bits of MSR[SIBRCA] individually accumulate Service Interface Bus (SIB) return codes 0 through 7 respectively for data accesses and unsuccessful instruction accesses. Whenever a SIB data access completes, the corresponding bit of MSR[SIBRCA] is set depending on the return code, regardless of whether it causes a machine check as configured in MSR[SEM]. Instruction accesses to SIB memory spaces only update MSR[SIBRCA] if they experience a non-zero return code and thereby cause a machine check. Successful instruction fetches to SIB do not update this field. MSR[SIBRCA] can be cleared by writing the field with **mtmsr**, and the field is always cleared by taking any interrupt.

8.10.13 PIR – Processor Identification Register

Register Access: SPR 286, Read-only

Table 1.51: PIR – Processor Identification Register

Bits	Field	Description
0:15	Reserved	
16:31	PIR	The PPE 42 PIR is architected as 16 bits. The contents of the PIR are specific to each instantiation of the PPE 42 core, however the intention is that the PIR uniquely identifies an instance of the PPE 42 core in a multi-core system. Although the PIR is read-only as an SPR, the contents of the PIR originate outside of the PPE 42 core, and it is possible that an instantiation of the PPE 42 core may provide a method to modify the PIR.

8.10.14 PVR – Processor Version Register

Register Access: SPR 287, Read-only

Table 1.52: PVR – Processor Version Register

Bits	Field	Description
0:7	CMN x'42'	Core Model Number This field always contains the value x'42' for a PPE 42 core.
8:11	CVN	Core Version Number PPE 42 (Original) = 0x0 PPE 42X (Extensions) = 0x1
12:31	ISVI	Instance-Specific Version Information The contents of this field will be documented with each instance of the PPE 42 core. Although the PVR is read-only as an SPR, the contents of PVR[ISVI] originate outside of the PPE 42 core, and it is possible that an instantiation of the PPE 42 core may provide a method to modify this field.

8.10.15 SPRG0 – SPR General 0

Register Access: SPR 272, Read/Write; XIR, Read/Write

Table 1.53: SPRG0 – SPR General 0

Bits	Field	Description
0:31	SPRG0	SPRG0 is a 32-bit scratch register. SPRG0 is architected both as an SPR and an XIR. Each PPE 42 instance will document the method to access SPRG0 as an XIR. SPRG0 can only be written as an XIR when the processor is halted (XSR[HS] = '1'). .

8.10.16 SRR0 – Save Restore Register 0

Register Access: SPR 26, Read/Write; PPE42X: XIR, Read-only

Table 1.54: SRR0 – Save Restore Register 0

Bits	Field	Description
0:29	SRR0	Taking an interrupt sets SRR0 to the current instruction address, if the current instruction did not complete due to the interrupt. Interrupts taken after the current instruction completes will set SRR0 to the address of the next sequential instruction. Executing an **rfi** instruction updates the value of the IAR (the NIA) from SRR0.
30:31	Reserved	Ignored on write; Always read as '00'

8.10.17 SRR1 – Save Restore Register 1

Register Access: SPR 27, Read/Write

Table 1.55: SRR1 – Save Restore Register 1

Bits	Field	Description
0:32	SRR1	SRR1 is an image of the Machine State Register (MSR). All fields marked reserved in the MSR are also marked reserved in SRR1. For details of MSR fields see section 8.10.12, MSR – Machine State Register. Taking an interrupt sets SRR1 to the current value of the MSR, after all instructions preceding the interrupt have completed to the point of reporting any exceptions they might possibly report, but before any interrupt-specific changes have been made to the MSR. Executing an **rfi** instruction updates the value of the MSR from SRR1.

8.10.18 TCR – Timer Control Register

Register Access: SPR 340, Read/Write

Table 1.56: TCR – Timer Control Register

Bits	Field	Description	Notes
0:1	WP	Watchdog Timer (WDT) Period 00 WDT uses `timer[0]` 01 WDT uses `timer[1]` 10 WDT uses `timer[2]` 11 WDT uses `timer[3]`	This field selects which of 4 external timebase inputs `timer[0:3]` causes TSR[WIS] to be set. The presence and frequency of any external timebase is instance specific.
2:3	WRC	Watchdog Timer Reset Control 00 No WDT reset will occur 01 WDT reset action is a soft reset 10 WDT reset action is a hard reset 11 WDT reset action force-halts the PPE core	If TCR[WRC] is not '00', TSR[WIS] is '1' and the external timebase input selected by TCR[WP] is asserted then the associated watchdog action will occur. The effect of resetting the PPE 42 core is described in section 3, *Initialization,Reset, and Starting Execution*. The effect of resetting the environment will be documented with each instantiation of the PPE 42 core. If a watchdog action halts the core, the core is force-halted. This means that normal halt processing is bypassed and the core immediately (within two cycles) enters the halted state. In the event of a WDT reset or halt action, TCR[WRC] is copied to TSR[WRS]. TCR[WRC] is cleared by all resets.
4	WIE	Watchdog Interrupt Enable 0 Watchdog interrupt is disabled 1 Watchdog interrupt is enabled	A watchdog interrupt occurs when TCR[WIE] = '1', TSR[WIS] = '1', MSR[EE] = '1' and the watchdog interrupt is the highest priority interrupt pending. However if TCR[WRC] is not '00', then a watchdog timeout with TSR[WIS] = '1' causes the action specified by TCR[WRC].
5	DIE	Decrementer Interrupt Enable 0 Decrementer interrupt is disabled 1 Decrementer interrupt is enabled	A decrementer interrupt occurs when TCR[DIE] = '1', TSR[DIS] = '1', MSR[EE] = '1' and the decrementer interrupt is the highest priority interrupt pending.
6:7	FP	Fixed Interval Timer (FIT) Period 00 FIT uses `timer[0]` 01 FIT uses `timer[1]` 10 FIT uses `timer[2]` 11 FIT uses `timer[3]`	This field selects which of 4 external timebase inputs causes TSR[FIS] to be set. The presence and frequency of any external timebase is instance specific.
8	FIE	Fixed Interval Timer (FIT) Interrupt Enable 0 FIT interrupt is disabled 1 FIT interrupt is enabled	A FIT interrupt occurs when TCR[FIE] = '1', TSR[FIS] = '1', MSR[EE] = '1' and the FIT interrupt is the highest priority interrupt pending.
9:10	DS	Decrementer (DEC) Select 00 DEC decrements every cycle 01 DEC decrements when `timer[1] active` 10 DEC decrements when `dec_timer` is active 11 DEC decrements when `timer[3] active`	This field selects whether the decrementer decrements every processor cycle (TCR[DS] = '00'), or only on cycles that the external timer[1], `dec_timer, or timer[3]` inputs are active. Note that DS bit 1 is reserved in PPE42 and implemented on PPE42X.
11:31	Reserved		

8.10.19 TSR – Timer Status Register

Register Access: SPR 336, All Fields Read/Write 1 to Clear (Normal Operation) or Direct Write (Ramming)

Table 1.57: TSR – Timer Status Register

Bits	Field	Description	Notes
0	ENW	Enable Next Watchdog 0 See notes 1 See notes	If TSR[ENW] = '0', then the watchdog event simply sets TSR[ENW] = '1'. If TSR[ENW] = '1', then the watchdog event either generates a watchdog exception or a watchdog reset action.
1	WIS	Watchdog Timer Interrupt Status 0 No new watchdog timer event has occurred 1 A watchdog timer event has occurred	If TSR[ENW] = '1', then TSR[WIS] is set on any clock cycle in which the external timebase input selected by TCR[WP] is asserted.
2:3	WRS	Watchdog Timer Reset Status 00 No watchdog timer reset has occurred 01 Last watchdog action was a soft reset 10 Last watchdog action was a hard reset 11 Last watchdog action halted the processor	These two bits are set to the contents of TCR[WRC] when a reset or halt is caused by the Watchdog Timer mechanism. These bits are unchanged if the processor is reset by any mechanism other than the Watchdog Timer mechanism. Software is responsible for clearing these bits after a reset or Watchdog timeout event.
4	DIS	Decrementer Interrupt Status 0 No new decrementer event has occurred 1 A decrementer event has occurred	TSR[DIS] is set on any cycle that changes $(DEC)_0$ from '0' to '1', including by **mtspr** instructions targeting DEC. A decrementer interrupt occurs when TCR[DIE] = '1', TSR[DIS] = '1', MSR[EE] = '1' and the decrementer interrupt is the highest priority interrupt pending.
5	FIS	Fixed Interval Timer (FIT) Interrupt Status 0 No new FIT event has occurred 1 A FIT event has occurred	This field is set on any clock cycle in which the external timebase input selected by TCR[FP] is asserted. A FIT interrupt occurs when TCR[FIE] = '1', TSR[FIS] = '1', MSR[EE] = '1' and the FIT interrupt is the highest priority interrupt pending.
6:31	Reserved		

Note:

If the processor is halted and an **mttsr Rx** instruction is rammed, then all defined fields of the TSR are updated directly from the contents of the GPR **Rx**. This allows debugging code to directly set timer interrupt status, and/or restore the TSR to an original state after debugging.

8.10.20 XCR – External Control Register

Register Access: XIR; Write-only

Table 1.58: XCR – External Control Register

Bits	Field	Description	Mode	Notes
0	Reserved			
1:3	CMD	Command 000 Clear Debug Status 001 Halt 010 Resume 011 Single-step 100 Toggle XSR[TRH] 101 Soft Reset 110 Hard Reset 111 Force Halt	WO	All external commands to the processor are consolidated into a single field of the write-only XCR. **000 – Clear Debug Status** This command clears XSR[TRAP, IAC, RDAC, WDAC] to '0'. This command is only implemented and acknowledged when XSR[HS] = '1' (the processor is halted). **001 – Halt** This command sets XSR[HCP], which requests the processor to halt after the execution of the current instruction (if any) is complete. A processor reporting XSR[HCP] = '1' may still be executing a single previously in-flight, single-stepped or rammed instruction. XSR[HS] = '1' indicates that the processor is both halted and not executing an instruction. Executing this command when XSR[HCP] is already '1' has no effect on the operation of the core. **010 – Resume** This command causes a halted processor to resume execution by clearing XSR[HCP]. This command is only implemented and acknowledged when either XSR[HCP] = '0' (the processor is already running), or XSR[HS] = '1' (the processor is truly halted). **011 – Single-step** This command causes a halted processor to fetch and execute the single instruction addressed by the IAR, unless an asynchronous exception is pending, in which case no instruction is executed and asynchronous interrupt processing takes place as controlled by the MSR. This command is only implemented and acknowledged when XSR[HS] = '1' (the processor is truly halted). **100 – Toggle XSR[TRH]** This command toggles (inverts) the value of XSR[TRH]. **101 – Soft Reset** **110 – Hard Reset** The '101' and '110' commands initiate a hard or soft reset respectively. Note that there is no provision for holding a PPE 42 core in a reset state. However, if either of these commands is issued and XSR[HCP] is also '1', then the processor will execute the reset sequence and then halt with the IAR addressing the system reset interrupt vector. Issuing the '010' (resume) command in this state will then cause the processor to resume execution. **111 – Force Halt** This command forces the core to transition immediately (within two cycles) to the halted state, without necessarily completing the current instruction, and without synchronizing with the memory subsystem. Force-halting is a method of last resort, and halts the core from any state, including hung memory access and reset states. A force-halted core is guaranteed to be able to correctly ram any instruction other than a load, store, cache management instruction or **sync**. It may be necessary to reset the system in order to single-step, ram a memory access instruction or **sync**, or synchronize after ramming or single-stepping.

Bits	Field	Description	Mode	Notes
4:31	Reserved			

8.10.21 XER – Fixed Point Exception Register

Register Access: SPR 1, Read/Write

Table 1.59: XER – Fixed Point Exception Register

Bits	Field	Description	Notes
0	SO	Summary Overflow 0 No overflow has occurred 1 Overflow has occurred	The PPE 42 core only partially implements the XER as architected by the Power ISA.
1	OV	Overflow 0 No overflow has occurred 1 Overflow has occurred	
2	CA	Carry 0 Carry has not occurred 1 Carry has occurred	
3:31	Reserved		

8.10.22 XSR – External Status Register

Register Access: XIR; PPE42: Read-only, PPE42X: Read/Write

Table 1.60: XSR – External Status Register

Bits	Field	Description	Notes
0	HS	Halted State 0 Processor not in halted state 1 Processor in halted state	This bit reads as '0' whenever the processor is in the process of executing instructions, including normal instruction execution and halted execution of single-stepped and rammed instructions. This bit reads as '1' if the processor is in the halted state. The processor is in the halted state whenever XSR[HCP] = '1' and the processor is not executing an instruction that was in-flight when XSR[HSR] became '1', and is not executing a single-stepped or rammed instruction. Whenever XSR[HS] = '1', the XSR[HC] field details the most recent condition that caused the processor to halt.
1:3	HC	Halt Condition 000 None of the below 001 XCR[CMD] written '111' 010 WDT halt 011 Unmaskable interrupt halt 100 Debug halt 101 DBCR halt 110 `halt_req` input active 111 Hardware failure	This field summarizes the last condition that caused the processor to halt, and is only valid when XSR[HS] = '1'. The processor may have halted due to the following conditions: • XCR[CMD] was written with '001', or XSR[HCP] remains '1' during single-stepping or ramming. This is the “none of the below” case. • XCR[CMD] was written with '111' to force-halt the processor. • A second watchdog timer (WDT) event occurred while TCR[WRC] = '11'. • A machine check interrupt occurred with MSR[ME] = '0', or a program, instruction storage, data storage or alignment interrupt occurred with MSR[UIE] = '0' and MSR[ME] = '0'. The cause can be further diagnosed using other bits in XSR along with the value of the IAR, which will address the associated interrupt vector. • An enabled debug event occurred. The event can be further diagnosed using XSR bits 7:8,and 12:13,. • The executing program wrote DBCR[RST] with '11' to halt the processor. • The external `halt_req` input was active. • An unrecoverable hardware failure was detected by the processor. This field is cleared during all processor resets, whenever XCR[CMD] is written with '010' (resume) or '011' (single-step), or whenever an instruction is rammed by writing the IR on a halted processor. While single-stepping and ramming, XCR[HC] remains '000' unless one of the other enumerated conditions occurs. In the case that two or more halt conditions become active on the same cycle, then the condition with the highest priority is reported, where “none of the below” is the lowest priority and a hardware failure is the highest priority condition. Once XSR[HC] takes on a non-zero value it becomes “locked”, and subsequently occurring halt conditions will not be reported until after the field has been cleared as described above. This means, for example, that if the processor is already halted when XCR[CMD] is externally written with '001', then the original halt condition can still be recovered.
4	HCP	Halt Condition Present 0 No halt condition present 1 Halt condition is present	If this field is read as '1', it indicates that a halt condition is present, and the processor is either truly halted (if XSR[HS] = '1'), or the processor will halt after the current in-flight instruction (which may be a single-stepped or rammed instruction) completes execution. Writing XCR[CMD] with '001' (halt) sets XSR[HCP]. Processor halts due to all other conditions (the external `halt_req` signal, MSR-controlled unmaskable interrupt halts, watchdog timeout halts, DBCR[RST] halts, unrecoverable hardware errors and halts due to debug events) also set XSR[HCP].
Continued next page			

XSR – External Status Register – *Continued from previous page*			
Bits	**Field**	**Description**	**Notes**
5	RIP	Ramming In progress 0 Unspecified 1 Ramming in progress	This field is set when a ramming operation is initiated by writing the IR on a halted processor, and cleared during the processing of the rammed instruction. Programmers should always use the condition XSR[HS] = '1' to determine when the rammed instruction has completed, not this field. This field is also cleared during all processor resets and forced halts.
6	SIP	Single-step In Progress 0 Unspecified 1 Single-step in progress	This bit is set when a single-step is initiated on a halted processor by writing XCR[CMD] = '011' (single-step), and cleared during processing of the single-stepped instruction (e.g. when the hardware either completes the instruction or detects an exception that has modified the state of the processor to take an interrupt instead). This field is also cleared during all processor resets and forced halts. However, programmers should always use the condition XSR[HS] = '1' to determine when the single-stepped instruction has completed, not this field.
7	TRAP	**trap** Instruction Debug Event 0 Event did not occur 1 Event occurred	This bit is set if a **trap** instruction is executed when DBCR[TRAP] = '1'. The occurrence of this event also causes the processor to halt. This bit is cleared by writing XCR[CMD] = '000' on a halted processor.
8	IAC	Instruction Address Compare Debug Event 0 Event did not occur 1 Event occurred	This bit is set if DBCR[IACE] = '1' and the current instruction address matches the contents of the DACR. The occurrence of this event also causes the processor to halt. This bit is cleared by writing XCR[CMD] = '000' on a halted processor. Note that the IAC debug halt is implemented while single-stepping. This means that if the event conditions hold, the instruction at the IAR is not fetched and not executed. The IAC debug halt is not implemented while ramming.
9:11	SIBRC	SIB Return Code	This field provides external, read-only access of the current value of the like-named field of the MSR.
12	RDAC	Read Data Address Compare Debug Event 0 Event did not occur 1 Event occurred	This bit is set if either DBCR[DACE] = '10' or DBCR[DACE] = '11', and a load-type data address matches the DACR, or if DBCR[ZACE] = '1' and a load-type data address is all zeroes. The occurrence of this event causes the processor to halt. This bit is cleared by writing XCR[CMD] = '000' on a halted processor. Load-type instructions consist of all loads, **dcbq** and **dcbt**. Note that the RDAC debug halt is implemented while single-stepping. This means that if the event conditions hold, the instruction at the IAR will be fetched but not executed. The RDAC debug halt is not implemented while ramming. Also note PPE 42X removes the restriction that DBCR[IACE] = '0',
13	WDAC	Write Data Address Compare Debug Event 0 Event did not occur 1 Event occurred	This bit is set if either DBCR[DACE] = '01' or DBCR[DACE] = '11', and a store-type data address matches the DACR, or if DBCR[ZACE] = '1' and a store-type data address is all zeroes.. The occurrence of this event causes the processor to halt. This bit is cleared by writing XCR[CMD] = '000' on a halted processor. Store-type instructions consist of all stores, **dcbf**, **dcbi** and **dcbz**. Note that the WDAC debug halt is implemented while single-stepping. This means that if the event conditions hold, the instruction at the IAR will be fetched but not executed. The WDAC debug halt is not implemented while ramming. Also note PPE 42X removes the restriction that DBCR[IACE] = '0',
Continued next page			

XSR – External Status Register – *Continued from previous page*			
Bits	**Field**	**Description**	**Notes**
14	WS	Wait State	This field provides external, read-only access to the current value of MSR[WE]. Note that unlike all other bits mirrored from the MSR and ISR, this field appears in a different location and with a different name in the XSR vs. the MSR.
15	TRH	Timers Run While Halted 0 Timers freeze when halted 1 Timers run when halted	If this field is '0', then all external time sources are disabled whenever the halt condition is present (XSR[HCP] = '1'). This means that the decrementer (DEC) does not decrement, and no new Fixed Interval Timer (FIT) or Watchdog Timer (WDT) events will become pending. If this field is '1' then all external time sources are active, and all time-based events and actions will occur as expected, even if the halt condition is present. Note in particular that WDT reset and halt actions may occur even on an otherwise “halted” core if TCR[WRC] is not '00'. XSR[TRH] is not written directly. Instead, writing XCR[CMD] with '100' toggles (inverts) the current value of the field.
16:19	SMS	State Machine State	For details please see PPE 42 core hardware design documentation.
20	LP	Low Priority	This field provides external, read-only access of the current value of the like-named field of the MSR.
21	EP	MSR[EE] Maskable Event Pending	For PPE42, this field provides external, read-only access of the current value of the like-named field of the ISR. For PPE42X, this field freezes state whenever XCR[HC] is greater than 0x1 and DBCR[HDFI]==1.
22	EE	Value of MSR[EE]	Reserved for PPE42. For PPE42X, this field provides external, read-only access to the value of MSR[EE].
23	Reserved		
24	PTR	Program Interrupt from **trap**	These fields provides external, read-only access of the current value of the like-named fields of the ISR.
25	ST	Data Interrupt Caused by a Store	
26:27	Reserved		
28	MFE	Multiple Fault Error	These fields provides external, read-only access of the current value of the like-named fields of the ISR.
29:31	MCS	Machine Check Status	

9 Instruction Set

PPE 42 implements an extended subset of the 32-bit Power ISA V2.07 specification. This section details the PPE 42 instruction set.

9.1 Instruction Set Origin and Portability

The PPE 42 core implements the large majority of the instructions defined in the Power ISA User Instruction Set Architecture (Book I) for 32-bit processors, including two instructions from the Legacy Integer Multiply-Accumulate Instructions category. PPE 42 also implements a subset of the cache management instructions defined in the Power ISA Virtual Environment Architecture (Book II), as well as a subset of instructions defined in the Power ISA Operating Architecture – Embedded (Book III-E). Finally, PPE 42 defines a number of implementation-specific instructions that are not part of any standard.

Although a subset of PPE 42 programs will execute on Power ISA platforms, strict compatibility was not a requirement driving the architecture. The primary benefits of modeling PPE 42 after the Power ISA are to take advantage of the well-developed Power ISA ecosystem, and the breadth of user experience in programming the Power ISA, in a way that provides an area-efficient core today with the opportunity to expand capability in the future while maintaining backwards compatibility.

PPE 42 is *not* Power ISA compliant, and programs optimized for PPE 42 will not be portable in binary form. However, many PPE 42 binary programs originating from source code written or compiled without use of PPE 42 specific instructions will execute correctly on 32-bit Power ISA platforms that utilize standard PowerPC application binary interfaces (ABIs). This feature allows the development of libraries that can be shared between PPE and other Power ISA platforms. In particular, PPE 42 programs that only use the subset of instructions defined by the Power ISA User Instruction Set Architecture (other than the Legacy Multiply-Accumulate instructions, and instructions referencing special-purpose registers) would be portable to other Power ISA platforms. PPE 42 programs using Power ISA cache-management instructions may also be portable with the caveat that PPE 42 allows certain behaviors not specified by the Power ISA.

PPE 42 programs will execute correctly on the PPE 42X, with the only exception being the Machine Check vector. Programs written or compiled with the use of PPE 42X specific instructions will cause program execeptions when executed on the PPE 42, allowing the added intructions to be emulated if desired.

PPE 42 programs must be modified to either:

1. add a "b -32" instruction at interrupt vector 0x020
2. move the machine check code to vector 0x020 and put "b 32" instruction at interrupt vector 0x000

In many cases the Power ISA provides that all or part of the behavior of certain instruction forms is undefined in the architecture. This manual fully defines the behavior of almost every instruction with respect to the state of the PPE 42 core, either by specifying that certain instruction forms are illegal, and/or by fully specifying register contents after instruction execution. Programmers should be aware of the impact this may have on portability as documented with each such instruction. PPE 42 also specifies synchronization behavior for certain instructions that is not part of or extends the Power ISA specification. Portable programs should not make use of these behaviors. Only the PPE 42 specific **dcbq** instruction leaves certain GPR contents in an instance-specific state after execution.

The specification origins of PPE 42 instructions are captured in the tables below. The fact that an instruction originates from the Power ISA does not always imply that it is fully compliant with the Power ISA; See each individual instruction description for details.

In the tables, the syntax “[**o**]” indicates that an instruction has an overflow-enabled form, which updates the XER[SO,OV] fields, and a non-overflow-enabled form. The syntax “[**.**]” indicates that an instruction has a

recording form, which updates CR[CR0], whereas the non-recording form does not. The instructions marked with “*” are actually assembler extended mnemonics for the PPE 42 restricted forms of more general Power ISA instructions.

Table 1.61: Power ISA User Instruction Set Architecture (Book I, Base) Instructions

add[o][.] **addc[o][.]** **adde[o][.]** **addi** **addic** **addic.** **addis** **addme[o][.]** **addze[o][.]**	**and[.]** **andc[.]** **andi.** **andis.** **b** **bc** **bcctr** **bclr**	**cmplw*** **cmplwi*** **cmpw*** **cmpwi*** **cntlzw[.]** **eqv[.]** **extsb[.]** **extsh[.]**	**lbz** **lbzu** **lbzx** **lhz** **lhzu** **lhzx** **lwz** **lwzu** **lwzx**	**mfcr** **mfspr** **mtcr0*** **mtspr** **nand[.]** **neg[o][.]** **nor[.]** **or[.]** **orc[.]** **ori** **oris**	**rlwimi[.]** **rlwinm[.]** **rlwnm[.]** **slw[.]** **srw[.]** **sraw[.]** **srawi[.]**	**stb** **stbu** **stbx** **sth** **sthu** **sthx** **stw** **stwu** **stwx**	**subf[o][.]** **subfc[o][.]** **subfe[o][.]** **subfic** **subfme[o][.]** **subfze[o][.]**	**tw*** **xor[.]** **xori** **xoris**

Table 1.62: Power ISA User Instruction Set Architecture (Book I, Legacy Integer Multiply-Accumulate) Instructions

mullhw[.] **mullhwu[.]**	PPE42X (not PPE42): **mulli** **mullw[o][.]**

Table 1.63: Power ISA User Instruction Set Architecture (Book I, 64-bit Fixed-Point Rotate) Instructions

PPE42X (not PPE42): **rldicl[.]** **rldicr[.]** **rldimi[.]**

Table 1.64: Power ISA Virtual Environment (Book II) Instructions

dcbf **dcbt** **dcbz**	**sync**

Table 1.65: Power ISA Operating Environment Architecture – Embedded (Book III-E) Instructions

dcbi **mfmsr** **mtmsr**	**rfi** **wrtee** **wrteei**

Table 1.66: PPE 42 Architecture-Specific Instructions

bnbw **bnbwi** **clrbwbc** **clrbwibc**	**cmplwbc** **cmpwbc** **cmpwibc**	**dcbq**	**lvd** **lvdu** **lvdx**	**stvd** **stvdu** **stvdx**

Table 1.67: PPE 42X Architecture-Specific Instructions

lsku **stsku**	**slvd** **srvd**

9.2 Rationale for the PPE 42 Instruction Set

The overriding goal of the PPE 42 architecture is to balance required functionality and good performance against area and complexity, both in the PPE 42 core and in compatible memory subsystems. Implementing an extended subset of the Power ISA Books I, II and III-E allows synergies of design, verification, tools and firmware development within the IBM POWER Systems organization that would not be realized if any other architecture for PPE 42 had been chosen.

The target applications of the PPE 42 core require atomic access to 64-bit control registers in a flat 32-bit address space encompassing both control registers and normal memory. However it is rare for these applications to perform computations requiring full 64-bit arithmetic (other than shift and rotate operations which are implemented by PPE 42X). Therefore PPE 42 is architected as a 32-bit machine, and the virtual doubleword load and store instructions provide the required 64-bit access. PPE 42 implements the majority of Power ISA 32-bit load and store instructions.

For design simplicity PPE 42 does not implement instructions that would require multiple execution states or “microcode”. Therefore multiple-word, string word, 32-bit multiply and divide instructions are omitted. The 16 x 16 hardware multiply is sufficient for simple scaling of sensor data, and can also be used to implement 32 x 32 multiplies with a few instructions in software. PPE 42X implements 32-bit low word multiply instructions since they are occasionally used by the compiler to index into multi-dimensional array-type data structures. The lack of multiple-word and string-word instructions is mitigated somewhat by the presence of virtual doubleword load and store instructions. Sign-extended halfword loads and the rarely used update-indexed load and store forms are also omitted, but are easily and faithfully emulated by a single additional instruction.

Also for simplicity PPE 42 only implements CR[CR0], similar to other small microcontrollers that maintain a single set of condition codes. Since only a single CR field is defined there is no need for CR logical instructions, and the rarely used **mcrxr** instruction is also omitted. Note that CR manipulation code that executes correctly on PPE 42 will also execute correctly on a Power ISA platform since no legal PPE 42 instruction can modify a CR field not defined by PPE 42.

PPE 42 implements the **sync** instruction as there is otherwise no generic mechanism to differentiate execution synchronization from a memory barrier. Since the PPE 42 core is not aggressively pipelined and executes in-order, the Power ISA **eieio** and **isync** instruction can be emulated with **sync** if required. Since PPE 42 does not implement privilege levels there is no need for the system call **sc**, which can be easily emulated with a normal subroutine call.

The initial instantiations of PPE 42 were designed with at most a small instruction buffer, therefore no I-cache management instructions are provided. The data caches in these instantiations support software management however, leading to the requirement for several D-cache management instructions. Finally, although the Power ISA architects the **dcread** instruction for querying the data cache for debugging, the **dcread** instruction requires two extra SPRs to function. PPE 42 defines the implementation-specific **dcbq** instruction to simplify the architecture.

9.2.1 PPE 42X Added Instructions

PPE 42X implements two instructions (**mulli** and **mullw**), previously omitted from the original PPE42, for more efficient compiled code. Only the two 32-bit multiplies yielding the low order word of the product are included to allow the compiler to index into data structures without unnecessary emulation using 16-bit multiply instructions.

PPE 42X also implements three of the six arithmetic rotate instructions (**rldimi, rldicr, rldicl**) from the 64-bit Power ISA for more efficient processing of bit fields within virtual doublewords as they are used to access 64-bit control registers. The remaining three (**rldic, rldcl, rldcr**) are not implemented due to lack of anticipated usage, and the behavior of these can be emulated by PPE 42X using a combination of two instructions, a 64-bit shift and rotate-then-clear immediate. Additionally, adding these MDS-form instructions would require definition of new virtual doubleword opcodes unique to PPE.

9.2.2 PPE 42 New Instructions

PPE 42 defines fourteen new instructions that are not part of the Power ISA, to support virtual doubleword operations, code-space efficient compare-and-branch, and a simplified form of data cache query. A primary requirement for PPE 42 is the ability to perform loads and stores of virtual doublewords using base plus immediate displacement addressing to any 32-bit address, regardless of address alignment. Since this instruction form requires a full 16-bit immediate displacement, PPE 42 claims four currently unused or reserved Power ISA primary opcodes for **lvd**, **lvdu**, **stvd** and **stvdu**. It is possible that these primary opcodes will be specified by the Power ISA in the future, and if so, it may not be possible to even emulate (by way of a program interrupt) PPE 42 code on future Power ISA platforms.

The PPE 42 architecture also claims the currently illegal Power ISA primary opcode 1 for a set of *fused compare and branch* instructions, implementing a special form of extended opcode. Five types of fused compare-branch operations are encoded on opcode 1:

- Register-register arithmetic compare and conditional branch
- Register-register logical compare and conditional branch
- Register-short-immediate arithmetic compare and conditional branch
- Conditional branch on a bit 0 or non-0, with the bit number specified either as an immediate or in a register
- Clearing a bit, followed by a branch on whether the resulting register value is 0 or non-0, with the bit number specified either as an immediate or in a register

Fusing compare-and-branch operations into a single instruction can reduce code space requirements, sometimes dramatically. Fusing compare-and-branch also potentially increases performance by reducing the number of instructions fetched and reducing I-cache misses, even though the fused compare-branch does not actually execute in fewer cycles than the two underlying instructions. Also note that non-destructive bit testing is not supported by the Power ISA, and clearing a bit by an index held in a register is requires multiple instructions in the Power ISA, requiring at least one other register to be destroyed.

The three remaining new instructions are defined as extended opcodes for primary opcode 31. The extended opcodes used are all currently undefined in the Power ISA, and were selected to simplify decoding. Again, if future Power ISA specifications define these extended opcodes it may not be possible to even emulate PPE 42 code on future Power ISA platforms.

Table 1.68: PPE 42 Architecture-Specific Opcodes

Mnemonic	Primary Opcode	Extended Opcode
bnbw **bnbwi** **clrbwbc** **clrbwibc** **cmplwbc** **cmpwbc** **cmpwibc**	1	Special; See each instruction
dcbq	31	406
lvd	5	N/A

Mnemonic	Primary Opcode	Extended Opcode
lvdu	9	N/A
lvdx	31	17
stvd	6	N/A
stvdu	22	N/A
stvdx	31	145

9.2.3 PPE 42X New Instructions

The PPE42X implements two instructions slightly modified from the 64-bit Power ISA, to define 64-bit virtual doubleword shift operations, where the shift amount is instead specified in a 32-bit (single word) register. Therefore, **sld** and **srd** are instead defined as **slvd** and **srvd**.

The PPE42X defines two new instructions that are not part of the Power ISA, to allow faster execution of interrupts and function calls. These instructions are similar to load and store multiple register instructions defined in the Power ISA (but not implemented by the PPE) except they include a fixed subset of General Purpose Registers and may also include a subset of Special Purpose Registers as well. A single instruction causes a **stack frame** in EABI format to be either pushed to (**stsku**) or popped from (**lsku**) a specified region of memory with the address register updated with the resultant stack pointer. These instructions are a slightly modified DS-form where the lower bit of the DS field has a special meaning as described in PPE 42X Specific Instruction Format.

Table 1.69: PPE 42X Architecture-Specific Opcodes

Mnemonic	Primary Opcode	Extended Opcode
lsku	58	3
slvd	31	59
srvd	31	571
stsku	62	3

9.3 Instruction Formats

Instructions are four bytes long, and instruction addresses are always word-aligned. It is architecturally impossible for the PPE 42 core to generate an unaligned instruction address.

Since the PPE 42 architecture largely derives from the Power ISA, readers seeking a detailed introduction to Power ISA instruction formats and instruction field usage are referred to the Power ISA specification. Briefly, instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. The remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- **Defined:** These instruction fields contain values, such as opcodes and extended opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.
- **Variable:** These fields contain operands, such as general purpose register selectors and immediate values, that can vary from instance to instance. The instruction format diagrams specify the operands in variable fields. Some variable fields only support a limited range of values.
- **Reserved:** Bits in a reserved field should be set to ‘0’. In the instruction format diagrams, reserved fields are shaded.

If any bit in a defined field does not contain the expected value, of if a variable field contains an illegal value, the instruction is not valid, and an illegal instruction exception occurs.

In keeping with the Power ISA architectural direction for embedded processors, the PPE 42 core ignores all bits in reserved fields. Ignoring reserved fields allows upward compatibility to later implementations that may define the reserved fields to specify extended semantics for instructions, as long as the base semantics of the instruction remains unchanged.

9.3.1 PPE 42 Specific Instruction Format

PPE 42 defines one new instruction format for the fused compare-branch instructions encoded on primary opcode 1. This instruction format is illustrated below.

Table 1.70: PPE 42 Specific ***FCB*** Instruction Format

0 5	6 7	8	9 10	11 15	16 20	21 30	31
OPCODE	FCBXO	PX	BIX	RA	RB	BDX	LK
OPCODE	FCBXO	PX	BIX	RA	UIX	BDX	LK
OPCODE	FCBXO	PX	BBXO	RA	RB	BDX	LK
OPCODE	FCBXO	PX	BBXO	RA	BNX	BDX	LK

Field	Description
OPCODE	The primary opcode, always 1 for PPE 42 fused compare-branch forms
FCBXO	An extended opcode for all fused compare-branch forms
PX	Polarity selection for comparison against a CR[CR0] bit
BIX	CR[CR0] field selection for word comparison forms
BBXO	A secondary extended opcode for bit comparison forms
RA, RB	GPR specifiers
UIX	A 5-bit unsigned immediate for **cmpwibc**
BNX	An immediate big-endian bit number for immediate bit forms
BDX	A 10-bit encoding of a signed, word-aligned 12-bit branch displacement
LK	Branch with link option field

9.3.2 PPE 42X Specific Instruction Format

PPE 42X defines one new instruction format DD-form, which is a slightly modified DS-form.
This format can also be represented as a DS-form with DS = DD || CX.

Table 1.71: PPE 42X Specific DD Instruction Format

0 5	6 10	11 15	16 28	29	30 31
OPCODE	RS	RA	DD	CX	XO
OPCODE	RT	RA	DD	CX	XO

Field	Description
OPCODE	The primary opcode
XO	An extended opcode for all DD forms. This is always 3 for PPE 42X.
RS, RT, RA	GPR specifiers. These must reference the same register (typically 1) in PPE 42X.
DD	Immediate field used to specify a 13-bit signed two's complement integer which is concatenated on the right with 0b000 and sign-extended to 32 bits.
CX	Context save/restore specifier. When 1, specifies 10 doubleword memory accesses.

	When 0, specifies 0,1,or 2 doubleword memory accesses dependent on the value of DD.

9.4 Alphabetical Instruction Listing

Descriptions of the PPE 42 core instructions implemented in hardware follow, in alphabetical order. Each description contains the following elements:

- Instruction names (mnemonic and full)
- Instruction syntax
- Instruction format diagram
- Pseudocode description
- Prose description
- Registers altered
- PPE 42 Restrictions
- Architecture notes identifying the associated Power ISA Architecture component and other information

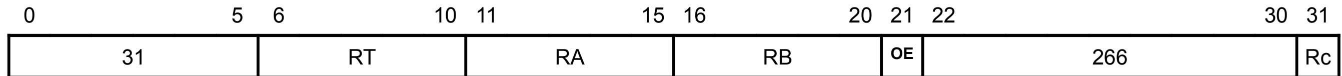
Where appropriate, instruction descriptions list invalid instruction forms and exceptions, and provide programming notes. Following the Power ISA, the PPE 42 assembler provides numerous extended mnemonics for many instruction forms, and these extended mnemonics are documented with the instruction and are also summarized in Instruction Set Mnemonics List.

Each instruction description lists the registers altered by the instruction. Some register changes are explicitly detailed in the instruction description (for example, the target register of a load instruction). In other cases registers are changed, but the details of the change are not included in the instruction description. This category frequently includes the Condition Register (CR) and the Fixed-Point Exception Register (XER). For discussions of the semantics of the CR and XER see section 2, *Programming Model.*

9.4.1 add

Add

add	RT, RA, RB	OE = '0', Rc = '0'
add.	RT, RA, RB	OE = '0', Rc = '1'
addo	RT, RA, RB	OE = '1', Rc = '0'
addo.	RT, RA, RB	OE = '1', Rc = '1'



(RT) ← (RA) + (RB)

The sum of the contents of register RA and register RB is placed into register RT.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

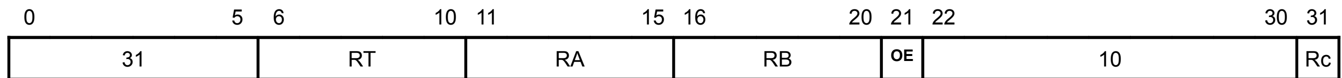
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.2 addc

Add Carrying

addc	RT, RA, RB	OE = '0', Rc = '0'
addc.	RT, RA, RB	OE = '0', Rc = '1'
addco	RT, RA, RB	OE = '1', Rc = '0'
addco.	RT, RA, RB	OE = '1', Rc = '1'



(RT) ← (RA) + (RB)
if (RA) + (RB) $>^{u}$ $2^{32} - 1$ then
XER[CA] ← 1
else
XER[CA] ← 0

The sum of the contents of register RA and register RB is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

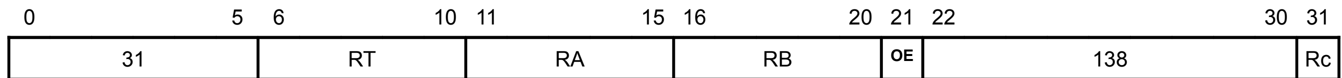
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.3 adde

Add Extended

adde	RT, RA, RB	OE = '0', Rc = '0'
adde.	RT, RA, RB	OE = '0', Rc = '1'
addeo	RT, RA, RB	OE = '1', Rc = '0'
addeo.	RT, RA, RB	OE = '1', Rc = '1'



(RT) ← (RA) + (RB) + XER[CA]
if (RA) + (RB) + XER[CA] $>^u$ $2^{32} - 1$ then
    XER[CA] ← 1
else
    XER[CA] ← 0

The sum of the contents of register RA, register RB and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

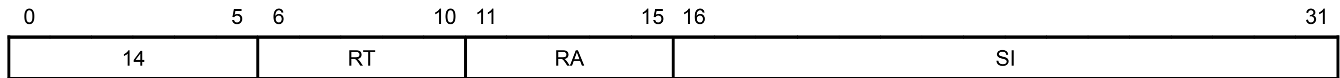
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.4 addi

Add Immediate

addi RT, RA, SI



0 5	6 10	11 15	16 31
14	RT	RA	SI

(RT) ← (RA|0) + EXTS(SI)

If the RA field is 0, the SI field, sign-extended to 32 bits, is placed into register RT.

If the RA field is nonzero, the sum of the contents of register RA and the contents of the SI field, sign-extended to 32 bits, is placed into register RT.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

The **addi** instruction can be used to load an immediate value into a register. An extended mnemonic is provided to convey the idea that no addition is being performed but merely data movement from the immediate field of the instruction to a register.

Load a 16-bit signed immediate value into register Rx.

li RT, value (equivalent to: **addi RT, 0, value**)

An extended mnemonic is also provided to subtract an immediate value from a register.

subi RT, RA, value (equivalent to **addi RT, RA, -value**)

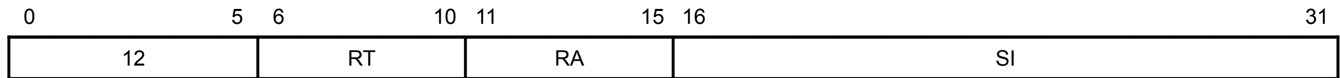
The **la** (load address) extended mnemonic is another form of **addi** with a syntax suggesting the generation of a base plus displacement (signed offset) address.

la RT, offset(RA) (equivalent to **addi RT, RA, offset**)

9.4.5 addic

Add Immediate Carrying

addic RT, RA, SI



(RT) ← (RA) + EXTS(SI)
If (RA) + EXTS(SI) $>^{u}$ $2^{32} - 1$ then
XER[CA] ← 1
else
XER[CA] ← 0

The sum of the contents of register RA and the contents of the SI field, sign-extended to 32 bits, is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

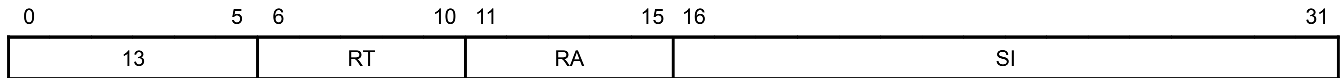
An extended mnemonic is also provided to subtract an immediate value from a register with carrying.

subic RT, RA, value (equivalent to addic RT, RA, -value)

9.4.6 addic.

Add Immediate Carrying and Record

addic. RT, RA, SI



(RT) ← (RA) + EXTS(SI)
If (RA) + EXTS(SI) $>^u 2^{32} - 1$ then
XER[CA] ← 1
else
XER[CA] ← 0

The sum of the contents of register RA and the contents of the SI field, sign-extended to 32 bits, is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

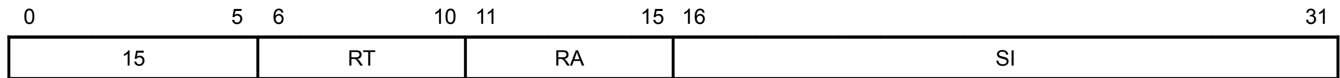
An extended mnemonic is also provided to subtract an immediate value from a register with carrying and recording.

subic. RT, RA, value (equivalent to addic. RT, RA, -value)

9.4.7 addis

Add Immediate Shifted

addis RT, RA, SI



$(RT) \leftarrow (RA|0) + (SI \mid\mid {}^{16}0)$

If the RA field is 0, the SI field is concatenated on its right with sixteen 0-bits and placed into register RT.

If the RA field is nonzero, the contents of register RA are added to the contents of the extended SI field. The sum is stored into register RT.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Programming Note

An **addis** instruction followed by an **ori** instruction stores an arbitrary 32-bit value in a GPR, as shown in the following example:

addis RT, 0, high 16 bits of value

ori RT, RT, low 16 bits of value

Extended Mnemonics

The **addis** instruction can be used to load an immediate value into a register. An extended mnemonic is provided to convey the idea that no addition is being performed but merely data movement from the immediate field of the instruction to a register.

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.

lis RT,value (equivalent to: addis RT, 0, value)

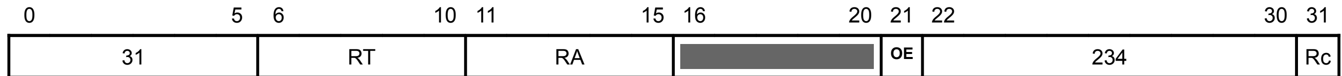
An extended mnemonic is also provided to subtract an immediate value from a register.

subis RT, RA, value (equivalent to addis RT, RA, -value)

9.4.8 addme

Add to Minus One Extended

addme	RT, RA	OE = '0', Rc = '0'
addme.	RT, RA	OE = '0', Rc = '1'
addmeo	RT, RA	OE = '1', Rc = '0'
addmeo.	RT, RA	OE = '1', Rc = '1'



(RT) ← (RA) + XER[CA] + (-1)
if (RA) + XER[CA] + (-1) $>^{u}$ $2^{32} - 1$ then
XER[CA] ← 1
else
XER[CA] ← 0

The sum of the contents of register RA, XER[CA] and -1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]LT, GT, EQ, SO if Rc contains a '1'
- XER[SO, OV] if OE contains a '1'

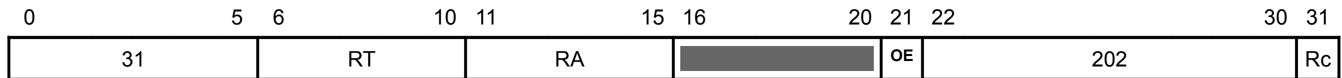
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.9 addze

Add to Zero Extended

addze	RT, RA	OE = '0', Rc = '0'
addze.	RT, RA	OE = '0', Rc = '1'
addzeo	RT, RA	OE = '1', Rc = '0'
addzeo.	RT, RA	OE = '1', Rc = '1'



(RT) ← (RA) + XER[CA]
if (RA) + XER[CA] $>^u$ $2^{32} - 1$ then
    XER[CA] ← 1
else
    XER[CA] ← 0

The sum of the contents of register RA and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

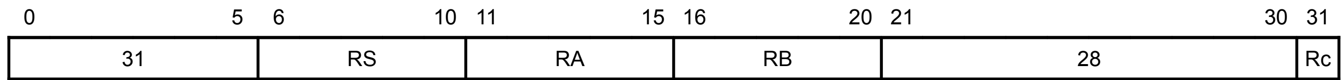
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.10 and

AND

and	RA, RS, RB	Rc = '0'
and.	RA, RS, RB	Rc = '1'



(RA) ← (RS) ∧ (RB)

The contents of register RS are ANDed with the contents of register RB. The result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

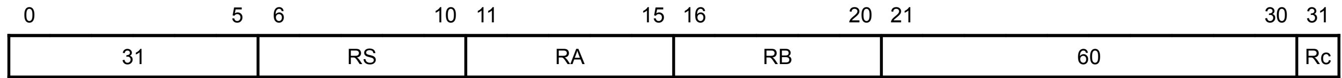
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.11 andc

AND with Complement

andc	RA, RS, RB	Rc = '0'
andc.	RA, RS, RB	Rc = '1'



0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31
31	RS	RA	RB	60	Rc

$(RA) \leftarrow (RS) \wedge \neg(RB)$

The contents of register RS are ANDed with the complement of the contents of register RB. The result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

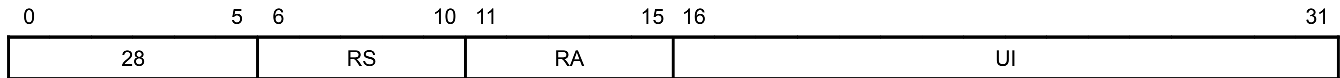
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.12 andi.

AND Immediate

andi. RA, RS, UI



0 5	6 10	11 15	16 31
28	RS	RA	UI

$(RA) \leftarrow (RS) \land (^{16}0 \,||\, UI)$

The UI field is extended to 32 bits by concatenating 16 0-bits on its left. The contents of register RS are ANDed with the extended UI field. The result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]$_{LT, GT, EQ, SO}$

Programming Note

The **andi.** Instruction can test whether any of the 16 least-significant bits in a GPR are 1-bits.

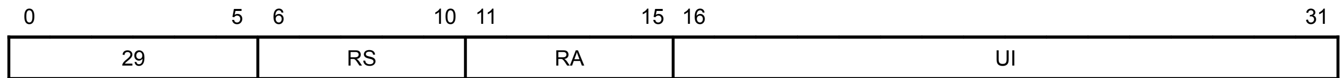
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.13 andis.

AND Immediate Shifted

andis. RA, RS, UI



0 5	6 10	11 15	16 31
29	RS	RA	UI

$(RA) \leftarrow (RS) \land (UI \,||\, {}^{16}0)$

The UI field is extended to 32 bits by concatenating 16 0-bits on its right. The contents of register RS are ANDed with the extended UI field. The result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$

Programming Note

The **andis.** Instruction can test whether any of the 16 most-significant bits in a GPR are 1-bits.

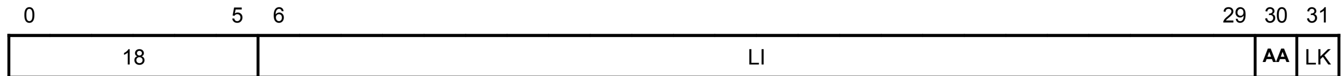
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.14 b

Branch

b	target	AA = '0', LK = '0'
bl	target	AA = '0', LK = '1'
ba	target	AA = '1', LK = '0'
bla	target	AA = '1', LK = '1'



If AA = '1' then
    NIA ← EXTS(LI || ^{2}0)
else
    NIA ← CIA + EXTS(LI || ^{2}0)

if LK = 1 then
    (LR) ← CIA + 4

PC ← NIA

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the LI field and sign-extending the result to 32 bits.

If the AA field contains a '1' then the base address is 0. If the AA field is a '0' then the base address is the address of the branch instruction, which is also the current instruction address (CIA).

Program flow is transferred to the NIA.

If the LK field contains a '1' then (CIA + 4) is placed into the LR.

Registers Altered

- LR if LK contains a '1'

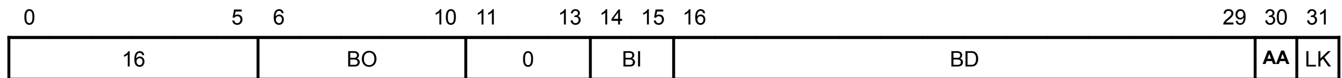
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.15 bc

Branch Conditional

bc	BO, BI, target	AA = '0', LK = '0'
bcl	BO, BI, target	AA = '0', LK = '1'
bca	BO, BI, target	AA = '1', LK = '0'
bcla	BO, BI, target	AA = '1', LK = '1'



0–5	6–10	11–13	14–15	16–29	30	31
16	BO	0	BI	BD	AA	LK

If BO_2 = 0 then
    CTR ← CTR – 1

if (BO_2 = 1 ∨ ((CTR = 0) = BO_3)) ∧ (BO_0 = 1 ∨ ($CR[CR0]_{BI}$ = BO_1)) then
    If AA = '1' then
        NIA ← EXTS(BD || ^{2}0)
    else
        NIA ← CIA + EXTS(BD || ^{2}0)
else
    NIA ← CIA + 4

if LK = 1 then
    (LR) ← CIA + 4

PC ← NIA

If bit 2 of the BO field contains a '0' then the Count Register (CTR) decrements.

The BO field controls options that determine when program flow is transferred to the NIA. The BI field specifies a bit of CR[CR0] to be used as the condition of the branch.

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BD field and sign-extending the result to 32 bits.

If the AA field contains a '1' then the base address is 0. If the AA field is a '0' then the base address is the address of the branch instruction, which is also the current instruction address (CIA).

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Registers Altered

- CTR if BO_2 contains a '0'
- LR if LK contains a '1'

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture. Since PPE 42 only defines CR[CR0], PPE 42 requires that bits 11:13 of the instruction be explicitly set as '000', otherwise an illegal instruction exception occurs.

The Power ISA provides for branch prediction hints to be encoded in the BO field. These hints are ignored by the PPE 42 core.

Extended Mnemonics

The BO and BI fields of conditional branch instructions control whether the branch is taken, and specify other side effects. It is rare for programmers to code conditional branches by explicitly specifying BO and BI. Instead programmers typically use one of the numerous extended mnemonic forms for conditional branches.

Providing an extended mnemonic for every possible combination of the BO and BI fields would be neither useful nor practical. The most generally useful extended mnemonics for conditional branches are listed in the tables below. For complete information on extended mnemonics for branches please refer to the Power ISA specification.

Because PPE 42 only implements CR[CR0], it is never necessary to specify a CR field when coding conditional branch instructions for PPE 42 using the extended mnemonics listed below. Instead one can simply program for example

blt target

for a conditional branch if less than to the target.

Any conditional branch extended mnemonic can be extended with optional “l” and “a” suffixes to specify with LR update and absolute addressing respectively. If an LR update + absolute form branch is required then the suffix “la” must be used.

Table 1.72: Conditional Branch Extended Mnemonics Incorporating Conditions

Branch Semantics	***bc***	***bcl***	***bca***	***bcla***
Branch if less than	blt	bltl	blta	bltla
Branch if less than or equal	ble	blel	blea	blela
Branch if greater than	bgt	bgtl	bgta	bgtla
Branch if greater than or equal	bge	bgel	bgea	bgela
Branch if equal	beq	beql	beqa	beqla
Branch if not equal	bne	bnel	bnea	bnela
Branch is summary overflow	bso	bsol	bsoa	bsola
Branch if not summary overflow	bns	bnsl	bnsa	bnsla

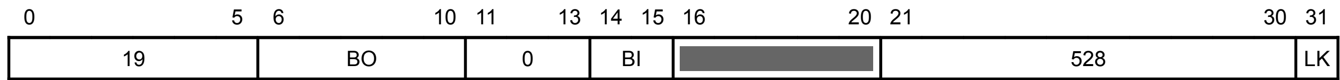
Table 1.73: Conditional Branch Extended Mnemonics With CTR Decrement

Branch Semantics	***bc***	***bcl***	***bca***	***bcla***
Decrement CTR, branch if CTR nonzero	bdnz	bdnzl	bdnza	bdnzla
Decrement CTR, branch if CTR zero	bdz	bdzl	bdza	bdzla

9.4.16 bcctr

Branch Conditional to Count Register

bcctr	BO, BI	LK = '0'
bcctrl	BO, BI	LK = '1'



0 5	6 10	11 13	14 15	16 20	21 30	31
19	BO	0	BI		528	LK

if (BO_2 = 1 ∨ ((CTR = 0) = BO_3)) ∧ (BO_0 = 1 ∨ ($CR[CR0]_{BI}$ = BO_1)) then
NIA ← $CTR_{0:29}$ || $^{2}0$
else
NIA ← CIA + 4

if LK = 1 then
(LR) ← CIA + 4

PC ← NIA

The BO field controls options that determine when program flow is transferred to the NIA. The BI field specifies a bit of CR[CR0] to be used as the condition of the branch.

The NIA is the effective address of the branch. The NIA is formed by concatenating the 30 most significant bits of CTR with two 0-bits on the right.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

If BO_2 = 0 then an illegal instruction exception is generated. Also see the Architecture Note below.

Registers Altered

- LR if LK contains a '1'

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture. Since PPE 42 only defines CR[CR0], PPE 42 requires that bits 11:13 of the instruction be explicitly set as '000', otherwise an illegal instruction exception occurs.

The Power ISA specifies that if bit 2 of the BO field contains a '0' then the instruction form is invalid. The PPE 42 core generates an illegal instruction exception in this case.

The Power ISA provides for branch prediction hints to be encoded in the BO field, and in bits 19:20 of the instruction. These hints are ignored by the PPE 42 core.

Extended Mnemonics

The BO and BI fields field of **bcctr** and **bcctrl** control whether the branch is taken, and specify other side effects. It is rare for programmers to code **bcctr** and **bcctrl** by explicitly specifying BO and BI. Instead programmers typically use one of the numerous extended mnemonics.

Providing an extended mnemonic for every possible combination of the BO and BI fields would be neither useful nor practical. The most generally useful extended mnemonics for **bcctr** and **bcctrl** are listed in the tables below. For complete information on extended mnemonics for branches please refer to the Power ISA specification.

Because PPE 42 only implements CR[CR0], it is never necessary to specify a CR field when coding **bcctr** and **bcctrl** instructions for PPE 42 using the extended mnemonics listed below. Instead one can simply program for example

bctr

for an unconditional branch to the CTR, and

bltctr

for a conditional branch if less than to the CTR.

Any **bcctr** extended mnemonic can be extended with an optional “l” suffix to specify with LR update.

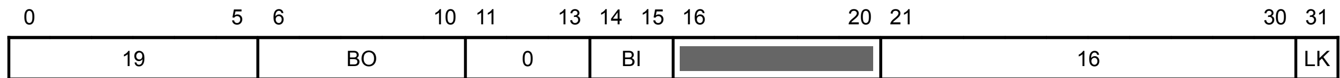
Table 1.74: **bcctr** and **bcctrl** Extended Mnemonics

Branch Semantics (Branch to CTR)	***bctr***	***bctrl***
Unconditional branch	bctr	bctrl
Branch if less than	bltctr	bltctrl
Branch if less than or equal	blectr	blectrl
Branch if greater than	bgtctr	bgtctrl
Branch if greater than or equal	bgectr	bgectrl
Branch if equal	beqctr	beqctrl
Branch if not equal	bnectr	bnectrl
Branch is summary overflow	bsoctr	bsoctrl
Branch if not summary overflow	bnsctr	bnsctrl

9.4.17 bclr

Branch Conditional to Link Register

bclr	BO, BI	LK = '0'
bclrl	BO, BI	LK = '1'



If $BO_2 = 0$ then
  $CTR \leftarrow CTR - 1$

if $(BO_2 = 1 \lor ((CTR = 0) = BO_3)) \land (BO_0 = 1 \lor (CR[CR0]_{BI} = BO_1))$ then
  $NIA \leftarrow (LR)_{0:29} \,||\, {}^{2}0$
else
  $NIA \leftarrow CIA + 4$

if LK = 1 then
  $(LR) \leftarrow CIA + 4$

$PC \leftarrow NIA$

If bit 2 of the BO field contains a '0' then the Count Register (CTR) decrements.

The BO field controls options that determine when program flow is transferred to the NIA. The BI field specifies a bit of CR[CR0] to be used as the condition of the branch.

The NIA is the effective address of the branch. The NIA is formed by concatenating the 30 most significant bits of LR with two 0-bits on the right.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Registers Altered

- CTR if BO_2 contains a '0'
- LR if LK contains a '1'

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture. Since PPE 42 only defines CR[CR0], PPE 42 requires that bits 11:13 of the instruction be explicitly set as '000', otherwise an illegal instruction exception occurs.

The Power ISA provides for branch prediction hints to be encoded in the BO field, and in bits 19:20 of the instruction. These hints are ignored by the PPE 42 core.

Extended Mnemonics

The BO and BI fields field of **bclr** and **bclrl** control whether the branch is taken, and specify other side effects. It is rare for programmers to code **bclr** and **bclrl** by explicitly specifying BO and BI. Instead programmers typically use one of the numerous extended mnemonics.

Providing an extended mnemonic for every possible combination of the BO and BI fields would be neither useful nor practical. The most generally useful extended mnemonics for **bclr** and **bclrl** are listed in the tables below. For complete information on extended mnemonics for branches please refer to the Power ISA specification.

Because PPE 42 only implements CR[CR0], it is never necessary to specify a CR field when coding **bclr** and **bclr** instructions for PPE 42 using the extended mnemonics listed below. Instead one can simply program for example

blr

for an unconditional branch to the LR, and

bltlr

for a conditional branch if less than to the LR.

Any **bclr** extended mnemonic can be extended with an optional “l” suffix to specify with LR update.

Table 1.75: **bclr and bclrl** Extended Mnemonics Involving Conditions

Branch Semantics (Branch to LR)	***bclr***	***bclrl***
Unconditional branch	blr	blrl
Branch if less than	bltlr	bltlrl
Branch if less than or equal	blelr	blelrl
Branch if greater than	bgtlr	bgtlrl
Branch if greater than or equal	bgelr	bgelrl
Branch if equal	beqlr	beqlrl
Branch if not equal	bnelr	bnelrl
Branch is summary overflow	bsolr	bsolrl
Branch if not summary overflow	bnslr	bnslrl

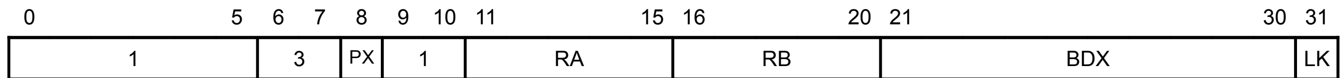
Table 1.76: **bclr** and **bclrl** Extended Mnemonics With CTR Decrement

Branch Semantics (Branch to LR)	***bclr***	***bclrl***
Decrement CTR, branch if CTR nonzero	bdnzlr	bdnzlrl
Decrement CTR, branch if CTR zero	bdzlr	bdzlrl

9.4.18 bnbw

Branch on Not Bit Word

bnbw	PX, RA, RB, target	LK = '0'
bnbwl	PX, RA, RB, target	LK = '1'



0–5	6–7	8	9–10	11–15	16–20	21–30	31
1	3	PX	1	RA	RB	BDX	LK

m ← MASK($(RB)_{27:31}$, $(RB)_{27:31}$)
r ← ((RA) ∧ m)

CR[$CR0_0$] ← r_0
CR[$CR0_1$] ← ¬r_0 ∧ ¬(r = 0)
CR[$CR0_2$] ← (r = 0)
CR[$CR0_3$] ← XER[SO]

if PX = CR[$CR0_2$] then
    NIA ← CIA + EXTS(BDX || ^{2}0)
else
    NIA ← CIA + 4

if LK = 1 then
    (LR) ← CIA + 4

A mask is generated having a single 1-bit at the bit position specified by bits 27:31 of RB. The contents of register RA are ANDed with the generated mask. CR[CR0] is updated to reflect the results of the AND operation and the value of XER[SO] is placed into CR[$CR0_3$]. The result of the AND operation is discarded.

The branch is taken if PX = CR[$CR0_2$]. Since (CR[$CR0_2$] = '1') = ($(RA)_{(RB[27:31])}$ = 0), the PX field specifies the inverted polarity of the bit being tested.

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- $CR[CR0]_{LT, GT, EQ, SO}$
- LR if LK contains a '1'

Architecture Note

This instruction is specific to the PPE 42 architecture.

CR[CR0] is updated as if the instruction being executed were

rlwinm. Rx, RA, 0, MB, ME

where MB = ME = $(RB)_{27:31,}$ without the update of Rx.

Programming Note

Programmers should take care to distinguish **bnbwi[l]** from **bnbw[l],** including the extended mnemonic forms. In general, the assembler can not distinguish an immediate bit number specified for **bnbwi[l]** from a GPR number specified for **bnbw[l]**, and will not be able to detect a programming error where the two mnemonics are confused.

Extended Mnemonics

Extended mnemonics are provided to allow specification of the bit value to test for embedded in the mnemonic. For example:

bb0w RA, RB, target (equivalent to **bnbw 1, RA, RB, target**)

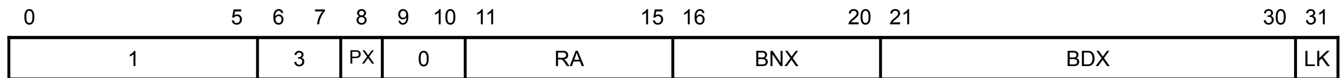
Table 1.77: Extended Mnemonics for **bnbw[l]**

Branch Semantics	***bnbw***	***bnbwl***
Branch on bit zero word	**bb0w**	**bb0wl**
Branch on bit one word	**bb1w**	**bb1wl**

9.4.19 bnbwi

Branch on Not Bit Word Immediate

bnbwi	PX, RA, BNX, target	LK = '0'
bnbwil	PX, RA, BNX, target	LK = '1'



$m \leftarrow \text{MASK}(\text{BNX}, \text{BNX})$
$r \leftarrow ((\text{RA}) \land m)$

$\text{CR}[\text{CR0}_0] \leftarrow r_0$
$\text{CR}[\text{CR0}_1] \leftarrow \neg r_0 \land \neg(r = 0)$
$\text{CR}[\text{CR0}_2] \leftarrow (r = 0)$
$\text{CR}[\text{CR0}_3] \leftarrow \text{XER}[\text{SO}]$

if PX = $\text{CR}[\text{CR0}_2]$ then
    $\text{NIA} \leftarrow \text{CIA} + \text{EXTS}(\text{BDX} \,||\, {}^2 0)$
else
    $\text{NIA} \leftarrow \text{CIA} + 4$

if LK = 1 then
    $(\text{LR}) \leftarrow \text{CIA} + 4$

A mask is generated having a single 1-bit at the bit position specified in the BNX field. The contents of register RA are ANDed with the generated mask. CR[CR0] is updated to reflect the results of the AND operation and the value of XER[SO] is placed into $\text{CR}[\text{CR0}_3]$. The result of the AND operation is discarded.

The branch is taken if PX = $\text{CR}[\text{CR0}_2]$. Since $(\text{CR}[\text{CR0}_2] = \text{'1'}) = ((\text{RA})_{\text{BNX}} = 0)$, the PX field specifies the inverted polarity of the bit being tested.

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The register RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- $\text{CR}[\text{CR0}]_{\text{LT, GT, EQ, SO}}$
- LR if LK contains a '1'

Architecture Note

This instruction is specific to the PPE 42 architecture.

CR[CR0] is updated as if the instruction being executed were

rlwinm. Rx, Ra, 0, BNX, BNX

without the update of Rx.

Programmers should take care to distinguish **bnbwi[l]** from **bnbw[l],** including the extended mnemonic forms. In general, the assembler can not distinguish an immediate bit number specified for **bnbwi[l]** from a GPR number specified for **bnbw[l]**, and will not be able to detect a programming error where the two mnemonics are confused.

Extended Mnemonics

Extended mnemonics are provided to allow specification of the bit value to test for embedded in the mnemonic. For example:

bb0wi RA, n, target (equivalent to **bnbwi 1, RA, n, target**)

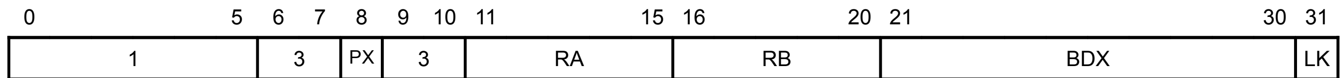
Table 1.78: Extended Mnemonics for **bnbwi[l]**

Branch Semantics	***bnbwi***	***bnbwil***
Branch on bit zero word immediate	**bb0wi**	**bb0wil**
Branch on bit one word immediate	**bb1wi**	**bb1wil**

9.4.20 clrbwbc

Clear Bit Word and Branch Conditional

clrbwbc	PX, RA, RB, target	LK = '0'
clrbwbcl	PX, RA, RB, target	LK = '1'



0–5	6–7	8	9–10	11–15	16–20	21–30	31
1	3	PX	3	RA	RB	BDX	LK

m ← MASK($(RB)_{27:31}$, $(RB)_{27:31}$)
r ← ((RA) ∧ ¬m)
(RA) ← r

CR[$CR0_0$] ← r_0
CR[$CR0_1$] ← ¬r_0 ∧ ¬(r = 0)
CR[$CR0_2$] ← (r = 0)
CR[$CR0_3$] ← XER[SO]

if PX = CR[$CR0_2$] then
    NIA ← CIA + EXTS(BDX || $^{2}0$)
else
    NIA ← CIA + 4

if LK = 1 then
    (LR) ← CIA + 4

A mask is generated having a single 1-bit at the bit position specified by bits 27:31 of RB. The contents of register RA are ANDed with the complement of the generated mask. CR[CR0] is updated to reflect the results of the AND operation and the value of XER[SO] is placed into CR[$CR0_3$]. The result of the AND operation is placed into RA.

The branch is taken if PX = CR[$CR0_2$], that is, based on whether the value placed into RA is either 0 (PX = '1') or non-0 (PX = '0').

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]LT, GT, EQ, SO
- LR if LK contains a '1'

Architecture Note

This instruction is specific to the PPE 42 architecture.

CR[CR0] is updated as if the instruction being executed were

rlwinm. RA, RB, 0, MB, ME

where MB = $((RB)_{27:31} + 1) \% 31$, and ME = $((RB)_{27:31} - 1) \% 31$.

Programming Note

Programmers should take care to distinguish **clrbwibc[l]** from **clrbwbc[l],** including the extended mnemonic forms. In general, the assembler can not distinguish an immediate bit number specified for **clrbwibc[l]** from a GPR number specified for **clrbwbc[l]**, and will not be able to detect a programming error where the two mnemonics are confused.

The **clrbwbc** instruction supports time and code-space efficient iteration over sparse bit vectors, where the bits to process are identified using **cntlzw**. For example, assume R28 contains a bit vector to process:

```
       bwz    R28, done        # No bits to process
loop:
       cntlzw R3, R28
       … Process based on bit number held in R3
       clrbwbnz R28, R3, loop # Clear bit; Iterate until no more bits
done:
```

In general, the most efficient way to simply clear a bit whose index is held in a register is to use the **clrbw.** extended mnemonic.

Extended Mnemonics

Extended mnemonics are provided consistent with the way that other zero and non-zero branches are named. For example:

clrbwbz RA, RB, target (equivalent to **clrbwbc 1, RA, RB, target**)

Table 1.79: Extended Mnemonics for **clrbwbc[l]**

Branch Semantics	***clrbwbc***	***clrbwbcl***
Branch if final result is 0	**clrbwbz**	**clrbwbzl**
Branch if final result is not zero	**clrbwbnz**	**clrbwbnzl**

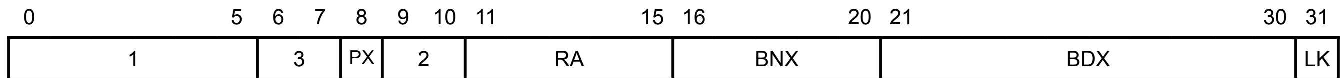
The most efficient way to simply clear a bit whose index is held in a register is to use the **clrbwbc** instruction with a redundant branch to the next sequential instruction. The **clrbw.** extended mnemonic is provided for this purpose. The mnemonic includes the '.' (dot) prefix since the true nature of the operation (a compare that sets CR[CR0] followed by a branch) is hidden.

clrbw. RA, RB (equivalent to **clrbwbz RA, RB, $ + 4**)

9.4.21 clrbwibc

Clear Bit Word Immediate and Branch Conditional

clrbwibc	PX, RA, BNX, target	LK = '0'
clrbwibcl	PX, RA, BNX, target	LK = '1'



0–5	6–7	8	9–10	11–15	16–20	21–30	31
1	3	PX	2	RA	BNX	BDX	LK

m ← MASK(BNX, BNX)
r ← ((RA) ∧ ¬m)
(RA) ← r

$CR[CR0_0] \leftarrow r_0$
$CR[CR0_1] \leftarrow \neg r_0 \wedge \neg(r = 0)$
$CR[CR0_2] \leftarrow (r = 0)$
$CR[CR0_3] \leftarrow XER[SO]$

if PX = $CR[CR0_2]$ then
    NIA ← CIA + EXTS(BDX || $^{2}0$)
else
    NIA ← CIA + 4

if LK = 1 then
    (LR) ← CIA + 4

A mask is generated having a single 1-bit at the bit position specified in the BNX field. The contents of register RA are ANDed with the complement of the generated mask. CR[CR0] is updated to reflect the results of the AND operation and the value of XER[SO] is placed into $CR[CR0_3]$. The result of the AND operation is placed into RA.

The branch is taken if PX = $CR[CR0_2]$, that is, based on whether the value placed into RA is either 0 (PX = '1') or non-0 (PX = '0').

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The register RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$
- LR if LK contains a '1'

Architecture Note

This instruction is specific to the PPE 42 architecture.

CR[CR0] is updated as if the instruction being executed were

rlwinm. RA, RA, 0, (BNX + 1) % 32, (BNX – 1) % 32.

Programming Note

Programmers should take care to distinguish **clrbwibc[l]** from **clrbwbc[l],** including the extended mnemonic forms. In general, the assembler can not distinguish an immediate bit number specified for **clrbwibc[l]** from a GPR number specified for **clrbwbc[l]**, and will not be able to detect a programming error where the two mnemonics are confused.

Unless a branch on the result is required, clearing a bit from a register based on an immediate bit number is most efficiently encoded using the **clrbwi** extended mnemonic.

Extended Mnemonics

Extended mnemonics are provided consistent with the way that other zero and non-zero branches are named. For example:

clrbwibz RA, n, target (equivalent to **clrbwibc 1, RA, n, target**)

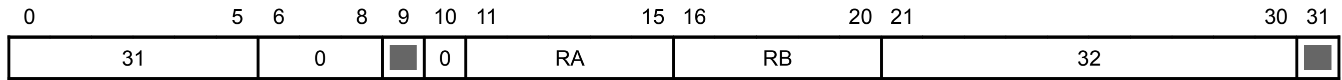
Table 1.80: Extended Mnemonics for **clrbwibc[l]**

Branch Semantics	***clrbwibc***	***clrbwibcl***
Branch if final result is 0	**clrbwibz**	**clrbwibzl**
Branch if final result is not zero	**clrbwibnz**	**clrbwibnzl**

9.4.22 cmplw

Compare Logical Word

cmplw RA, RB



0–5	6–8	9	10	11–15	16–20	21–30	31
31	0	■	0	RA	RB	32	■

$CR[CR0] \leftarrow 0$
if (RA) $<^u$ (RB) then $CR[CR0_0] \leftarrow 1$
if (RA) $>^u$ (RB) then $CR[CR0_1] \leftarrow 1$
if (RA) = (RB) then $CR[CR0_2] \leftarrow 1$
$CR[CR0_3] \leftarrow XER[SO]$

The contents of register RA are compared with the contents of register RB using a 32-bit unsigned compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into $CR[CR0_3]$.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below.

Registers Altered

- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$

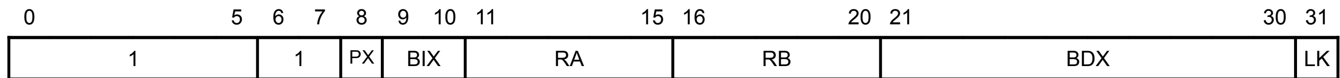
Architecture Note

cmplw is actually an assembler extended mnemonic for the more general **cmpl** instruction of the Power ISA User Instruction Set Architecture. PPE 42 only supports word comparisons (instruction bit 10 = 0) with update of CR[CR0] (instruction bits 6:8 = 0). Any other form of the **cmpl** instruction will cause an illegal instruction exception.

9.4.23 cmplwbc

Compare Logical Word and Branch Conditional

cmplwbc	PX, BIX, RA, RB, target	LK = '0'
cmplwbcl	PX, BIX, RA, RB, target	LK = '1'



0 ... 5	6 ... 7	8	9 ... 10	11 ... 15	16 ... 20	21 ... 30	31
1	1	PX	BIX	RA	RB	BDX	LK

CR[CR0] ← 0
if (RA) $<^u$ (RB) then CR[$CR0_0$] ← 1
if (RA) $>^u$ (RB) then CR[$CR0_1$] ← 1
if (RA) = (RB) then CR[$CR0_2$] ← 1
CR[$CR0_3$] ← XER[SO]

if ((BIX ≠ 3) ∧ (PX = CR[$CR0_{BIX}$])) ∨ ((BIX = 3) ∧ (PX = CR[$CR0_2$])) then
    NIA ← CIA + EXTS(BDX || ^{2}0)
else
    NIA ← CIA + 4

If BIX = 3 then

    (RA) ← ¬(RB) + (RA) + 1

if LK = 1 then
    (LR) ← CIA + 4

The contents of register RA are compared with the contents of register RB using a 32-bit unsigned compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into CR[$CR0_3$].

The BIX field specifies a bit of CR[CR0] to be used as the condition of the branch, and the PX field specifies the polarity of the CR[CR0] bit used to determine if the branch is taken or untaken.

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If BIX = 3 then the 32-bit value computed to generate CR[$CR0_{0:2}$] is placed into RA, regardless of whether the branch is taken or not taken.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- $CR[CR0]_{LT, GT, EQ, SO}$
- LR if LK contains a '1'
- RA if BIX = 3

Architecture Note

This instruction is specific to the PPE 42 architecture.

This instruction modifies CR[CR0] identically to the Power ISA **cmplw** instruction, including the update of $CR[CR0_3]$ with XER[SO]. This instruction does not modify XER[SO] or XER[OV] however; only the "o" forms of arithmetic instructions modify XER[SO, OV]. Therefore the BIX = 3 form is used to implement an equal test with GPR writeback rather than a test for XER[SO].

Note that the subtraction performed as the underlying comparison is RA – RB, not RB – RA as per the PPE 42 **subf*** instructions.

Programming Note

The **cmplwbc** instruction can be used both as a side-effect free compare-and-branch (BIX < 3), as well as to compute and store a difference, and branch on the original equality of RA and RB (BIX = 3). The latter form allows **cmplwbc** to be used to control iteration terminating with RA = RB, that is, with a final difference of 0.

Note however that the BIX = 2 and BIX = 3 forms of **cmplwbc** (for branches based on equality) behave identically to **cmpwbc** with BIX = 2 or BIX = 3 respectively, unless the intention is to use the CR after the initial compare-and-branch, and a CR recording a logical comparison is required.

Extended Mnemonics

The PX and BIX fields of the **cmplwbc[l]** instruction control whether the branch is taken, and whether the comparison result is stored back into RB. It not necessary for programmers to code **cmplwbc** by explicitly specifying PX and BIX however. Instead, programmers typically use one of the numerous extended mnemonic forms.

The first set of extended mnemonics support using **cmplwbc** as a side-effect-free compare and branch instruction for logical comparisons.

For example,

cmplwble RA, RB, target (equivalent to **cmplwbc 0, 1, RA, RB, target**)

Table 1.81: **cmplwbc[l]** Extended Mnemonics for Side-Effect-Free Compare-and-Branch

Branch Semantics	***cmplwbc***	***cmplwbcl***
Branch if less than	**cmplwblt**	**cmplwbltl**
Branch if less than or equal	**cmplwble**	**cmplwblel**
Branch if greater than	**cmplwbgt**	**cmplwbgtl**
Branch if greater than or equal	**cmplwbge**	**cmplwbgel**
Branch if equal	**cmplwbeq**	**cmplwbeql**
Branch if not equal	**cmplwbne**	**cmplwbnel**

The second set of extended mnemonics are used for the forms of **cmplwbc** that provide a 0/non-0 test and also update RA with the difference of RA and RB. For example:

sublwbz RA, RB, target (equivalent to **cmplwbc 1, 3, RA, RB, target**)

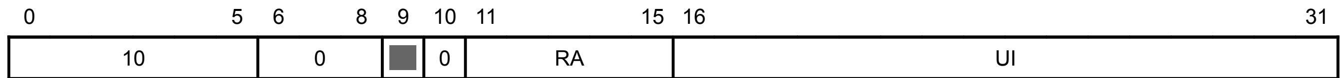
Table 1.82: **cmplwbc[l]** Extended Mnemonics for Compare-and-Branch with Update

Branch Semantics	***cmplwbc***	***cmplwbcl***
Subtract and branch if zero	**sublwbz**	**sublwbzl**
Subtract and branch if not zero	**sublwbnz**	**sublwbnzl**

9.4.24 cmplwi

Compare Logical Word Immediate

cmplwi RA, UI



CR[CR0] $\leftarrow$ 0
if (RA) $<^{u}$ $^{16}0$ || UI then CR[$CR0_0$] $\leftarrow$ 1
if (RA) $>^{u}$ $^{16}0$ || UI then CR[$CR0_1$] $\leftarrow$ 1
if (RA) = $^{16}0$ || UI then CR[$CR0_2$] $\leftarrow$ 1
CR[$CR0_3$] $\leftarrow$ XER[SO]

The UI field is zero extended on the left to 32 bits. The contents of register RA are compared with the extended UI field using a 32-bit unsigned compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into CR[$CR0_3$].

Restrictions

The register RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below.

Registers Altered

- CR[CR0]$_{LT,\ GT,\ EQ,\ SO}$

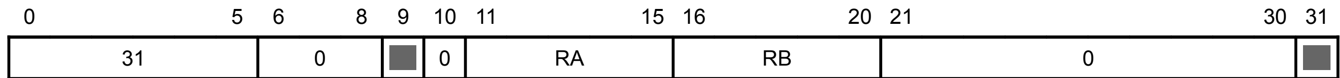
Architecture Note

cmplwi is actually an assembler extended mnemonic for the more general **cmpli** instruction of the Power ISA User Instruction Set Architecture. PPE 42 only supports word comparisons (instruction bit 10 = 0) with update of CR[CR0] (instruction bits 6:8 = 0). Any other form of the **cmpli** instruction will cause an illegal instruction exception.

9.4.25 cmpw

Compare Word

cmpw RA, RB



CR[CR0] ← 0
if (RA) < (RB) then CR[$CR0_0$] ← 1
if (RA) > (RB) then CR[$CR0_1$] ← 1
if (RA) = (RB) then CR[$CR0_2$] ← 1
CR[$CR0_3$] ← XER[SO]

The contents of register RA are compared with the contents of register RB using a 32-bit signed compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into CR[$CR0_3$].

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below.

Registers Altered

- CR[CR0]$_{LT, GT, EQ, SO}$

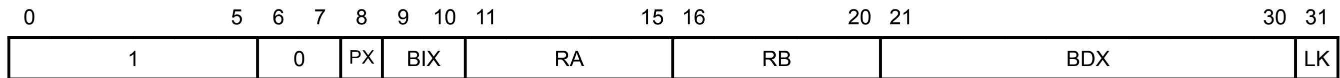
Architecture Note

cmpw is actually an assembler extended mnemonic for the more general **cmp** instruction of the Power ISA User Instruction Set Architecture. PPE 42 only supports word comparisons (instruction bit 10 = 0) with update of CR[CR0] (instruction bits 6:8 = 0). Any other form of the **cmp** instruction will cause an illegal instruction exception.

9.4.26 cmpwbc

Compare Word and Branch Conditional

cmpwbc	PX, BIX, RA, RB, target	LK = '0'
cmpwbcl	PX, BIX, RA, RB, target	LK = '1'



CR[CR0] ← 0
if (RA) < (RB) then CR[$CR0_0$] ← 1
if (RA) > (RB) then CR[$CR0_1$] ← 1
if (RA) = (RB) then CR[$CR0_2$] ← 1
CR[$CR0_3$] ← XER[SO]

if ((BIX ≠ 3) ∧ (PX = CR[$CR0_{BIX}$])) ∨ ((BIX = 3) ∧ (PX = CR[$CR0_2$])) then
    NIA ← CIA + EXTS(BDX || ^{2}0)
else
    NIA ← CIA + 4

If BIX = 3 then

    (RA) ← ¬(RB) + (RA) + 1

if LK = 1 then
    (LR) ← CIA + 4

The contents of register RA are compared with the contents of register RB using a 32-bit signed compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into CR[$CR0_3$].

The BIX field specifies a bit of CR[CR0] to be used as the condition of the branch, and the PX field specifies the polarity of the CR[CR0] bit used to determine if the branch is taken or untaken.

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If BIX = 3 then the 32-bit value computed to generate CR[$CR0_{0:2}$] is placed into RA, regardless of whether the branch is taken or not taken.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- CR[CR0]LT, GT, EQ, SO
- LR if LK contains a '1'
- RA if BIX = 3

Architecture Note

This instruction is specific to the PPE 42 architecture.

This instruction modifies CR[CR0] identically to the Power ISA **cmpw** instruction, including the update of CR[$CR0_3$] with XER[SO]. This instruction does not modify XER[SO] or XER[OV] however; only the “o” forms of arithmetic instructions modify XER[SO, OV]. Therefore the BIX = 3 form is used to implement an equal test with GPR writeback rather than a test for XER[SO].

Note that the subtraction performed as the underlying comparison is RA – RB, not RB – RA as per the PPE 42 **subf*** instructions.

Programming Note

The **cmpwbc** instruction can be used both as a side-effect free compare-and-branch (BIX < 3), as well as to compute and store a difference, and branch on the original equality of RA and RB (BIX = 3). The latter form allows **cmpwbc** to be used to control iteration terminating with RA = RB, that is, with a final difference of 0.

Extended Mnemonics

The PX and BIX fields of the **cmpwbc[l]** instruction control whether the branch is taken, and whether the comparison result is stored back into RA. It not necessary for programmers to code **cmpwbc** by explicitly specifying PX and BIX however. Instead, programmers typically use one of the numerous extended mnemonic forms.

The first set of extended mnemonics support using **cmpwbc** as a side-effect-free compare and branch instruction for signed comparisons. For example,

cmpwbeq RA, RB, target (equivalent to **cmpwbc 1, 2, RA, RB, target**)

Table 1.83: **cmpwbc[l]** Extended Mnemonics for Side-Effect-Free Compare-and-Branch

Branch Semantics	***cmpwbc***	***cmpwbcl***
Branch if less than	**cmpwblt**	**cmpwbltl**
Branch if less than or equal	**cmpwble**	**cmpwblel**
Branch if greater than	**cmpwbgt**	**cmpwbgtl**
Branch if greater than or equal	**cmpwbge**	**cmpwbgel**
Branch if equal	**cmpwbeq**	**cmpwbeql**
Branch if not equal	**cmpwbne**	**cmpwbnel**

The second set of extended mnemonics are used for the forms of **cmpwbc** that provide a 0/non-0 test and also update RA with the difference of RA and RB. For example:

subwbz RA, RB, target (equivalent to **cmpwbc 1, 3, RA, RB, target**)

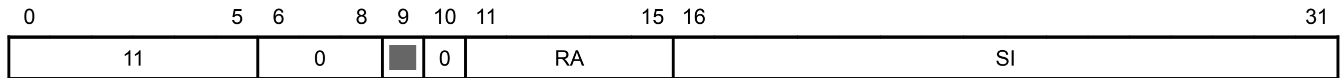
Table 1.84: **cmpwbc[l]** Extended Mnemonics for Compare-and-Branch with Update

Branch Semantics	***cmpwbc***	***cmpwbcl***
Subtract and branch if zero	**subwbz**	**subwbzl**
Subtract and branch if not zero	**subwbnz**	**subwbnzl**

9.4.27 cmpwi

Compare Word Immediate

cmpwi RA, SI



CR[CR0] ← 0
if (RA) < EXTS(SI) then CR[$CR0_0$] ← 1
if (RA) > EXTS(SI) then CR[$CR0_1$] ← 1
if (RA) = EXTS(SI) then CR[$CR0_2$] ← 1
CR[$CR0_3$] ← XER[SO]

The SI field is sign extended to 32 bits. The contents of register RA are compared with the extended SI field using a 32-bit signed compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into CR[$CR0_3$].

Restrictions

The register RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below.

Registers Altered

- CR[CR0]$_{LT, GT, EQ, SO}$

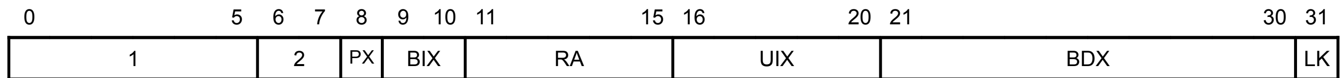
Architecture Note

cmpwi is actually an assembler extended mnemonic for the more general **cmpi** instruction of the Power ISA User Instruction Set Architecture. PPE 42 only supports word comparisons (instruction bit 10 = 0) with update of CR[CR0] (instruction bits 6:8 = 0). Any other form of the **cmpi** instruction will cause an illegal instruction exception.

9.4.28 cmpwibc

Compare Word Immediate and Branch Conditional

cmpwibc	PX, BIX, RA, UIX, target	LK = '0'
cmpwibcl	PX, BIX, RA, UIX, target	LK = '1'



CR[CR0] ← 0
if (RA) < ^{27}0 || UIX then CR[$CR0_0$] ← 1
if (RA) > ^{27}0 || UIX then CR[$CR0_1$] ← 1
if (RA) = ^{27}0 || UIX then CR[$CR0_2$] ← 1
CR[$CR0_3$] ← XER[SO]

if ((BIX ≠ 3) ∧ (PX = CR[$CR0_{BIX}$])) ∨ ((BIX = 3) ∧ (PX = CR[$CR0_2$])) then
    NIA ← CIA + EXTS(BDX || ^{2}0)
else
    NIA ← CIA + 4

If BIX = 3 then

    (RA) ← ¬(^{27}0 || UIX) + (RA) + 1

if LK = 1 then
    (LR) ← CIA + 4

The UIX field is zero-extended on the left to 32 bits. The contents of register RA are compared with the extended UIX field using a 32-bit signed compare. CR[CR0] is updated to reflect the results of the compare and the value of XER[SO] is placed into CR[$CR0_3$].

The BIX field specifies a bit of CR[CR0] to be used as the condition of the branch, and the PX field specifies the polarity of the CR[CR0] bit used to determine if the branch is taken or untaken.

The NIA is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BDX field and sign-extending the result to 32 bits. The base address is the address of the branch instruction, which is also the CIA.

If BIX = 3 then the 32-bit value computed to generate CR[$CR0_{0:2}$] is placed into RA, regardless of whether the branch is taken or not taken.

If the LK field contains a '1' then (CIA + 4) is placed into the LR, regardless of whether the branch is taken or not taken.

Restrictions

The register RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- $CR[CR0]_{LT, GT, EQ, SO}$
- LR if LK contains a '1'
- RA if BIX = 3

Architecture Note

This instruction is specific to the PPE 42 architecture.

This instruction modifies CR[CR0] identically to the **cmpwi** instruction, including the update of $CR[CR0_3]$ with XER[SO]. This instruction does not modify XER[SO] or XER[OV] however; only the “o” forms of arithmetic instructions modify XER[SO, OV]. Therefore the BIX = 3 form is used to implement an equal test with GPR writeback rather than a test for XER[SO].

Programming Note

The **cmpwibc** instruction can be used both as a side-effect free compare-and-branch (BIX < 3), as well as to compute and store a difference, and branch on the original equality of RA and UIX (BIX = 3). The latter form allows **cmpwibc** to be used to control iteration terminating with RA = UIX, that is, with a final difference of 0.

Note that standard Power ISA application binary interfaces (ABIs) specify that the CTR is a volatile register, that is, the value of CTR need not be saved and restored by a subroutine. When iterating over code sequences that include subroutine calls, it will be more time and code-space efficient to iterate with the **subwib[n]z** extended mnemonic targeting a non-volatile GPR, rather than iterating with the CTR and saving and restoring the CTR across subroutine calls.

Although **cmpwibc** can be used to compare a GPR against 0, in general this only provides an advantage immediately after a load, or when comparing the values of subroutine arguments. In general it is more efficient to use recording forms of arithmetic instructions (if available) followed by a conditional branch, rather than using non-recording forms followed by **cmpwibc** with a 0 immediate value. For example, to test an *n*-bit field of a word for a zero value, the sequence

```
extrwi. RA, RS, n, b
beq   target
```

executes in 3 cycles, while the equivalent

```
extrwi RA, RS, n, b
bwz   RA, target
```

executes in 4 cycles in the PPE 42 core.

Extended Mnemonics

The PX and BIX fields of the **cmpwibc[l]** instruction control whether the branch is taken, and whether the comparison result is stored back into RA. It not necessary for programmers to code **cmpwibc** by explicitly specifying PX and BIX however. Instead, programmers typically use one of the numerous extended mnemonic forms.

One set of extended mnemonics support using **cmpwibc** as a side-effect-free compare and branch instruction. For example,

cmpwibeq RA, n, target (equivalent to **cmpwibc 1, 2, RA, n, target**)

Table 1.85: **cmpwibc[l]** Extended Mnemonics for Side-effect-free Compare-and-branch

Branch Semantics	***cmpwibc***	***cmpwibcl***
Branch if less than	**cmpwiblt**	**cmpwibltl**
Branch if less than or equal	**cmpwible**	**cmpwiblel**
Branch if greater than	**cmpwibgt**	**cmpwibgtl**
Branch if greater than or equal	**cmpwibge**	**cmpwibgel**
Branch if equal	**cmpwibeq**	**cmpwibeql**
Branch if not equal	**cmpwibne**	**cmpwibnel**

Since comparison against 0 is such a common operation, extended mnemonics are also provided for side-effect-free compare-and-branch against an implicit 0 value. For example:

bwz RA, target (Branch on Word Zero; equivalent to **cmpwibc 1, 2, RA, 0, target**)

Table 1.86: **cmpwibc[l]** Extended Mnemonics for Side-effect-free Compare-and-branch against 0

Branch Semantics	***cmpwibc***	***cmpwibcl***
Branch if less than 0	**bwltz**	**bwltzl**
Branch if less than or equal to 0	**bwlez**	**bwlezl**
Branch if greater than 0	**bwgtz**	**bwgtzl**
Branch if greater than or equal to 0	**bwgez**	**bwgezl**
Branch if equal to 0	**bwz**	**bwzl**
Branch if not equal to 0	**bwnz**	**bwnzl**

The final set of extended mnemonics are provided for the forms of **cmpwibc** that provide a 0/non-0 test and also update RA with the difference of RA and UIX. For example:

subwibz RA, n, target (equivalent to **cmpwibc 1, 3, RA, n, target**)

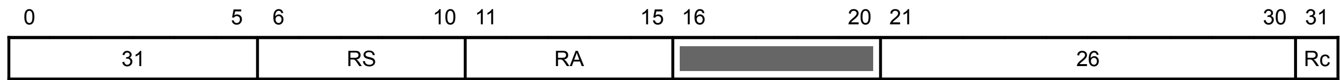
Table 1.87: **cmpwibc[l]** Extended Mnemonics for Compare-and-Branch with Update

Branch Semantics	***cmpwibc***	***cmpwibcl***
Subtract and branch if zero	**subwibz**	**subwibzl**
Subtract and branch if not zero	**subwibnz**	**subwibnzl**

9.4.29 cntlzw

Count Leading Zeros Word

cntlzw	RA, RS	Rc = '0'
cntlzw.	RA, RS	Rc = '1'



n ← 0
do while n < 32
    if $(RS)_n$ = 1 then leave
    n ← n + 1
(RA) ← n

The consecutive leading 0 bits in register RS are counted. The count is placed into register RA.

The count ranges from 0 through 32, inclusive.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

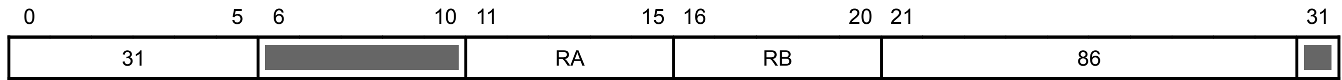
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.30 dcbf

Data Cache Block Flush

dcbf RA, RB



EA ← (RA | 0) + (RB)
DCBF(EA)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

If the data block containing the byte addressed by EA is in the data cache and marked as modified (stored into), the data block is copied back to main storage and then marked invalid in the data cache. If the data block is not marked as modified, it is marked invalid in the data cache. The operation is performed whether or not the EA is marked as cacheable.

If the data block at the EA is not in the data cache, no operation is performed.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

Exceptions

A data storage exception is signaled if a load of the EA would signal a data storage exception.

This instruction is considered a “store” with respect to DACR debug exceptions.

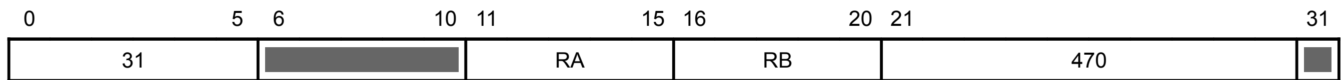
Architecture Note

The **dbcf** instruction is part of the Power ISA Book II specification. PPE 42 does not support the hint bits (instruction bits 9:10) specified by the Power ISA, which may affect the portability of programs using this instruction.

9.4.31 dcbi

Data Cache Block Invalidate

dcbi RA, RB



EA ← (RA | 0) + (RB)
DCBI(EA)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

If the data block containing the byte addressed by EA is in the data cache, the data block is marked invalid, regardless of whether the EA is marked as cacheable. If modified data existed in the data block before the operation of this instruction, that data is lost.

If the data block containing the byte addressed by EA is not in the data cache, no operation is performed.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

Exceptions

A data storage exception is signaled if a store to the EA would signal a data storage exception.

This instruction is considered a “store” with respect to DACR debug exceptions.

Architecture Note

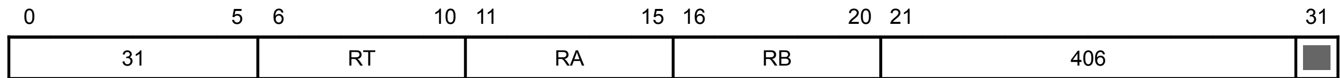
The **dcbi** instruction is part of the Power ISA Book III-E specification.

The Power ISA allows implementations to implement **dcbi** in such a way that modified data in the cache line is first flushed back to memory before invalidating the line. PPE 42 requires that the data cache *not* flush modified data, but simply invalidate the line.

9.4.32 dcbq

Data Cache Block Query

dcbq RT, RA, RB



QA ← (RA | 0) + (RB)
B ← DCBQ(QA)

if CacheBlockDefined(B) ∧ CacheBlockValid(B) then
$(RT)_{0:TagSize-1}$ ← CacheTag(B)
$(RT)_{TagSize:28}$ ← *Instance Specific*

$(RT)_{29}$ ← CacheBlockError(B)
$(RT)_{30}$ ← CacheBlockModified(B)
$(RT)_{31}$ ← CacheBlockValid(B)
else
$(RT)_{0:30}$ ← *Instance Specific*
$(RT)_{31}$ ← 0

A query address QA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The QA identifies a cache block in the data cache to be queried. The format of the query address is instance-specific, and will be documented with each instantiation of the PPE 42 core.

The storage subsystem converts the QA into a cache block descriptor B. If B is a legal descriptor for a data cache block, and the cache block is valid then

- The high order bits of register RT are set to the cache tag of cache block B.
- Bit 29 of RT is set to '1' if the cache block B is marked as being in error, otherwise '0'.
- Bit 30 of RT is set to '1' if the cache block B is marked as modified, otherwise to '0'.
- Bit 31 of RT is set to '1'.
- All other bits of RT not covered by the above specifications take on an instance-specific value.

If a data cache is present and B is not a valid descriptor for a cache block, then the storage subsystem signals a load-type data storage exception. If a data cache is not present then the storage subsystem responds to all **dcbq** requests with a value that sets $(RT)_{31}$ to 0 (invalid).

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Exceptions

The storage subsystem signals a load-type data storage exception if a data cache is present and the QA specifies an invalid cache block descriptor B.

This instruction is considered a “load” with respect to DAC debug exceptions. The “load” address in this case is the query address QA.

Architecture Note

The **dbcq** instruction is specific to the PPE 42 specification. If a data cache is present then PPE 42 requires a minimum cache block size of 8 bytes in order to implement **dcbq**.

Programming Notes

The **dcbq** instruction supports cache invalidation, cache restoration and examining data cache contents. If the data cache block referred to by QA is in a valid state, then the value returned by **dcbq** is a valid EA of a byte in the cache block regardless of how the implementation sets implementation-specific bits. This EA can be used as-is for a subsequent **dcbi**, **dcbf**, **dcbt** or **dcbz** operation, or suitably masked as an EA for loads and stores.

The data cache is not required to synchronize outstanding operations before responding to a **dcbq** request from the core, which means that the data cache tag and contents may change after the **dcbq** is executed. Programs using **dcbq** should always do so with interrupts disabled, and always issue a **sync** instruction prior to using **dcbq** to guarantee that the data cache contents are stable.

To invalidate the entire data cache a program can iterate over all valid query addresses and perform a sequence similar to the following, after first issuing a **sync** instruction to guarantee that the contents of the data cache are stable:

```
        . . .                # Compute next QA into R3
        dcbq   R4, R0, R3    # Load query result into R4
        bb0wi  R4, 31, 1f    # Ignore invalid lines
        dcbi   R0, R4        # Invalidate the block
1:
```

To copy the contents of a data cache block to another location (for example as a debugging procedure) a sequence like the following could be used. The example assumes a 32-byte cache line.

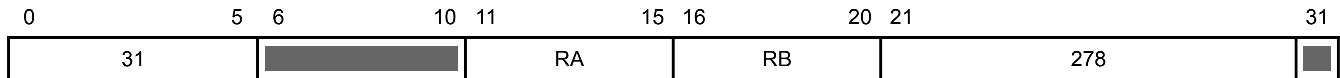
```
        . . .                # QA is in R3, target address in R4
        dcbq   R5, R0, R3    # Load query result into R5
        bb0wi  R5, 31, 1f    # Ignore invalid lines
        clrrwi R3, R5, 5     # Compute cache block base address into R3
        subi   R3, R3, 8     # Prepare to index by 8 bytes

        # Copy 4 * 8 bytes
        lvdu   D28, R3, 8
        stvdu  D28, R4, 8
        lvdu   D28, R3, 8
        stvdu  D28, R4, 8
        lvdu   D28, R3, 8
        stvdu  D28, R4, 8
        lvdu   D28, R3, 8
        stvdu  D28, R4, 8
1:             .
```

9.4.33 dcbt

Data Cache Block Touch

dcbt RA, RB



EA ← (RA | 0) + (RB)
DCBT(EA)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

If the data block containing the byte addressed by the EA is not in the data cache and the EA is marked as cacheable, the block is read from main storage into the data cache. If the data block at the EA is in the data cache, or if the EA is marked as non-cacheable, no operation is performed.

Note that the **dcbt** instruction may complete before the data is partially or fully loaded into the data cache. Execution of a **dcbt** instruction may cause a modified cache block to be written back to storage in order to make room for the new block.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

Exceptions and Synchronization

The storage subsystem signals a data storage exception if a load of the EA would signal a data storage exception.

This instruction is considered a “load” with respect to DACR debug exceptions.

PPE 42 allows the storage subsystem to include the completion of cache line fills hinted by **dcbt** in the criteria for completion of the **sync** instruction. The precise behavior of **dcbt** with respect to exceptions and synchronization will be documented with each instantiation of the PPE 42 core.

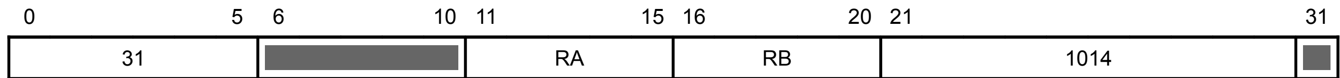
Architecture Note

The **dbct** instruction is part of the Power ISA Book II specification. The PPE 42 implementation of **dbct** differs from the Power ISA specification in several ways, including synchronization and lack of support for the hint bits (instruction bits 6:10), which may affect the portability of programs using this instruction.

9.4.34 dcbz

Data Cache Block Zero

dcbz RA, RB



EA ← (RA | 0) + (RB)
DCBZ(EA)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

If the data block containing the byte addressed by the EA is in the data cache and the EA is marked as cacheable, the data in the cache block is set to 0.

If the data block at the EA is not in the data cache and the EA is marked as cacheable, a cache block is established and set to 0. Note that nothing is read from main storage, as described in the programming note. Execution of a **dcbz i**nstruction may cause a modified cache block to be written back to storage in order to make room for the new block.

If the storage subsystem does not signal an alignment exception when presented with a **dcbz** request for a non-cacheable EA, then the storage subsystem must emulate the instruction by setting all bytes in the putative cache block containing the byte addressed by EA to 0. Otherwise the alignment exception handler may emulate the operation if required.

Restrictions

The registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

Exceptions

An alignment exception is signaled if the EA is non-cacheable and the storage subsystem does not emulate the operation.

A data storage exception is signaled if a store to the EA would signal a data storage exception.

This instruction is considered a “store” with respect to DACR debug exceptions.

Architecture Note

The **dbcz** instruction is part of the Power ISA Book II specification.

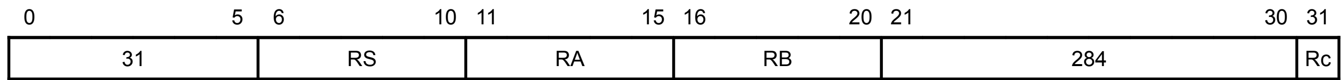
Programming Notes

Because the **dcbz** instruction can establish an address in the data cache without copying the contents of that address from main storage, the address established might be invalid with respect to the storage subsystem. A subsequent operation can cause the cached data at the invalid address to be copied back to main storage, for example, to make room for a new cache block; a machine check exception can occur under these circumstances.

9.4.35 eqv

Equivalent

eqv	RA, RS, RB	Rc = '0'
eqv.	RA, RS, RB	Rc = '1'



0 - 5	6 - 10	11 - 15	16 - 20	21 - 30	31
31	RS	RA	RB	284	Rc

$(RA) \leftarrow \neg((RS) \oplus (RB))$

The contents of register RS are XORed with the contents of register RB. The complemented result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

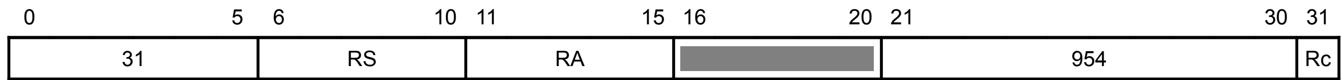
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.36 extsb

Extend Sign Byte

extsb	RA, RS	Rc = '0'
extsb.	RA, RS	Rc = '1'



$s \leftarrow (RS)_{24}$
$RA_{0:23} \leftarrow {}^{24}s$
$RA_{24:31} \leftarrow (RS)_{24:31}$

Bits 24:31 of register RS are copied to bits 24:31 of register RA, and bits 0:23 of RA are filled with a copy of bit 24 of RS.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RS
- $CR[CR0]_{LT,\ GT,\ EQ,\ SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.37 extsh

Extend Sign Halfword

extsh	RA, RS	Rc = '0'
extsh.	RA, RS	Rc = '1'

0–5	6–10	11–15	16–20	21–30	31
31	RS	RA		922	Rc

$s \leftarrow (RS)_{16}$
$RA_{0:15} \leftarrow {}^{16}s$
$RA_{16:31} \leftarrow (RS)_{16:31}$

Bits 16:31 of register RS are copied to bits 16:31 of register RA, and bits 0:15 of RA are filled with a copy of bit 16 of RS.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RS
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

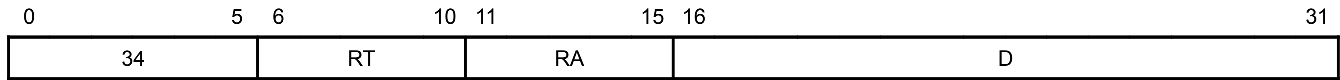
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.38 lbz

Load Byte and Zero

lbz RT, D(RA)



0 ... 5	6 ... 10	11 ... 15	16 ... 31
34	RT	RA	D

EA ← (RA | 0) + EXTS(D)
(RT) ← $^{24}0$ || MS(EA, 1)

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The byte at the EA is loaded into $RT_{24:31}$. $RT_{0:23}$ are set to 0.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.39 lbzu

Load Byte and Zero with Update

lbzu RT, D(RA)

0 5	6 10	11 15	16 31
35	RT	RA	D

$EA \leftarrow (RA) + EXTS(D)$
$(RA) \leftarrow EA$
$(RT) \leftarrow {}^{24}0 \,||\, MS(EA, 1)$

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The byte at the EA is loaded into $RT_{24:31}$. $RT_{0:23}$ are set to 0.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- RA

Invalid Instruction Forms

- RA = RT
- RA = 0

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.40 lbzx

Load Byte and Zero Indexed

lbzx RT, RA, RB

0 5	6 10	11 15	16 20	21 31	
31	RT	RA	RB	87	■

$EA \leftarrow (RA \mid 0) + (RB)$
$(RT) \leftarrow {}^{24}0 \parallel MS(EA, 1)$

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The byte at the EA is loaded into $RT_{24:31}$. $RT_{0:23}$ are set to 0.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

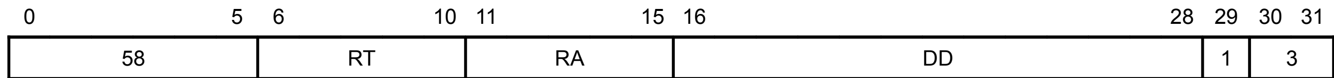
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.41 lcxu

Load stack Context with Update

lcxu RT, DD(RA)



0–5	6–10	11–15	16–28	29	30–31
58	RT	RA	DD	1	3

EA ← (RA) + EXTS(DD || ^{3}0)
(EDR) ← EA

DW(1..10) = VDR30, VDR28, SRR0 || SRR1, XER || CTR, VDR(9,7,5,3,0), CR || SPRG0
i ← 1
do while i ≤ 10
    (DW(i)) ← MS(EA - (i || ^{3}0), 8)
    i ← i + 1

(LR) ← MS(EA + 4, 4)
(RA) ← EA

An EA is formed by adding a displacement to the base address in register RA. The 32-bit displacement is obtained by sign-extending the 13-bit DD field to 32 bits after concatenating 3 0-bits on its right.

The contents of ten doublewords are loaded from consecutively descending memory locations starting at (EA - 8) into the following series of registers: VDR30, VDR28, (SRR0 and SRR1), (XER and CTR), VDR registers (9,7,5,3,0), and (CR and SPRG0).

The word at (EA + 4) is placed into the Link Register. The EA is then placed into register RA.

Restrictions

The registers RT and RA must be the same and defined in the PPE 42 core architecture, the displacement must be a positive number no greater than 32 kilobytes and must specify at least 1 doubleword, and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- EDR

Invalid Instruction Forms

- RA = 0
- RT ≠ RA
- $(DD_0 = '1') \vee (DD_{1:12} < x'00B')$

Exceptions

An alignment exception is signaled if the address in register RA is not doubleword-aligned, regardless if an unaligned operation is supported by the memory subsystem.

Synchronous exceptions caused by the execution of this instruction or a previously executed imprecise store must suppress update of RT and LR. The EDR reflects the address of the word or doubleword being accessed when the memory subsystem raised this exception. Only the GPRs and SPRs whose values had been successfully read from memory before this exception was signaled are updated.

Like other PPE 42 instructions, asynchronous exceptions that occur during the execution of this instruction are held pending until all registers altered by this instruction have been updated.

Architecture Notes

This instruction is implemented by PPE 42X, is specific to the PPE architecture, and is not part of the original PPE 42 Instruction Set. This instruction is a modified DS-Form, where DS is defined as DD || 1. The similar **lsku** instruction provides a subset of this instruction's function.

The EDR is used as a “scratch pad” to store the new Stack Pointer address during execution of this instruction and will be overwritten if this instruction experiences a precise or imprecise synchronous exception.

Unlike other load with update forms, RT is not updated from memory. Update of the register specified by RA is suppressed when its VDR doubleword is read from memory, such that the Stack Pointer is not modified if a synchronous exception occurs during execution of this instruction.

Programming Note

Although the DD field specifies the number of doublewords, assemblers should instead support the specification of a byte offset for programming consistency with other instruction forms, discarding the three rightmost bits of the offset when forming the DD field of the assembled instruction.

Usage Notes

This instruction is intended to be used to *pop* the a subset of the PPE's GPR and SPR context from a stack in memory, in a format compatible with the PowerPC EABI used for the call stack, with a single instruction and to be used in concert with the corresponding **stcxu** instruction. To ensure the stack pointer update is an atomic operation that cannot be interrupted, a load-with-update instruction form is used. Note that this instruction does not check for consistency that the Back Pointer word located in memory at EA, written by a previous stack *push* operation, is equal to the updated stack pointer that will be written into RA, since the Back Pointer is included in the EABI only for stack debug.

Registers RT and RA must point to the same GPR, which contains the current stack pointer, and the updated stack pointer is written into register RT. The call stack pointer is typically kept in GPR(1) as per the EABI. For context switches due to interrupts, the programmer may choose to share the call stack by using GPR(1) or to create a separate machine stack using a different GPR. In practice, RA will be less than 14 to honor the non-volatile definition in the EABI.

The immediate field DD contains the number of doublewords of the stack frame being read and must be a positive number specifying no more than 32 kilobytes and at least 11 doublewords, one of which contains the stack frame header. The full context data contains all PPE-defined GPRs except R2 and R13, and a subset of the PPE-defined SPRs: CR, SPRG0, XER, CTR, SRR0, and SRR1. If DD specifies more than 11 doublewords, previous instructions should have already restored additional context from the extra room that had been left in the stack frame.

The **lcxu** instruction may be emulated by the following sequence of instructions, where the stack pointer is not updated until the end, in case an interrupt occurs during the sequence:

```
lwz Ry,0(RA)        # Read Back chain
lwz Rx,4(Ry)        # Read Link register
mtlr Rx             # Restore Link register
lvd CONTEXT[0..9],-8..80(Ry) # Restore 10 doublewords of context
mr RT, Ry           # Move Stack Pointer
```

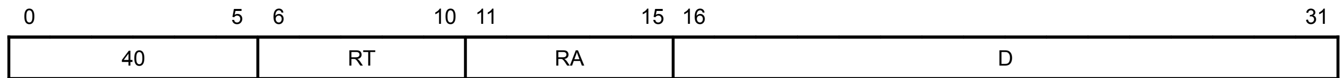
The PPE context is read (“popped”) off the stack as follows:

Stack	Address	Size (bytes)	Content Read by lcxu
Stack Data (if any)	X	X	X
Stack Frame Header	SP+4	4	Read into LR
	New Stack Pointer SP = (RA) + D (written into RA)	4	[ignore previous back pointer]
Previous Stack Frame Data	SP-8	8	Read into R31 (if RA !=31) Read into R30 (if RA !=30)
	SP-16	8	Read into R29 (if RA !=29) Read into R28 (if RA !=28)
	SP-24	8	Read into SRR1 Read into SRR0
	SP-32	8	Read into CTR Read into XER
	SP-40	8	Read into R10 (if RA !=10) Read into R9 (if RA !=9)
	SP-48	8	Read into R8 (if RA !=8) Read into R7 (if RA !=7)
	SP-56	8	Read into R6 (if RA !=6) Read into R5 (if RA !=5)
	SP-64	8	Read into R4 (if RA !=4) Read into R3 (if RA !=3)
	SP-72	8	Read into R1 (if RA !=1) Read into R0 (if RA !=0)
	SP-80	8	Read into SPRG0 Read into CR
	optional, if D>88 SP-88 ... (RA)+8	D-88	[ignore Variable Context]
Previous Stack Frame Header	(RA)+4	4	[ignore next LR placeholder]
	Previous Stack Pointer = (RA)	4	[ignore Back Pointer]

9.4.42 lhz

Load Halfword and Zero

lhz RT, D(RA)



0 5	6 10	11 15	16 31
40	RT	RA	D

EA ← (RA | 0) + EXTS(D)
(RT) ← $^{16}0$ || MS(EA, 2)

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is loaded into $RT_{16:31}$. $RT_{0:15}$ are set to 0.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

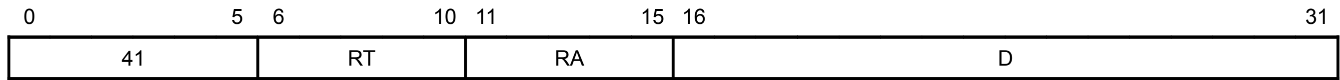
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.43 lhzu

Load Halfword and Zero with Update

lhzu RT, D(RA)



0 5	6 10	11 15	16 31
41	RT	RA	D

EA ← (RA) + EXTS(D)
(RA) ← EA
(RT) ← $^{16}0$ || MS(EA, 2)

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The halfword at the EA is loaded into $RT_{16:31}$. $RT_{0:15}$ are set to 0.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- RA

Invalid Instruction Forms

- RA = RT
- RA = 0

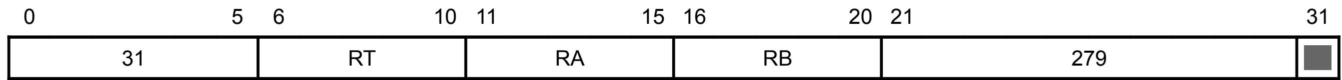
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.44 lhzx

Load Halfword and Zero Indexed

lhzx RT, RA, RB



EA ← (RA | 0) + (RB)
(RT) ← ^{16}0 || MS(EA, 2)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The halfword at the EA is loaded into $RT_{16:31}$. $RT_{0:15}$ are set to 0.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

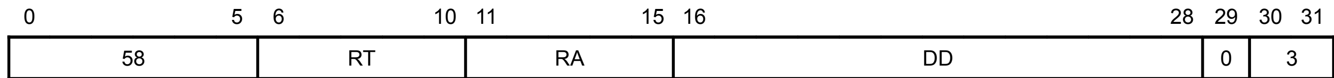
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.45 lsku

Load Stack frame with Update

lsku RT, DD(RA)



0–5	6–10	11–15	16–28	29	30–31
58	RT	RA	DD	0	3

EA ← (RA) + EXTS(DD || $^{3}0$)
(EDR) ← EA

n ← DD-1
if n > 0 then
    (VDR30) ← MS(EA - 8, 8)
if n > 1 then
    (VDR28) ← MS(EA -16, 8)

(LR) ← MS(EA + 4, 4)
(RA) ← EA

An EA is formed by adding a displacement to the base address in register RA. The 32-bit displacement is obtained by sign-extending the 13-bit DD field to 32 bits after concatenating 3 0-bits on its right.

The number of doublewords to read *n* is determined by the value of in the DD field minus 1.
If *n* is greater than zero, the doubleword at (EA – 8) is loaded into VDR30.
If *n* is greater than one, the doubleword at (EA -16) is loaded into VDR28.

The word at (EA + 4) is placed into the Link Register. The EA is then placed into register RA.

Restrictions

The registers RT and RA must be the same and defined in the PPE 42 core architecture, the displacement must be a positive number no greater than 32 kilobytes and must specify at least 1 doubleword, and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- EDR

Invalid Instruction Forms

- RA = 0
- RT ≠ RA
- $(DD_0 = '1') \lor (DD_{1:12} = {}^{12}0)$

Exceptions

An alignment exception is signaled if the address in register RA is not doubleword-aligned, regardless if an unaligned operation is supported by the memory subsystem.

Synchronous exceptions caused by the execution of this instruction or a previously executed imprecise store must suppress update of RT and LR. The EDR reflects the address of the word or doubleword being accessed when the memory subsystem raised this exception. Only the GPRs and SPRs whose values had been successfully read from memory before this exception was signaled are updated.

Like other PPE 42 instructions, asynchronous exceptions that occur during the execution of this instruction are held pending all registers altered by this instruction have been updated.

Architecture Notes

This instruction is implemented by PPE 42X, is specific to the PPE architecture, and is not part of the original PPE 42 Instruction Set. This instruction is a modified DS-Form, where DS is defined as DD || 0. The similar **lcxu** instruction provides a superset of this instruction's function.

The EDR is used as a “scratch pad” to store the new Stack Pointer address during execution of this instruction and will be overwritten if this instruction experiences a precise or imprecise synchronous exception.

Unlike other load with update forms, RT is not updated from memory. Update of the register specified by RA from memory is suppressed, such that the Stack Pointer is not modified if an intermediate exception occurs during execution of this instruction.

Programming Note

Although the DD field specifies the number of doublewords, assemblers should instead support the specification of a byte offset for programming consistency with other instruction forms, discarding the three rightmost bits of the offset when forming the DD field of the assembled instruction.

Usage Notes

This instruction is intended to be used to *pop* the a subset of the PPE's GPR and SPR context from a stack in memory, in a format compatible with the PowerPC EABI used for the call stack, with a single instruction and to be used in concert with the corresponding **stsku** instruction. To ensure the stack pointer update is an atomic operation that cannot be interrupted, a load-with-update instruction form is used. Note that this instruction does not check for consistency that the Back Pointer word located in memory at EA, written by a previous stack *push* operation, is equal to the updated stack pointer that will be written into RA, since the Back Pointer is included in the EABI only for stack debug.

Registers RT and RA must point to the same GPR, which contains the current stack pointer, and the updated stack pointer is written into register RT. The call stack pointer is typically kept in GPR(1) as per the EABI. For context switches due to interrupts, the programmer may choose to share the call stack by using GPR(1) or to create a separate machine stack using a different GPR. In practice, RA will be less than 14 to honor the non-volatile definition in the EABI.

The immediate field DD contains the number of doublewords of the stack frame being read and must be a positive number specifying no more than 32 kilobytes and at least 1 doubleword, which is the stack frame header. The full context data contains all PPE-defined GPRs except R2 and R13, and a subset of the PPE-defined SPRs: CR, SPRG0, XER, CTR, SRR0, and SRR1. If DD specifies more than 3 doublewords, previous instructions should have already restored additional context from the extra room that had been left in the stack frame.

The **lsku** instruction may be emulated by the following sequence of instructions, where the stack pointer is not updated until the end, in case an interrupt occurs during the sequence:

```
lwz Ry,0(RA)        # Read Back chain
lwz Rx,4(Ry)        # Read Link register
mtlr Rx             # Restore Link register
lvd CONTEXT[0..1],-8..16(Ry) # Restore 0,1,2 doublewords of context
mr RT, Ry           # Move Stack Pointer
```

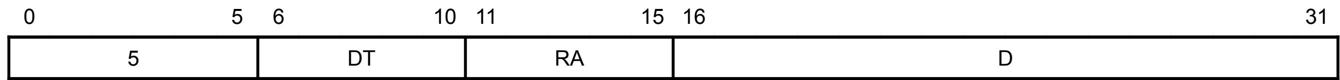
The context is read (“popped”) off the stack as follows:

Stack	Address	Size (bytes)	Content Read by lsku
Stack Data (if any)	X	X	X
Stack Frame Header	SP+4	4	Read into LR
	New Stack Pointer SP = (RA) + D (written into RA)	4	[ignore previous back pointer]
Previous Stack Frame Data	SP-8 (if D>8)	8	Read into R31 (if RA !=31) Read into R30 (if RA !=30)
	SP-16 (if D>16)	8	Read into R29 (if RA !=29) Read into R28 (if RA !=28)
	optional, if D>24 SP-24 ... (RA)+8	D-24	[ignore Variable Context]
Previous Stack Frame Header	(RA)+4	4	[ignore next LR placeholder]
	Previous Stack Pointer = (RA)	4	[ignore Back Pointer]

9.4.46 lvd

Load Virtual Doubleword

lvd DT, D(RA)



0 5	6 10	11 15	16 31
5	DT	RA	D

EA ← (RA | 0) + EXTS(D)
(DT) ← MS(EA, 8)

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The doubleword at the EA is placed into virtual doubleword register DT.

Restrictions

The registers DT and RA must be defined in the PPE 42 core architecture, and DT must specify a valid virtual doubleword register, otherwise an illegal instruction exception occurs.

Registers Altered

- DT

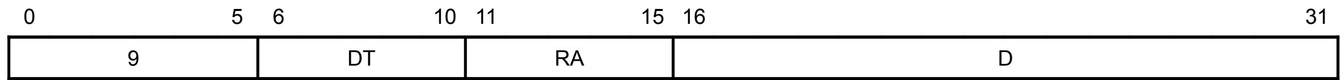
Architecture Note

This instruction is specific to the PPE 42 architecture.

9.4.47 lvdu

Load Virtual Doubleword with Update

lvdu DT, D(RA)



0 ... 5	6 ... 10	11 ... 15	16 ... 31
9	DT	RA	D

EA ← (RA) + EXTS(D)
(RA) ← EA
(DT) ← MS(EA, 8)

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The doubleword at the EA is placed into the virtual doubleword register DT.

Restrictions

The registers DT and RA must be defined in the PPE 42 core architecture, DT must specify a valid virtual doubleword register and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- DT
- RA

Invalid Instruction Forms

- RA = DT
- RA = (DT + 1) % 32
- RA = 0

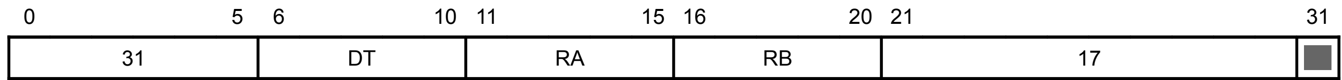
Architecture Note

This instruction is specific to the PPE 42 architecture.

9.4.48 lvdx

Load Virtual Doubleword Indexed

lvdx DT, RA, RB



EA ← (RA | 0) + (RB)
(DT) ← MS(EA, 8)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The doubleword at the EA is placed into virtual doubleword register DT.

Restrictions

The registers DT, RA and RB must be defined in the PPE 42 core architecture, and DT must specify a valid virtual doubleword register, otherwise an illegal instruction exception occurs.

Registers Altered

- DT

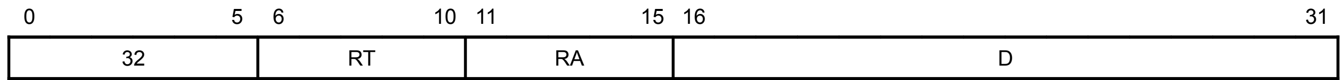
Architecture Note

This instruction is specific to the PPE 42 architecture.

9.4.49 lwz

Load Word and Zero

lwz RT, D(RA)



0 – 5	6 – 10	11 – 15	16 – 31
32	RT	RA	D

EA ← (RA | 0) + EXTS(D)
(RT) ← MS(EA, 4)

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

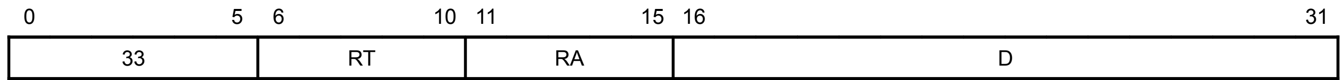
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.50 lwzu

Load Word and Zero with Update

lwzu RT, D(RA)



EA ← (RA) + EXTS(D)
(RA) ← EA
(RT) ← MS(EA, 4)

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The EA is placed into register RA.

The word at the EA is placed into register RT.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- RA

Invalid Instruction Forms

- RA = RT
- RA = 0

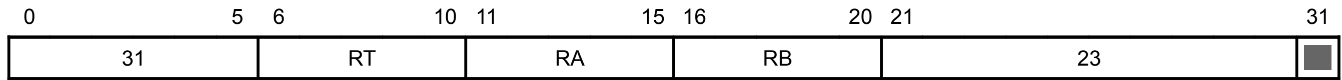
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.51 lwzx

Load Word and Zero Indexed

lwzx RT, RA, RB



EA ← (RA | 0) + (RB)
(RT) ← MS(EA, 4)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The word at the EA is placed into register RT.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.52 mfcr

Move From Condition Register

mfcr RT

$(RT)_{0:3} \leftarrow CR[CR0]$
$(RT)_{4:31} \leftarrow {}^{28}0$

The contents of CR[CR0] are copied to the four high-order bits of RS. The twenty-eight low-order bits of RT are cleared.

Restrictions

The register RT must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below.

Registers Altered

- RT

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture. Since PPE 42 only defines CR[CR0], the other Power ISA CR fields are treated by this instruction as if they contained 0.

9.4.53 mfmsr

Move from Machine State Register

mfmsr RT

(RT) ← (MSR)

The contents of the MSR are placed into RT.

Restrictions

The register RT must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Architecture Note

The **mfmsr** instruction is defined in the Power ISA Book III-E specification.

The **mfmsr** instruction is context synchronizing under normal conditions, but is not context synchronizing when rammed. For more information on ramming and single-stepping semantics see section 7, *Debugging.*

9.4.54 mfspr

Move From Special Purpose Register

mfspr RT, SPRN

$SPRN \leftarrow SPRF_{5:9} \ || \ SPRF_{0:4}$
$(RT) \leftarrow (SPR(SPRN))$

The contents of the SPR specified by the SPRF field are placed into register RT. See section 8.6, *Special Purpose Registers* for a listing of SPR mnemonics and corresponding SPRN values.

Restrictions

The register RT must be defined in the PPE 42 core architecture, and the SPRN (SPRF) field must specify a valid SPR number, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Programming Note

The SPRN value specified in the assembler language coding of the **mfspr** instruction refers to an SPR *number*. The assembler handles the unusual register number encoding to generate the SPRF field.

Although MSR and CR are considered special-purpose registers, they do not have SPR numbers and can not be read using **mfspr**. To read the MSR use **mfmsr**. To read the CR[CR0] field use **mfcr**.

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture. Unlike the Power ISA, PPE 42 does not implement processor-mode based access restrictions on SPRs, although PPE 42 does implement a small set of SPRs that are read-only to all programs.

Extended Mnemonics

Either the assembler or an assembler programming header file will typically define extended mnemonics or macros respectively to encode **mfspr** for every valid PPE 42 SPR. The expected extended mnemonics and/or macro names are listed in the table below.

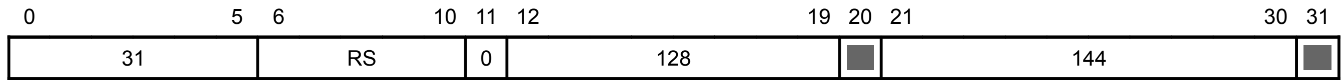
Table 1.88: Extended Mnemonics for **mfspr**

mfctr **mfdacr** **mfdbcr** **mfdec** **mfedr** **mfisr** **mfivpr** **mflr**	**mfpir** **mfpvr** **mfsprg0** **mfsrr0** **mfsrr1** **mftcr** **mftsr** **mfxer**

9.4.55 mtcr0

Move to CR[CR0]

mtcr0 RS



CR[CR0] ← $(RS)_{0:3}$

The four high-order bits of RS are copied to CR[CR0].

Restrictions

The register RS must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below.

Registers Altered

- CR[CR0]

Architecture Note

mtcr0 is actually an assembler extended mnemonic for the more general **mtcrf** instruction of the Power ISA User Instruction Set Architecture. PPE 42 only supports updating CR[CR0] as specified by instruction bits 12:19 = 128. Any other form of the **mtcrf** instruction will cause an illegal instruction exception.

Although **mtocrf** (bit 11 = '1') is the new preferred form of the legacy **mtcrf** instruction, PPE 42 implements the legacy form for compatibility with older IBM embedded processor architectures. Note that CR manipulation code that executes correctly on PPE 42 will also execute correctly on a Power ISA platform since no legal PPE 42 instruction can modify a CR field not defined by PPE 42.

9.4.56 mtmsr

Move to Machine State Register

mtmsr RS

(MSR) ← (RS)

If the processor is not halted then
    If a pending interrupt is enabled then
        NIA ← Address of highest priority interrupt
        $(SRR0)_{0:29}$ ← $((CIA) + 4)_{0:29}$
        (SRR1) ← (MSR)
        (MSR) ← (MSR) modified by the taking of the interrupt
    else if MSR[WE] = '1' then
        WAIT()

The contents of SRR1 are placed into the MSR. If the processor is halted, that is, if the **mtmsr** instruction is being single-stepped or rammed, then no other state change occurs (apart from the normal update of the IAR during single-stepping). The following description covers the case that the processor is not halted.

If the new MSR value enables one or more pending interrupts, then the highest priority interrupt is taken. The value placed into SRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address of the instruction following **mtmsr**), the MSR is placed into SRR1, and the MSR is then modified by the taking of the interrupt as described in section 4, *Interrupts and Exceptions.*

Otherwise, if the new MSR value includes MSR[WE] = '1', then the processor enters the WAIT state. The WAIT state is described in section 2.11.5.2, *WAIT mode.*

If the new MSR value does not enable any pending interrupts or cause the WAIT state to be entered, then execution continues under control of the new MSR value.

Restrictions

The register RS must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- MSR

Architecture Note

The **mtmsr** instruction is defined in the Power ISA Book III-E specification.

The **mtmsr** instruction is context synchronizing under normal conditions, but is not context synchronizing when rammed. For more information on ramming and single-stepping semantics see section 7, *Debugging.*

9.4.57 mtspr

Move To Special Purpose Register

mtspr SPRN, RS

SPRN ← $SPRF_{5:9}$ || $SPRF_{0:4}$
(SPR(SPRN)) ← (RS)

The contents of register RS are placed into the SPR specified by the SPRF field. See section 8.6, *Special Purpose Registers* for a listing of SPR mnemonics and corresponding SPRN values.

Restrictions

The register RS must be defined in the PPE 42 core architecture, and the SPRN (SPRF) field must specify a valid, writable SPR number otherwise an illegal instruction exception occurs. The read-only SPRs are the IVPR, PIR and PVR.

Registers Altered

- RT

Programming Note

The SPRN value specified in the assembler language coding of the **mtspr** instruction refers to an SPR *number*. The assembler handles the unusual register number encoding to generate the SPRF field.

Although MSR and CR are considered special-purpose registers, they do not have SPR numbers and can not be set using **mtspr**. To set the MSR use **mtmsr**. To set the CR[CR0] field use **mtcr0**.

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture. Unlike the Power ISA, PPE 42 does not implement processor-mode based access restrictions on SPRs, although PPE 42 does implement a small set of SPRs that are read-only to all programs.

The PPE 42 architecture guarantees that **mtspr** for all SPRs is fully executed prior to fetching the subsequent instruction. This means that if the effect of **mtspr** is to cause or unmask an interrupt, the next instruction executed after the **mtspr** will be the instruction at the associated interrupt vector address.

Extended Mnemonics

Either the assembler or an assembler programming header file will typically define extended mnemonics or macros respectively to encode **mtspr** for every valid, writable PPE 42 SPR. The expected extended mnemonics and/or macro names are listed in the table below.

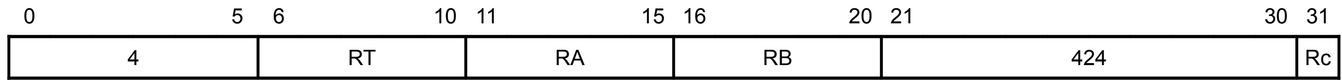
Table 1.89: Extended Mnemonics for **mtspr**

mtctr **mtdacr** **mtdbcr**	**mtdec** **mtedr** **mtisr**	**mtlr** **mtsprg0** **mtsrr0** **mtsrr1**	**mttcr** **mttsr** **mtxer**

9.4.58 mullhw

Multiply Low Halfword to Word Signed

mullhw	RT, RA, RB	Rc = '0'
mullhw.	RT, RA, RB	Rc = '1'



0 - 5	6 - 10	11 - 15	16 - 20	21 - 30	31
4	RT	RA	RB	424	Rc

$(RT)_{0:31} \leftarrow (RA)_{16:31} \times (RB)_{16:31}$ signed

The low-order halfword of RA is multiplied by the low-order halfword of RB. The resulting signed product replaces the contents of RT.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

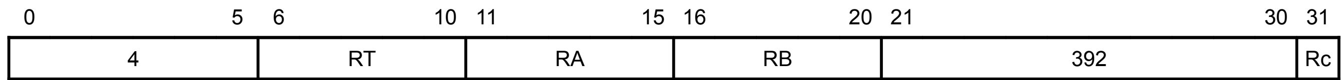
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture, Legacy Integer Multiply-Accumulate Instructions, and as such will not be portable to every Power ISA system.

9.4.59 mullhwu

Multiply Low Halfword to Word Unsigned

mullhwu	RT, RA, RB	Rc = '0'
mullhwu.	RT, RA, RB	Rc = '1'



0 5	6 10	11 15	16 20	21 30	31
4	RT	RA	RB	392	Rc

$(RT)_{0:31} \leftarrow (RA)_{16:31}$ x $(RB)_{16:31}$ unsigned

The low-order halfword of RA is multiplied by the low-order halfword of RB. The resulting unsigned product replaces the contents of RT.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

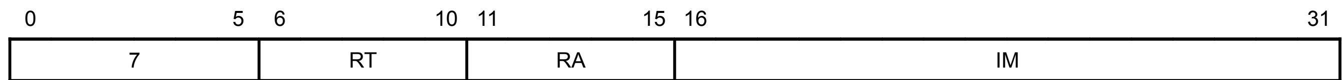
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture, Legacy Integer Multiply-Accumulate Instructions, and as such will not be portable to every Power ISA system.

9.4.60 mulli

Multiply Low Immediate

mulli RT, RA, IM



$prod_{0:47} \leftarrow$ (RA) x EXTS(IM)
(RT) $\leftarrow prod_{16:47}$

The 48-bit product of register RA and the sign-extended IM field is formed. Both register RA and the IM field are interpreted as signed quantities. The least significant 32 bits of the product are placed into register RT.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT

Programming Note

The least significant 32 bits of the product are correct, regardless of whether register RA and field IM are interpreted as signed or unsigned numbers.

Architecture Notes

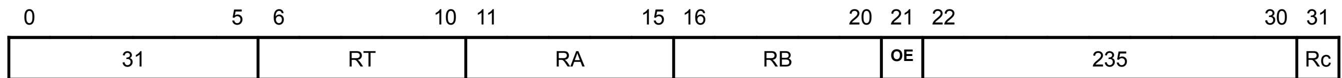
This instruction is part of the Power ISA User Instruction Set Architecture.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

9.4.61 mullw

Multiply Low Word

mullw	RT, RA, RB	OE = '0', Rc = '0'
mullw.	RT, RA, RB	OE = '0', Rc = '1'
mullwo	RT, RA, RB	OE = '1', Rc = '0'
mullwo.	RT, RA, RB	OE = '1', Rc = '1'



$prod_{0:63} \leftarrow$ (RA) x (RB) signed
(RT) $\leftarrow prod_{32:63}$

The 64-bit signed product of register RA and register RB is formed. The least significant 32 bits of the result is placed into register RT.

If the signed product cannot be represented in 32 bits and OE = 1, XER[SO, OV] are set to ‘1’.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

Programming Note

The least significant 32 bits of the product are correct, regardless of whether register RA and register RB are interpreted as signed or unsigned numbers. The overflow indication is correct only if the operands are regarded as signed numbers.

Architecture Notes

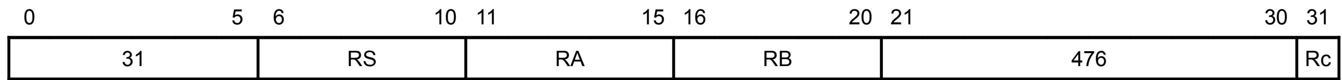
This instruction is part of the Power ISA User Instruction Set Architecture.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

9.4.62 nand

NAND

nand	RA, RS, RB	Rc = '0'
nand.	RA, RS, RB	Rc = '1'



0 – 5	6 – 10	11 – 15	16 – 20	21 – 30	31
31	RS	RA	RB	476	Rc

$(RA) \leftarrow \neg((RS) \wedge (RB))$

The contents of register RS are ANDed with the contents of register RB. The complement of the result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

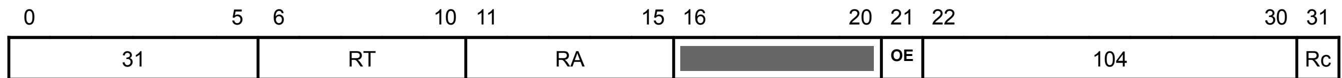
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.63 neg

Negate

neg	RT, RA	OE = '0', Rc = '0'
neg.	RT, RA	OE = '0', Rc = '1'
nego	RT, RA	OE = '1', Rc = '0'
nego.	RT, RA	OE = '1', Rc = '1'



(RT) ← ¬(RA) + 1

The sum of the complement of the contents of register RA and 1 is placed into register RT.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

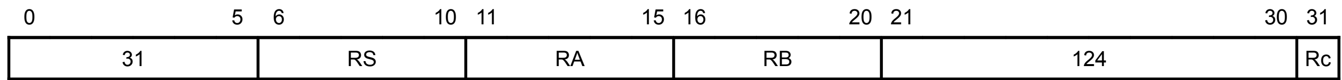
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.64 nor

NOR

nor	RA, RS, RB	Rc = '0'
nor.	RA, RS, RB	Rc = '1'



0–5	6–10	11–15	16–20	21–30	31
31	RS	RA	RB	124	Rc

(RA) ← ¬((RS) ∨ (RB))

The contents of register RS are ORed with the contents of register RB. The complement of the result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

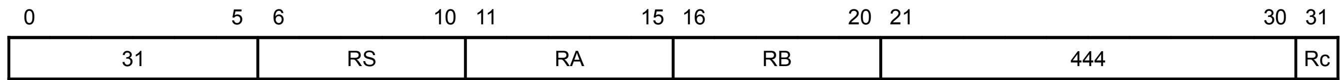
The **not** mnemonic copies the complement of register Ry to register Rx. This mnemonic can be coded with a final “.” to cause the Rc bit to be set in the underlying instruction.

not Rx,Ry (equivalent to: n**or Rx,Ry,Ry**)

9.4.65 or

OR

or	RA, RS, RB	Rc = '0'
or.	RA, RS, RB	Rc = '1'



0–5	6–10	11–15	16–20	21–30	31
31	RS	RA	RB	444	Rc

(RA) ← (RS) ∨ (RB)

The contents of register RS are ORed with the contents of register RB. The result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

Several Power ISA instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

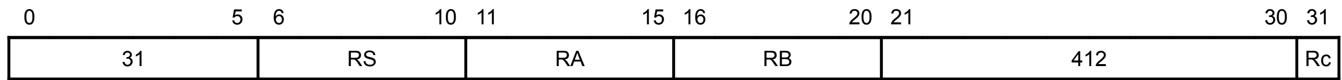
The **mr** mnemonic copies the contents of register Ry to register Rx. This mnemonic can be coded with a final “.” to cause the Rc bit to be set in the underlying instruction.

mr Rx,Ry (equivalent to: **or Rx,Ry,Ry**)

9.4.66 orc

OR with Complement

orc	RA, RS, RB	Rc = '0'
orc.	RA, RS, RB	Rc = '1'



0 5	6 10	11 15	16 20	21 30	31
31	RS	RA	RB	412	Rc

(RA) ← (RS) ∨ ¬(RB)

The contents of register RS are ORed with the complement of the contents of register RB. The result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

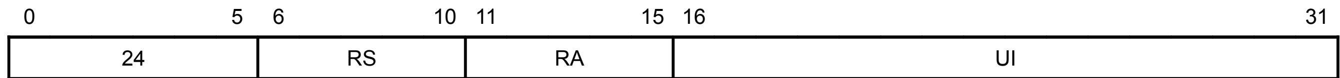
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.67 ori

OR Immediate

ori RA, RS, UI



0 5	6 10	11 15	16 31
24	RS	RA	UI

(RA) ← (RS) ∨ (^{16}0 || UI)

The UI field is extended to 32 bits by concatenating 16 0-bits on its left. The contents of register RS are ORed with the extended UI field. The result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

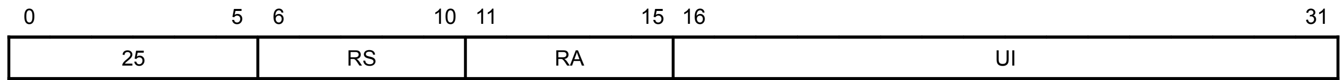
Many Power ISA instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the preferred form of no-op, however the PPE 42 core does not perform any run-time optimization related to no-ops.

nop (equivalent to: **ori 0,0,0**)

9.4.68 oris

OR Immediate Shifted

oris RA, RS, UI



0 5	6 10	11 15	16 31
25	RS	RA	UI

(RA) ← (RS) ∨ (UI || ^{16}0)

The UI field is extended to 32 bits by concatenating 16 0-bits on its right. The contents of register RS are ORed with the extended UI field. The result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.69 rfi

Return from Interrupt

rfi

(IAR) ← $(SRR0)_{0:29}$ || $^{2}0$
(MSR) ← (SRR1)

If the processor is not halted then
    If a pending interrupt is enabled then
        NIA ← Address of highest priority interrupt
        $(SRR0)_{0:29}$ ← $(IAR)_{0:29}$
        (SRR1) ← (MSR)
        (MSR) ← (MSR) modified by the taking of the interrupt
    else if MSR[WE] = '1' then
        WAIT()
    else
        NIA ← (IAR)

The **rfi** instruction is used to return from an interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of SRR0 are placed into the IAR, and the contents of SRR1 are placed into the MSR. If the processor is halted, that is, if the **rfi** instruction is being single-stepped or rammed, then no other state change occurs (other than the normal update of the IAR during single-stepping). The following description covers the case that the processor is not halted.

If the new MSR value enables one or more pending interrupts, then the highest priority interrupt is taken. In this case the value placed into SRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred, that is, the address in SRR0 is effectively unmodified. The MSR is placed into SRR1 and the MSR is then modified by the taking of the interrupt as described in section 4, *Interrupts and Exceptions.*

Otherwise, if the new MSR value does not enable any pending exceptions but includes MSR[WE] = '1', then the processor enters the WAIT state. The WAIT state is described in section 2.11.5.2, WAIT mode.

If the new MSR value does not enable any pending interrupts and includes MSR[WE] = '0', then the next instruction is fetched, under control of the new MSR value, from the address contained in SRR0.

Registers Altered

- MSR

Architecture Note

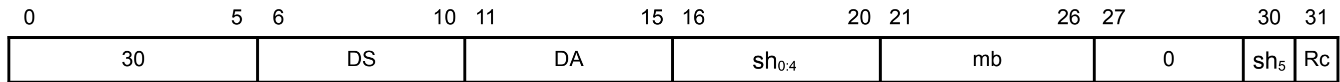
The **rfi** instruction is defined in the Power ISA Book III-E specification.

The **rfi** instruction is context synchronizing under normal conditions, but is not context synchronizing when rammed. For more information on ramming and single-stepping semantics see section 7, *Debugging.*

9.4.70 rldicl

Rotate Left Doubleword Immediate then Clear Left

rldicl	DA, DS, SH, MB	Rc = '0'
rldicl.	DA, DS, SH, MB	Rc = '1'



$n \leftarrow sh_5 \,||\, sh_{0:4}$
$r \leftarrow ROTL_{64}((DS), n)$
$b \leftarrow mb_5 \,||\, mb_{0:4}$
$m \leftarrow MASK(b, 63)$
$(DA) \leftarrow r \wedge m$

The contents of register DS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in bit position 63, with 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register DA.

Restrictions

The registers DS and DA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- DA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a '1'

Architecture Notes

This instruction is part of the 64-bit Power ISA User Instruction Set Architecture.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

Extended Mnemonics

The extended mnemonics for **rldicl** can all be coded with an optional "." to specify the recording form that updates CR[CR0].

Table 1.90: Extended Mnemonics for **rldicl**

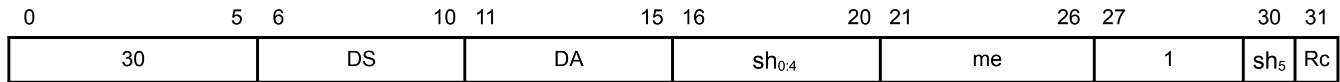
Mnemonic	Equivalent	Description
clrldi[.] DA, DS, n	**rldicl[.] DA, DS, 0, n**	Clear left doubleword immediate. (n < 64) $(DA)_{0:n-1} \leftarrow {}^{n}0$ $(DA)_{n:64} \leftarrow (DS)_{n:64}$
extrdi[.] DA, DS, n, b	**rldicl[.] DA, DS, b+n, 64-n**	Extract and right justify doubleword immediate. (n > 0) $(DA)64\text{-}n{:}63 \leftarrow (DS)_{b:b+n-1}$ $(DA)_{0:63-n} \leftarrow {}^{64-n}0$
srdi[.] DA, DS, n	**rldicl[.] DA, DS, 64-n, n**	Shift right doubleword immediate. (n < 64) $(DA)_{n:63} \leftarrow (DS)_{0:63-n}$

		$(DA)_{0:n-1} \leftarrow {}^{n}0$

9.4.71 rldicr

Rotate Left Doubleword Immediate then Clear Right

rldicr	DA, DS, SH, ME	Rc = '0'
rldicr.	DA, DS, SH, ME	Rc = '1'



n ← sh_5 || $sh_{0:4}$
r ← $ROTL_{64}$((DS), n)
e ← me_5 || $me_{0:4}$
m ← MASK(0, e)
(DA) ← r ∧ m

The contents of register DS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at bit 0 and ending in bit position specified in the ME field, with 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register DA.

Restrictions

The registers DS and DA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- DA
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Notes

This instruction is part of the 64-bit Power ISA User Instruction Set Architecture.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

Extended Mnemonics

The extended mnemonics for **rldicr** can all be coded with an optional “.” to specify the recording form that updates CR[CR0].

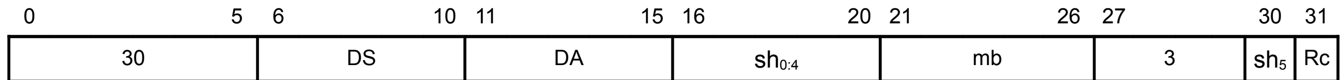
Table 1.91: Extended Mnemonics for **rldicr**

Mnemonic	Equivalent	Description
clrrdi[.] RA, RS, n	**rldicr[.] RA, RS, 0, 63-n**	Clear right doubleword immediate. (n < 64) $(DA)_{64-n:63} \leftarrow {}^{n}0$ $(DA)_{0:n-1} \leftarrow (DS)_{0:n-1}$
extldi[.] DA, DS, n, b	**rldicr[.] DA, DS, b, n-1**	Extract and left justify doubleword immediate. (n >0) $(DA)_{0:n-1} \leftarrow (DS)_{b:b+n-1}$ $(DA)_{n:63} \leftarrow {}^{64-n}0$
sldi[.] DA, DS, n	**rldicr[.] DA, DS, n, 63-n**	Shift left doubleword immediate. (n < 64) $(DA)_{0:63-n} \leftarrow (DS)_{n:63}$ $(DA)_{64-n:63} \leftarrow {}^{n}0$

9.4.72 rldimi

Rotate Left Doubleword Immediate then Mask Insert

rldimi	DA, DS, SH, MB	Rc = '0'
rldimi.	DA, DS, SH, MB	Rc = '1'



$n \leftarrow sh_5 \,||\, sh_{0:4}$
$r \leftarrow ROTL_{64}((DS), n)$

$m \leftarrow MASK(mb_5 \,||\, mb_{0:4}, \neg n)$

$(DA) \leftarrow (r \wedge m) \vee ((DA) \wedge \neg m)$

The contents of register DS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by 63-SH, with 0-bits elsewhere. The rotated data is inserted into register DA, in positions corresponding to the bit positions in the mask that contain a 1-bit.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is inserted into register DA, in positions corresponding to the bit positions in the mask that contain a 1-bit.

Restrictions

The registers DS and DA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- DA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the 64-bit Power ISA User Instruction Set Architecture.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

Extended Mnemonics

The extended mnemonics for **rldimi** can all be coded with an optional “.” to specify the recording form that updates CR[CR0].

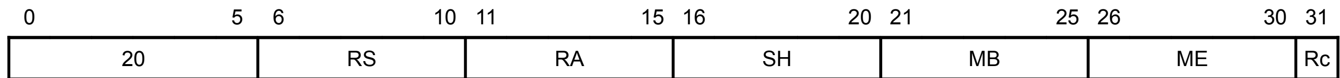
Table 1.92: Extended Mnemonics for **rldimi**

Mnemonic	Equivalent	Description
insrdi[.] DA, DS, n, b	**rldimi[.] DA, DS, 64–b-n, b**	Insert from right doubleword immediate. (n > 0) $(DA)_{b:b+n-1} \leftarrow (DS)_{64-n:63}$ Other bits of DA are unchanged.

9.4.73 rlwimi

Rotate Left Word Immediate then Mask Insert

rlwimi	RA, RS, SH, MB, ME	Rc = '0'
rlwimi.	RA, RS, SH, MB, ME	Rc = '1'



$r \leftarrow \text{ROTL}((\text{RS}), \text{SH})$
$m \leftarrow \text{MASK}(\text{MB}, \text{ME})$

$(\text{RA}) \leftarrow (r \land m) \lor ((\text{RA}) \land \neg m)$

The contents of register RS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field, with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is inserted into register RA, in positions corresponding to the bit positions in the mask that contain a 1-bit.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

The extended mnemonics for **rlwimi** can all be coded with an optional “.” to specify the recording form that updates CR[CR0].

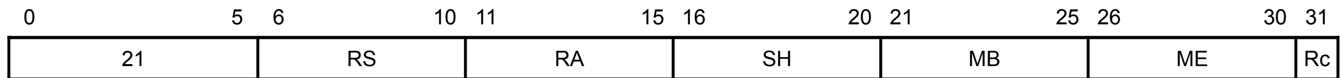
Table 1.93: Extended Mnemonics for **rlwimi**

Mnemonic	Equivalent	Description
inslwi[.] RA, RS, n, b	**rlwimi[.] RA, RS, 32–b, b, b+n+1**	Insert from left word immediate. (n > 0) $(\text{RA})_{b:b+1} \leftarrow (\text{RS})_{0:n-1}$ Other bits of RA are unchanged.
insrwi[.] RA, RS, n, b	**rlwimi[.] RA, RS, 32–b-n, b, b+n+1**	Insert from right word immediate. (n > 0) $(\text{RA})_{b:b+n-1} \leftarrow (\text{RS})_{32-n:31}$ Other bits of RA are unchanged.

9.4.74 rlwinm

Rotate Left Word Immediate then AND with Mask

rlwinm	RA, RS, SH, MB, ME	Rc = '0'
rlwinm.	RA, RS, SH, MB, ME	Rc = '1'



$r \leftarrow ROTL((RS), SH)$
$m \leftarrow MASK(MB, ME)$
$(RA) \leftarrow r \land m$

The contents of register RS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field, with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

The extended mnemonics for **rlwinm** can all be coded with an optional "." to specify the recording form that updates CR[CR0].

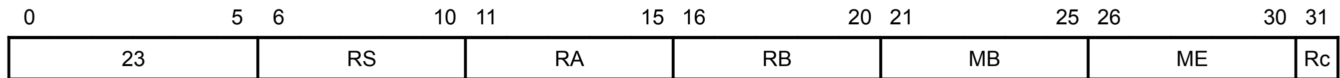
Table 1.94: Extended Mnemonics for **rlwinm**

Mnemonic	Equivalent	Description
clrbwi[.] RA, RS, b	**rlwinm[.] RA, RS, 0, (b+1) % 31, (b-1) % 31**	Clear bit from word immediate (b < 32) $(RA)_X \leftarrow (RS)_X$ for X = 0,...,31 and X ≠ b $(RA)_b \leftarrow$ '0'
clrlwi[.] RA, RS, n	**rlwinm[.] RA, RS, 0, n, 31**	Clear left word immediate. (n < 32) $(RA)_{0:n-1} \leftarrow {}^{n}0$ $(RA)_{n:32} \leftarrow (RS)_{n:32}$
clrrwi[.] RA, RS, n	**rlwinm[.] RA, RS, 0, 0, 31-n**	Clear right word immediate. (n < 32) $(RA)_{32-n:31} \leftarrow {}^{n}0$ $(RA)_{0:n-1} \leftarrow (RS)_{0:n-1}$
extlwi[.] RA, RS, n, b	**rlwinm[.] RA, RS, b, 0, n-1**	Extract and left justify word immediate. (n >0) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow {}^{32-n}0$
extrwi[.] RA, RS, n, b	**rlwinm[.] RA, RS, b+n, 32-n, 31**	Extract and right justify word immediate. (n > 0) (RA)32-n:31 $\leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow {}^{32-n}0$
rotlwi[.] RA, RS, n	**rlwinm[.] RA, RS, n, 0, 31**	Rotate left word immediate. (RA) ← ROTL((RS), n)
rotrwi[.] RA, RS, n	**rlwinm[.] RA, RS, 32-n, 0, 31**	Rotate right word immediate. (RA) ← ROTL((RS), 32-n)
slwi[.] RA, RS, n	**rlwinm[.] RA, RS, n, 0, 31-n**	Shift left word immediate. (n < 32) $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^{n}0$
srwi[.] RA, RS, n	**rlwinm[.] RA, RS, 32-n, n, 31**	Shift right word immediate. (n < 32) $(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow {}^{n}0$

9.4.75 rlwnm

Rotate Left Word then AND with Mask

rlwnm	RA, RS, RB, MB, ME	Rc = '0'
rlwnm.	RA, RS, RB, MB, ME	Rc = '1'



r ← ROTL((RS), $(RB)_{27:31}$)
m ← MASK(MB, ME)
(RA) ← r ∧ m

The contents of register RS are rotated left by the number of bit positions specified by the contents of register $(RB)_{27:31}$. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field, with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

The extended mnemonics for **rlwnm** can all be coded with an optional “.” to specify the recording form that updates CR[CR0].

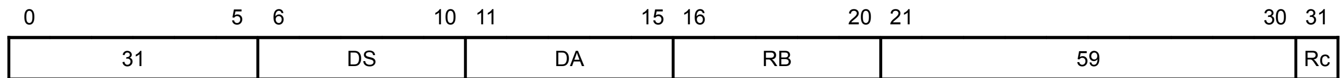
Table 1.95: Extended Mnemonics for **rlwnm**

Mnemonic	Equivalent	Description
rotlw[.] RA, RS, RB	**rlwnm[.] RA, RS, RB, 0, 31**	Rotate left word (RA) ← ROTL((RS), $(RB)_{27:31}$)

9.4.76 slvd

Shift Left Virtual Doubleword

slvd	DA, DS, RB	Rc = '0'
slvd.	DA, DS, RB	Rc = '1'



$n \leftarrow (RB)_{26:31}$

$r \leftarrow ROTL((DS), n)$

if $(RB)_{25}$ = 0 then

    $m \leftarrow MASK(0, 63 - n)$

else

    $m \leftarrow {}^{64}0$

$(DA) \leftarrow r \wedge m$

The contents of register DS are shifted left by the number of bits specified by the contents of register $RB_{26:31}$. Bits shifted left out of the most significant bit are lost, and 0-bits fill vacated bit positions on the right. The result is placed into register DA.

If RB_{25} = ‘1’, register DA is set to zero.

In other words, **slvd** shifts the contents of DS by the contents of RB modulo 64 bits.

Restrictions

The registers DS, DA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- DA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Notes

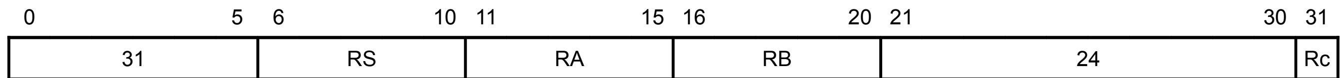
This instruction is similar to **sld** of the 64-bit Power ISA User Instruction Set Architecture, where RB is instead defined as a 64-bit register.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

9.4.77 slw

Shift Left Word

slw	RA, RS, RB	Rc = '0'
slw.	RA, RS, RB	Rc = '1'



0 — 5	6 — 10	11 — 15	16 — 20	21 — 30	31
31	RS	RA	RB	24	Rc

n ← $(RB)_{27:31}$
r ← ROTL((RS), n)
if $(RB)_{26}$ = 0 then
  m ← MASK(0, 31 − n)
else
  m ← $^{32}0$
(RA) ← r ∧ m

The contents of register RS are shifted left by the number of bits specified by the contents of register $RB_{27:31}$. Bits shifted left out of the most significant bit are lost, and 0-bits fill vacated bit positions on the right. The result is placed into register RA.

If RB_{26} = '1', register RA is set to zero.

In other words, **slw** shifts the contents of RS by the contents of RB modulo 64 bits.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]LT, GT, EQ, SO if Rc contains a '1'

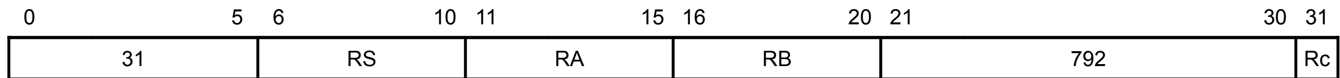
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.78 sraw

Shift Right Algebraic Word

sraw	RA, RS, RB	Rc = '0'
sraw.	RA, RS, RB	Rc = '1'



0–5	6–10	11–15	16–20	21–30	31
31	RS	RA	RB	792	Rc

$n \leftarrow (RB)_{27:31}$
$r \leftarrow ROTL((RS), 32 - n)$
if $(RB)_{26} = 0$ then
    $m \leftarrow MASK(n, 31)$
else
    $m \leftarrow {}^{32}0$
$s \leftarrow (RS)_0$

$(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$

$XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$

The contents of register RS are shifted right by the number of bits specified by the contents of register $RB_{27:31}$. Bits shifted right out of the most significant bit are lost. Bit RS_0 is replicated to fill the vacated bit positions on the left. The result is placed into register RA.

If register RS contains a negative number and any 1-bits were shifted out of the least-significant bit position, XER[CA] is set to 1. Otherwise XER[CA] is set to 0.

If RB_{26} = ‘1’, register RA and XER[CA] are set to RS_0.

In other words, **sraw** shifts the contents of RS by the contents of RB modulo 64 bits.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

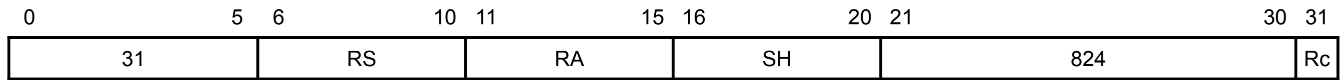
Programming Note

An **sraw** instruction followed by **addze** can be used to quickly divide a signed integer (RS) by $2^{(RB) \% 64}$.

9.4.79 srawi

Shift Right Algebraic Word Immediate

srawi	RA, RS, SH	Rc = '0'
srawi.	RA, RS, SH	Rc = '1'



0–5	6–10	11–15	16–20	21–30	31
31	RS	RA	SH	824	Rc

$n \leftarrow SH$
$r \leftarrow ROTL((RS), 32 - n)$
$m \leftarrow MASK(n, 31)$
$s \leftarrow (RS)_0$

$(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$

$XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$

The contents of register RS are shifted right by the number of bits specified by the SH field. Bits shifted right out of the most significant bit are lost. Bit RS_0 is replicated to fill the vacated bit positions on the left. The result is placed into register RA.

If register RS contains a negative number and any 1-bits were shifted out of the least-significant bit position, XER[CA] is set to 1. Otherwise XER[CA] is set to 0.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

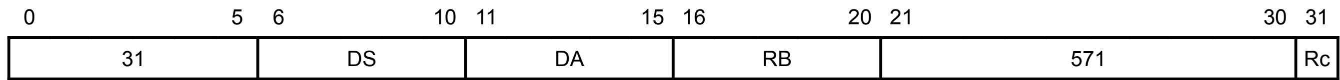
Programming Note

An **srawi** instruction followed by **addze** can be used to quickly divide a signed integer (RS) by 2^{SH}.

9.4.80 srvd

Shift Right Virtual Doubleword

srvd	DA, DS, RB	Rc = '0'
srvd.	DA, DS, RB	Rc = '1'



0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
31	DS	DA	RB	571	Rc

n ← $(RB)_{26:31}$
r ← ROTL((DS), 64-n)
if $(RB)_{25}$ = 0 then
    m ← MASK(n, 63)
else
    m ← $^{64}0$
(DA) ← r ∧ m

The contents of register DS are shifted right by the number of bits specified by the contents of register $RB_{26:31}$. Bits shifted right out of the least significant bit are lost, and 0-bits fill vacated bit positions on the left. The result is placed into register DA.

If RB_{25} = ‘1’, register DA is set to zero.

In other words, **srvw** shifts the contents of DS by the contents of RB modulo 64 bits.

Restrictions

The registers DS, DA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- DA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Notes

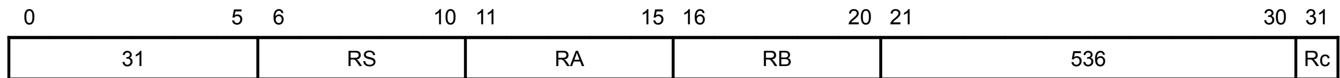
This instruction is similar to **srd** of the 64-bit Power ISA User Instruction Set Architecture, where RB is instead defined as a 64-bit register.

This instruction is implemented by PPE 42X and is not part of the original PPE 42 Instruction Set.

9.4.81 srw

Shift Right Word

srw	RA, RS, RB	Rc = '0'
srw.	RA, RS, RB	Rc = '1'



0 5	6 10	11 15	16 20	21 30	31
31	RS	RA	RB	536	Rc

$n \leftarrow (RB)_{27:31}$
$r \leftarrow ROTL((RS), 32 - n)$
if $(RB)_{26}$ = 0 then
    $m \leftarrow MASK(n, 31)$
else
    $m \leftarrow {}^{32}0$
$(RA) \leftarrow r \wedge m$

The contents of register RS are shifted right by the number of bits specified by the contents of register $RB_{27:31}$. Bits shifted right out of the least significant bit are lost, and 0-bits fill vacated bit positions on the left. The result is placed into register RA.

If RB_{26} = ‘1’, register RA is set to zero.

In other words, **srw** shifts the contents of RS by the contents of RB modulo 64 bits.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

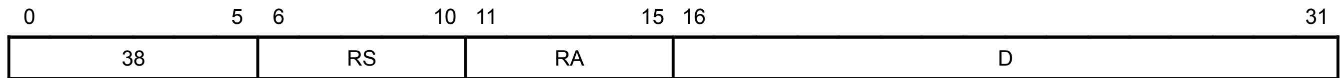
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.82 stb

Store Byte

stb RS, D(RA)



0 5	6 10	11 15	16 31
38	RS	RA	D

EA ← (RA | 0) + EXTS(D)
MS(EA, 1) ← $(RS)_{24:31}$

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

Bits 24:31 of register RS are stored into the byte addressed by the EA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

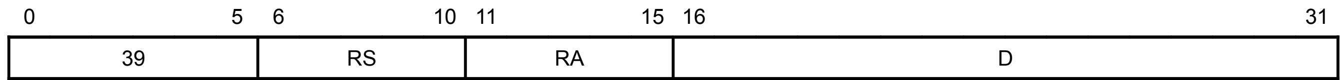
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.83 stbu

Store Byte with Update

stbu RS, D(RA)



0–5	6–10	11–15	16–31
39	RS	RA	D

EA ← (RA) + EXTS(D)
MS(EA, 1) ← $(RS)_{24:31}$
(RA) ← EA

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits.

Bits 24:31 of register RS are stored into the byte addressed by the EA.

The EA is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

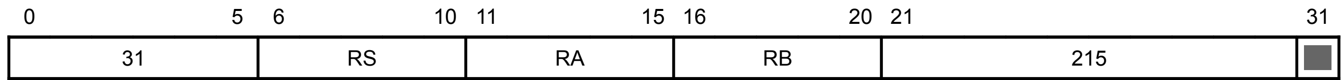
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.84 stbx

Store Byte Indexed

stbx RS, RA, RB



EA ← (RA | 0) + (RB)
MS(EA, 1) ← $(RS)_{24:31}$

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

Bits 24:31 of register RS are stored into the byte addressed by the EA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

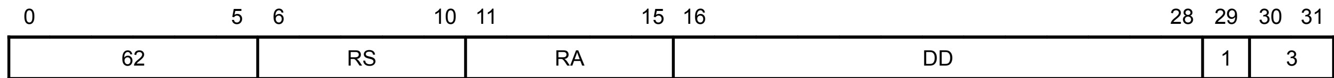
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.85 stcxu

Store stack Context with Update

stcxu RS, DD(RA)



0 – 5	6 – 10	11 – 15	16 – 28	29	30 – 31
62	RS	RA	DD	1	3

EA ← (RA)
MS(EA + 4, 4) ← (LR)
(EDR) ← (RA)

DW(1..10) = VDR30, VDR28, SRR0 || SRR1, XER || CTR, VDR(9,7,5,3,0), CR || SPRG0
i ← 1
do while i ≤ 10
    MS(EA - (i || ^{3}0), 8) ← (DW(i))
    i ← i + 1

EA ← (RA) + EXTS(DD || ^{3}0)
MS(EA, 4) ← (RS)
(RA) ← EA

An EA is formed by the contents of register RA. The contents of the Link Register are stored into the word at (EA + 4). The contents of ten doublewords are stored into consecutively descending memory locations starting at (EA - 8) from the following series of registers: VDR30, VDR28, SRR0 concatenated on the left of SRR1, XER concatenated on the left of CTR, VDR registers (9,7,5,3,0), and CR concatenated on the left of SPRG0.

A second EA is formed by adding a displacement to the base address in register RA. The 32-bit displacement is obtained by sign-extending the 13-bit DD field to 32 bits after concatenating 3 0-bits on its right. The contents of register RS are stored into the word addressed by this second EA. This EA is then placed into register RA.

Restrictions

The registers RS and RA must be the same and defined in the PPE 42 core architecture, the displacement must be a negative number and must specify at least 11 doublewords, and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- EDR

Invalid Instruction Forms

- RA = 0
- RA ≠ RS
- $(DD_0 = '0') \lor (DD_{1:12} >^{u} x'FF5')$

Exceptions

An alignment exception is signaled if the address in register RA is not doubleword aligned, regardless if an unaligned operation is supported by the memory subsystem.

Synchronous exceptions caused by the execution of this instruction or a previously executed imprecise store must suppress the update of RA. The EDR reflects the address of the word or doubleword being accessed when the memory subsystem raised this exception. Only memory locations accessed before this exception was signaled are updated.

Since this instruction is storage synchronizing, asynchronous exceptions that occur during the execution of this instruction are held pending until all stores generated by this instruction have been completed by the memory subsystem.

Architecture Notes

This instruction is implemented by PPE 42X, is specific to the PPE architecture, and is not part of the original PPE 42 Instruction Set. This instruction is a modified DS-Form, where DS is defined as DD || 1, and is storage synchronizing. The similar **stsku** instruction provides a subset of this instruction's function.

The EDR is used as a “scratch pad” to store the current Stack Pointer address during execution of this instruction and will be overwritten if this instruction experiences a precise or imprecise synchronous exception.

Programming Note

Although the DD field specifies the number of doublewords, assemblers should instead support the specification of a byte offset for programming consistency with other instruction forms, discarding the three rightmost bits of the offset when forming the DD field of the assembled instruction.

Usage Notes

This instruction is intended to be used to *push* a subset of the PPE's GPR and SPR context onto a stack in memory, in a format compatible with the PowerPC EABI used for the call stack, with a single instruction and to be used in concert with the corresponding **lcxu** instruction. To ensure the stack pointer update is an atomic operation that cannot be interrupted, a store-with-update instruction form is used. Note that this instruction also populates the Back Pointer word located in memory at the updated stack pointer address to facilitate debug as per the EABI.

Registers RS and RA must point to the same GPR, which contains the current stack pointer, and the updated stack pointer is written into register RA. The call stack pointer is typically kept in GPR(1) as per the EABI. For context switches due to interrupts, the programmer may choose to share the call stack by using GPR(1) or to create a separate machine stack using a different GPR.

The immediate field DD contains the number of doublewords, in 2's complement, of the stack frame being written and must be a negative number specifying at least 11 doublewords, one of which contains the stack frame header. The maximum stack frame size supported by this instruction is 32 kilobytes due to the maximum stack pointer displacement that can be represented. The full context data contains all PPE-defined GPRs except R2 and R13, and a subset of the PPE-defined SPRs: CR, SPRG0, XER, CTR, SRR0, and SRR1. If DD specifies more than 11 doublewords, extra room is left in the stack frame for subsequent instructions to store additional context that needs to be preserved.

The **stcxu** instruction may be emulated by the following sequence of instructions, where ordering is important in case an interrupt occurs during the sequence and D is the instruction DD field multiplied by 8:

```
mflr Rx             # Get Link register
stwu RS,D(RA)       # Save Back chain and move SP atomically
stw  Rx,4-D(RA)     # Save Link register
```

```
stvd CONTEXT[0..9],-8..80(RS) # Save 10 doublewords of context
```

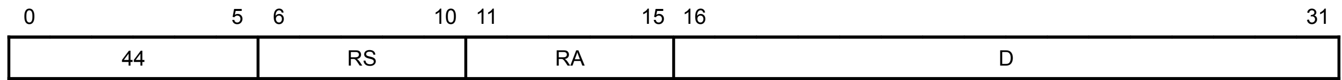
The specified PPE context is written (“pushed”) onto the stack as follows:

Stack	Address	Size (bytes)	New Content Written by stcxu	Old Content
Previous Stack Data (if any)	X	X	[unmodified]	Previous Stack Frames
Previous Stack Frame Header	SP+4	4	LR	[next LR Placeholder]
	Stack Pointer SP = (RA)	4	[unmodified] Back Pointer	Back Pointer [Previous SP]
New Stack Frame Data	SP-8	8	R31 R30	[Undefined]
	SP-16	8	R29 R28	
	SP-24	8	SRR1 SRR0	
	SP-32	8	CTR XER	
	SP-40	8	R10 R9	
	SP-48	8	R8 R7	
	SP-56	8	R6 R5	
	SP-64	8	R4 R3	
	SP-72	8	R1 R0	
	SP-80	8	SPRG0 CR	
	optional, if D>88 SP-88 ... SP + D + 8	D-88	[unmodified] Variable Context placeholder	
New Stack Frame Header	SP + D + 4	4	[unmodified] next LR placeholder	
	SP + D (written into RA)	4	Back Pointer = (RS)	

9.4.86 sth

Store Halfword

sth RS, D(RA)



0–5	6–10	11–15	16–31
44	RS	RA	D

EA ← (RA | 0) + EXTS(D)
MS(EA, 2) ← $(RS)_{16:31}$

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

Bits 16:31 of register RS are stored into the halfword addressed by the EA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

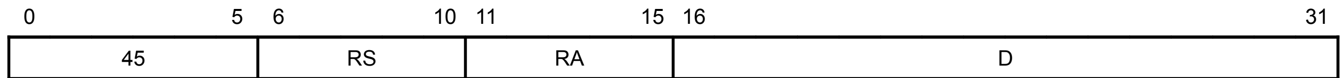
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.87 sthu

Store Halfword with Update

sthu RS, D(RA)



$EA \leftarrow (RA) + EXTS(D)$
$MS(EA, 2) \leftarrow (RS)_{16:31}$
$(RA) \leftarrow EA$

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits.

Bits 16:31 of register RS are stored into the halfword addressed by the EA.

The EA is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

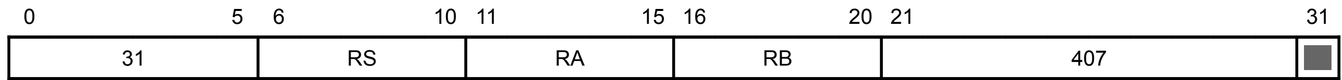
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.88 sthx

Store Halfword Indexed

sthx RS, RA, RB



EA ← (RA | 0) + (RB)
MS(EA, 2) ← $(RS)_{16:31}$

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

Bits 16:31 of register RS are stored into the halfword addressed by the EA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

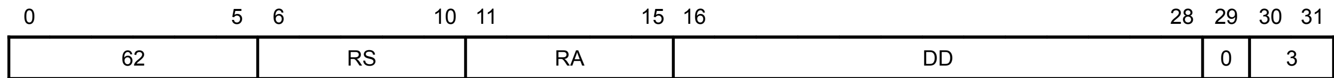
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.89 stsku

Store Stack frame with Update

stsku RS, DD(RA)



EA ← (RA)
MS(EA + 4, 4) ← (LR)
(EDR) ← (RA)

n ← ¬ DD
if n > 0 then
    MS(EA - 8), 8) ← (VDR30)
if n > 1 then
    MS(EA -16, 8) ← (VDR28)

EA ← (RA) + EXTS(DD || ^{3}0)
MS(EA, 4) ← (RS)
(RA) ← EA

An EA is formed by the contents of register RA. The contents of the Link Register are stored into the word at (EA + 4). The number of doublewords to store *n* is determined by the one's complement of the DD field. if n is greater than zero, VDR30 is stored into the doubleword at (EA - 8). If n is greater than one, VDR28 is stored into the doubleword at (EA -16).

A second EA is formed by adding a displacement to the base address in register RA. The 32-bit displacement is obtained by sign-extending the 13-bit DD field to 32 bits after concatenating 3 0-bits on its right. The contents of register RS are stored into the word addressed by this second EA. This EA is then placed into register RA.

Restrictions

The registers RS and RA must be the same and defined in the PPE 42 core architecture, the displacement must be a negative number, and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- EDR

Invalid Instruction Forms

- RA = 0
- RA ≠ RS
- DD_0 = '0'

Exceptions

An alignment exception is signaled if the address in register RA is not doubleword aligned, regardless if an unaligned operation is supported by the memory subsystem.

Synchronous exceptions caused by the execution of this instruction or a previously executed imprecise store must suppress the update of RA. The EDR reflects the address of the word or doubleword being accessed when the memory subsystem raised this exception. Only memory locations accessed before this exception was signaled are updated.

Since this instruction is storage synchronizing, asynchronous exceptions that occur during the execution of this instruction are held pending until all stores generated by this instruction have been completed by the memory subsystem.

Architecture Notes

This instruction is implemented by PPE 42X, is specific to the PPE architecture, and is not part of the original PPE 42 Instruction Set. This instruction is a modified DS-Form, where DS is defined as DD || 0, and is storage synchronizing. The similar **stcxu** instruction provides a superset of this function.

The EDR is used as a “scratch pad” to store the current Stack Pointer address during execution of this instruction and will be overwritten if this instruction experiences a precise or imprecise synchronous exception.

Programming Note

Although the DD field specifies the number of doublewords, assemblers should instead support the specification of a byte offset for programming consistency with other instruction forms, discarding the three rightmost bits of the offset when forming the DD field of the assembled instruction.

Usage Notes

This instruction is intended to be used to *push* a subset of the PPE's GPR and SPR context onto a stack in memory, in a format compatible with the PowerPC EABI used for the call stack, with a single instruction and to be used in concert with the corresponding **lsku** instruction. To ensure the stack pointer update is an atomic operation that cannot be interrupted, a store-with-update instruction form is used. Note that this instruction also populates the Back Pointer word located in memory at the updated stack pointer address to facilitate debug as per the EABI.

Registers RS and RA must point to the same GPR, which contains the current stack pointer, and the updated stack pointer is written into register RA. The call stack pointer is typically kept in GPR(1) as per the EABI.

The immediate field DD contains the number of doublewords, in 2's complement, of the stack frame being written and must be a negative number specifying at least 1 doubleword containing the stack frame header. The maximum stack frame size supported by this instruction is 32 kilobytes due to the maximum stack pointer displacement that can be represented. The stack data optionally contains VDR30 and in addition VDR28. If DD specifies more than 3 doublewords, extra room is left in the stack frame for subsequent instructions to store additional context that needs to be preserved.

The **stsku** instruction may be emulated by the following sequence of instructions, where ordering is important in case an interrupt occurs during the sequence and D is the instruction DD field multiplied by 8:

```
mflr Rx             # Get Link register
stwu RS,D(RA)       # Save Back chain and move SP atomically
stw  Rx,4-D(RA)     # Save Link register
stvd CONTEXT[0..1],-8..16(RS) # Save 0,1,2 doublewords of context
```

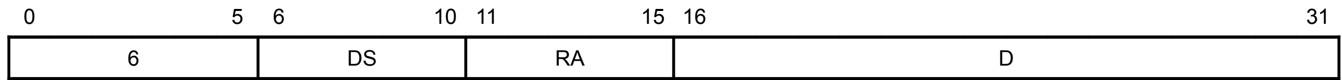
The specified PPE context is written ("pushed") onto the stack as follows:

Stack	Address	Size (bytes)	New Content Written by stsku	Old Content
Previous Stack Data (if any)	X	X	[unmodified]	Previous Stack Frames
Previous Stack Frame Header	SP+4	4	LR	[next LR Placeholder]
	Stack Pointer SP = (RA)	4	[unmodified] Back Pointer	Back Pointer [Previous SP]
New Stack Frame Data	SP-8 (if D>8)	8	R31 R30	[Undefined]
	SP-16 (if D>16)	8	R29 R28	
	optional, if D>24 SP-24 ... SP + D + 8	D-24	[unmodified] Variable Context placeholder	
New Stack Frame Header	SP + D + 4	4	[unmodified] next LR placeholder	
	SP + D (written into RA)	4	Back Pointer = (RS)	

9.4.90 stvd

Store Virtual Doubleword

stvd DS, D(RA)



0 5	6 10	11 15	16 31
6	DS	RA	D

EA ← (RA | 0) + EXTS(D)
MS(EA, 8) ← (DS)

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The contents of virtual doubleword register DS are stored into the doubleword at the EA.

Restrictions

The registers DS and RA must be defined in the PPE 42 core architecture, and DS must specify a valid virtual doubleword register, otherwise an illegal instruction exception occurs.

Registers Altered

- None

Architecture Note

This instruction is specific to the PPE 42 architecture.

9.4.91 stvdu

Store Virtual Doubleword with Update

stvdu DS, D(RA)

0 5	6 10	11 15	16 31
22	DS	RA	D

EA ← (RA) + EXTS(D)
MS(EA, 8) ← (DS)
(RA) ← EA

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits.

The contents of virtual doubleword register DS are stored into the doubleword at the EA.

The EA is placed into register RA.

Restrictions

The registers DS and RA must be defined in the PPE 42 core architecture, DS must specify a valid virtual doubleword register and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

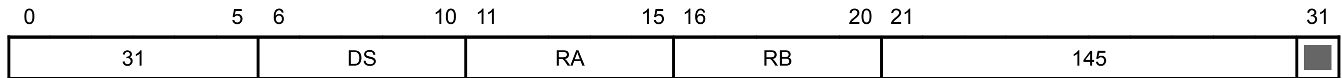
Architecture Note

This instruction is specific to the PPE 42 architecture.

9.4.92 stvdx

Store Virtual Doubleword Indexed

stvdx DS, RA, RB



EA ← (RA | 0) + (RB)
MS(EA, 8) ← (DS)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The contents of virtual doubleword register DS are stored into the doubleword at the EA.

Restrictions

The registers DS, RA and RB must be defined in the PPE 42 core architecture, and DS must specify a valid virtual doubleword register, otherwise an illegal instruction exception occurs.

Registers Altered

- None

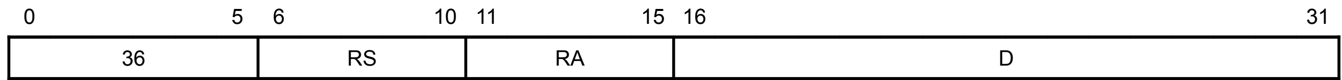
Architecture Note

This instruction is specific to the PPE 42 architecture.

9.4.93 stw

Store Word

stw RS, D(RA)



EA ← (RA | 0) + EXTS(D)
MS(EA, 4) ← (RS)

An EA is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The contents of register RS are stored at the EA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

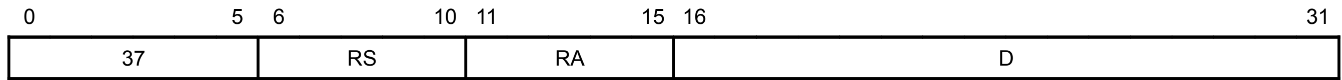
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.94 stwu

Store Word with Update

stwu RS, D(RA)



EA ← (RA) + EXTS(D)
MS(EA, 4) ← (RS)
(RA) ← EA

An EA is formed by adding a displacement to the base address in register RA. The displacement is obtained by sign-extending the 16-bit D field to 32 bits.

The contents of register RS are stored into the word at the EA.

The EA is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture and the instruction form must be valid (see below), otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

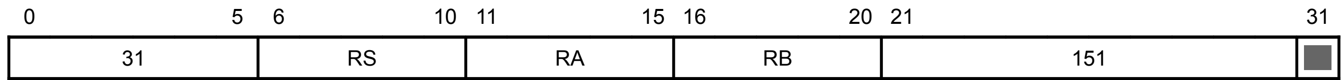
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.95 stwx

Store Word Indexed

stwx RS, RA, RB



EA ← (RA | 0) + (RB)
MS(EA, 4) ← (RS)

An EA is formed by adding an index to the base address in register RA. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of RA otherwise.

The contents of register RS are stored into the word at the EA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- None

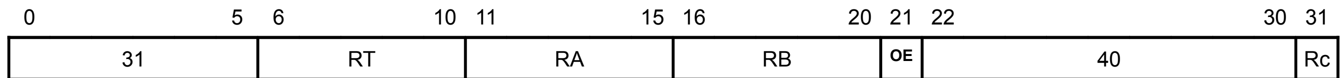
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.96 subf

Subtract From

subf	RT, RA, RB	OE = '0', Rc = '0'
subf.	RT, RA, RB	OE = '0', Rc = '1'
subfo	RT, RA, RB	OE = '1', Rc = '0'
subfo.	RT, RA, RB	OE = '1', Rc = '1'



(RT) ← ¬(RA) + (RB) + 1

The sum of the ones complement of register RA, register RB and 1 is placed into register RT.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

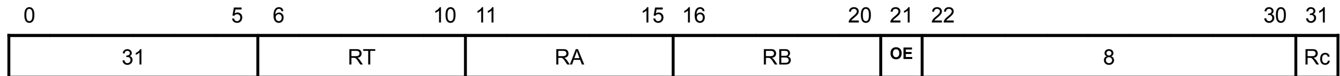
The *Subtract From* instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more “normal” order, in which the third operand is subtracted from the second. The mnemonic can be coded with a final “o” and/or “.” to cause the OE and/or Rc bit to be set in the underlying instruction.

sub[o][.] Rx,Ry,Rz (equivalent to: **subf[o][.] Rx,Rz,Ry**)

9.4.97 subfc

Subtract From Carrying

subfc	RT, RA, RB	OE = '0', Rc = '0'
subfc.	RT, RA, RB	OE = '0', Rc = '1'
subfco	RT, RA, RB	OE = '1', Rc = '0'
subfco.	RT, RA, RB	OE = '1', Rc = '1'



(RT) ← ¬(RA) + (RB) + 1

if ¬(RA) + (RB) + 1 $>^{u}$ 2^{32} – 1 then
    XER[CA] ← 1
else
    XER[CA] ← 0

The sum of the ones complement of register RA, register RB and 1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

The *Subtract From* instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more “normal” order, in which the third operand is subtracted from the second. The mnemonic can be coded with a final “o” and/or “.” to cause the OE and/or Rc bit to be set in the underlying instruction.

subc[o][.] Rx,Ry,Rz (equivalent to: **subfc[o][.] Rx,Rz,Ry**)

9.4.98 subfe

Subtract From Extended

subfe	RT, RA, RB	OE = '0', Rc = '0'
subfe.	RT, RA, RB	OE = '0', Rc = '1'
subfeo	RT, RA, RB	OE = '1', Rc = '0'
subfeo.	RT, RA, RB	OE = '1', Rc = '1'

0 – 5	6 – 10	11 – 15	16 – 20	21	22 – 30	31
31	RT	RA	RB	OE	136	Rc

(RT) ← ¬(RA) + (RB) + XER[CA]

if ¬(RA) + (RB) + XER[CA] $>^u 2^{32} - 1$ then

XER[CA] ← 1

else

XER[CA] ← 0

The sum of the ones complement of register RA, register RB and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Restrictions

The registers RT, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

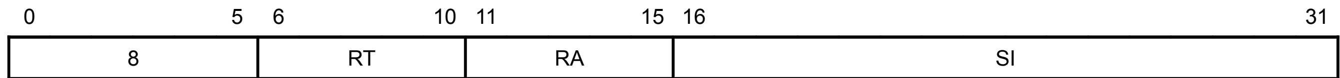
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.99 subfic

Subtract From Immediate Carrying

subfic RT, RA, SI



0 5	6 10	11 15	16 31
8	RT	RA	SI

(RT) ← ¬(RA) + EXTS(SI) + 1

If ¬(RA) + EXTS(SI) + 1 $>^u$ $2^{32} - 1$ then
    XER[CA] ← 1
else
    XER[CA] ← 0

The sum of the ones complement of RA, the contents of the SI field sign-extended to 32 bits, and 1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

Extended Mnemonics

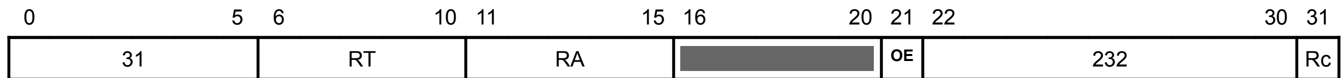
PPE 42 does not implement the Power ISA **neg** (negate) instruction, however the assembler provides a negate-with-carry extended mnemonic.

negc RT, RA (equivalent to **subfic RT, RA, 0**)

9.4.100 subfme

Subtract From Minus One Extended

subfme	RT, RA	OE = '0', Rc = '0'
subfme.	RT, RA	OE = '0', Rc = '1'
subfmeo	RT, RA	OE = '1', Rc = '0'
subfmeo.	RT, RA	OE = '1', Rc = '1'



0–5	6–10	11–15	16–20	21	22–30	31
31	RT	RA		OE	232	Rc

(RT) ← ¬(RA) + XER[CA] + (-1)

if ¬(RA) + XER[CA] + (-1) $>^u$ 2^{32} – 1 then
XER[CA] ← 1
else
XER[CA] ← 0

The sum of the ones complement of register RA, XER[CA] and -1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

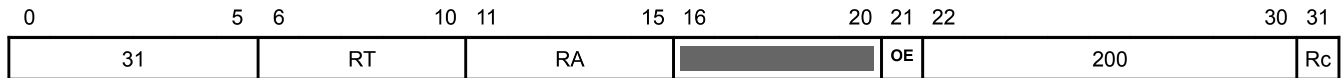
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.101 subfze

Subtract From Zero Extended

subfze	RT, RA	OE = '0', Rc = '0'
subfze.	RT, RA	OE = '0', Rc = '1'
subfzeo	RT, RA	OE = '1', Rc = '0'
subfzeo.	RT, RA	OE = '1', Rc = '1'



(RT) ← ¬(RA) + XER[CA]

if ¬(RA) + XER[CA] $>^u$ $2^{32} - 1$ then
    XER[CA] ← 1
else
    XER[CA] ← 0

The sum of the ones complement of register RA and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Restrictions

The registers RT and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RT
- XER[CA]
- CR[CR0]LT, GT, EQ, SO if Rc contains a ‘1’
- XER[SO, OV] if OE contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.102 sync

Synchronize

sync

The **sync** instruction guarantees that all instructions initiated by the processor preceding the **sync** instruction are completed before the **sync** instruction is completed, and that no subsequent instructions are initiated by the processor until after the **sync** instruction is completed. When the **sync** instruction is completed, all storage accesses that were initiated by the processor before the **sync** instruction will have been completed with respect to all mechanisms that access storage.

Registers Altered

- None

Architecture Note

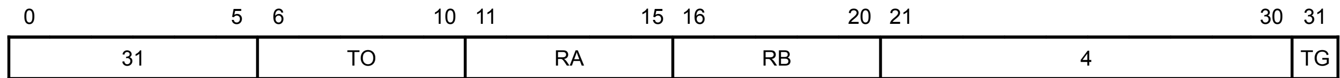
This instruction is part of the Power ISA Virtual Environment Architecture. PPE 42 does not support any the options provided by the Power ISA instruction controlling the type of barrier provided by **sync.**

The **sync** instruction is both context synchronizing and storage synchronizing under all conditions, and is the only way to guarantee context synchronization while ramming. For more information on ramming and single-stepping semantics see section 7, *Debugging.*

9.4.103 tw

Trap Word

tw TO,RA,RB



The **tw** instruction causes either a program exception or a debug event, that is either treated as a no-op or halts the processor. The behavior of **tw** is governed by both the TO and TG fields as well as the current setting of DBCR[TRAP] as shown in the table below.

Table 1.96: Behavior of the **tw** instruction

TO	DBCR[TRAP]	TG	Behavior
0	xx	0	The **tw** is treated as a no-op for code marking and debug.
(TO ≠ 0) ∧ (TO ≠31)	xx	0	The **tw** causes a program exception.
31	0	0	
31	1	0	The **tw** causes the processor to halt and sets XSR[TRAP] ← '1'.
x	xx	1	On PPE42, the **tw** behaves the same as TG=0 above. On PPE42X, **tw** is treated as a no-op for code marking and debug.

Restrictions

If TG = '0', the registers RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs. Also see the Architecture Note below. On PPE42X, if TG = '1' there are no restrictions.

Registers Altered

None if PPE42X and TG = '1',
else:

- ISR (If TO = 31 and DBCR[TRAP] = '0' and the program interrupt is the highest priority interrupt.)
- EDR (If TO = 31 and DBCR[TRAP] = '0' and the program interrupt is the highest priority interrupt.)
- XSR (If TO = 31 and DBCR[TRAP] = '1').

Programming Note

This instruction is inserted into the execution stream by a debugger to implement breakpoints or trace markers, and is not typically used by application code. Using TO = '0' or TG = '1' allows the user to “mark” or “tag” regions of code during simulation or when collecting debug traces to collect information that can be consumed to analyze hardware test coverage or PPE code coverage. Collecting only these **mark** and **marktag** records in the PPE debug trace allows for collection of longer traces by effectively compressing trace information into the just the code flow that led to the scenario of interest.

Architecture Note

The PPE 42 **tw** instruction only implements a subset of the instruction functionality defined in the Power ISA User Instruction Set Architecture. The behavior of the PPE 42 **tw** is generally consistent with the Power

ISA Book III-E architecture, although PPE 42 implements a modification of the Book III-E debugging infrastructure.

On PPE 42, bit 31 (now named TG) was reserved. When TG=0, PPE 42 & 42X only implements the unconditional form of **tw** (all instruction bits 6:10 set to all '0' or all '1'). Any other form of the **tw** instruction will cause an illegal instruction exception. For consistency and future compatibility PPE 42 requires that RA and RB denote valid PPE 42 GPR numbers, even though the GPR contents have no effect on the behavior of the instruction. **tw** instructions with invalid RA or RB fields are reported as illegal instructions, not as **tw** instructions.

On PPE 42X, when TG = 1 is set, the RA and RB fields may contain any value, i.e. RA and RB are not required to denote valid GPR numbers.

Extended Mnemonics

Unconditional debug trap instruction, to insert a breakpoint as defined by DBCR[TRAP]:

trap (equivalent to: **tw 31, 0, 0** with **TG**=0)

twu X, Y (equivalent to: **tw 31, X, Y** with **TG**=0)

where X and Y are valid GPR numbers, used to provide encoded information in the EDR for code workarounds and facilitate debug of detected error conditions when the code halts the PPE.

Unconditional debug mark instruction, executed as a no-op that puts a marker into the debug trace:

mark X, Y (equivalent to: **tw 0, X, Y** with **TG**=0)

where X and Y are valid GPR numbers, used to provide encoded information in the instruction stream for debug or code flow analysis.

Unconditional debug mark tag instruction, executed as a no-op that puts a marker into the debug trace:

marktag A (equivalent to: **tw (A/1024), (A/32)^x'1F', A^x'1F'** with **TG**=1)

similar to **mark** except it supports only a single parameter, which can specify a value from 0 to x'7FFF', used to provide encoded information in the instruction stream for debug or code flow analysis.

9.4.104 wrtee

Write External Enable

wrtee RS

$(MSR)_{16} \leftarrow (RS)_{16}$

If the processor is not halted then
    If a pending interrupt is enabled then
        NIA ← Address of highest priority interrupt
        $(SRR0)_{0:29} \leftarrow ((CIA) + 4)_{0:29}$
        (SRR1) ← (MSR)
        (MSR) ← (MSR) modified by the taking of the interrupt

The MSR[EE] (bit 16) is set to the value of bit 16 of register RS. If the processor is halted, that is, if the **wrtee** instruction is being single-stepped or rammed, then no other state change occurs (other than the normal update of the IAR during single-stepping). The following description covers the case that the processor is not halted.

If the new MSR value enables one or more pending interrupts, then the highest priority interrupt is taken. The value placed into SRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address of the instruction following **wrtee**), the MSR is placed into SRR1 and the MSR is then modified by the taking of the interrupt as described in section 4, *Interrupts and Exceptions.*

Note that if the processor executes **wrtee,** then by definition MSR[WE] is not set and the processor continues executing instructions.

Restrictions

The register RS must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- MSR

Architecture Note

This instruction is part of the Power ISA Operating Architecture Environment – Embedded (Book III-E).

The PPE 42 architecture guarantees that **wrtee** is fully executed prior to fetching the subsequent instruction. This means that if the effect of **wrtee** is to unmask an interrupt, the next instruction executed after the **wrtee** will be the instruction at the associated interrupt vector address.

Programming Note

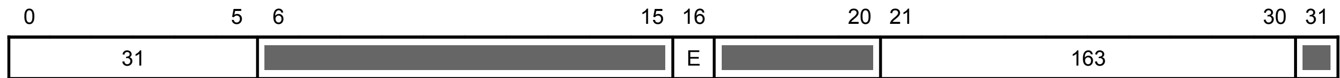
This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

```
mfmsr Rn # Save EE in Rn[16]
wrteei 0 # Turn off EE
•        # Code with EE disabled
•
•
wrtee Rn #Restore EE without affecting any MSR changes that occurred in
         #the disabled code
```

9.4.105 wrteei

Write External Enable Immediate

wrteei E



$(MSR)_{16} \leftarrow E$

If the processor is not halted then

If a pending interrupt is enabled then

NIA ← Address of highest priority interrupt

$(SRR0)_{0:29} \leftarrow ((CIA) + 4)_{0:29}$

(SRR1) ← (MSR)

(MSR) ← (MSR) modified by the taking of the interrupt

The MSR[EE] (bit 16) is set to the value of bit 16 of the instruction. If the processor is halted, that is, if the **wrteei** instruction is being single-stepped or rammed, then no other state change occurs. The following description covers the case that the processor is not halted.

If the new MSR value enables one or more pending interrupts, then the highest priority interrupt is taken. The value placed into SRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address of the instruction following **wrteei**), the MSR is placed into SRR1 and the MSR is then modified by the taking of the interrupt as described in section 4, *Interrupts and Exceptions.*

Note that if the processor executes **wrteei,** then by definition MSR[WE] is not set and the processor continues executing instructions.

Registers Altered

- MSR

Architecture Note

This instruction is part of the Power ISA Operating Architecture Environment – Embedded (Book III-E).

The PPE 42 architecture guarantees that **wrteei** is fully executed prior to fetching the subsequent instruction. This means that if the effect of **wrteei** is to unmask an interrupt, the next instruction executed after the **wrteei** will be the instruction at the associated interrupt vector address.

Programming Note

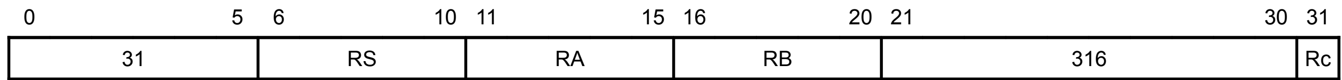
This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

```
mfmsr Rn # Save EE in Rn[16]
wrteei 0 # Turn off EE
•        # Code with EE disabled
•
•
wrtee Rn #Restore EE without affecting any MSR changes that occurred in
         #the disabled code
```

9.4.106 xor

XOR

xor	RA, RS, RB	Rc = '0'
xor.	RA, RS, RB	Rc = '1'



0 — 5	6 — 10	11 — 15	16 — 20	21 — 30	31
31	RS	RA	RB	316	Rc

$(RA) \leftarrow (RS) \oplus (RB)$

The contents of register RS are XORed with the contents of register RB. The result is placed into register RA.

Restrictions

The registers RS, RA and RB must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA
- CR[CR0]$_{LT, GT, EQ, SO}$ if Rc contains a ‘1’

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.107 xori

XOR Immediate

xori RA, RS, UI

0 5	6 10	11 15	16 31
26	RS	RA	UI

$(RA) \leftarrow (RS) \oplus ({}^{16}0 \| UI)$

The UI field is extended to 32 bits by concatenating 16 0-bits on its left. The contents of register RS are XORed with the extended UI field. The result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA

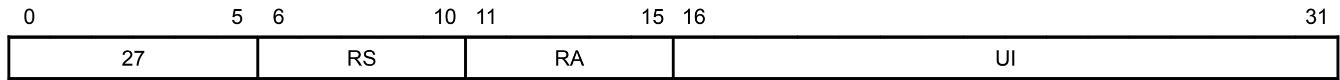
Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.4.108 xoris

XOR Immediate Shifted

xoris RA, RS, UI



0 5	6 10	11 15	16 31
27	RS	RA	UI

$(RA) \leftarrow (RS) \oplus (UI \,||\, ^{16}0)$

The UI field is extended to 32 bits by concatenating 16 0-bits on its right. The contents of register RS are XORed with the extended UI field. The result is placed into register RA.

Restrictions

The registers RS and RA must be defined in the PPE 42 core architecture, otherwise an illegal instruction exception occurs.

Registers Altered

- RA

Architecture Note

This instruction is part of the Power ISA User Instruction Set Architecture.

9.5 Instruction Set Mnemonics List

The following table contains the PPE 42 core instruction set, including required extended mnemonics. A short functional description is included for each mnemonic, as well as the instruction operands and notation. All mnemonics are listed alphabetically, without regard to whether the mnemonic is realized in hardware or software. When an instruction supports multiple hardware mnemonics (for example, **b**, **ba**, **bl**, **bla** are all forms of **b**), the instruction is alphabetized under the root form. The hardware instructions are described in detail in Alphabetical Instruction Listing which is also alphabetized under the root form.

Table 1.97: PPE 42 Instruction Syntax Summary

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		
add.			CR[CR0]	
addo			XER[SO, OV]	
addo.			CR[CR0] XER[SO, OV]	
addc	RT, RA, RB	Add (RA) to (RB). Place result in RT. Place carry-out in XER[CA].		
addc.			CR[CR0]	
addco			XER[SO, OV]	
addco.			CR[CR0] XER[SO, OV]	
adde	RT, RA, RB	Add XER[CA], (RA), (RB). Place result in RT. Place carry-out in XER[CA].		
adde.			CR[CR0]	
addeo			XER[SO, OV]	
addeo.			CR[CR0] XER[SO, OV]	
addi	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT.		
addic	RT, RA, IM	Add EXTS(IM) to (RA|0). Place result in RT. Place carry-out in XER[CA].		
addic.			CR[CR0]	
addis	RT, RA, IM	Add (IM || $^{16}0$) to (RA|0). Place result in RT.		

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		
addme.			CR[CR0]	
addmeo			XER[SO, OV]	
addmeo.			CR[CR0] XER[SO, OV]	
addze	RT, RA	Add XER[CA] to (RA). Place result in RT. Place carry-out in XER[CA].		
addze.			CR[CR0]	
addzeo			XER[SO, OV]	
addzeo.			CR[CR0] XER[SO, OV]	
and	RA, RS, RB	AND (RS) with (RB). Place result in RA.		
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with ¬(RB). Place result in RA.		
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with ($^{16}0$ || IM). Place result in RA.	CR[CR0]	
andis.	RA, RS, IM	AND (RS) with (IM || $^{16}0$). Place result in RA.	CR[CR0]	
b	target	Branch unconditional relative. LI ← $(\text{target} - \text{CIA})_{6:29}$ NIA ← CIA + EXTS(LI || $^{2}0$)		
ba		Branch unconditional absolute. LI ← $\text{target}_{6:29}$ NIA ← EXTS(LI || $^{2}0$)		
bl		Branch unconditional relative. LI ← $(\text{target} - \text{CIA})_{6:29}$ NIA ← CIA + EXTS(LI || $^{2}0$)	(LR) ← CIA + 4.	
bla		Branch unconditional absolute. LI ← $\text{target}_{6:29}$ NIA ← EXTS(LI || $^{2}0$)	(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bc	BO, BI, target	Branch conditional relative. BD ← $(\text{target} - \text{CIA})_{16:29}$ NIA ← CIA + EXTS(BD || $^{2}0$)	CTR if BO_2 = '0'.	BI < 4 (CR0 only)
bca		Branch conditional absolute. BD ← $\text{target}_{16:29}$ NIA ← EXTS(BD || $^{2}0$)	CTR if BO_2 = '0'.	
bcl		Branch conditional relative. BD ← $(\text{target} - \text{CIA})_{16:29}$ NIA ← CIA + EXTS(BD || $^{2}0$)	CTR if BO_2 = '0'. (LR) ← CIA + 4.	
bcla		Branch conditional absolute. BD ← $\text{target}_{16:29}$ NIA ← EXTS(BD || $^{2}0$)	CTR if BO_2 = '0'. (LR) ← CIA + 4.	
bcctr	BO, BI	Branch conditional to address in CTR. Using (CTR) at exit from instruction, NIA ← $\text{CTR}_{0:29}$ || $^{2}0$.	CTR if BO_2 = '0'.	BI < 4 (CR0 only)
bcctrl			CTR if BO_2 = '0'. (LR) ← CIA + 4.	
bclr	BO, BI	Branch conditional to address in LR. Using (LR) at entry to instruction, NIA ← $\text{LR}_{0:29}$ || $^{2}0$.	CTR if BO_2 = '0'.	BI < 4 (CR0 only)
bclrl			CTR if BO_2 = '0'. (LR) ← CIA + 4.	
bctr	BO, BI	Branch unconditionally to address in CTR. Extended mnemonic for **bcctr** 20,0		
bctrl		Extended mnemonic for **bcctrl** 20,0	(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bdnz	target	Decrement CTR. Branch if CTR ≠ ‘0’. Extended mnemonic for **bc** 16,0,target		
bdnza		Extended mnemonic for **bca** 16,0,target		
bdnzl		Extended mnemonic for **bcl** 16,0,target	(LR) ← CIA + 4.	
bdnzla		Extended mnemonic for **bcla** 16,0,target	(LR) ← CIA + 4.	
bdnzlr	target	Decrement CTR. Branch if CTR ≠ ‘0’ to address in LR. Extended mnemonic for **bclr** 16,0		
bdnzlrl		Extended mnemonic for **bclrl** 16,0	(LR) ← CIA + 4.	
bdnzf **bdnzfa** **bdnzfl** **bdnzfla** **bdnzflr** **bdnzflrl** **bdnzt** **bdnzta** **bdnztl** **bdnztla** **bdnztlr** **bdnztlrl**	cr_bit	NOT REQUIRED Not generally useful extended mnemonic.		cr_bit < 4 (CR0 only)
bdz	target	Decrement CTR. Branch if CTR = ‘0’. Extended mnemonic for **bc** 18,0,target		
bdza		Extended mnemonic for **bca** 18,0,target		
bdzl		Extended mnemonic for **bcl** 18,0,target	(LR) ← CIA + 4.	
bdzla		Extended mnemonic for **bcla** 18,0,target	(LR) ← CIA + 4.	
bdzlr	target	Decrement CTR. Branch if CTR = ‘0’ to address in LR. Extended mnemonic for **bclr** 18,0		
bdzlrl		Extended mnemonic for **bclrl** 18,0	(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bdzf **bdzfa** **bdzfl** **bdzfla** **bdzflr** **bdzflrl** **bdzt** **bdzta** **bdztl** **bdztla** **bdztlr** **bdztlrl**	cr_bit	NOT REQUIRED Not generally useful extended mnemonic.		cr_bit < 4 (CR0 only)
beq	[0,] target	Branch if equal. Extended mnemonic for **bc** 12,2,target		CR0 only
beqa		Extended mnemonic for **bca** 12,2,target		
beql		Extended mnemonic for **bcl** 12,2,target	(LR) ← CIA + 4.	
beqla		Extended mnemonic for **bcla** 12,2,target	(LR) ← CIA + 4.	
beqctr	[0]	Branch if equal to address in CTR. Extended mnemonic for **bcctr** 12,2		CR0 only
beqctrl		Extended mnemonic for **bcctrl** 12,2	(LR) ← CIA + 4.	
beqlr	[0]	Branch if equal to address in LR. Extended mnemonic for **bclr** 12,2		CR0 only
beqlrl		Extended mnemonic for **bclrl** 12,2	(LR) ← CIA + 4.	
bf **bfa** **bfl** **bfla** **bfctr** **bfctrl** **bflr** **bflrl**	cr_bit	NOT REQUIRED Not generally useful extended mnemonic.		cr_bit < 4 (CR0 only)

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bge	[0,] target	Branch if greater than or equal. Extended mnemonic for **bc** 4,0,target		CR0 only
bgea		Extended mnemonic for **bca** 4,0,target		
bgel		Extended mnemonic for **bcl** 4,40,target	(LR) ← CIA + 4.	
bgela		Extended mnemonic for **bcla** 4,0,target	(LR) ← CIA + 4.	
bgectr	[0]	Branch if greater than or equal to address in CTR. Extended mnemonic for **bcctr** 4,0		CR0 only
bgectrl		Extended mnemonic for **bcctrl** 4,0	(LR) ← CIA + 4.	
bgelr	[0]	Branch if greater than or equal to address in LR. Extended mnemonic for **bclr** 4,0		CR0 only
bgelrl		Extended mnemonic for **bclrl** 4, 0	(LR) ← CIA + 4.	
bgt	[0,] target	Branch if greater than. Extended mnemonic for **bc** 12,1,target		CR0 only
bgta		Extended mnemonic for **bca** 12,1,target		
bgtl		Extended mnemonic for **bcl** 12,1,target	(LR) ← CIA + 4.	
bgtla		Extended mnemonic for **bcla** 12,1,target	(LR) ← CIA + 4.	
bgtctr	[0]	Branch if greater than to address in CTR. Extended mnemonic for **bcctr** 12,1		CR0 only
bgtctrl		Extended mnemonic for **bcctrl** 12,1	(LR) ← CIA + 4.	
bgtlr	[0]	Branch if greater than to address in LR. Extended mnemonic for **bclr** 12,1		CR0 only
bgtlrl		Extended mnemonic for **bclrl** 12,1	(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
ble	[0,] target	Branch if less than or equal. Extended mnemonic for **bc** 4,1,target		CR0 only
blea		Extended mnemonic for **bca** 4,1,target		
blel		Extended mnemonic for **bcl** 4,1,target	(LR) ← CIA + 4.	
blela		Extended mnemonic for **bcla** 4,1,target	(LR) ← CIA + 4.	
blectr	[0]	Branch if less than or equal to address in CTR. Extended mnemonic for **bcctr** 4,1		CR0 only
blectrl		Extended mnemonic for **bcctrl** 4,1	(LR) ← CIA + 4.	
blelr	[0]	Branch if less than or equal to address in LR. Extended mnemonic for **bclr** 4,1		CR0 only
blelrl		Extended mnemonic for **bclrl** 4,1	(LR) ← CIA + 4.	
blr		Branch unconditionally to address in LR. Extended mnemonic for **bclr** 20,0		
blrl		Extended mnemonic for **bclrl** 20,0	(LR) ← CIA + 4.	
blt	[0,] target	Branch if less than. Extended mnemonic for **bc** 12,0,target		CR0 only
blta		Extended mnemonic for **bca** 12,0,target		
bltl		Extended mnemonic for **bcl** 12,0,target	(LR) ← CIA + 4.	
bltla		Extended mnemonic for **bcla** 12,0,target	(LR) ← CIA + 4.	
bltctr	[0]	Branch if less than to address in CTR. Extended mnemonic for **bcctr** 12,0		CR0 only
bltctrl		Extended mnemonic for **bcctrl** 12,0	(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bltlr	[0]	Branch if less than to address in LR. Extended mnemonic for **bclr** 12,0		CR0 only
bltlrl		Extended mnemonic for **bclrl** 12,0	(LR) ← CIA + 4.	
bne	[0,] target	Branch if not equal. Extended mnemonic for **bc** 4,2,target		CR0 only
bnea		Extended mnemonic for **bca** 4,2,target		
bnel		Extended mnemonic for **bcl** 4,2,target	(LR) ← CIA + 4.	
bnela		Extended mnemonic for **bcla** 4,2,target	(LR) ← CIA + 4.	
bnectr	[0]	Branch if not equal to address in CTR. Extended mnemonic for **bcctr** 4,2		CR0 only
bnectrl		Extended mnemonic for **bcctrl** 4,2	(LR) ← CIA + 4.	
bnelr	[0]	Branch if not equal to address in LR. Extended mnemonic for **bclr** 4,2		CR0 only
bnelrl		Extended mnemonic for **bclrl** 4,2	(LR) ← CIA + 4.	
bng	[0,] target	Branch if not greater than. Extended mnemonic for **bc** 4,1,target		CR0 only
bnga		Extended mnemonic for **bca** 4,1,target		
bngl		Extended mnemonic for **bcl** 4,1,target	(LR) ← CIA + 4.	
bngla		Extended mnemonic for **bcla** 4,1,target	(LR) ← CIA + 4.	
bngctr	[0]	Branch if not greater than to address in CTR. Extended mnemonic for **bcctr** 4,1		CR0 only
bngctrl		Extended mnemonic for **bcctrl** 4,1	(LR) ← CIA + 4.	

Mnemonic	Operands	Function	Other Registers Changed	Difference from PowerPC 405-S ISA*
bnglr	[0]	Branch if not greater than to address in LR. Extended mnemonic for **bclr** 4,1		CR0 only
bnglrl		Extended mnemonic for **bclrl** 4,1	(LR) ← CIA + 4.	
bnl	[0,] target	Branch if not less than. Extended mnemonic for **bc** 4,0,target		CR0 only
bnla		Extended mnemonic for **bca** 4,0,target		
bnll		Extended mnemonic for **bcl** 4,0,target	(LR) ← CIA + 4.	
bnlla		Extended mnemonic for **bcla** 4,0,target	(LR) ← CIA + 4.	
bnlctr	[0]	Branch if not less than to address in CTR. Extended mnemonic for **bcctr** 4,0		CR0 only
bnlctrl		Extended mnemonic for **bcctrl** 4,0	(LR) ← CIA + 4.	
bnllr	[0]	Branch if not less than to address in LR. Extended mnemonic for **bclr** 4,0		CR0 only
bnllrl		Extended mnemonic for **bclrl** 4,0	(LR) ← CIA + 4.	
bns	[0,] target	Branch if not summary overflow. Extended mnemonic for **bc** 4,3,target		CR0 only
bnsa		Extended mnemonic for **bca** 4,3,target		
bnsl		Extended mnemonic for **bcl** 4,3,target	(LR) ← CIA + 4.	
bnsla		Extended mnemonic for **bcla** 4,3,target	(LR) ← CIA + 4.	
bnsctr	[0]	Branch if not summary overflow to address in CTR. Extended mnemonic for **bcctr** 4,3		CR0 only
bnsctrl		Extended mnemonic for **bcctrl** 4,3	(LR) ← CIA + 4.	
bnslr	[0]	Branch if not summary overflow to address in LR. Extended mnemonic for **bclr** 4,3		CR0 only
bnslrl		Extended mnemonic for **bclrl** 4,3	(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bnu **bnua** **bnul** **bnula** **bnuctr** **bnuctrl** **bnulr** **bnulrl**	[0]	NOT REQUIRED Not generally useful extended mnemonic.		CR0 only
bso	[0,] target	Branch if summary overflow. Extended mnemonic for **bc** 12,3,target		CR0 only
bsoa		Extended mnemonic for **bca** 12,3,target		
bsol		Extended mnemonic for **bcl** 12,3,target	(LR) ← CIA + 4.	
bsola		Extended mnemonic for **bcla** 12,3,target	(LR) ← CIA + 4.	
bsoctr	[0]	Branch if summary overflow to address in CTR. Extended mnemonic for **bcctr** 12,3		CR0 only
bsoctrl		Extended mnemonic for **bcctrl** 12,3	(LR) ← CIA + 4.	
bsolr	[0]	Branch if summary overflow to address in LR. Extended mnemonic for **bclr** 12,3		CR0 only
bsolrl		Extended mnemonic for **bclrl** 12,3	(LR) ← CIA + 4.	
bt **bta** **btl** **btla** **btctr** **btctrl** **btlr** **btlrl**	[0]	NOT REQUIRED Not generally useful extended mnemonic.		cr_bit < 4 (CR0 only)
bb0w	RA, RB, target	Fused compare bit test and branch if not set. Branch if the bit in (RA) as selected by (RB) is zero. Extended mnemonic for **bnbw[l] 1, RA, RB, target**	CR0	New for PPE 42
bb0wl			CR0 (LR) ← CIA + 4.	
bb0wi	RA, BNX, target	Fused compare bit test and branch if not set. Branch if the bit in (RA) as selected by (RB) is zero. Extended mnemonic for **bnbwi[l] 1, RA, BNX, target**	CR0	New for PPE 42
bb0wil			CR0 (LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bb1w	RA, RB, target	Fused compare bit test and branch if set. Branch if the bit in (RA) as selected by (RB) is one. Extended mnemonic for **bnbw[l] 0, RA, RB, target**	CR0	New for PPE 42
bb1wl			CR0 (LR) ← CIA + 4.	
bb1wi	RA, BNX, target	Fused compare bit test and branch if set. Branch if the bit in (RA) as selected by BNX is one. Extended mnemonic for **bnbwi[l] 0, RA, BNX, target**	CR0	New for PPE 42
bb1wil			CR0 (LR) ← CIA + 4.	
bnbw	PX, RA, RB, target	Fused compare bit test (using register) and branch. Branch if the bit in (RA) as selected by (RB) is the *inverse* of the value provided by PX. CR[CR0] is updated as if the instruction being executed were **rlwinm. Rx, RA, 0, MB,** ME where MB = ME = $(RB)_{27:31}$, *without* the update of Rx.	CR0	New for PPE 42
bnbwl			CR0 (LR) ← CIA + 4.	
bnbwi	PX, RA, BNX, target	Fused compare bit test (using immediate) and branch. Branch if the bit in (RA) as selected by BNX is the *inverse* of the value provided by PX. CR[CR0] is updated as if the instruction being executed were **rlwinm. Rx, Ra, 0, BNX, BNX** *without* the update of Rx.	CR0	New for PPE 42
bnbwil			CR0 (LR) ← CIA + 4.	
bwgez	RA, target	Extended Mnemonics for fused compare and branch if RA is greater than or equal to zero. Equivalent to **cmpwibc[l] 0, 0, RA, 0, target**		New for PPE 42
bwgezl			(LR) ← CIA + 4.	
bwgtz	RA, target	Extended Mnemonics for fused compare and branch if RA is greater than zero. Equivalent to **cmpwibc[l] 1, 1, RA, 0, target**		New for PPE 42
bwgtzl			(LR) ← CIA + 4.	
bwlez	RA, target	Extended Mnemonics for fused compare and branch if RA is less than or equal to zero. Equivalent to **cmpwibc[l] 0, 1, RA, 0, target**		New for PPE 42
bwlezl			(LR) ← CIA + 4.	
bwltz	RA, target	Extended Mnemonics for fused compare and branch if RA is less than zero. Equivalent to **cmpwibc[l] 1, 0, RA, 0, target**		New for PPE 42
bwltzl			(LR) ← CIA + 4.	
bwnz	RA, target	Extended Mnemonics for fused compare and branch if RA is *not* equal to zero. Equivalent to **cmpwibc[l] 0, 2, RA, 0, target**		New for PPE 42
bwnzl			(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
bwz	RA, target	Extended Mnemonics for fused compare and branch if RA is equal to zero. Equivalent to **cmpwibc[l] 1, 2, RA, 0, target**		New for PPE 42
bwzl			(LR) ← CIA + 4.	
bun **buna** **bunl** **bunla** **bunctr** **bunctrl** **bunlr** **bunlrl**	[0]	NOT REQUIRED Not generally useful extended mnemonic.		CR0 only
clrbw.	RA, RB	Clear bit in RA whose index is contained in (RB). Note this is only available as a “dot-form” since CR always gets updated. Extended mnemonic for **clrbwbz RA, RB, $ + 4**	CR0	New for PPE 42
clrbwi.	RA, BNX	Clear bit in RA whose index is contained in BNX. Note this is only available as a “dot-form” since CR always gets updated. Extended mnemonic for **clrbwbz RA, RB, $ + 4**	CR0	New for PPE 42
clrbwbz	RA, RB, target	Fused clear bit and branch on zero result. Clear the bit in (RA) as selected by (RB). Branch if the result in (RA) = 0. Extended mnemonic for **clrbwbc 1, RA, RB, target**	CR0	New for PPE 42
clrbwbzl			CR0 (LR) ← CIA + 4.	
clrbwibz	RA, BNX, target	Fused clear bit and branch on zero result. Clear the bit in (RA) as selected by BNX. Branch if the result in (RA) = 0. Extended mnemonic for **clrbwibc 1, RA, BNX, target**	CR0	New for PPE 42
clrbwibzl			CR0 (LR) ← CIA + 4.	
clrbwbnz	RA, RB, target	Fused clear bit and branch on non-zero result. Clear the bit in (RA) as selected by (RB). Branch if the result in (RA) ≠ 0. Extended mnemonic for **clrbwbc 0, RA, RB, target**	CR0	New for PPE 42
clrbwbnzl			CR0 (LR) ← CIA + 4.	
clrbwibnz	RA, BNX, target	Fused clear bit and branch on non-zero result. Clear the bit in (RA) as selected by BNX. Branch if the result in (RA) ≠ 0. Extended mnemonic for **clrbwibc 0, RA, BNX, target**	CR0	New for PPE 42
clrbwibnzl			CR0 (LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
clrbwbc	PX, RA, RB, target	Fused clear bit and branch on result test. Clear the bit in (RA) as selected by (RB). Branch if the result in (RA) = 0 when PX=1. Branch if the result in (RA) ≠ 0 when PX=0. CR[CR0] is updated as if the instruction being executed were **rlwinm. RA, RB, 0, MB, ME** where MB = $((RB)_{27:31} + 1)$ % 31, and ME = $((RB)_{27:31} - 1)$ % 31.	CR0	New for PPE 42
clrbwbcl			CR0 (LR) ← CIA + 4.	
clrbwibc	PX, RA, BNX, target	Fused clear bit and branch on result test. Clear the bit in (RA) as selected by BNX. Branch if the result in (RA) = 0 when PX=1. Branch if the result in (RA) ≠ 0 when PX=0. CR[CR0] is updated as if the instruction being executed were **rlwinm. RA, RA, 0, (BNX + 1) % 32, (BNX – 1) % 32**.	CR0	New for PPE 42
clrbwibcl			CR0 (LR) ← CIA + 4.	
clrlwi	RA, RS, n	Clear left immediate. (n < 32) (RA)0:n−1 ← n0 Extended mnemonic for rlwinm RA,RS,0,n,31		
clrlwi.		Extended mnemonic for rlwinm. RA,RS,0,n,31	CR[CR0]	
clrlslwi	RA, RS, b, n	Clear left and shift left immediate. (n ≤ b < 32) (RA)b−n:31−n ← (RS)b:31 (RA)32−n:31 ← n0 (RA)0:b−n−1 ← b−n0 Extended mnemonic for rlwinm RA,RS,n,b−n,31−n		
clrlslwi.		Extended mnemonic for rlwinm. RA,RS,n,b−n,31−n	CR[CR0]	
clrrwi	RA, RS, n	Clear right immediate. (n < 32) (RA)32−n:31 ← n0 Extended mnemonic for rlwinm RA,RS,0,0,31−n		
clrrwi.		Extended mnemonic for rlwinm. RA,RS,0,0,31−n	CR[CR0]	
cmp	0, 0, RA, RB	Compare (RA) to (RB), signed. Results in CR0. Note: only word compare.		CR0 only
cmpi	0, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR0. Note: only word compare.		CR0 only
cmpl	0, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR0. Note: only word compare.		CR0 only
cmpli	0, 0, RA, IM	Compare (RA) to ($^{16}0$ || IM), unsigned. Results in CR0. Note: only word compare.		CR0 only

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
cmplw	[0,] RA, RB	Compare Logical Word. Extended mnemonic for cmpl 0,0,RA,RB		CR0 only
cmplwbc	PX, BIX, RA, RB, target	Fused Compare Logical Word and Branch Conditional. Compare (RA) to (RB), unsigned. Results in CR0 as per the **cmplw** instruction. If BIX=3, update (RA) ← (RA) – (RB) and branch based on equality of the operation (if PX = CR[$CR0_2$]). Otherwise, RA is unchanged and the BIX field specifies which bit of, and the PX field specifies the polarity of, the CR[CR0] bit used to determine if the branch is taken or untaken.		New for PPE 42
cmplwbcl			(LR) ← CIA + 4.	
cmplwblt **cmplwble** **cmplwbgt** **cmplwbge** **cmplwbeq** **cmplwbne**	RA, RB, target	Extended Mnemonics for fused compare logical word and branch, without modifying RA. The mnemonic determines the value of PX and BIX to indicate when the branch should be taken. **cmplwbc[l] PX, BIX, RA, RB, target**		New for PPE 42
cmplwbltl **cmplwblel** **cmplwbgtl** **cmplwbgel** **cmplwbeql** **cmplwbnel**			(LR) ← CIA + 4.	
cmplwi	[0,] RA, IM	Compare Logical Word Immediate. Extended mnemonic for **cmpli 0,0,RA,IM**		CR0 only
cmpw	[0,] RA, RB	Compare Word. Extended mnemonic for **cmp 0,0,RA,RB**		CR0 only
cmpwbc	PX, BIX, RA, RB, target	Fused Compare Word and Branch Conditional. Identical **to cmplwbc[l]** except with a *signed* compare. Note: if BIX=3, RA is updated.		New for PPE 42
cmpwbcl			(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
cmpwblt **cmpwble** **cmpwbgt** **cmpwbge** **cmpwbeq** **cmpwbne**	RA, RB, target	Extended Mnemonics for fused compare word and branch, without modifying RA. The mnemonic determines the value of PX and BIX to indicate when the branch should be taken. **cmpwbc[l] PX, BIX, RA, RB, target**		New for PPE 42
cmpwbltl **cmpwblel** **cmpwbgtl** **cmpwbgel** **cmpwbeql** **cmpwbnel**			(LR) ← CIA + 4.	
cmpwi	[0,] RA, IM	Compare Word Immediate. Extended mnemonic for **cmpi 0,0,RA,IM**		CR0 only
cmpwibc	PX, BIX, RA, UIX, target	Fused Compare Word Immediate and Branch Conditional. Identical **to cmpwbc[l]** except with an *immediate* compare value. Note: if BIX=3, RA is updated.		New for PPE 42
cmpwibcl			(LR) ← CIA + 4.	
cmpwiblt **cmpwible** **cmpwibgt** **cmpwibge** **cmpwibeq** **cmpwibne**	RA, UIX, target	Extended Mnemonics for fused compare word immediate and branch, without modifying RA. The mnemonic determines the value of PX and BIX to indicate when the branch should be taken. **cmpwibc[l] PX, BIX, RA, RB, target**		New for PPE 42
cmpwibltl **cmpwiblel** **cmpwibgtl** **cmpwibgel** **cmpwibeql** **cmpwibnel**			(LR) ← CIA + 4.	
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		
cntlzw.			CR[CR0]	
crand **crandc** **crclr** **creqv** **crmove** **crnand** **crnor** **crnot** **cror** **crorc** **crset** **crxor**		NOT IMPLEMENTED No CR-logical operations (CR0 only)		X

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
dcba		NOT IMPLEMENTED		X
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address (RA|0) + (RB).		
dcbi	RA, RB	Invalidate the data cache block which contains the effective address (RA|0) + (RB).		
dcbq	RA, RB	Query the data cache block which contains the effective address (RA|0) + (RB).		NEW for PPE 42
dcbst		NOT IMPLEMENTED		X
dcbt	RA, RB	Load the data cache block which contains the effective address (RA|0) + (RB).		
dcbtst		NOT IMPLEMENTED		X
dcbz	RA, RB	Zero the data cache block which contains the effective address (RA|0) + (RB).		
dccci		NOT IMPLEMENTED		X
dcread		NOT IMPLEMENTED PPE 42 provides dcbq instead.		X
divw **divw.** **divwo** **divwo.** **divwu** **divwu.** **divwuo** **divwuo.**		NOT IMPLEMENTED		X
eieio		NOT IMPLEMENTED Emulate with sync instruction		E
eqv	RA, RS, RB	Equivalence of (RS) with (RB). $(RA) \leftarrow \neg((RS) \oplus (RB))$		
eqv.			CR[CR0]	
extlwi	RA, RS, n, b	Extract and left justify immediate. (n > 0) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow {}^{32-n}0$ Extended mnemonic for rlwinm RA,RS,b,0,n−1		
extlwi.		Extended mnemonic for rlwinm. RA,RS,b,0,n−1	CR[CR0]	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
extrwi	RA, RS, n, b	Extract and right justify immediate. (n > 0) (RA)32−n:31 ← (RS)b:b+n−1 (RA)0:31−n ← 32−n0 Extended mnemonic for rlwinm RA,RS,b+n,32−n,31		
extrwi.		Extended mnemonic for rlwinm. RA,RS,b+n,32−n,31	CR[CR0]	
extsb	RA, RS	Extend the sign of byte $(RS)_{24:31}$. Place the result in RA.		
extsb.			CR[CR0]	
extsh	RA, RS	Extend the sign of halfword $(RS)_{16:31}$. Place the result in RA.		
extsh.			CR[CR0]	
icbi **icbt** **iccci** **icread**		NOT IMPLEMENTED PPE 42 usage does not need instruction cache management.		X
inslwi	RA, RS, n, b	Insert from left immediate. (n > 0) (RA)b:b+n−1 ← (RS)0:n−1 Extended mnemonic for rlwimi RA,RS,32−b,b,b+n−1		
inslwi.		Extended mnemonic for rlwimi. RA,RS,32−b,b,b+n−1	CR[CR0]	
insrwi	RA, RS, n, b	Insert from right immediate. (n > 0) (RA)b:b+n−1 ← (RS)32−n:31 Extended mnemonic for rlwimi RA,RS,32−b−n,b,b+n−1		
insrwi.		Extended mnemonic for rlwimi. RA,RS,32−b−n,b,b+n−1	CR[CR0]	
isync		NOT IMPLEMENTED Not needed since PPE 42 is not superscalar or pipelined		X
la	RT, D(RA)	Load address. (RA ≠ ‘0’) D is an offset from a base address that is assumed to be (RA). (RT) ← (RA) + EXTS(D) Extended mnemonic for addi RT,RA,D		
lbz	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← $^{24}0$ || MS(EA,1).		

Mnemonic	Operands	Function	Other Registers Changed	Difference from PowerPC 405-S ISA*
lbzu	RT, D(RA)	Load byte from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← ^{24}0 || MS(EA,1). Update the base address, (RA) ← EA.		
lbzux		NOT IMPLEMENTED Emulate with addi followed by lbzx		E
lbzx	RT, RA, RB	Load byte from EA = (RA|0) + (RB) and pad left with zeroes, (RT) ← ^{24}0 || MS(EA,1).		
lcxu	RT, D(RA)	PPE 42: NOT IMPLEMENTED PPE 42X: Load word from EA = (RA) + EXTS(D) and place in EDR. Load 10 doublewords from memory between (RA) and EA into VDR(30), VDR(28), (SRR0 || SRR1, XER || CTR, VDR(9,7,5,3,0), CR || SPRG0). Load word from EA+4 into LR. Update the base address, (RA) ← EA. Extended mnemonic for lsku RT, D V 4(RA)		Not supported by PPE 42. New for PPE 42X
lha **lhau** **lhaux** **lhax**		NOT IMPLEMENTED Emulate with lhz form followed by extsh		E
lhbrx		NOT IMPLEMENTED		X
lhz	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← ^{16}0 || MS(EA,2).		
lhzu	RT, D(RA)	Load halfword from EA = (RA|0) + EXTS(D) and pad left with zeroes, (RT) ← ^{16}0 || MS(EA,2). Update the base address, (RA) ← EA.		
lhzux		NOT IMPLEMENTED Emulate with addi followed by lhzx		E
lhzx	RT, RA, RB	Load halfword from EA = (RA|0) + (RB) and pad left with zeroes, (RT) ← ^{16}0 || MS(EA,2).		
li	RT, IM	Load immediate. (RT) ← EXTS(IM) Extended mnemonic for addi RT,0,value		
lis	RT, IM	Load immediate shifted. (RT) ← (IM || ^{16}0) Extended mnemonic for addis RT,0,value		

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
lmw		NOT IMPLEMENTED Emulate with lvd and lwz combination.		E
lswi **lswx**		NOT IMPLEMENTED **Can be emulated with lvd, lwz, lhz, and lbz combination.**		E
lsku	RT,D(RA)	PPE 42: NOT IMPLEMENTED PPE 42X: If D ∧ 4 = 4 then perform function described by **lcxtu** else Load word from EA = (RA) + EXTS(D) and place in EDR. Load 0,1, or 2 doublewords, dependent on the value of D, from memory between (RA) and EA into VDR(30) and/or VDR(28). Load word from EA+4 into LR. Update the base address, (RA) ← EA.		Not supported by PPE 42. New for PPE 42X
lvd	DT, D(RA)	Load doubleword from EA = (RA|0) + EXTS(D) and place in DT, (DT) ← MS(EA,8).		New for PPE 42
lvdu	DT, D(RA)	Load doubleword from EA = (RA|0) + EXTS(D) and place in DT, (DT) ← MS(EA,8). Update the base address, (RA) ← EA.		New for PPE 42
lvdx	DT, RA, RB	Load doubleword from EA = (RA|0) + (RB) and place in DT, (DT) ← MS(EA,8).		New for PPE 42
lwarx		NOT IMPLEMENTED PPE 42 usage does not require multi-processor synchronization (reservation)		X
lwbrx		NOT IMPLEMENTED		X
lwz	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) ← MS(EA,4).		
lwzu	RT, D(RA)	Load word from EA = (RA|0) + EXTS(D) and place in RT, (RT) ← MS(EA,4). Update the base address, (RA) ← EA.		
lwzux		NOT IMPLEMENTED Emulated with addi followed by lwzx.		E
lwzx	RT, RA, RB	Load word from EA = (RA|0) + (RB) and place in RT, (RT) ← MS(EA,4).		
macchw[o][s][u][.] **machhw[o][s][u][.]** **maclhw[o][s][u][.]**		NOT IMPLEMENTED		X
mark	RA, RB	Debug Mark when instruction bit 31 =0. Extended mnemonic for tw 0,RA,RB	ISR, EDR with program interrupt when RA or RB are invalid.	Places Mark in the debug trace containing the content of RA and RB fields

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
marktag	A	Debug Mark Tag when instruction bit 31 = 1. Extended mnemonic for tw (A/1024), (A/32)^x'1F', A^x'1F'		Places Mark in the debug trace containing the content of the TO, RA and RB fields
mcrf		NOT IMPLEMENTED Not needed, CR0 field only.		X
mcrxr		NOT IMPLEMENTED		X
mfcr	RT	Move from CR to RT, (RT) ← (CR).		
mfdbsr **mfdcr**		NOT IMPLEMENTED		X
mfmsr	RT	Move from MSR to RT, (RT) ← (MSR).		
mfdacr **mfdbcr** **mfdec** **mfedr** **mfisr** **mfivpr**	RT	Move from special purpose register (SPR) SPRN. Extended mnemonic for mfspr RT,SPRN See *Special Purpose Registers* for listing of valid SPRN values.		NEW for PPE 42
mfctr **mflr** **mfpid** **mfpvr** **mfsprg0** **mfsrr0** **mfsrr1** **mftcr** **mftsr** **mfxer**	RT			
mfspr	RT, SPRN	Move from SPR to RT, (RT) ← (SPR(SPRN)).		
mftb **mftbu**		NOT IMPLEMENTED		X
mr	RT, RS	Move register. (RT) ← (RS) Extended mnemonic for or RT,RS,RS		
mr.		Extended mnemonic for or. RT,RS,RS	CR[CR0]	
mtcr		NOT IMPLEMENTED Use legacy mtcrf form instead.		X

Mnemonic	Operands	Function	Other Registers Changed	Difference from PowerPC 405-S ISA*
mtcrf	128, RS	The first operand is the FXM field, which determines which bits to move into CR. FXM_0 must be '1' to select CR0, all other bits must be '0'. Therefore the first operand must always be 128.		FXM = 128 (0x80) to select only CR0
mtcr0	RS	Move bits 0:3 of RS into CR0 Extended mnemonic for mtcrf 128, RS Note: mtocrf form is not supported.		Only CR0
mtdbsr **mtdcr**		NOT IMPLEMENTED		X
mtmsr	RS	Move to MSR from RS, (MSR) ← (RS).		
mtdacr **mtdbcr** **mtdec** **mtedr** **mtisr** **mtivpr**	RS	Move to SPR SPRN. Extended mnemonic for mtspr SPRN,RS See *Special Purpose Registers* for listing of valid SPRN values.		NEW for PPE 42
mtctr **mtlr** **mtsprg0** **mtsrr0** **mtsrr1** **mttcr** **mttsr** **mtxer**	RS			
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) ← (RS).		
mulchw **mulchw.** **mulchwu** **mulchwu.** **mulhhw** **mulhhw.** **mulhhwu** **mulhhwu.**		NOT IMPLEMENTED Must compute with combination of mullhw and other arithmetics.		X
mullhw	RT, RA, RB	(RT)0:31 ← (RA)16:31 x (RB)16:31 signed		
mullhw.			CR[CR0]	
mullhwu	RT, RA, RB	(RT)16:31 ← (RA)16:31 x (RB)16:31 unsigned		
mullhwu.			CR[CR0]	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
mulhw **mulhw.** **mulhwu** **mulhwu.**		NOT IMPLEMENTED Must compute with combination of mullhw and other arithmetics.		X
mulli	RT, RA, IM	PPE 42: NOT IMPLEMENTED Must compute with combination of mullhw and other arithmetics. PPE 42X: (RT)0:31 ← least significant word (RA)0:31 x EXTS(IM) (signed or unsigned)		X
mullw **mullw.**	RT, RA, RB	PPE 42: NOT IMPLEMENTED Must compute with combination of mullhw and other arithmetics. PPE 42X: (RT)0:31 ← least significant word (RA)0:31 x (RB)0:31 (signed or unsigned)	CR[CR0]	X
mullwo **mullwo.**	RT, RA, RB	PPE 42: NOT IMPLEMENTED Must compute with combination of mullhw and other arithmetics. PPE 42X: (RT)0:31 ← least significant word (RA)0:31 x (RB)0:31 (signed or unsigned)	CR[CR0] XER[SO, OV]	X
nand	RA, RS, RB	NAND (RS) with (RB). Place result in RA.		
nand.	RA, RS, RB	NAND (RS) with (RB). Place result in RA.	CR[CR0]	
neg	RT, RA	Negative (twos complement) of RA. (RT) ← ¬(RA) + 1		
neg.	RT, RA	Negative (twos complement) of RA. (RT) ← ¬(RA) + 1	CR[CR0]	
nego	RT, RA	Negative (twos complement) of RA. (RT) ← ¬(RA) + 1	XER[SO, OV]	
nego.	RT, RA	Negative (twos complement) of RA. (RT) ← ¬(RA) + 1	CR[CR0] XER[SO, OV]	
nmacchw[o][s][u][.] **nmachhw[o][s][u][.]** **nmachlw[o][s][u][.]***		NOT IMPLEMENTED		X
nop		Preferred no-op, triggers optimizations based on no-ops. Extended mnemonic for ori 0,0,0		

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		
nor.			CR[CR0]	
not	RA, RS	Complement register. (RA) ← ¬(RS) Extended mnemonic for nor RA,RS,RS		
not.		Extended mnemonic for nor. RA,RS,RS	CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with ¬(RB). Place result in RA.		
orc.			CR[CR0]	
ori	RA, RS, IM	OR (RS) with (^{16}0 || IM). Place result in RA.		
oris	RA, RS, IM	OR (RS) with (IM || ^{16}0). Place result in RA.		
rfci		NOT IMPLEMENTED		X
rfi		Return from interrupt. (PC) ← (SRR0). (MSR) ← (SRR1).		
rldicl	DA, DS, SH, MB	PPE 42: NOT IMPLEMENTED Must be emulated using using multiple 32-bit rotates PPE 42X:Rotate left doubleword immediate, then clear left r ← ROTL((DS), SH) m ← MASK(MB, 63) (DA) ← r ∧ m		Not supported by PPE 42. New for PPE 42X.
rldicl.			CR[CR0]	
rldicr	DA, DS, SH, ME	PPE 42: NOT IMPLEMENTED Must be emulated using using multiple 32-bit rotates PPE 42X: r ← ROTL((DS), SH) m ← MASK(0, ME) (DA) ← r ∧ m		Not supported by PPE 42. New for PPE 42X.
rldicr.			CR[CR0]	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
rldimi	DA, DS, SH, MB	PPE 42: NOT IMPLEMENTED Must be emulated using using multiple 32-bit rotates PPE 42X: Rotate left doubleword immediate, then insert according to mask. r ← ROTL((DS), SH) m ← MASK(MB, ¬SH) (DA) ← (r ∧ m) ∨ ((DA) ∧ ¬m)		Not supported by PPE 42. New for PPE 42X.
rldimi.			CR[CR0]	
rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask. r ← ROTL((RS), SH) m ← MASK(MB, ME) (RA) ← (r ∧ m) ∨ ((RA) ∧ ¬m)		
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. r ← ROTL((RS), SH) m ← MASK(MB, ME) (RA) ← (r ∧ m)		
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. r ← ROTL((RS), (RB)27:31) m ← MASK(MB, ME) (RA) ← (r ∧ m)		
rlwnm.			CR[CR0]	
rotlw	RA, RS, RB	Rotate left. (RA) ← ROTL((RS), (RB)27:31) Extended mnemonic for rlwnm RA,RS,RB,0,31		
rotlw.		Extended mnemonic for rlwnm. RA,RS,RB,0,31	CR[CR0]	
rotlwi	RA, RS, n	Rotate left immediate. (RA) ← ROTL((RS), n) Extended mnemonic for rlwinm RA,RS,n,0,31		
rotlwi.		Extended mnemonic for rlwinm. RA,RS,n,0,31	CR[CR0]	
rotrwi	RA, RS, n	Rotate right immediate. (RA) ← ROTL((RS), 32−n) Extended mnemonic for rlwinm RA,RS,32−n,0,31		
rotrwi.		Extended mnemonic for rlwinm. RA,RS,32−n,0,31	CR[CR0]	
sc		NOT IMPLEMENTED		X

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
slvd	DA, DS, RB	PPE 42: NOT IMPLEMENTED Must be emulated using using multiple 32-bit shifts PPE 42X: Shift left (DS) by $(RB)_{27:31}$. n ← $(RB)_{27:31}$. r ← ROTL((DS), 64 – n). if $(RB)_{26}$ = 0 then m ← MASK(0, 63-n) else m ← $^{64}0$. (DA) ← r ∧ m.		Not supported by PPE 42. New for PPE 42X.
slvd.			CR[CR0]	
slw	RA, RS, RB	Shift left (RS) by $(RB)_{27:31}$. n ← (RB)27:31. r ← ROTL((RS), n). if $(RB)_{26}$ = '0' then m ← MASK(0, 31 – n) else m ← $^{32}0$. (RA) ← r ∧ m.		
slw.			CR[CR0]	
slwi	RA, RS, n	Shift left immediate. (n < 32) $(RA)_{0:31-n}$ ← $(RS)_{n:31}$ $(RA)_{32-n:31}$ ← n0 Extended mnemonic for rlwinm RA,RS,n,0,31−n		
slwi.		Extended mnemonic for rlwinm. RA,RS,n,0,31−n	CR[CR0]	
sraw	RA, RS, RB	Shift right algebraic (RS) by $(RB)_{27:31}$. n ← $(RB)_{27:31}$. r← ROTL((RS), 32 – n). if $(RB)_{26}$ = '0' then m ← MASK(n, 31) else m ← $^{32}0$. s← $(RS)_0$. (RA) ← (r ∧ m) ∨ (^{32}s ∧ ¬m). XER[CA] ← s ∧ ((r ∧ ¬m) ≠ '0').		
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. n ← SH. r ← ROTL((RS), 32 – n). m ← MASK(n, 31). s ← $(RS)_0$. (RA) ← (r ∧ m) ∨ (^{32}s ∧ ¬m). XER[CA] ← s ∧ ((r ∧ ¬m)≠'0').		
srawi.			CR[CR0]	
srvd	DA, DS, RB	PPE 42: NOT IMPLEMENTED Must be emulated using using muliple 32-bit shifts PPE 42X: Shift right (DS) by $(RB)_{27:31}$. n ← $(RB)_{27:31}$. r ← ROTL((DS), 64 – n). if $(RB)_{26}$ = 0 then m ← MASK(n, 63) else m ← $^{64}0$. (DA) ← r ∧ m.		Not supported by PPE 42. New for PPE 42X
srvd.			CR[CR0]	

<table>
<tr><th>Mnemonic</th><th>Operands</th><th>Function</th><th>Other Registers Changed</th><th>Difference from PowerPC 405-S ISA*</th></tr>
<tr><td>srw</td><td rowspan="2">RA, RS, RB</td><td rowspan="2">Shift right (RS) by $(RB)_{27:31}$. n ← $(RB)_{27:31}$.
r ← ROTL((RS), 32 – n).
if $(RB)_{26}$ = 0 then m ← MASK(n, 31) else m ← ${}^{32}0$.
(RA) ← r ∧ m.</td><td></td><td></td></tr>
<tr><td>srw.</td><td>CR[CR0]</td><td></td></tr>
<tr><td>srwi</td><td rowspan="2">RA, RS, n</td><td>Shift right immediate. (n < 32)
$(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow {}^{n}0$
Extended mnemonic for rlwinm RA,RS,32−n,n,31</td><td></td><td></td></tr>
<tr><td>srwi.</td><td>Extended mnemonic for rlwinm. RA,RS,32−n,n,31</td><td>CR[CR0]</td><td></td></tr>
<tr><td>stb</td><td>RS, D(RA)</td><td>Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + EXTS(D).</td><td></td><td></td></tr>
<tr><td>stbu</td><td>RS, D(RA)</td><td>Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + EXTS(D).
Update the base address,
(RA) ← EA.</td><td></td><td></td></tr>
<tr><td>stbux</td><td></td><td>NOT IMPLEMENTED
Emulate with addi followed by stbx</td><td></td><td>E</td></tr>
<tr><td>stbx</td><td>RS, RA, RB</td><td>Store byte $(RS)_{24:31}$ in memory at EA = (RA|0) + (RB).</td><td></td><td></td></tr>
<tr><td>sth</td><td>RS, D(RA)</td><td>Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D).</td><td></td><td></td></tr>
<tr><td>sthbrx</td><td></td><td>NOT IMPLEMENTED</td><td></td><td>X</td></tr>
<tr><td>stcxu</td><td>RS, DS(RA)</td><td>PPE 42: NOT IMPLEMENTED
PPE 42X:
EA ← (RA). Store LR into word at EA + 4.
Store 10 doublewords into memory between (RA) and EA = (RA) + EXTS(D) from VDR(30), VDR(28), SRR0 || SRR1, XER || CTR, VDR(9,7,5,3,0), and CR || SPRG0.
Store RS into word at (EA).
Update the base address, (RA) ← EA.
Extended mnemonic for stsku RS, DS∨4(RA)</td><td></td><td>Not supported by PPE 42. New for PPE 42X</td></tr>
<tr><td>sthu</td><td>RS, D(RA)</td><td>Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + EXTS(D).
Update the base address,
(RA) ← EA.</td><td></td><td></td></tr>
<tr><td>sthux</td><td></td><td>NOT IMPLEMENTED
Emulate with addi followed by sthx</td><td></td><td>E</td></tr>
</table>

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
sthx	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at EA = (RA|0) + (RB).		
stmw		NOT IMPLEMENTED Emulate with stvd and stw combination.		E
stswi **stswx**		NOT IMPLEMENTED **Can be emulated with stvd, stw, sth, and stb combination.**		E
stsku	RS, DS(RA)	PPE 42: NOT IMPLEMENTED PPE 42X: If D ∧ 4 = 4 then perform function described by **stcxtu** else EA ← (RA). Store LR into word at EA + 4. Store 0,1, or 2 doublewords, dependent on the value of D, into memory between (RA) and EA = (RA) + EXTS(D) from VDR(30), VDR(28), SRR0 || SRR1, XER || CTR, VDR(9,7,5,3,0), and CR || SPRG0. Store RS into word at (EA). Update the base address, (RA) ← EA.		Not supported by PPE 42. New for PPE 42X
stvd	DS, D(RA)	Store virtual doubleword (DS) in memory at EA = (RA|0) + EXTS(D).		New for PPE 42
stvdu	DS, D(RA)	Store virtual doubleword (DS) in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) ← EA.		New for PPE 42
stvdx	DS, RA, RB	Store virtual doubleword (DS) in memory at EA = (RA|0) + (RB).		New for PPE 42
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D).		
stwbrx		NOT IMPLEMENTED		X
stwcx.		NOT IMPLEMENTED PPE 42 usage does not require multi-processor synchronization (reservation)		X
stwu	RS, D(RA)	Store word (RS) in memory at EA = (RA|0) + EXTS(D). Update the base address, (RA) ← EA.		
stwux		NOT IMPLEMENTED Emulate with addi followed by stwx		E
stwx	RS, RA, RB	Store word (RS) in memory at EA = (RA|0) + (RB).		

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
sub	RT, RA, RB	Subtract (RB) from (RA). (RT) ← ¬(RB) + (RA) + 1. Extended mnemonic for subf RT,RB,RA		
sub.		Extended mnemonic for subf. RT,RB,RA	CR[CR0]	
subo		Extended mnemonic for subfo RT,RB,RA	XER[SO, OV]	
subo.		Extended mnemonic for subfo. RT,RB,RA	CR[CR0] XER[SO, OV]	
subc	RT, RA, RB	Subtract (RB) from (RA). (RT) ← ¬(RB) + (RA) + 1. Place carry-out in XER[CA]. Extended mnemonic for subfc RT,RB,RA		
subc.		Extended mnemonic for subfc. RT,RB,RA	CR[CR0]	
subco		Extended mnemonic for subfco RT,RB,RA	XER[SO, OV]	
subco.		Extended mnemonic for subfco. RT,RB,RA	CR[CR0] XER[SO, OV]	
subf	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1.		
subf.			CR[CR0]	
subfo			XER[SO, OV]	
subfo.			CR[CR0] XER[SO, OV]	
subfc	RT, RA, RB	Subtract (RA) from (RB). (RT) ← ¬(RA) + (RB) + 1. Place carry-out in XER[CA].		
subfc.			CR[CR0]	
subfco			XER[SO, OV]	
subfco.			CR[CR0] XER[SO, OV]	
subfe	RT, RA, RB	Subtract (RA) from (RB) with carry-in. (RT) ← ¬(RA) + (RB) + XER[CA]. Place carry-out in XER[CA].		
subfe.			CR[CR0]	
subfeo			XER[SO, OV]	
subfeo.			CR[CR0] XER[SO, OV]	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
subfic	RT, RA, IM	Subtract (RA) from EXTS(IM). (RT) ← ¬(RA) + EXTS(IM) + 1. Place carry-out in XER[CA].		
subfme	RT, RA, RB	Subtract (RA) from (–1) with carry-in. (RT) ← ¬(RA) + (–1) + XER[CA]. Place carry-out in XER[CA].		
subfme.			CR[CR0]	
subfmeo			XER[SO, OV]	
subfmeo.			CR[CR0] XER[SO, OV]	
subfze	RT, RA, RB	Subtract (RA) from zero with carry-in. (RT) ← ¬(RA) + XER[CA]. Place carry-out in XER[CA].		
subfze.			CR[CR0]	
subfzeo			XER[SO, OV]	
subfzeo.			CR[CR0] XER[SO, OV]	
subi	RT, RA, IM	Subtract EXTS(IM) from (RA|0). Place result in RT. Extended mnemonic for addi RT,RA,−IM		
subic	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. Extended mnemonic for addic RT,RA,−IM		
subic.			CR[CR0]	
subis	RT, RA, IM	Subtract (IM || $^{16}0$) from (RA|0). Place result in RT. Extended mnemonic for addis RT,RA,−IM		
subwbnz	RA, RB, target	Fused Subtract Word and Branch if Not Zero. Extended mnemonic for: **cmpwbc[l] 0,3,RA, RB, target**		New for PPE 42
subwbnzl			(LR) ← CIA + 4.	
subwbz	RA, RB, target	Fused Subtract Word and Branch if Zero. Extended mnemonic for: **cmpwbc[l] 1,3,RA, RB, target**		New for PPE 42
subwbzl			(LR) ← CIA + 4.	

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
subwibnz	RA, UIX, target	Fused Subtract Word Immediate and Branch if Not Zero. Extended mnemonic for: **cmpwibc[l] 0,3,RA, UIX, target**		New for PPE 42
subwibnzl			(LR) ← CIA + 4.	
subwibz	RA, UIX, target	Fused Subtract Word Immediate and Branch if Zero. Extended mnemonic for: **cmpwibc[l] 1,3,RA, UIX, target**		New for PPE 42
subwibzl			(LR) ← CIA + 4.	
sync		Synchronization. All instructions that precede **sync** complete before any instructions that follow **sync** begin. When **sync** completes, all storage accesses initiated prior to **sync** will have completed.		
tlbia **tlbre** **tlbrehi** **tlbrelo** **tlbsx** **tlbsx.** **tlbsync** **tlbwe** **tlbwehi** **tlbwelo**		NOT IMPLEMENTED		X
trap		Trap unconditionally, with instruction bit 31 = 0. Extended mnemonic for tw 31,0,0		
tweq **twge** **twgt** **twle** **twlge** **twlgt** **twlle** **twllt** **twlng** **twlnl** **twlt** **twne** **twng** **twnl**		NOT IMPLEMENTED Extended mnemonics for conditional tw		X
tw	TO, RA, RB	Requires instruction bit 31 = 0. No-op with Debug mark when TO=0 or TG=1 else Trap exception is generated when TO=31, causing either a program interrupt or a halt based on DBCR. Other TO values cause program interrupt, unless TG=1.	XSR if DBCR[TRAP] = '1' and TO=31 else ISR, EDR with program interrupt when TO≠0 or when RA or RB are invalid.	Always unconditional

Mnemonic	*Operands*	*Function*	*Other Registers Changed*	*Difference from PowerPC 405-S ISA**
tweqi **twgei** **twgti** **twlei** **twlgei** **twlgti** **twllei** **twllti** **twlngi** **twlnli** **twlti** **twnei** **twngi** **twnli**		NOT IMPLEMENTED Extended mnemonics for conditional twi		X
twi		NOT IMPLEMENTED		X
twu	RA, RB	Trap unconditionally. Extended mnemonic for tw 31,RA,RB Note: requires instruction bit 31 = 0.	XSR if DBCR[TRAP] = '1' ISR, EDR with program interrupt when RA or RB are invalid.	
wrtee	RS	Write value of RS_{16} to MSR[EE].		
wrteei	E	Write value of E to MSR[EE].		
xor	RA, RS, RB	XOR (RS) with (RB). Place result in RA.		
xor.	RA, RS, RB	XOR (RS) with (RB). Place result in RA.	CR[CR0]	
xori	RA, RS, IM	XOR (RS) with (${}^{16}0$ || IM). Place result in RA.		
xoris	RA, RS, IM	XOR (RS) with (IM || ${}^{16}0$). Place result in RA.		

* Notes:

X = Not implemented, no equivalent implementation in PPE 42.

E = Not implemented, can be perfectly emulated with two or more PPE 42 instructions.