

IEEE Standard for a 32-bit Microprocessor Architecture

Sponsor

**Microprocessor and Microcomputer Standards Committee
of the
IEEE Computer Society**

Approved March 17, 1994

IEEE Standards Board

Abstract: A 32-bit microprocessor architecture, available to a wide variety of manufacturers and users, is defined. The standard includes the definition of the instruction set, register model, data types, instruction op-codes, and coprocessor interface.

Keywords: computer, instruction set architecture, ISA, microprocessor

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1995 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 1995. Printed in the United States of America.

ISBN 1-55937-428-4

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE standards documents may involve the use of patented technology. Their approval by the Institute of Electrical and Electronics Engineers does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

(This introduction is not a part of IEEE Std 1754-1994, IEEE Standard for a 32-bit Microprocessor Architecture.)

The purpose of Std 1754-1994 is to define a 32-bit microprocessor architecture based on version 8 of the SPARC microprocessor architecture. The standard is intended for chip implementors and system software developers.

Structure of the standard

This standard comprises the following:

- Seven clauses
 - Clause 1, Overview, describes the IEEE 1754 architecture in high-level terms and explains the conventions used in this standard.
 - Clause 2, Definitions, special word usage, abbreviations, and acronyms, defines terms used throughout the standard.
 - Clause 3, Architectural overview, describes the IEEE 1754 processor, its instructions set, and its trap-handling capability.
 - Clause 4, Data formats, introduces the data formats supported by IEEE 1754.
 - Clause 5, Registers, describes the IEEE 1754 register set including integer unit (IU) registers, floating-point unit (FPU) registers, and coprocessor (CP) registers.
 - Clause 6, Instructions, describes the IEEE 1754 instruction set, including instruction formats and the various instruction categories.
 - Clause 7, Traps, describes the IEEE 1754 trap model.
- Four normative annexes
 - Annex A, Instruction definitions, defines each IEEE 1754 instruction, including its opcode, suggested assembly language syntax, and any traps it can generate.
 - Annex B, ISP descriptions, presents the required semantics of each IEEE 1754 instruction in instruction set processor (ISP) notation.
 - Annex C, IEEE 754 implementation requirements for IEEE 1754, describes the required IEEE 1754 floating-point implementation.
 - Annex D, IEEE 1754 implementation dependencies, describes the allowed implementation dependencies of the IEEE 1754 standard.
- Nine informative annexes
 - Annex E, Opcodes and condition codes, presents all IEEE 1754 opcodes and condition codes.
 - Annex F, IEEE 1754 Reference MMU architecture, describes the architecture of the IEEE 1754 Reference MMU, an optional memory management unit (MMU) that can support the IEEE 1754 architecture.
 - Annex G, Suggested ASI assignments, documents suggested address space identifier (ASI) assignments for IEEE 1754.
 - Annex H, Example integer multiplication and division routines, presents software multiplication and division routines for IEEE 1754.
 - Annex I, Suggested assembly language syntax, suggests assembly language for each IEEE 1754 instruction.
 - Annex J, Software considerations, discusses various software considerations when implementing system software to run on IEEE 1754-based processors.
 - Annex K, Instruction set summary, graphically summarizes the IEEE 1754 instruction set.
 - Annex L, Non-IEEE 1754 architectural extensions, describes the mechanisms provided for extending the IEEE 1754 instruction set.
 - Annex M provides a bibliography.
- An informative index

Conventions

• Fonts in standard

In this standard, fonts are used as follows:

- *Italic* is used for names of general-purpose registers, instruction fields, and read-only register status fields. For example, “The *rs1* field contains the address of the *r* register.”
- Typewriter font is used for literals throughout the annexes.
- **Bold** is used for emphasis. For example, “. . .the delay instruction is **not** executed. . . .”
- UPPERCASE items are acronyms, instruction names, or register mode fields that can be written by software. Some common acronyms appear in Clause 2, “Definitions, special word usage, abbreviations, and acronyms.” Note that names of some instructions contain both uppercase and lowercase letters.
- Underbar characters join words in register, register field, exception, or trap names. For example, “The integer_condition_code field. . . .”
- Square brackets [] indicate an addressed field in a register or a numbered register in a register file. For example, “*r*[0] is zero.”
- Angle brackets < > indicate a bit number or colon-separated range of bit numbers within a field; for example, “Bits FSR<29:28> and FSR<12> are unused.”
- Reduced-size font is used for notes.
- *Italic Helvetica* font is used for names of processor exceptions and floating-point trap types.

• References to implementation dependencies

Definitions of IEEE 1754 architecture implementation dependencies are preceded by the notation “**Impl. Dep. #*nn***”. The number *nn* is used to enumerate the dependencies in Annex D, “IEEE 1754 implementation dependencies.” References to IEEE 1754 architecture implementation dependencies are followed by the notation “(impl. dep. #*nn*)”.

• Number base notation

Numbers throughout this standard are decimal (base-10) numbers unless otherwise indicated. Within the text, numbers in other bases are followed by a numeric subscript indicating their base (for example, 11001₂, FFFF0000₁₆). Within C and assembly language examples, numbers may be preceded by “0x” to indicate base-16 (hexadecimal) notation (for example, 0xffff0000).

• Notes

This manual provides several different types of information in notes. This information appears in a reduced-size font with the word “NOTE” or “NOTES” in the left-hand margin.

- Programming notes contain incidental information about programming for the IEEE 1754 architecture.
- Implementation notes contain information that may be specific to an implementation, or that may differ in different implementations.

Previously copyrighted and patented material

The IEEE Standards Board calls attention to the fact that it is claimed that IEEE Std 1754-1994 is the subject of patents owned by Sun Microsystems, Inc. The IEEE takes no position with respect to patent validity. Sun Microsystems, Inc., has assured IEEE that it is willing to grant a license on these patents on reasonable and

nondiscriminatory terms to anyone wishing to obtain such a license. The undertakings of Sun Microsystems, Inc., in this respect are on file with the IEEE Standards Department, and the license details may be obtained from Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043-1100.

Committee lists

At the time this standard was approved, the membership of the IEEE 1754 Working Group was as follows:

Phillip W. Huelson, *Chair*
David L. Weaver, *Editor*
Stephen L. Diamond, *Secretary (1990–91)*

Dennis Allison	Dave Gustavson	David Poole
Rick Avra	Jerry Huck	Bernard Peuto
Harrison Beasley	Jim Hull	Sanjev Renjen
Clyde Camp	John Hyde	“Somas” Somasundaram
Greg Blanck	David James	Steve Studulski
Paul Borrill	Joseph Jeffreys	Chris Thomson
Jeff Broughton	Steve Krueger	Rich Trauben
Patrick Buffet	Tim McCoy	Rudolf Usselmann
David Ditzel	Bruce McKeever	Armando Viteri
Martin Eisenmann	James Monaco	James J. Whelan
Doug Finke	Craig Nelson	Mike Williams
	Joan Pendleton	

The following persons were on the balloting committee:

Charles Abernathy	Graham Connolly	Frances Habib
Barbara P. Aichinger	Robert Cook	Richard A. Hagan
Scott Akers	John Croll	Sandor V. Halasz
Richard P. Ames	Robert Crowder	Leland E. Hawkins
Robert G. Andersen	Dean Dapkewicz	George E. Heidorn
Roger Andersen	Rob Davies	Hans H. Heilborn
Elmer Arment	Randy Day	David Heimann
Bjorn Bakka	Stephen Vincent Dennis	John L. Hennessy
Harrison A. Beasley	Stephen L. Diamond	Frank Hom
Paul L. Bendeck	Eduardo Diaz	Rand A. Hoven
Manisha Bhagwatt	John Dicristina	Jerry Huck
Dileep Bhandarkar	David Ditzel	Phil Huelson
Dilip K. Bhavsar	Richard Doucette	Jim Hull
Robert Bismuth	Samuel Duncan	John Hyde
Prasajd Biswas	Raymond A. DuPont	Aron K. Insinga
Greg J. Blanck	Sourav K. Dutta	Jim Isaak
Paul L. Borrill	David L. Duxstad	Ed Jacques
Anna Boyd	Wilhelm P. Evertz	Raj Jain
John C. Brightwell	Aram Falsabi	David V. James
Jeffrey M. Broughton	Ernest R. Feasel	Kenneth A. Jansen
Mike Brown	John G. Fletcher	Paul Jensen
Russ Brown	Robert H. Follett	Sverre Johansen
Patrick Buffet	Gordon Force	David R. Jones
Bob Cargnoni	Robert M. Fries	Chris Joseph
John Carrillo	Donald Gaubatz	Jagdish Joshi
Don L. Chamberlain	Tayeb Giuma	W. Kahan
C. H. Chen	Steve Glaser	Michael Kantrowitz
Allan Christie	Douglas N. Good	John Kapp
Robert Colwell	Francis Gravel	Giora Katzir
Rich Connelly	David B. Gustavson	Waiman Fred Ki

Yoon J. Kim
Gary Kimura
Larry Kinney
Hemant Kishan
Richard J. C. Kissel
John F. Klebes
Wayne E. Klindt
Lee Knoch
John Kowaleski
Scott Kramer
Ernst H. Kristiansen
Steven D. Krueger
Masahito Kubo
Thomas M. Kurihara
Andy Ladd
Maureen Ladd
Douglas Laird
Karen Lentz
Bruce D. Lightner
William Lindow
Stig-Goran Lindqvist
Rollins Linser
Wayne M. Loucks
Dave Lunger
Stephen Mabbs
Michael J. Mahon
Chittaranjan Mallipeddi
Don Marshall
Susan Mason
Andrew J. McCrocklin
Montgomery C. McGraw
Bruce McKeever
Gary McMillian
Peter Michels
Steven M. Millendorf
Viki Moldenhauer
James Monaco
James D. Mooney

Dale Morris
James Mott
Robert Muglia
Hans Mulder
Radha Murthy
Mark Naylor
Kenneth E. Nicholas
Tadahiko Nishimukai
Daniel C. O'Connor
Rose G. O'Donnell
David A. Orbits
Michael Orlovsky
Francisco Pataro
Chandresh J. Patel
Nital P. Patwa
Donald Paylovich
Michael Pelletier
Joan Pendelton
Gianluca Piccinini
Dee Pipes
Dave Plauser
Fred Pollack
Roland F. Portman
Bruno R. Preiss
Luis Quijano
Andrew Radle
Robert Reichard
Paul Reuter
William T. Rhoades
Gary S. Robinson
Douglas B. Robinson
Fred U. Rosenberger
Chaarudatta Ruddrakshi
Anidudh Sahni
Donald Senzig
Amitabh Shah
Mahmoud Shahram
John Shakshober
Ravi Shankar

Steve Sherman
Kerry Shore
Eugene L. Shrock
Patrick M. Simonich
Patricia A. Simonson
Frank M. Smith
Dongzhuang Su
Javin M. Taylor
Randy Taylor
Donn S. Terry
Jeffrey A. Thomas
Markham Thomas
Chris Thomson
Marc Tremblay
Robert Tripi
Kris Uebeersax
Rudolf Usselmann
Andre Vachon
Steven M. Valentine
Carl Warren
David Weaver
Richard J. Westmore
Leslie John Weston
James J. Whelan
Martin J. Whittaker
Robert Widlicka
Mark Williams
Bryan Willman
Andrew Wilson
Peter J. Wilson
Roland Winkvist
Gregory Winston
David Wong
Tom Woodburn
Jim Wooldridge
Ricky Yeung
Oren Yuen
Janusz Zalewski

When the IEEE Standards Board approved this standard on March 17, 1994, it had the following membership:

Wallace S. Read, *Chair*

Donald C. Loughry, *Vice Chair*

Andrew G. Salem, *Secretary*

Gilles A. Baril
Bruce B. Barrow
José A. Berrios de la Paz
Clyde R. Camp
James Costantino
Stephen L. Diamond
Donald C. Fleckenstein
Jay Forster*
Ramiro Garcia

Donald N. Heirman
Richard J. Holleman
Jim Isaak
Ben C. Johnson
Sonny Kasturi
Lorraine C. Kevra
E. G. "Al" Kiener
Ivor N. Knight

Joseph L. Koepfinger*
D. N. "Jim" Logothetis
L. Bruce McClung
Marco W. Migliaro
Mary Lou Padgett
Arthur K. Reilly
Ronald H. Reimer
Gary S. Robinson
Leonard L. Tripp

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall
Richard B. Engelman
David E. Soffrin
Stanley I. Warshaw

Paula M. Kelty
IEEE Standards Project Editor

SPARC is a registered trademark of SPARC International, Inc.
Multibus II is a registered trademark of Intel Corporation.
UNIX is a registered trademark of X/Open Company Ltd.

Contents

CLAUSE	PAGE
1. Overview.....	1
1.1 Scope	1
1.1.1 IEEE 1754 attributes	1
1.2 IEEE 1754 system components	1
1.2.1 Reference MMU	1
1.2.2 Supervisor software	2
1.3 Binary compatibility	2
1.4 IEEE 1754 features	3
1.5 Conformance to IEEE 1754	3
1.6 Conformance document	4
1.7 References	4
2. Definitions, special word usage, abbreviations, and acronyms	5
2.1 Definitions	5
2.2 Special word usage	8
2.3 Acronyms and abbreviations	8
3. Architectural overview	9
3.1 IEEE 1754 processor	9
3.1.1 Integer unit (IU)	9
3.1.2 Floating-point unit (FPU)	10
3.1.3 Coprocessor (CP)	10
3.2 Instructions	10
3.2.1 Memory access	10
3.2.2 Arithmetic/logical/shift	12
3.2.3 Control transfer	12
3.2.4 State register access	13
3.2.5 Floating-point operate (FPop)	13
3.2.6 Coprocessor operate (CPop)	13
3.3 Traps	13
4. Data formats.....	15
4.1 Overview	15
4.2 Signed integer byte	15
4.3 Signed integer halfword	16
4.4 Signed integer word	16
4.5 Signed integer double	16
4.6 Unsigned integer byte	16
4.7 Unsigned integer halfword	17
4.8 Unsigned integer word	17
4.9 Unsigned integer double	17
4.10 Unsigned tagged word	17
4.11 Signed tagged word	18
4.12 Floating-point single precision	18
4.13 Floating-point double precision	18
4.14 Floating-point quad precision	19
5. Registers	23
5.1 IU <i>r</i> registers	23

CLAUSE	PAGE
5.1.1	Windowed <i>r</i> registers 23
5.1.2	Overlapping of windows 24
5.1.3	Special <i>r</i> registers 24
5.1.4	Doubleword operands 24
5.1.5	Register usage 25
5.2	IU Control/status registers 26
5.2.1	Processor state register (PSR) 27
5.2.2	Window invalid mask register (WIM) 29
5.2.3	Trap base register (TBR) 30
5.2.4	Multiply/divide register (Y) 30
5.2.5	Program counters (PC, nPC) 30
5.2.6	Ancillary state registers (ASR) 31
5.2.7	Implementation-dependent extensions register 31
5.2.8	IU deferred-trap queue 31
5.3	FPU <i>f</i> registers 32
5.3.1	Double and quad operands 32
5.4	FPU control/status registers 32
5.4.1	Floating-point state register (FSR) 33
5.4.2	Floating-point exception fields 37
5.4.3	FSR conformance 38
5.4.4	Floating-point deferred-trap queue (FQ) 39
5.5	CP registers 39
6.	Instructions 41
6.1	Instruction execution 41
6.2	Instruction formats 41
6.2.1	Instruction fields 42
6.3	Instruction categories 43
6.3.1	Memory access instructions 44
6.3.2	Integer arithmetic instructions 45
6.3.3	Control-transfer instructions (CTIs) 46
6.3.4	SAVE instruction 52
6.3.5	RESTORE instruction 52
6.3.6	State register access 52
6.3.7	Floating-point operate (FPop) instructions 53
6.3.8	Coprocessor operate (CPop) instructions 53
6.3.9	Reserved instructions 53
7.	Traps 55
7.1	Trap categories 55
7.1.1	Reset trap 55
7.1.2	Precise trap 55
7.1.3	Deferred trap 56
7.1.4	Disrupting trap 56
7.2	Trap model 57
7.3	Trap control 58
7.3.1	ET and PIL control 58
7.3.2	TEM control 58
7.4	Trap identification 59
7.4.1	Trap type (<i>tt</i>) 59
7.4.2	Trap priorities 59
7.5	Trap definition 61

CLAUSE	PAGE
7.5.1	Reset trap 61
7.5.2	Non-reset traps 62
7.5.3	Error_state 63
7.5.4	Processor execution state changes upon traps 63
7.6	Exception/interrupt descriptions 64
7.7	Trap actions 66

ANNEX

Annex A	Instruction definitions (normative)..... 67
A.1	Add instructions 70
A.2	Branch on coprocessor condition codes instructions 71
A.3	Branch on floating-point condition codes (FBfcc) instructions 73
A.4	Branch on integer condition codes (Bicc) instructions 75
A.5	Call and link instruction 77
A.6	Coprocessor operate instructions 78
A.7	Divide instructions (64 bit÷32 bit) 79
A.8	Floating-point add and subtract instructions 81
A.9	Floating-point compare instructions 82
A.10	Convert floating point to integer instructions 83
A.11	Convert between floating-point formats instructions 84
A.12	Convert integer to floating-point instructions 85
A.13	Floating-point move instructions 86
A.14	Floating-point multiply and divide instructions 87
A.15	Floating-point operate (FPop) instructions 88
A.16	Floating-point square-root instructions 89
A.17	Flush instruction memory 90
A.18	Jump and link instruction 91
A.19	Load coprocessor instructions 92
A.20	Load floating-point instructions 93
A.21	Load integer instructions 94
A.22	Load integer from alternate space instructions 96
A.23	Load-store unsigned byte instruction 98
A.24	Load-store unsigned byte in alternate space instruction 99
A.25	Logical instructions 100
A.26	Multiply instructions (32-bit) 101
A.27	Multiply step instruction 102
A.28	NOP instruction 103
A.29	Read state register instructions 104
A.30	Return from trap instruction 105
A.31	SAVE and RESTORE instructions 107
A.32	SETHI instruction 108
A.33	Shift instructions 109
A.34	STBAR instruction 110
A.35	Store coprocessor instructions 111
A.36	Store floating-point instructions 113
A.37	Store integer instructions 115
A.38	Store integer into alternate space instructions 116
A.39	Subtract instructions 117
A.40	SWAP register with memory instruction 118
A.41	SWAP register with alternate space memory instruction 119
A.42	Tagged add instructions 120

ANNEX	PAGE
A.43 Tagged subtract instructions	121
A.44 Trap on integer condition codes instruction	122
A.45 Unimplemented instruction	124
A.46 Write state register instructions	125
Annex B ISP descriptions (normative)	129
B.1 ISP notation	129
B.2 Processor external interface definition	130
B.2.1 Interface macros	130
B.2.2 Interface signals	131
B.3 Register field definitions	132
B.4 Instruction field definitions	133
B.5 Temporary variable definitions	134
B.6 Processor states and instruction dispatch	134
B.7 Instruction dispatch	136
B.8 Floating-point execution	136
B.9 Traps	137
B.10 Instruction definitions	139
B.10.1 Add instructions	139
B.10.2 Branch on integer condition instructions	139
B.10.3 CALL instruction	140
B.10.4 Coprocessor branch on condition instructions	140
B.10.5 Divide instructions	140
B.10.6 Floating-point branch on condition instructions	141
B.10.7 Flush instruction memory	142
B.10.8 Jump and link (JMPL) instruction	142
B.10.9 Load instructions	142
B.10.10 Load-store unsigned byte instructions	144
B.10.11 Logical instructions	145
B.10.12 Multiply instructions	145
B.10.13 Multiply step instruction	146
B.10.14 NOP instruction	146
B.10.15 Read state register instructions	146
B.10.16 Return from trap (RETT) instruction	146
B.10.17 SAVE and RESTORE instructions	147
B.10.18 SETHI instruction	148
B.10.19 Shift instructions	148
B.10.20 Store barrier instruction	148
B.10.21 Store instructions	148
B.10.22 Subtract instructions	150
B.10.23 Swap register with memory instructions	150
B.10.24 Tagged add instructions	151
B.10.25 Tagged subtract instructions	152
B.10.26 Trap on integer condition (Ticc) instructions	152
B.10.27 Unimplemented instruction	153
B.10.28 Write state register instructions	153
B.11 Floating-point operate instructions	154
B.11.1 Convert between floating-point formats instructions	154
B.11.2 Convert floating-point to integer	155
B.11.3 Convert integer to floating-point instructions	155
B.11.4 Floating-point add and subtract instructions	155
B.11.5 Floating-point compare instructions	155

ANNEX	PAGE
B.11.6	Floating-point move instructions 156
B.11.7	Floating-point multiply and divide instructions 156
B.11.8	Floating-point square root instructions 156
Annex C	IEEE 754 implementation requirements for IEEE 1754 (normative) 157
C.1	Traps inhibit results 157
C.2	NaN operand and result definitions 157
C.2.1	Untrapped floating-point result in different format than operands 158
C.2.2	Untrapped floating-point result in same format as operands 158
C.3	Trapped underflow definition (FSR.TEM.UFM=1) 159
C.4	Untrapped underflow definition (FSR.TEM.UFM=0) 159
C.5	Integer overflow definition 160
Annex D	IEEE 1754 implementation dependencies (normative)..... 161
D.1	Definition of an architectural implementation dependency 161
D.2	Hardware dependencies 161
D.3	Categories of architectural implementation dependencies 161
D.4	List of implementation dependencies 162
D.5	Implementation dependency groups 169
D.5.1	IU-related implementation dependencies 169
D.5.2	Coprocessor-related implementation dependencies 170
D.5.3	Floating-point-related implementation dependencies 170
D.5.4	Special registers and address space-related implementation dependencies 171
D.5.5	Trap-related implementation dependencies 172
D.5.6	MMU-related implementation dependencies 173
Annex E	Opcodes and condition codes (informative) 175
Annex F	IEEE 1754 Reference MMU architecture (informative) 181
F.1	Introduction 181
F.2	Overview 181
F.3	Software architecture 181
F.3.1	Contexts 185
F.3.2	Page table descriptors (PTDs) 185
F.3.3	Page table entry (PTE) 186
F.3.4	MMU flush and probe model 188
F.4	Hardware architecture 191
F.4.1	Accessing MMU registers 191
F.4.2	Control register 192
F.4.3	Context table pointer register 193
F.4.4	Context register 193
F.4.5	Diagnostic registers 193
F.5	MMU fault status register (MFSR) 194
F.6	MMU fault address register (MFAR) 197
F.7	Operation 197
F.7.1	Reset 197
F.7.2	Miss processing 198
F.7.3	Referenced and modified bit updates 198
Annex G	Suggested ASI assignments (informative) 199
G.1	Introduction 199
G.2	ASI summary 199

ANNEX	PAGE
G.3 Detailed descriptions	201
G.3.1 ASI = 0 (reserved)	201
G.3.2 ASI = 1 (unassigned)	201
G.3.3 ASI = 2 (unassigned system registers)	201
G.3.4 ASI = 3 (MMU flush/probe)	201
G.3.5 ASI = 4 (MMU registers)	202
G.3.6 ASI = 5 (MMU I diagnostic)	202
G.3.7 ASI = 6 (MMU D/I&D diagnostic)	202
G.3.8 ASI = 7 (MMU I/O diagnostic)	202
G.3.9 ASI = 8 (user I)	202
G.3.10 ASI = 9 (supervisor I)	203
G.3.11 ASI = A ₁₆ (user D)	203
G.3.12 ASI = B ₁₆ (supervisor D)	203
G.3.13 ASI = C ₁₆ (I-cache tag)	203
G.3.14 ASI = D ₁₆ (I-cache data)	203
G.3.15 ASI = E ₁₆ (D/I&D-cache tag)	203
G.3.16 ASI = F ₁₆ (D/I&D-cache data)	203
G.3.17 ASI = 10 ₁₆ –14 ₁₆ (flush I&D)	204
G.3.18 ASI = 17 ₁₆ (block copy)	204
G.3.19 ASI = 1C ₁₆ (flush I)	204
G.3.20 ASI = 1F ₁₆ (block fill)	205
G.3.21 ASI = 20 ₁₆ –2F ₁₆ (pass-through)	205
G.3.22 ASI = 30 ₁₆ –7F ₁₆ (unassigned)	205
G.3.23 ASI = 80 ₁₆ –FF ₁₆ (reserved)	205
Annex H Example integer multiplication and division routines (informative)	207
H.1 Signed multiplication	207
H.2 Unsigned multiplication	209
H.3 Division	211
H.3.1 Program 1	212
H.3.2 Program 2	212
H.3.3 Program 3	213
H.3.4 Program 4	215
H.3.5 Program 5	216
H.3.6 Program 6	217
Annex I Suggested assembly language syntax (informative)	223
I.1 Notation used	223
I.1.1 Register names	223
I.1.2 Special symbol names	224
I.1.3 Values	224
I.1.4 Labels	225
I.1.5 Other operand syntax	225
I.1.6 Comments.....	225
I.2 Syntax design	225
I.3 Synthetic instructions	226
Annex J Software considerations (informative)	229
J.1 Registers	229
J.1.1 <i>In</i> and <i>out</i> registers	229
J.1.2 <i>Local</i> registers	230
J.1.3 Register windows and %sp	231

ANNEX	PAGE
J.1.4 Global registers	231
J.1.5 Floating-point registers	232
J.2 The memory stack	233
J.3 Functions returning aggregate values	234
J.4 Tagged arithmetic	235
J.5 Leaf procedure optimization	236
J.6 Example code	237
J.7 Register allocation within a window	238
J.8 Other register window usage models	238
J.9 Non-IEEE 1754 modes of operation	240
J.10 Self-modifying code	240
Annex K Instruction set summary (informative)	241
K.1 Introduction	241
K.2 Data transfer	241
K.3 Arithmetic and logical operations	241
K.4 Control transfer	242
K.5 Miscellaneous	242
K.6 Floating-point operations.....	243
Annex L Non-IEEE 1754 architectural extensions (informative).....	245
L.1 IEEE 1754 standard and non-standard operation	245
L.2 Addition of non-IEEE 1754 extensions	245
L.2.1 Support for little-endian addressing	246
L.3 Modification of existing IEEE 1754 instructions	246
L.4 Extending IEEE 1754 to 64 bits	246
Annex M Bibliography (informative)	247
Index	249

IEEE Standard for a 32-bit Microprocessor Architecture

1. Overview

IEEE Std 1754-1994 defines a 32-bit microprocessor architecture (referred to herein as IEEE 1754) that is derived from SPARC and other existing architectures. This standard includes, but is not limited to, the definition of the instruction set, register model, data types, instruction opcodes, and coprocessor interface for this architecture. IEEE Std 1754-1994 is relevant to microprocessor applications for which the attributes described in 1.1.1 are desirable.

IEEE 1754 is not intended to be the only IEEE standard microprocessor architecture; other standards may be defined in the future.

1.1 Scope

1.1.1 IEEE 1754 attributes

IEEE 1754 is a CPU **instruction set architecture (ISA)** derived from a reduced instruction set computer (RISC) lineage. As an architecture, IEEE 1754 allows for a spectrum of chip and system **implementations** at a variety of price/performance points for a range of applications including scientific/engineering, programming, real-time, and commercial.

IEEE 1754 is designed to be a target for optimizing compilers and easily pipelined hardware implementations.

IEEE 1754 is derived from SPARC, which was formulated at Sun Microsystems in 1985. SPARC is based on the RISC I & II designs engineered at the University of California at Berkeley from 1980 through 1982. RISC I & II, SPARC, and IEEE 1754 make use of register windows.

One difference between IEEE 1754 and the Berkeley RISC I & II is that IEEE 1754 provides greater flexibility to a compiler in its assignment of registers to program variables. IEEE 1754 is more flexible because register window management is not tied to procedure call and return instructions, as it is on the Berkeley machines. Instead, separate instructions (SAVE and RESTORE) provide register window management.

Note that supervisor software, not user software, manages the register windows. A supervisor can save a minimum number of registers (approximately 24) at the time of a context switch, thereby keeping context switch latency low.

1.2 IEEE 1754 system components

The architecture allows for a spectrum of input/output (I/O), memory management unit (MMU), and cache system subarchitectures. IEEE 1754 assumes that these elements are optimally defined by the specific requirements of particular systems. Note that they are invisible to nearly all user programs and the interfaces to them can be limited to localized modules in an associated operating system.

1.2.1 Reference MMU

The IEEE 1754 ISA does not mandate that a single MMU design be used for all system implementations. Rather, designers are free to use the MMU that is most appropriate for their application, or no MMU at all, if

they wish. An IEEE 1754 Reference MMU is specified, which is appropriate for a wide range of applications. See Annex F, “IEEE 1754 Reference MMU architecture,” for more information.

If an MMU other than the Reference MMU is used, it is recommended that it supports at least the following features:

- Supervisor versus user access privileges
- Writeable protection
- A page size of at least 4 Kbytes

1.2.2 Supervisor software

IEEE 1754 does not assume that all implementations will execute identical supervisor software. Thus, certain supervisor-visible traits of an implementation can be tailored to the requirements of the system. For example, IEEE 1754 allows for implementations with different instruction concurrency and different exception trap hardware.

1.3 Binary compatibility

IEEE 1754 provides an ISA. It is intended that when this architecture is combined with an operating environment, the resulting system should be able to provide binary compatibility of user programs across implementations. Furthermore, this allows a system developer to implement binary interfaces for application programs in either the IEEE 1754 hardware or in the operating system.

Although different IEEE 1754 systems will execute user programs at different rates, it shall otherwise be possible to generate the same results barring time dependencies in the programs.

This standard provides one possible architecture that could be used as a basis for transfer of binary applications between equivalent (“clone”) systems. Additional standards required to accomplish this must include the following:

- Standards for all necessary system interface services.
- Standards for **all** interfaces available to applications that are not part of the binary image of the application. This includes database access (e.g., SQL), graphics services (e.g., GKS or PHiGS), windowing/display services, network access services, data encoding/formatting (e.g., ODIF, STEP, EDI), services, management services, shell and utility services, and language-specific services (e.g., keyed record access), etc.
- Standards for service access (trap, call, exception, shared library, etc.) and all parameter passing (register use, stack, pointers) for **all** of the services and interfaces previously mentioned.
- Standards for all interface-defined data types (including strings) and interface-supported data formats (for example, ISO 10646 [B8]¹ character set for character/string data) used by all of the services above.
- Standards for application (binary) accessible hardware capabilities, such as MMU, floating-point hardware, implementation-dependent elements (number of registers, coprocessor access, I/O access, etc. [see Annex D, “IEEE 1754 implementation dependencies”]) are some examples that may also be required.
- Standard test methods and assertions for the above (including the interactions between each of the above service interfaces) to ensure that the specifications are sufficiently unambiguous to permit independent implementation and conformance testing.

¹The numbers in brackets preceded by the letter B correspond to those of bibliography in Annex M.

Application binary interface (ABI) specifications tend to require a specific licensed (proprietary) operating system with a number of explicit (and some implicit) specifications of the above elements. Dependence on specifications that are outside the control of a public consensus process is inconsistent with the open, non-proprietary objectives of this standard. The IEEE definition for “open system” (IEEE P1003.0 [B5]) indicates that only properly engineered applications can obtain the benefits of interoperability and portability. Additional engineering discipline and investment is needed to obtain an application that is portable and interoperable in conformance to source code level standards (such as ISO/IEC 9945-1:1990 [B7]) as well as transferable among ABI equivalent systems. Without such discipline, and conformance to the higher (source code) level standards, applications will not be any more portable to future generations of technology than other proprietary, implementation-dependent applications today.

1.4 IEEE 1754 features

IEEE 1754 includes the following principal features:

- A **linear address space** with 32-bit addressing.
- **Few and simple instruction formats.** All instructions are 32 bits wide and are aligned on 32-bit boundaries in memory. There are only three basic instruction formats and they feature uniform placement of opcode and register address fields. Only load and store instructions access memory and perform I/O.
- **Few addressing modes.** A memory address is given as either “register + register” or “register + immediate.”
- **Triadic register addresses.** Most instructions operate on two-register operands or one register and a constant, and place the result in a third register.
- A **“windowed” register file.** At any one instant, a program sees 8 global integer registers plus a 24-register window of a larger register file. The windowed registers often are used as a cache of procedure arguments, local values, and return addresses.
- **Floating-point.** The architecture provides an IEEE 754² compliant floating-point instruction set, operating on a separate register file that provides 32 single-precision (32-bit) operands.
- **Delayed control transfer.** The processor always fetches the next instruction after a delayed control-transfer instruction. It either executes it or not, depending on the control-transfer instruction’s “annul” bit.
- **Fast trap handlers.** Traps are vectored through a table and cause allocation of a fresh register window in the register file.
- **Tagged instructions.** The tagged add/subtract instructions assume that the two least-significant bits of the operands are tag bits.
- **Multiprocessor synchronization instructions.** One instruction performs an atomic read-then-set-memory operation; another performs an atomic exchange-register-with-memory operation; another is used to synchronize the order of shared memory operations as observed by processors.
- **Coprocessor.** The architecture defines a straightforward coprocessor instruction set, in addition to the floating-point instruction set.

1.5 Conformance to IEEE 1754

An implementation that conforms to the definitions and algorithms given in the normative parts of this document (including all clauses and all normative annexes) is an implementation of the IEEE 1754 ISA.

The IEEE 1754 architecture is a **model** that specifies unambiguously the behavior observed by user-provided **software** on IEEE 1754 systems. Therefore, it does not necessarily describe the operation of the **hardware** in any actual implementation.

²Information on references can be found in 1.7.

Some elements of the architecture are defined to be implementation-dependent. These elements include certain registers and operations that may vary from implementation to implementation, and are explicitly identified as such in this document using the notation “**Impl. Dep. #nn:**” (see “References to implementation dependencies” in the Introduction). Annex D, “IEEE 1754 implementation dependencies,” explains and describes all of these references.

A collection of components compliant to IEEE 1754 shall

- Provide a complete enumeration of implementation choices made that correspond to the list of permitted choices in Annex D of this standard. See 1.6, “Conformance document.”
- Fully implement each nonprivileged instruction or trap such that it is possible to emulate it. Note that it is not required for every nonprivileged instruction to be emulated, only that it be possible to emulate it.

Impl. Dep. #1: Each privileged instruction in an implementation may

- Be fully implemented,
- Cause a trap and be emulated, or
- Be unimplemented.

Implementation elements (such as instructions or registers) that appear in an implementation but are not defined in this document (or its updates) are not considered to be IEEE 1754 elements of that implementation.

It is the responsibility of the implementor to ensure conformance to this standard.

1.6 Conformance document

For every implementation claiming conformance to this standard, a conformance document shall be made available to the IEEE-specified Registration Authority. The conformance document shall

- Reference the name, number, and date of this standard.
- Contain a list of choices for all implementation dependencies listed in Annex D, “IEEE 1754 implementation dependencies.” In particular, assigned values must be documented (see D.3, “Categories of architectural implementation dependencies”).
- Specify how non-IEEE 1754 extensions (if any) have been provided, as recommended in Annex L, “Non-IEEE 1754 architectural extensions.”
- Specify how any aspect marked in the standard as “undefined” behaves in that implementation.

1.7 References

This standard shall be used in conjunction with the following publications. When they are superseded by an approved revision, the revision shall apply:

IEEE Std 754-1985 (Reaff 1990), IEEE Standard for Binary Floating-Point Arithmetic (ANSI).³

³IEEE publications are available from the Institute of Electrical and Electronics, Inc., 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

2. Definitions, special word usage, abbreviations, and acronyms

The following subclauses define the most important terms, abbreviations, and acronyms used in this standard. Terms within definitions that are highlighted in **bold** are defined in 2.1.

2.1 Definitions

2.1.1 address space identifier (ASI): An 8-bit field appended to the address by the **integer unit**. It identifies the address space being accessed and typically encodes whether the processor is in user or supervisor mode.

2.1.2 annul bit: A bit in a delayed control-transfer instruction that can cause the delay instruction to have no effect.

2.1.3 application program: A program executed with the processor in **user mode**.

NOTE—Statements made in this document regarding application programs may be inapplicable to programs (for example, debuggers) that have access to privileged processor state (e.g., as stored in a memory-image dump).

2.1.4 byte: Eight consecutive bits of data.

NOTE—A byte is not necessarily equivalent to a character.

2.1.5 coprocessor (CP): An optional processing unit (impl. dep. #4), the operation of which is partially defined by this standard.

2.1.6 coprocessor operate (CPop) instructions: Instructions that perform coprocessor calculations, as defined by the CPop1 and CPop2 opcodes. CPop instructions do not include CBccc instructions, nor loads and stores between memory and the coprocessor.

2.1.7 current window: The block of 24 *r* registers to which the current window pointer (CWP) points.

2.1.8 delay instruction: The instruction following a delayed control-transfer instruction. The delay instruction is always fetched, even if the delayed control transfer is an unconditional branch. However, the **annul bit** in the delayed control-transfer instruction can cause the delay instruction to be annulled (that is, to have no effect) if the branch is not taken (or in the branch-always case, if the branch is taken).

2.1.9 dispatch: Issue a fetched instruction to one or more functional units for execution.

2.1.10 doublet: Two bytes (16 bits) of data.

2.1.11 doubleword: An aligned **octlet**.

NOTE—The definition of this term is architecture-dependent, and so may be different from that used in other processor architectures.

2.1.12 *f* register: One of the floating-point registers.

2.1.13 floating-point exception: An *fp_exception* of one of the following floating-point trap types, caused by execution of an FPop: *unfinished_FPop*, *unimplemented_FPop*, *sequence_error*, *hardware_error*, *invalid_fp_register*, or floating-point *IEEE_754_exception* that occurs while the corresponding bit in FSR.TEM is set to 1.

2.1.14 floating-point IEEE-754 exception: A floating-point exception, as specified by IEEE Std 754-1985.

2.1.15 floating-point trap type: The specific type of floating-point exception that has occurred, encoded in the FSR.*ftt* field.

2.1.16 floating-point operate (FPop) instructions: Instructions that perform floating-point calculations, as defined by the FPop1 and FPop2 opcodes. FPop instructions do not include FBfcc instructions, nor loads and stores between memory and the FPU.

2.1.17 floating-point unit (FPU): A processing unit that contains a set of floating-point registers and performs floating-point operations, as defined by this standard.

2.1.18 halfword: An aligned **doublet**.

NOTE—The definition of this term is architecture-dependent, and so may differ from that used in other processor architectures.

2.1.19 hexlet: Sixteen bytes (128 bits) of data.

2.1.20 ignored: Used to describe an instruction field, the contents of which are arbitrary and have no effect on the execution of the instruction. The contents of an ignored field will continue to be ignored in future versions of the architecture. *See also:* **reserved; unused.**

2.1.21 implementation: Hardware and/or software that conforms to all the specifications of an ISA.

2.1.22 implementation-dependent: Describes an aspect of the architecture that may legitimately vary among implementations of the architecture. In many cases, the permitted range of variation is specified in the standard. When a range is specified, compliant implementations shall not deviate from that range. Compliant implementations shall not add to or deviate from this standard except in aspects described as **implementation-dependent**. (See Annex D, “IEEE 1754 implementation dependencies.”)

2.1.23 informative annex: An annex that is included in the standard for information only and is not an official part of the standard itself.

2.1.24 instruction field: A bit field within an instruction word.

2.1.25 instruction set architecture (ISA): An ISA defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. An ISA does not define clock-cycle times, cycles per instruction, data paths, etc. This standard defines an ISA.

2.1.26 integer unit: A processing unit that performs integer and control-flow operations and contains general-purpose integer registers and processor state registers, as defined by this standard.

2.1.27 next program counter (nPC): Contains the address of the instruction to be executed next (if a trap does not occur).

2.1.28 normative annex: An annex that is an official part of the standard but is placed after the clauses of the standard for convenience or to create a hierarchical distinction.

2.1.29 octlet: Eight bytes (64 bits) of data. Not to be confused with an “octet,” which has been used to describe 8 bits of data. In this document, the term **byte**, rather than “octet,” is used to describe 8 bits of data.

2.1.30 opcode: A bit pattern that identifies a particular instruction.

2.1.31 privileged: An instruction (or register) that can only be executed (or accessed) when the processor is in supervisor mode (PSR.S = 1). *See also:* **supervisor mode.**

2.1.32 processor: The combination of the IU, FPU, and CP (if present).

2.1.33 program counter (PC): Contains the address of the instruction currently being executed by the IU.

2.1.34 quadlet: Four bytes (32 bits) of data.

2.1.35 quadword: An aligned **hexlet**.

NOTE—The definition of this term is architecture-dependent, and so may differ from that used in other processor architectures.

2.1.36 *r* register: One of the integer registers.

2.1.37 *rs1*, *rs2*, *rd*: The register operands of an instruction. *rs1* and *rs2* are the source registers; *rd* is the destination register.

2.1.38 reserved: Used to describe an instruction field or register field that is reserved for definition by future versions of the architecture. A reserved field should only be written to zero by software. A reserved register field shall read as zero in hardware; software intended to run on future versions of IEEE 1754 should not assume that the field will read as zero. *See also:* **ignored; unused.**

2.1.39 supervisor mode: A processor state that is active when the S bit of the PSR is set (PSR.S = 1). *See also:* **privileged.**

2.1.40 supervisor software: Software that executes when the processor is in **supervisor mode**.

2.1.41 trap: A vectored transfer of control to supervisor software through a table, the address of which is specified by a privileged **integer unit** register referred to as the trap base register (TBR).

2.1.42 unassigned: Describes a value (for example, an **address space identifier [ASI]** number), the semantics of which are not architecturally mandated and may be determined independently by each implementation (preferably within any guidelines given).

2.1.43 undefined: Describes an aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, nor make any assumptions about, such an architectural feature or behavior. Use of such a feature may deliver random results, may or may not cause a trap, may vary among implementations, and may vary with time on a given implementation. Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes, such as allowing user software to access supervisor state, put the processor into supervisor state, or put the processor into an unrecoverable state.

2.1.44 unpredictable: Describes an aspect of the architecture that is nondeterministic. This term is used only to describe the targets of branches and the occurrence of certain traps in unusual situations. *See also:* **undefined.**

2.1.45 unused: Used to describe an instruction field or register field that is not currently defined by the architecture. When read by software, the value of an unused register field is undefined. However, since an unused field could be defined by a future version of the architecture, an unused field should only be written to zero by software. *See also:* **ignored; reserved.**

2.1.46 user mode: A processor state that is active when the S bit of the PSR is not set (when PSR.S = 0).

2.1.47 user application program: *See:* **application program**; also referred to as “user program.”

2.1.48 word: An aligned **quadlet**.

NOTE—The definition of this term is architecture-dependent, and so may differ from that used in other processor architectures.

2.2 Special word usage

2.2.1 may: An IEEE-defined word used to indicate a course of action permissible within the limits of the standard. *Synonym:* is permitted.

2.2.2 shall: An IEEE-defined word indicating requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted. *Synonym:* is required.

2.2.3 should: An IEEE-defined word indicating that among several possibilities one is recommended as particularly suitable. *Synonym:* is recommended.

2.3 Acronyms and abbreviations

ABI:	application binary interface
ASI:	address space identifier
ASR:	ancillary state register
CP:	coprocessor
CPop:	coprocessor operate
CQ:	coprocessor deferred-trap queue
CSR:	coprocessor state register
CTI:	control-transfer instruction
CWP:	current window pointer
DCTI:	delayed control-transfer instruction
EC:	enable coprocessor
EF:	enable floating-point unit
FPop:	floating-point operate
FPU:	floating-point unit
FQ:	floating-point deferred-trap queue
FSR:	floating-point state register
I/O:	input/output
IRL:	interrupt request level
ISA:	instruction set architecture
IU:	integer unit
MMU:	memory management unit
nPC:	next program counter
PC:	program counter
PIL:	processor interrupt level
PSR:	processor state register
RDASR:	read ancillary state register
RISC:	reduced instruction set computer
TBA:	trap base address
TBR:	trap base register
TEM:	trap enable mask
WIM:	window invalid mask
WRASR:	write ancillary state register
Y:	multiply/divide register

3. Architectural overview

IEEE 1754 is an **instruction set architecture (ISA)** with principal data types of 32-bit integer and 32-, 64-, and 128-bit IEEE 754 floating point. IEEE 1754 defines general-purpose integer, floating-point, and special state/status registers and basic instruction operations, all encoded in 32-bit instruction formats. The load/store instructions address a linear, 2^{32} -byte address space. In addition to the basic instructions, IEEE 1754 also provides instruction set support for an optional implementation-defined coprocessor (impl. dep. #4).

3.1 IEEE 1754 processor

A IEEE 1754 processor logically consists of an **integer unit (IU)**, a **floating-point unit (FPU)**, and an optional **coprocessor (CP)** (impl. dep. #4), each with its own registers. This organization allows for implementations with concurrency between integer, floating-point, and coprocessor instruction execution. The IU and FPU may be implemented by any combination of hardware and software in conformance with the provisions specified in 1.5, “Conformance to IEEE 1754.”

All of the registers—with the possible exception of the coprocessor’s—are 32 bits wide. Instruction operands are single registers, register pairs, register quadruples, or immediate constants.

When in `execute_state` (see B.6, “Processor states and instruction dispatch”), the processor can be in either of two modes: **supervisor** or **user**. In supervisor mode, the processor can execute any instruction, including the privileged (supervisor-only) instructions. In user mode, an attempt to execute a privileged instruction shall cause a trap to supervisor software. User application programs execute while the processor is in user mode.

3.1.1 Integer unit (IU)

The IU contains the general-purpose registers and controls the overall operation of the processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU and the CP.

An implementation of the IU shall contain from 40 to 520 general-purpose 32-bit *r* registers. This corresponds to a grouping of the registers into 8 **global** *r* registers, plus a circular stack of from 2 to 32 sets of 16 registers each, known as a **register window**. **Impl. Dep. #2:** Since the number of register windows present (NWINDOVS) is implementation-dependent, the total number of integer registers is implementation-dependent. An implementation could have differing numbers of register windows, provided that a run-time mechanism for determining NWINDOVS is provided.

At a given time, an instruction can access the 8 **globals** and a register **window** into the *r* registers. The 24-register window consists of a 16-register set—divided into 8 *in* and 8 *local* registers—together with the 8 *in* registers of an adjacent register set, addressable from the current window as its *out* registers.

The current window is specified by the current window pointer (CWP) field in the processor state register (PSR). Window overflow and underflow are detected via the window invalid mask (WIM) register, which is controlled by supervisor software. With appropriate support in supervisor software, the actual number of windows in a IEEE 1754 implementation is invisible to a user application program.

When the IU accesses memory, it appends an **address space identifier (ASI)** to the address. The ASI identifies the address space being accessed, and typically encodes whether the processor is in supervisor or user mode and whether the access is to instruction memory or to data memory.

3.1.2 Floating-point unit (FPU)

The FPU has thirty-two 32-bit floating-point f registers. Double-precision values occupy an even-odd pair of registers, and quad-precision values occupy a quad-aligned group of four registers. Thus, the floating-point registers can hold a maximum of either 32 single-precision, 16 double-precision, or 8 quad-precision values.

Floating-point load/store instructions are used to move data between the FPU and memory. The memory address is calculated by the IU. **Floating-point operate (FPop)** instructions perform the floating-point arithmetic operations and comparisons.

The floating-point data formats and instruction set conform to IEEE Std 754-1985. However, IEEE 1754 does not require that all aspects of the standard, such as gradual underflow, be implemented in hardware. **Impl. Dep. #3:** An implementation may indicate that a floating-point instruction did not produce a correct IEEE Std 754-1985 result by generating a special floating-point unfinished or unimplemented exception. Privileged-mode software shall emulate any functionality not present in the hardware.

If an FPU is not present or if the **enable floating-point (EF)** bit in the PSR is 0, an attempt to execute a floating-point instruction generates an *fp_disabled* trap. In any of these cases, privileged-mode software shall either enable the FPU and re-execute the trapped instruction, or emulate the trapped floating-point instruction.

3.1.3 Coprocessor (CP)

Impl. Dep. #4: The instruction set includes support for a single, implementation-dependent coprocessor. The coprocessor may contain its own set of registers, the actual configuration of which is implementation-defined but is nominally some number of 32-bit registers (impl. dep. #26). Coprocessor load/store instructions are used to move data between the coprocessor registers and memory. For each floating-point load/store in the instruction set, there is an analogous coprocessor load/store instruction.

If a CP is not present or if the **enable coprocessor (EC)** bit in the PSR is 0, an attempt to execute a coprocessor instruction generates a *cp_disabled* trap.

This standard describes a model of coprocessor operation in terms of an independent functional unit with an independent register set. Other models are possible and the actual operation is implementation-dependent (impl. dep. #4).

3.2 Instructions

Instructions fall into the following six basic categories, which are described in 3.2.1–3.2.6:

- Memory access
- Arithmetic/logical/shift
- Control transfer
- State register access
- Floating-point operate
- Coprocessor operate

3.2.1 Memory access

Load-store instructions are the only instructions that access memory. They use two r registers or an r register and a signed 13-bit immediate value to calculate a 32-bit, byte-aligned memory address. The IU appends to this address an ASI that identifies the address space being accessed. Normal loads and stores encode whether

the processor is in supervisor or user mode, and that it is a data access. Loads and stores to alternate space supply an arbitrary ASI value.

The destination field of the load/store instruction specifies either an *r* register, a pair of *r* registers, an *f* register, a pair of *f* registers, or a coprocessor register that supplies the data for a store or receives the data from a load.

Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and doubleword (64-bit) accesses. There are versions of integer load instructions that perform sign-extension on 8- and 16-bit values as they are loaded into a 32-bit destination register. Floating-point and coprocessor load and store instructions support word and doubleword memory accesses.

3.2.1.1 Memory alignment restrictions

Halfword accesses shall be **aligned** on 2-byte boundaries, word accesses (which include instruction fetches) shall be aligned on 4-byte boundaries, and doubleword accesses shall be aligned on 8-byte boundaries. An improperly aligned address in a load, store, or load-store instruction shall cause a trap to occur.

Quadword quantities in memory should be aligned on 16-byte boundaries. Since there are no quadword memory access instructions, no trap occurs if quadwords are not so aligned. Sixteen-byte alignment of quadwords in memory is recommended for compatibility with future versions of this standard, which may provide quadword memory access instructions requiring such alignment.

3.2.1.2 Addressing conventions

IEEE 1754 shall use a memory addressing convention in which the address of a doubleword, word, or halfword is the address of its most significant byte (sometimes referred to as **big-endian** byte order). Increasing the address means decreasing the significance of the unit being accessed. Addressing conventions are illustrated in Clause 6, “Instructions.”

For information about also supporting the **little-endian** addressing convention, see L.2.1, “Support for little-endian addressing.”

3.2.1.3 Load/store alternate

There are special, privileged versions of the load/store integer instructions, the **load/store alternate** instructions, that directly specify an arbitrary 8-bit address space identifier for the load/store data access. **Impl. Dep. #5:** Privileged load/store alternate instructions may be used by supervisor software to access special protected registers, such as MMU, cache control, and processor state registers, and other processor- or system-dependent values.

3.2.1.4 Separate instruction and data memories

Most specifications in this manual are written as if store instructions write to the same memory from which instructions are accessed. However, a system implementation containing an IEEE 1754 processor may explicitly partition instructions and data into independent instruction and data memories (caches), commonly referred to as a “Harvard architecture” or “split I & D caches.” If software that modifies its own instruction stream (self-modifying code) executes on a system where writes to data memory do not immediately appear in instruction memory, the software shall issue FLUSH instructions (or supervisor calls that have an equivalent effect) for the addresses to which new instructions were written. See J.10, “Self-modifying code.” A FLUSH instruction ensures that the data previously written by a store instruction is seen by subsequent instruction fetches from the given address.

In implementations with separate, noncoherent instruction and data memories, the processor must execute appropriate FLUSH instructions before any changes to data memory by that processor or any external device can be reliably fetched as instructions.

Although some processors define cache-control instructions, to provide flexibility to implementations, cache-control instructions are not specified in this standard. However, IEEE 1754 implementations with separate instruction and data memories are expected to provide appropriate control mechanisms for them.

3.2.1.5 Input/output (I/O)

IEEE 1754 assumes that I/O registers are accessed via load/store alternate instructions, normal load/store instructions, coprocessor instructions, or read/write ancillary state register (RDASR/WRASR) instructions. In the load/store alternate instructions case, the I/O registers can only be accessed by supervisor software. **Impl. Dep. #6:** If normal load/store instructions, coprocessor instructions, or RDASR/WRASR instructions are used, whether or not the I/O registers can be accessed outside of supervisor code is implementation-dependent. **Impl. Dep. #7:** The addresses and contents of I/O registers are implementation-dependent.

3.2.1.6 Memory synchronization

Two special instructions are used for synchronization of memory operations: FLUSH and STBAR. Their operation is explained in Annex A, “Instruction definitions.”

3.2.2 Arithmetic/logical/shift

Arithmetic/logical/shift instructions perform arithmetic, tagged arithmetic, logical, and shift operations. With one exception, these instructions compute a result that is a function of two source operands; the result is either written into a destination register or discarded. The exception is a specialized instruction, SETHI, which (along with a second instruction) can be used to create a 32-bit constant in an *r* register.

Shift instructions can be used to shift the contents of an *r* register left or right by a given distance. The shift distance may be specified by a constant in the instruction or by the contents of an *r* register.

Integer multiply instructions perform a signed or unsigned $32 \times 32 \rightarrow 64$ -bit operation. Integer divide instructions perform a signed or unsigned $64 \div 32 \rightarrow 32$ -bit operation. There are versions of multiply and divide that set the condition codes. Division by zero causes a trap.

Tagged arithmetic instructions assume that the least significant two bits of the operands are data-type “tags.” These instructions set the overflow condition code bit upon arithmetic overflow or if any of the operands’ tag bits are nonzero. There are also variants of these instructions that trap (in addition to setting the overflow condition code bit) when either of these conditions occurs.

3.2.3 Control transfer

Control-transfer instructions (CTIs) include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are **delayed control-transfer instructions (DCTIs)**; that is, the instruction immediately following the DCTI is executed before the control transfer to the target address is completed.

The instruction following a delayed control-transfer instruction is called a **delay** instruction. The delay instruction is always fetched, even if the delayed control transfer is an unconditional branch. However, a bit in the delayed control-transfer instruction can cause the delay instruction to be annulled (that is, to have no effect) if the branch is not taken (or in the branch always case, if the branch is taken).

Branch and CALL instructions use PC-relative displacements. The jump and link (JMPL) and return from trap (RETT) instructions use a register-indirect target address. They compute their target addresses as either the sum of two *r* registers, or the sum of an *r* register and a 13-bit signed immediate value. A branch instruction provides a displacement of ± 8 Mbytes, while the CALL instruction's 30-bit word displacement allows a control transfer to an arbitrary address in the 4 Gbyte address space.

3.2.4 State register access

The read/write state register instructions read and write the contents of software-visible state/status registers.

Impl. Dep. #8: Software can use RDASR/WRASR to read/write unique implementation-dependent processor registers. **Impl. Dep. #9:** Whether or not each of the implementation-dependent RDASR/WRASR instructions is privileged is implementation-dependent.

3.2.5 Floating-point operate (FPop)

FPop instructions perform all floating-point calculations. They are register-to-register instructions that operate upon the floating-point registers. Like arithmetic/logical/shift instructions, FPop's compute a result that is a function of one or two source operands. Specific floating-point operations are selected by a subfield of the FPop1/FPop2 instruction formats.

3.2.6 Coprocessor operate (CPop)

CPop instructions are defined by the implemented coprocessor, if any. These instructions are specified by the CPop1 and CPop2 instruction formats.

3.3 Traps

A **trap** is a vectored transfer of control to the operating system through a trap table that contains the first four instructions of each trap handler. The base address of the table is established by software in the trap base register (TBR). The displacement within the table is encoded in the type number of each trap. One half of the table is reserved for hardware traps; the other half is reserved for software traps generated by trap (Ticc) instructions.

A trap causes the current window pointer (CWP) to advance to the next register window and the hardware to write the program counters into two registers of the new window. The trap handler can access the saved PC and nPC and, in general, may freely use the 6 other local registers in the new window.

A trap may be caused by a Ticc instruction, an instruction-induced **exception**, or an **interrupt request** not directly related to a particular instruction. The processor behaves as though, before executing each instruction,

- a) It determines if there are any pending exceptions or interrupt requests, and
- b) If there are pending exceptions or interrupt requests, the processor selects the highest-priority exception or interrupt request and causes a trap.

See Clause 7, "Traps," for a complete description of traps.

4. Data formats

4.1 Overview

The IEEE 1754 architecture recognizes three fundamental data formats or types:

- Signed integer: 8, 16, 32, and 64 bits
- Unsigned integer: 8, 16, 32, and 64 bits
- Floating-point: 32, 64, and 128 bits

The format widths are defined as follows:

- Byte: 8 bits
- Halfword: 16 bits
- Word: 32 bits
- Tagged word: 32 bits (30-bit value plus 2 tag bits)
- Doubleword: 64 bits
- Quadword: 128 bits

The signed integer formats encode two's-complement whole numbers. The unsigned integer formats are general-purpose in that they do not encode any particular data type; they can represent a whole number, string, fraction, Boolean value, etc. The floating-point formats conform to IEEE Std 754-1985. In tagged words, the least significant two bits are treated as tag bits.

For consistency with the documentation for many big-endian architectures, especially SPARC, bits are numbered right to left (the least significant bit of a data item is numbered "0") in this standard. Note, however, that compilers assign bit fields in big-endian data items left to right; that is, starting with the most significant bit. For a more consistent big-endian bit numbering scheme, see IEEE Std 1596.5-1993 [B6].

Subclauses 4.2–4.11 illustrate the signed integer, unsigned integer, and tagged formats. Subclauses 4.12–4.14 illustrate the floating-point formats. In subclauses 4.5, 4.9, 4.13, and 4.14, the individual subwords of the multiword data formats are assigned names. The arrangement of the subformats in memory and processor registers based on these names is shown in table 1. Tables 2–5 define the integer and floating-point formats.

4.2 Signed integer byte

Figure 1 illustrates the signed integer byte format.

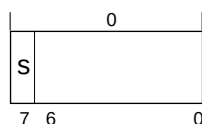


Figure 1—Signed integer byte data format

4.3 Signed integer halfword

Figure 2 illustrates the signed integer halfword format.

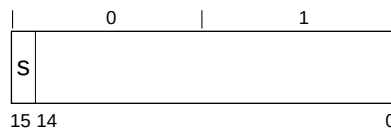


Figure 2—Signed integer halfword data format

4.4 Signed integer word

Figure 3 illustrates the signed integer word format.

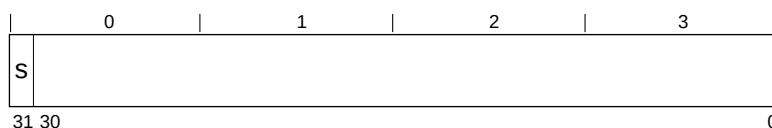


Figure 3—Signed integer word data format

4.5 Signed integer double

Figure 4 illustrates the signed integer double format.

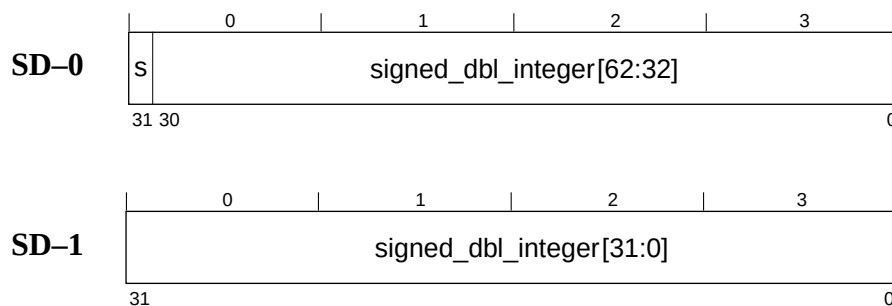


Figure 4—Signed integer double data format

4.6 Unsigned integer byte

Figure 5 illustrates the unsigned integer byte format.

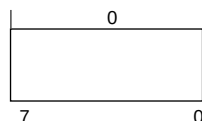


Figure 5—Unsigned integer byte data format

4.7 Unsigned integer halfword

Figure 6 illustrates the unsigned integer halfword format.

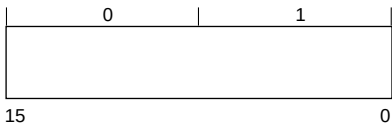


Figure 6—Unsigned integer halfword data format

4.8 Unsigned integer word

Figure 7 illustrates the unsigned integer word format.

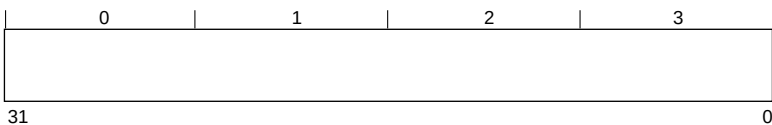


Figure 7—Unsigned integer word data format

4.9 Unsigned integer double

Figure 8 illustrates the unsigned integer double format.

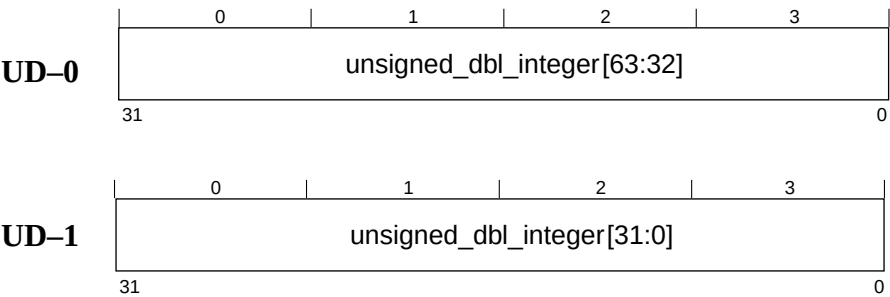


Figure 8—Unsigned integer double data format

4.10 Unsigned tagged word

Figure 9 illustrates the unsigned tag word format.



Figure 9—Unsigned tag word data format

4.11 Signed tagged word

Figure 10 illustrates the signed tag word format.

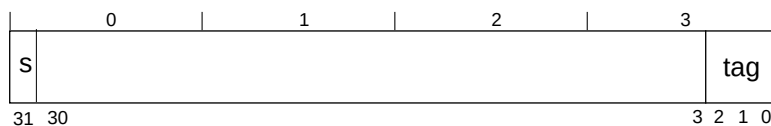


Figure 10—Signed tag word data format

4.12 Floating-point single precision

Figure 11 illustrates the floating-point single-precision format.

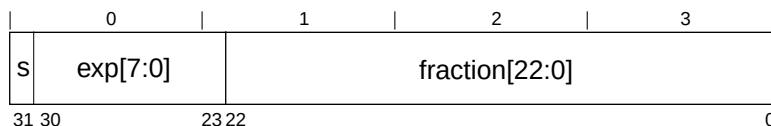


Figure 11—Floating-point single-precision data format

4.13 Floating-point double precision

Figure 12 illustrates the floating-point double-precision format.

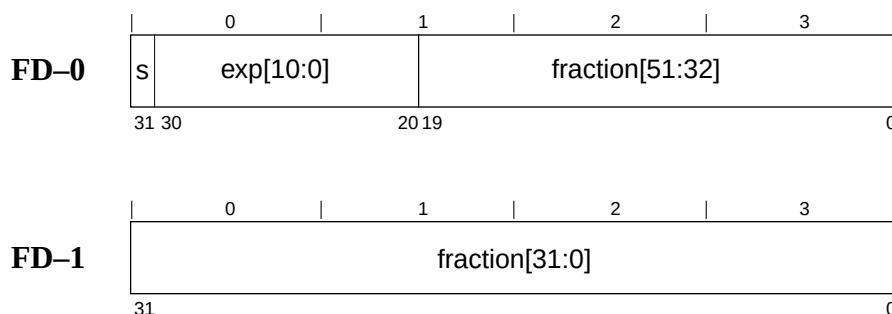


Figure 12—Floating-point double-precision data format

4.14 Floating-point quad precision

Figure 13 illustrates the floating-point quad-precision format.

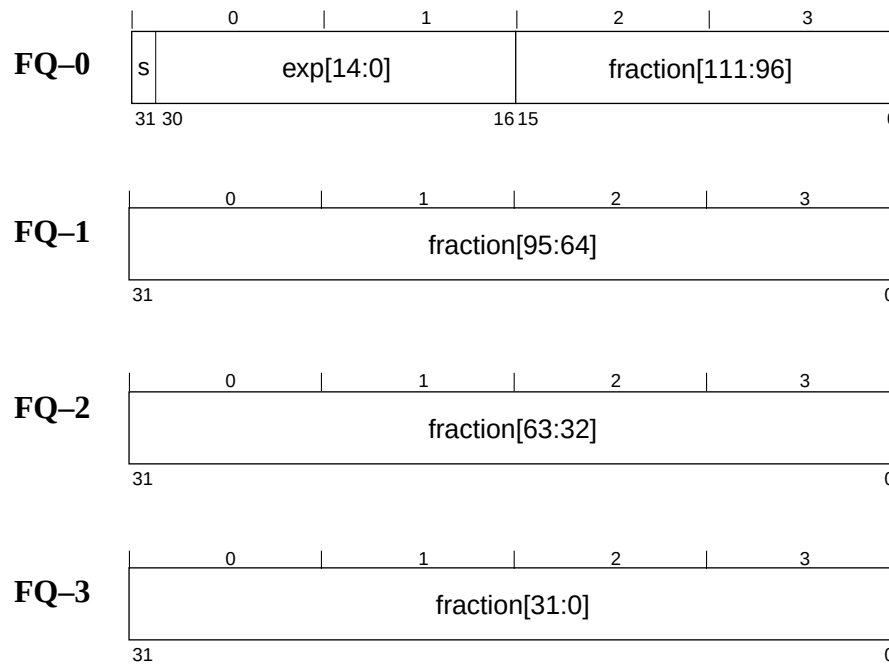


Figure 13—Floating-point quad-precision data format

Table 1—Arrangement of doublewords and quadwords in memory and registers

Sub-format name	Subformat field	Required memory address alignment	Memory address	Register number alignment	Reg nbr
SD-0	signed_dbl_integer[63:32]	0 mod 8	n	0 mod 2	r
SD-1	signed_dbl_integer[31:0]	4 mod 8	$n + 4$	1 mod 2	$r + 1$
UD-0	unsigned_dbl_integer[63:32]	0 mod 8	n	0 mod 2	r
UD-1	unsigned_dbl_integer[31:0]	4 mod 8	$n + 4$	1 mod 2	$r + 1$
FD-0	s:exp[10:0]:fraction[51:32]	0 mod 8	n	0 mod 2	f
FD-1	fraction[31:0]	4 mod 8	$n + 4$	1 mod 2	$f + 1$
FQ-0	s:exp[14:0]:fraction[111:96]	0 mod 16*	n	0 mod 4	f
FQ-1	fraction[95:64]	4 mod 16*	$n + 4$	1 mod 4	$f + 1$
FQ-2	fraction[63:32]	8 mod 16*	$n + 8$	2 mod 4	$f + 2$
FQ-3	fraction[31:0]	12 mod 16*	$n + 12$	3 mod 4	$f + 3$

* This alignment is recommended for compatibility with possible future extensions to this standard (see 3.2.1.1), although quadwords are only required to be doubleword-aligned.

Table 2—Signed/unsigned integer, and tagged format ranges

Data type	Width (bits)	Range
signed integer byte	8	-2^7 to 2^7-1
signed integer halfword	16	-2^{15} to $2^{15}-1$
signed integer word	32	-2^{31} to $2^{31}-1$
signed integer tagged word	32	-2^{29} to $2^{29}-1$
signed integer double	64	-2^{63} to $2^{63}-1$
unsigned integer byte	8	0 to 2^8-1
unsigned integer halfword	16	0 to $2^{16}-1$
unsigned integer word	32	0 to $2^{32}-1$
unsigned integer tagged word	32	0 to $2^{30}-1$
unsigned integer double	64	0 to $2^{64}-1$

Table 3—Floating-point single-precision format definition

s = sign (1 bit) e = biased exponent (8 bits) f = fraction (23 bits) u = undefined bit	
normalized value ($0 < e < 255$):	$(-1)^s \times 2^{e-127} \times 1.f$
subnormal value ($e = 0$):	$(-1)^s \times 2^{-126} \times 0.f$
zero ($e = 0$)	$(-1)^s \times 0$
signaling NaN:	$s = u$; $e = 255$ (max); $f = .0uu - uu$ (at least one bit of fraction shall be nonzero)
quiet NaN:	$s = u$; $e = 255$ (max); $f = .1uu - uu$
$-\infty$ (negative infinity):	$s = 1$; $e = 255$ (max); $f = .000 - 00$
$+\infty$ (positive infinity):	$s = 0$; $e = 255$ (max); $f = .000 - 00$

Table 4—Floating-point double-precision format definition

s = sign (1 bit) e = biased exponent (11 bits) f = fraction (52 bits) u = undefined bit	
normalized value ($0 < e < 2047$):	$(-1)^s \times 2^{e-1023} \times 1.f$
subnormal value ($e = 0$):	$(-1)^s \times 2^{-1022} \times 0.f$
zero ($e = 0$):	$(-1)^s \times 0$
signaling NaN:	$s = u$; $e = 2047$ (max); $f = .0uu - uu$ (at least one bit of fraction shall be nonzero)
quiet NaN:	$s = u$; $e = 2047$ (max); $f = .1uu - uu$
$-\infty$ (negative infinity):	$s = 1$; $e = 2047$ (max); $f = .000 - 00$
$+\infty$ (positive infinity):	$s = 0$; $e = 2047$ (max); $f = .000 - 00$

Table 5—Floating-point quad-precision format definition

s = sign (1 bit) e = biased exponent (15 bits) f = fraction (112 bits) u = undefined bit	
normalized value ($0 < e < 32767$):	$(-1)^s \times 2^{e-16383} \times 1.f$
subnormal value ($e = 0$):	$(-1)^s \times 2^{-16382} \times 0.f$
zero ($e = 0$):	$(-1)^s \times 0$
signaling NaN:	$s = u$; $e = 32767$ (max); $f = .0uu - uu$ (at least one bit of fraction shall be nonzero)
quiet NaN:	$s = u$; $e = 32767$ (max); $f = .1uu - uu$
$-\infty$ (negative infinity):	$s = 1$; $e = 32767$ (max); $f = .000 - 00$
$+\infty$ (positive infinity):	$s = 0$; $e = 32767$ (max); $f = .000 - 00$

5. Registers

An IEEE 1754 processor includes two types of registers: general-purpose (or “working”) data registers and control/status registers. The IU’s general-purpose registers are called *r* registers; the FPU’s general-purpose registers are called *f* registers. Coprocessor working registers are coprocessor-implementation-dependent (impl. dep. #26).

IU control/status registers include the following:

- Processor state register (PSR)
- Window invalid mask (WIM)
- Trap base register (TBR)
- Multiply/divide register (Y)
- Program counters (PC, nPC)
- Implementation-dependent ancillary state registers (ASRs) (impl. dep. #8)
- Implementation-dependent IU deferred-trap queue (impl. dep. #16)

FPU control/status registers include the following:

- Floating-point state register (FSR)
- Implementation-dependent floating-point deferred-trap queue (FQ) (impl. dep. #24)

Coprocessor (CP) control/status registers, if present, may include the following:

- Implementation-dependent coprocessor state register (CSR) (impl. dep. #27)
- Implementation-dependent coprocessor deferred-trap queue (CQ) (impl. dep. #28)

5.1 IU *r* registers

An implementation of the IU shall contain from 40 to 520 general-purpose 32-bit *r* registers. They are partitioned into 8 **global** registers, plus an implementation-dependent number of 16-register sets (impl. dep. #2). A register window consists of the current 8 **in** registers, 8 **local** registers, and 8 **out** registers. See table 6.

5.1.1 Windowed *r* registers

At a given time, an instruction can access the 8 **globals** and a 24-register **window** into the *r* registers. A register window comprises the 8 **in** and 8 **local** registers of a particular register set, together with the 8 **in** registers of an adjacent register set, which are addressable from the current window as **out** registers. See figure 14.

The number of windows or register sets, NWINDOWS, is implementation-dependent and ranges from 2 to 32 (impl. dep. #2). The total number of *r* registers in a given implementation is 8 (for the **globals**), plus the number of sets times 16 registers per set. Thus, the minimum number of *r* registers is 40 (2 sets plus 8 **globals**) and the maximum number is 520 (32 sets plus 8 **globals**).

Table 6—Window addressing

Windowed register address	<i>r</i> register address
<i>in</i> [0] – <i>in</i> [7]	<i>r</i> [24] – <i>r</i> [31]
<i>local</i> [0] – <i>local</i> [7]	<i>r</i> [16] – <i>r</i> [23]
<i>out</i> [0] – <i>out</i> [7]	<i>r</i> [8] – <i>r</i> [15]
<i>global</i> [0] – <i>global</i> [7]	<i>r</i> [0] – <i>r</i> [7]

The current window into the r registers is given by the current window pointer (CWP) field in the processor state register (PSR). The CWP is incremented by a RESTORE (or RETT) instruction and decremented by a SAVE instruction or a trap. Window overflow and underflow are detected via the window invalid mask (WIM) register, which is controlled by supervisor software.

The following algorithm is used to calculate the physical register number to which a windowed register n refers:

$$((n - 8) + (\text{CWP} \times 16)) \bmod (\text{NWINDOWS} \times 16)$$

5.1.2 Overlapping of windows

Each window shares its *ins* with one adjacent window and its *outs* with another. The *outs* of the CWP + 1 window are addressable as the *ins* of the current window, and the *outs* in the current window are the *ins* of the CWP – 1 window. The *locals* are unique to each window.

An r register with address o , where $8 \leq o \leq 15$, refers to exactly the same register as $(o + 16)$ does after the CWP is decremented by 1 (modulo NWINDOWS). Likewise, a register with address i , where $24 \leq i \leq 31$, refers to exactly the same register as address $(i - 16)$ does after the CWP is incremented by 1 (modulo NWINDOWS). See figure 14 and figure 15.

Since CWP arithmetic is performed modulo NWINDOWS, the highest numbered implemented window overlaps with window 0. The *outs* of window 0 are the *ins* of window NWINDOWS – 1. Implemented windows shall be contiguously numbered from 0 through NWINDOWS – 1.

NOTE—Since the procedure call instructions (CALL and JMPL) do not change the CWP, a procedure can be called without changing the window. See Annex J, “Software considerations.”

Because the windows overlap, the number of windows available to software is one less than the number of implemented windows, or NWINDOWS – 1. When the register file is full, the *outs* of the newest window are the *ins* of the oldest window, which still contains valid program data.

No assumptions can be made regarding the values contained in the *local* and *out* registers of a register window upon re-entering the window through a SAVE instruction. If, with traps enabled, a program executes a RESTORE followed by a SAVE, the resulting window’s *locals* and *outs* may not be valid after the SAVE, since a trap may have occurred between the RESTORE and the SAVE.

5.1.3 Special r registers

The utilization of four r registers is fixed, in whole or in part, by the architecture:

- The value of $r[0]$ is always zero. Note that $r[0]$ is equivalent to *global*[0].
- The CALL instruction writes its own address into register $r[15]$ (*out* register 7).
- When a trap occurs, the program counters PC and nPC are copied into registers $r[17]$ and $r[18]$ (*local* registers 1 and 2) of the trap’s new register window.

5.1.4 Doubleword operands

Instructions that access a doubleword in the r registers assume even-odd register alignment. The least significant bit of an r register address in these instructions is reserved and should be supplied as zero by software.

When the $r[0] - r[1]$ register pair is used as a destination, only $r[1]$ is modified. When the $r[0] - r[1]$ register pair is used as a source, the more significant 32 bits of the doubleword are zero and the less significant 32 bits of the doubleword are the contents of $r[1]$.

Impl. Dep. #10: An attempt to execute a doubleword load or store instruction that refers to a misaligned (odd) destination register number should cause an *illegal_instruction* trap. If a trap is not generated, the least significant bit of the destination register number shall be ignored.

NOTES

- 1—**Impl. Dep. #11:** During execution of a load doubleword integer or coprocessor (LDD, LDDA, or LDDC) instruction, if an exception is generated during the memory cycle in which the second word is being loaded, the destination register(s) may be modified before the trap is taken. A similar exception can occur during the store memory access but not the load access of a LSTUB, LDSTUBA, SWAP, or SWAPA instruction. Thus, the destination register may be modified before the trap is taken. See Clause 7, “Traps.”
- 2—**Impl. Dep. #12:** An implementation might cause a *data_access_exception* trap due to a “catastrophic” error during the “effective address + 4” memory access of a store-doubleword (STD, STDA, STDF, STDFQ, STDC, or STDCQ) instruction, even though the corresponding “effective address” access did not cause an error. Thus, memory data at the effective memory address may be modified before the trap is taken. (This cannot happen across a page boundary because of the doubleword-alignment restriction.) See Clause 7, “Traps.”

5.1.5 Register usage

See Annex J, “Software considerations,” for a description of conventional usage of the *r* registers.

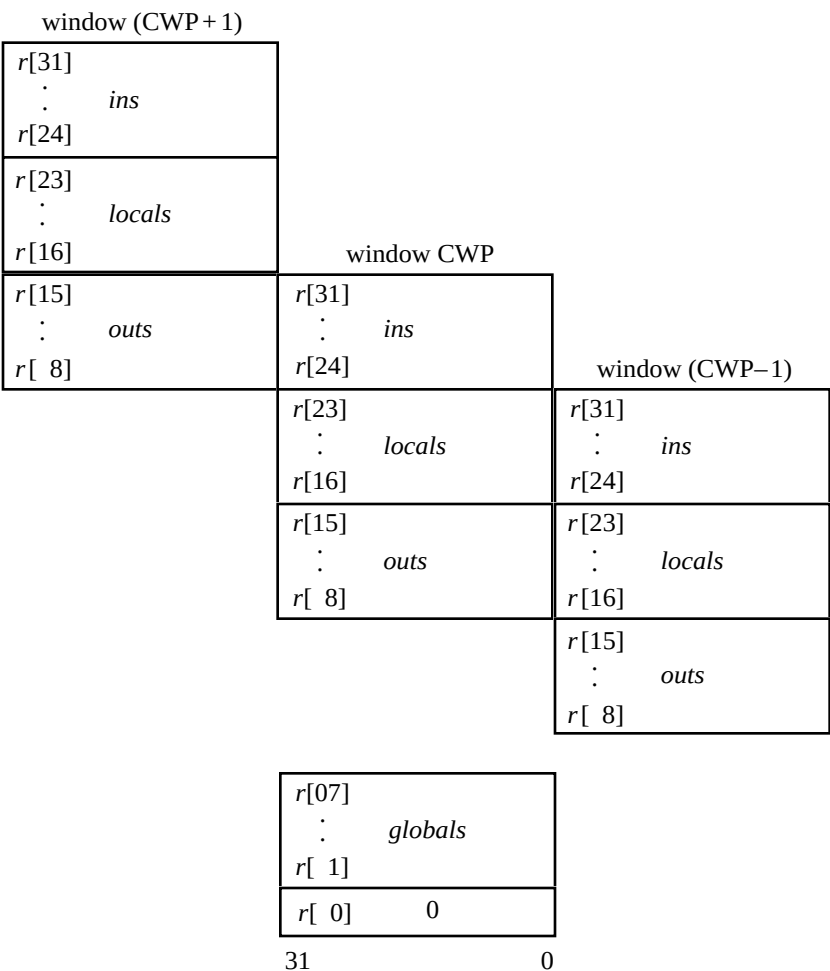


Figure 14—Three overlapping windows and eight global registers

In figure 15, $NWINDOWS = 8$. The register sets are indicated in *italics*. $CWP = 0$ and $WIM[7] = 1$. If the procedure using window $w0$ executes a **RESTORE**, window $w1$ will become the current window. If the procedure using window $w0$ executes a **SAVE**, a *window_overflow* trap will occur. The overflow trap handler uses the $w7$ *locals*.

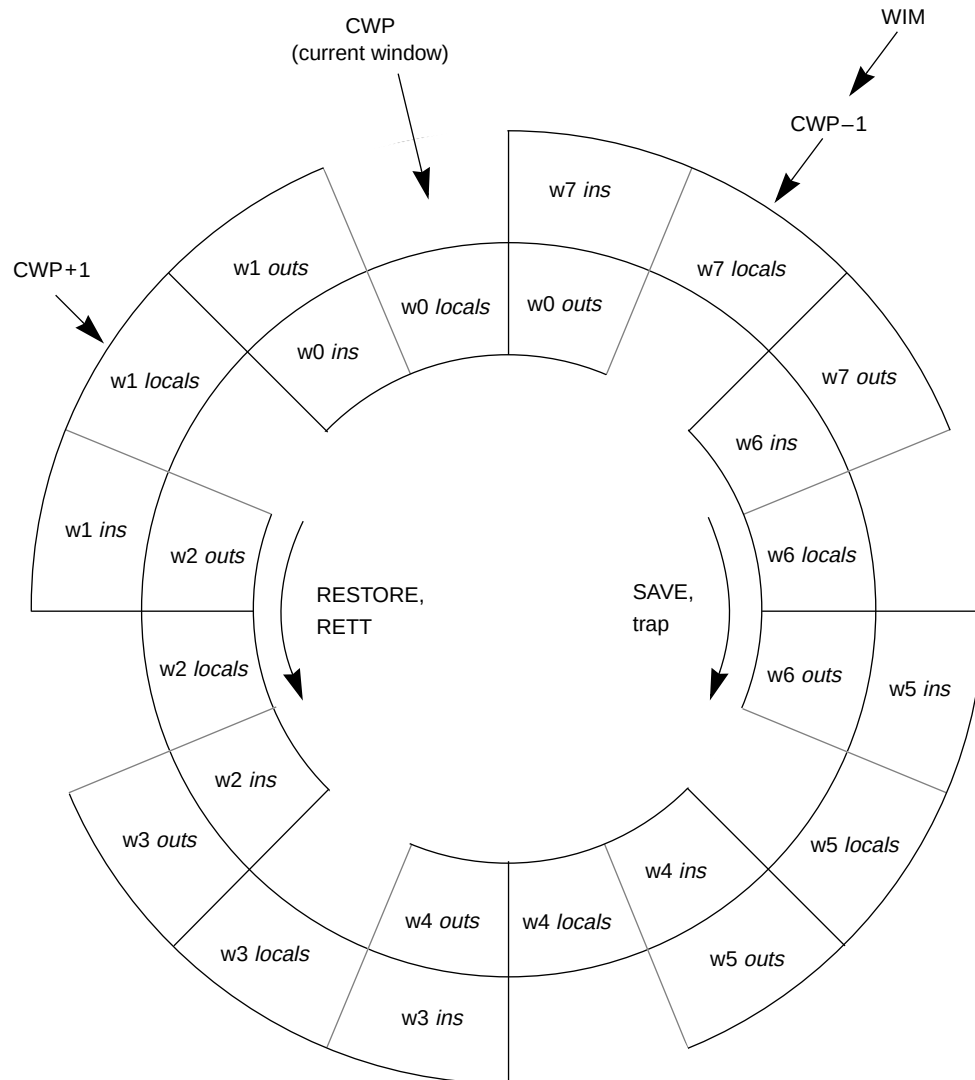


Figure 15—The windowed r registers (for $NWINDOWS=8$)

5.2 IU Control/status registers

The 32-bit IU control/status registers include the processor state register (PSR), the window invalid mask register (WIM), the trap base register (TBR), the multiply/divide (Y) register, the program counters (PC and nPC), optional, implementation-dependent ancillary state registers (ASRs) (impl. dep. #8), and the IU deferred-trap queue (impl. dep. #16).

5.2.1 Processor state register (PSR)

The 32-bit PSR contains various fields that control the processor and hold status information. The PSR can be modified by the SAVE, RESTORE, Ticc, and RETT instructions, and by all instructions that modify the condition codes. The privileged RDPSR and WRPSR instructions read and write the PSR directly. Figure 16 illustrates the fields in the PSR register.

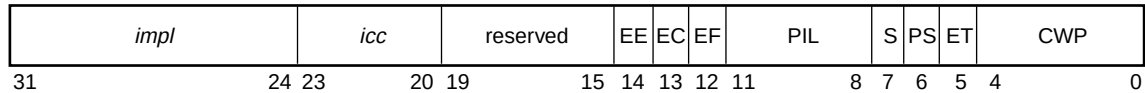


Figure 16—PSR fields

The PSR provides the following fields:

5.2.1.1 PSR_implementation (*impl*)

Impl. Dep. #13: Bits 31 through 24 are hardwired to uniquely identify an implementation or class of software-compatible implementations of the architecture. The hardware shall not change this field in response to a WRPSR instruction. This value is assigned as described in D.3, “Categories of architectural implementation dependencies.” Values $F0_{16}..FF_{16}$ are reserved and are not available for assignment.

5.2.1.2 PSR_integer_cond_codes (*icc*)

Bits 23 through 20 are the IU’s condition codes. These bits are modified by the arithmetic and logical instructions whose names end with the letters **cc** (e.g., ANDcc), and by the WRPSR instruction. The Bicc and Ticc instructions cause a transfer of control based on the value of these bits, which are illustrated in figure 17.

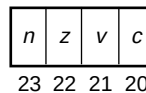


Figure 17—Integer condition codes (*icc*) fields of the PSR

5.2.1.2.1 PSR_icc_negative (*n*)

Bit 23 indicates whether the 32-bit two’s complement ALU result was negative for the last instruction that modified the *icc* field. A value of 1 indicates that the result was negative; a value of 0 indicates that it was not negative.

5.2.1.2.2 PSR_icc_zero (*z*)

Bit 22 indicates whether the 32-bit ALU result was zero for the last instruction that modified the *icc* field. A value of 1 indicates that the result was zero; a value of 0 indicates that it was nonzero.

5.2.1.2.3 PSR_icc_overflow (*v*)

Bit 21 indicates whether the ALU result was within the range of (was representable in) 32-bit two’s complement notation for the last instruction that modified the *icc* field. A value of 1 indicates that the result overflowed; a value of 0 indicates that it did not overflow.

5.2.1.2.4 PSR_icc_carry (c)

Bit 20 indicates whether a two's complement carry-out (or borrow) occurred for the last instruction that modified the *icc* field. The carry bit is set on addition if there is a carry out of bit 31 (see B.10.1). The carry bit is set on subtraction if there is a borrow out of bit 31 (see B.10.22). A value of 1 indicates that the operation generated a carry-out/borrow; a value of 0 indicates that it did not generate a carry-out/borrow.

5.2.1.3 PSR_reserved

Bits 19 through 15 are reserved. When read by a RDPSR instruction, these bits shall read as zeros. Supervisor software should only issue WRPSR instructions with zero values in this field. Software intended to run on processors compliant with future versions of IEEE 1754 should not expect the values of these bits when read to be undefined.

5.2.1.4 PSR_enable_extensions (EE)

Bit 14 determines whether non-IEEE 1754 extensions are enabled. If disabled (PSR.EE = 0), the processor operation shall be compliant with IEEE 1754. If enabled (PSR.EE = 1), the value in the IER register (see 5.2.7, "Implementation-dependent extensions register") determines the state of operational extensions in the processor. A *reset* trap sets PSR.EE to 0.

Impl. Dep. #14: Whether or not PSR.EE is implemented is implementation-dependent. If it is not implemented, PSR.EE shall read as 0 and writes to it shall be ignored.

5.2.1.5 PSR_enable_coprocessor (EC)

Bit 13 determines whether the implementation-dependent coprocessor is enabled. If disabled, a coprocessor instruction will trap. A value of 1 = enabled; a value of 0 = disabled. **Impl. Dep. #4:** If an implementation does not support a coprocessor in hardware, PSR.EC should always read as 0 and writes to it should be ignored.

NOTE—Software can use the EC bit to determine whether a particular process uses the CP. If a process does not use the CP, coprocessor registers need not be saved across a context switch.

5.2.1.6 PSR_enable_floating-point_unit (EF)

Bit 12 determines whether the FPU is enabled. It is set by software. If disabled, a floating-point instruction shall trap. A value of 1 = enabled; a value of 0 = disabled. **Impl. Dep. #15:** If an implementation does not support a hardware FPU, PSR.EF should always read as 0 and writes to it should be ignored.

NOTE—Software can use the EF bit to determine whether a particular process uses the FPU. If a process does not use the FPU, floating-point registers need not be saved across a context switch.

5.2.1.7 PSR_proc_interrupt_level (PIL)

Bits 11 (the most significant bit) through 8 (the least significant bit) identify the interrupt level above which the processor will accept an interrupt. Note that interrupt level 15 cannot be disabled by setting PSR.PIL to 15; IR 15 can be disabled only by setting PSR.ET to 0. See Clause 7, "Traps."

5.2.1.8 PSR_supervisor (S)

Bit 7 determines whether the processor is in supervisor or user mode. A value of 1 = supervisor mode; a value of 0 = user mode. A *reset* trap sets PSR.S to 1 (supervisor mode).

5.2.1.9 PSR_previous_supervisor (PS)

Bit 6 contains the value of the S bit at the time of the most recent trap.

5.2.1.10 PSR_enable_traps (ET)

Bit 5 determines whether traps are enabled. A trap automatically resets ET to 0. When ET = 0, an interrupt request is ignored and an exception trap causes the IU to enter error_state, which typically results in a reset trap that resumes execution at address zero. A value of 1 = traps enabled; a value of 0 = traps disabled. A reset trap sets PSR.ET to 0 (traps disabled). See Clause 7, “Traps.”

5.2.1.11 PSR_current_window_pointer (CWP)

Bits 4 (the most significant bit) through 0 (the least significant bit) comprise the current window pointer, a counter that identifies the current window into the r registers. The hardware decrements the CWP on traps and SAVE instructions and increments it on RESTORE and RETT instructions (modulo NWINDOWS).

5.2.2 Window invalid mask register (WIM)

The window invalid mask register (WIM) is controlled by supervisor software and is used by hardware to determine whether a window_overflow or window_underflow trap is to be generated by a SAVE, RESTORE, or RETT instruction. Figure 18 illustrates the fields in the window invalid mask register.

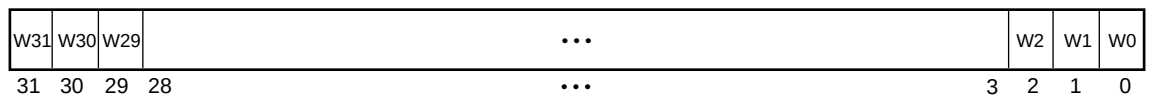


Figure 18—WIM fields

There is an active state bit in the WIM for each register set or window in an implementation. WIM[n] corresponds to the register set addressed when CWP = n.

When a SAVE, RESTORE, or RETT instruction executes, the current value of the CWP is compared against the WIM. If the SAVE, RESTORE, or RETT instruction would cause the CWP to point to an invalid register set, that is, one whose corresponding WIM bit equals 1 (WIM[CWP] = 1), a window_overflow or window_underflow trap is caused.

The WIM can be read by the privileged RDWIM instruction and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeros and values written to unimplemented bits are unused. A WRWIM with all bits set to 1, followed by a RDWIM, yields a bit vector in which the implemented windows (and only the implemented windows) are indicated by ones.

The WIM allows for implementations with up to 32 windows.

Although usual software convention sets a group of contiguous bits in WIM to 1, there is no requirement that WIM bits be set contiguously. See J.8, “Other register window usage models.”

5.2.3 Trap base register (TBR)

The trap base register (TBR) contains three fields that together comprise the address to which control is transferred when a trap occurs. Figure 19 illustrates the fields in the trap base register.

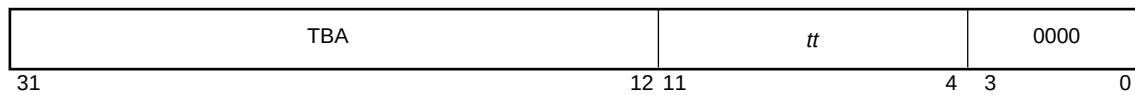


Figure 19—TBR fields

The TBR contains the following fields:

5.2.3.1 TBR_trap_base_address (TBA)

Bits 31 through 12 are the trap base address, which is established by supervisor software. It contains the most significant 20 bits of the trap table address. The TBA field is written by the WRTBR instruction.

5.2.3.2 TBR_trap_type (*tt*)

Bits 11 through 4 comprise the trap type (*tt*) field. This 8-bit field is written by the hardware when a trap (except *reset*) occurs, and retains its value until the next trap. It provides an offset into the trap table. The WRTBR instruction does not affect the *tt* field.

5.2.3.3 TBR_zero (0)

Bits 3 through 0 always read as zeros. The WRTBR instruction does not affect this field.

For compatibility with possible extensions to this architecture, supervisor software should only issue a WRTBR instruction with a zero value in this field.

See Clause 7, “Traps,” for additional information.

5.2.4 Multiply/divide register (Y)

The 32-bit Y register contains the most significant word of the double-precision product of an integer multiplication, as a result of either an integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction, or of a routine that uses the integer multiply step (MULScc) instruction. The Y register also holds the most significant word of the double-precision dividend for an integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

The Y register can be read and written directly with the RDY and WRY instructions.

5.2.5 Program counters (PC, nPC)

The 32-bit PC contains the address of the instruction currently being executed by the IU. The nPC holds the address of the next instruction to be executed (assuming a trap does not occur).

For a delayed control transfer, the instruction that immediately follows the transfer instruction is known as the delay instruction. Unless the control transfer instruction annuls it, this delay instruction is executed before control is transferred to the target. During execution of the delay instruction, the nPC points to the target of the control transfer instruction, while the PC points to the delay instruction. See Clause 6, “Instructions.”

The PC is read by a CALL, Bicc, FBfcc, CBccc, or JMPL instruction. The PC and nPC are written to two *local* registers during a trap. See Clause 7, “Traps,” for details.

5.2.6 Ancillary state registers (ASR)

IEEE 1754 provides for up to 30 ancillary state registers (ASRs), numbered from 2 to 31.

ASRs numbered 2..15 are reserved for future use by the architecture and should not be referenced by software. The result of referencing one of these ASRs is undefined.

ASRs numbered 16..31 are available for implementation-dependent uses (impl. dep. #8) such as timers, counters, diagnostic registers, self-test registers, and trap-control registers. An IU may implement from zero to sixteen of these ASRs. The semantics of accessing any of these ASRs is implementation-dependent. Whether or not access to a particular ASR is privileged is implementation-dependent (impl. dep. #9).

An ASR is read and written with the RDASR and WRASR instructions. A read or write ASR instruction is privileged if the accessed register is privileged.

5.2.7 Implementation-dependent extensions register

Impl. Dep. #14: The existence of the IER register and the meaning of specific values contained in the IER register are implementation-dependent.

If the enable extensions (EE) bit of the PSR (bit 14) is inactive (0), the value of the IER register is ignored and the processor is in standard IEEE 1754 mode.

If PSR.EE is active (1), then the IER shall be implemented. In this case, the mode of the processor is determined by the value in the IER register.

- If the IER register contains 00000000_{16} , the processor is in standard IEEE 1754 mode, memory is addressed using the big-endian convention, and all operations shall adhere to this standard. See 6.3.1.2 for information about the big-endian addressing convention.
- If the IER register contains any other value, all aspects of processor operation are implementation-dependent and no aspect of its operation is required to adhere to this standard. See Annex L, “Non-IEEE 1754 architectural extensions.” For information about also supporting the little-endian addressing convention, see L.2.1, “Support for little-endian addressing.”

A conforming implementation shall provide to user mode software a means to set the processor into a state where at least one of PSR.EE or IER is zero. Typically, the WRIER instruction will serve this purpose. See A.46, “Write state register instructions.”

5.2.8 IU deferred-trap queue

An implementation may contain zero or more deferred-trap queues. Such a queue contains sufficient state to implement resumable deferred traps caused by the IU. Note that *fp_exception* and *cp_exception* deferred traps are handled by the floating-point and coprocessor deferred-trap queues, respectively.

An IU deferred-trap queue can be read and written via privileged load/store alternate or read/write ancillary state register instructions.

Impl. Dep. #16: The contents and operation of an IU deferred-trap queue are implementation-dependent and are not visible to user application programs.

5.3 FPU f registers

The FPU contains thirty-two 32-bit floating-point f registers, which are numbered from $f[0]$ to $f[31]$. Unlike the windowed r registers, at a given time an instruction has access to any of the 32 f registers. The f registers can be read and written by FPop (FPop1/FPop2 format) instructions, and by load/store single/double floating-point instructions (LDF, LDDF, STF, STDF). See figure 20.

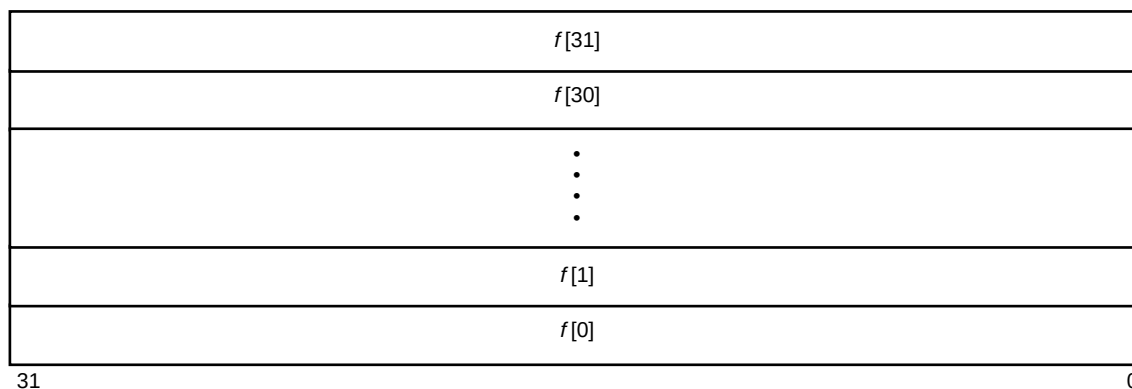


Figure 20—The f registers

5.3.1 Double and quad operands

A single f register can hold one single-precision operand. A double-precision operand requires an aligned pair of f registers, and a quad-precision operand requires an aligned quadruple of f registers. Thus, at a given time, the f registers can hold a maximum of 32 single-precision, 16 double-precision, or 8 quad-precision values.

Instructions that access a floating-point double in the f registers assume double alignment. The least-significant bit of a doubleword f register address specifier is reserved and shall be set to zero by software. Similarly, the least-significant two bits of a quadword f register address are reserved and shall be set to zero by software. See table 1.

Impl. Dep. #17: An attempt to execute an instruction that refers to a misaligned floating-point register operand (double-precision operand in a register whose number is not 0 **mod** 2, or quadruple-precision operand in a register whose number is not 0 **mod** 4) should cause an *fp_exception* trap, with FSR.ftt = 6 (*invalid_fp_register*).

5.4 FPU control/status registers

The 32-bit FPU control/status registers include a floating-point state register (FSR) that contains mode and status information about the FPU, and an optional, implementation-dependent, floating-point deferred-trap queue (FQ) (impl. dep. #24).

5.4.1 Floating-point state register (FSR)

The FSR register fields contain FPU mode and status information. The FSR is read and written by the STFSR and LDFSR instructions. Figure 21 illustrates the FSR.

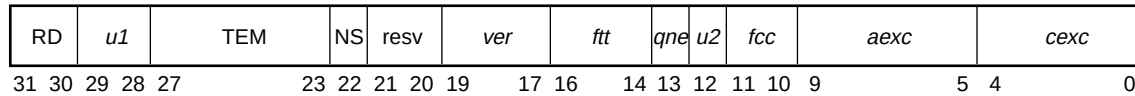


Figure 21—FSR fields

The FSR provides the following fields:

5.4.1.1 FSR_rounding_direction (RD)

Bits 31 and 30 select the rounding direction for floating-point results according to IEEE Std 754-1985.

Table 7—Rounding direction (RD) field of FSR

RD	Round toward
0	nearest (even, if tied)
1	0
2	$+\infty$
3	$-\infty$

5.4.1.2 FSR_unused (*u1*, *u2*)

Bits 29, 28, and 12 are unused. Software should only issue a LDFSR instruction with zero values in these bits, unless the values of these bits are exactly those derived from a previous STFSR. The values of these bits as read by an STFSR are undefined.

5.4.1.3 FSR_trap_enable_mask (TEM)

Bits 27 through 23 are enable bits for each of the five IEEE 754 floating-point exceptions that can be indicated in the current exception field (*cexc*). See figure 22. If a floating-point operate instruction generates one or more exceptions and the TEM bit corresponding to any of the exceptions is 1, an *fp_exception* trap is caused. A TEM value of 0 prevents that exception type from generating a trap.

5.4.1.4 FSR_nonstandard_fp (NS)

Impl. Dep. #18: Bit 22, when set to 1, causes the FPU to produce implementation-defined results that may not correspond to IEEE Std 754-1985. For instance, to obtain higher performance, implementations may convert a subnormal floating-point operand or result to zero when FSR.NS is set. IEEE 1754 implementations are permitted but not encouraged to deviate from IEEE 754 requirements when the nonstandard mode bit of the FSR is 1. In an implementation where the nonstandard floating-point mode operates identically to standard IEEE 1754 mode, the NS bit of the FSR should always read as 0, and writes to it should be ignored. FSR.NS has no effect on non-FPU operations. FSR.NS can be set or cleared by software.

5.4.1.5 FSR_reserved (*resv*)

Bits 21 and 20 are reserved. When read by an STFSR instruction, these bits shall read as zero. Software should only issue LDFSR instructions with zero values in these bits, unless the values of these bits are exactly those derived from a previous STFSR.

5.4.1.6 FSR_version (*ver*)

Impl. Dep. #19: Bits 19 through 17 identify a particular implementation of the FPU architecture. For each IEEE 1754 IU implementation (as identified by its PSR.*impl* field), there may be one or more FPU implementations, or none. This field identifies the particular FPU implementation present. The values of this field are assigned by the Registration Authority for IEEE 1754, and should only be referenced by system software. Version number 7 is reserved to indicate that no hardware floating-point controller is present.

5.4.1.7 FSR_floating-point_trap_type (*ftt*)

One of several conditions can cause a floating-point exception trap. When an *fp_exception* trap occurs, bits 16 through 14 of the FSR identify the cause of the exception, the floating-point trap type. After a floating-point exception occurs, the *ftt* field encodes the type of floating-point exception until an STFSR or another FPop is executed.

The *ftt* field can be read by the STFSR instruction. An LDFSR instruction does not affect *ftt*.

Supervisor-mode software that handles floating-point traps can execute an STFSR to determine the floating-point trap type. **Impl. Dep. #20:** Whether STFSR explicitly sets *ftt* to zeros is implementation-dependent; if STFSR does not set *ftt* to zeros, then the trap software shall ensure that a subsequent STFSR from user mode shows a value of zero for *ftt*.

NOTE—LDFSR cannot be used for this purpose since it leaves *ftt* unchanged, although executing a non-trapping FPop such as “fmovs %f0,%f0” prior to returning to user mode will zero *ftt*. *ftt* remains valid until the next FPop instruction completes execution.

The *ftt* field encodes the floating-point trap type according to table 8. Note that the value “7” is reserved for future expansion.

Table 8—Floating-point trap type (*ftt*) field of FSR

<i>ftt</i>	Trap type
0	(none)
1	<i>IEEE_754_exception</i>
2	<i>unfinished_FPop</i>
3	<i>unimplemented_FPop</i>
4	<i>sequence_error</i>
5	<i>hardware_error</i>
6	<i>invalid_fp_register</i>
7	(reserved)

The *sequence_error* and *hardware_error* trap types are unlikely to arise in the normal course of computation. They are essentially unrecoverable from the point of view of user application software.

In contrast, *IEEE_754_exception*, *unfinished_FPop*, and *unimplemented_FPop* will likely arise occasionally in the normal course of computation and shall be recoverable by supervisor software. Execution of a floating-point instruction causing an *fp_exception* trap of type *unfinished_FPop* shall be completed by software.