picoJava-II™ Programmer's Reference Manual

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Part No.: 805-2800-06
March 1999

Contents

Preface xvii

Part I. Understanding the picoJava-II Architecture

1. Overview 3

1.1 Purpose 3

1.2 Relationship to the Java Virtual Machine 4

1.3 Key Elements of the Core 4

2. Registers 5

2.1 Program Counter Register (PC) 5

2.2 Stack Management Registers 6

2.2.1 Local Variable Pointer Register (VARS) 6

2.2.2 FRAME Pointer Register (FRAME) 7

2.2.3 Top-of-Stack Pointer Register (OPTOP) 7

2.2.4 Minimum Value of Top-of-Stack Register (OPLIM) 8

2.2.5 Address of Deepest Stack Cache Entry Register (SC_BOTTOM) 8

2.3 Constant Pool Base Pointer Register (CONST_POOL) 9

2.4 Memory Protection Registers (USERRANGE1 and USERRANGE2) 10

2.5 Processor Status Register (PSR) 10

2.6 Trap Handler Address Register (TRAPBASE) 12

2.7 Monitor-Caching Registers 12

2.7.1 Lock Count Registers (LOCKCOUNT[0..1]) 13
2.7.2 Lock Address Registers (LOCKADDR[0..1]) 13
2.8 Garbage Collection Register (GC_CONFIG) 14
2.9 Breakpoint Registers 15
2.10 Implementation Registers 17
2.10.1 Version ID Register (VERSIONID) 17
2.10.2 Hardware Configuration Register (HCR) 18
2.11 Global Registers (GLOBAL[0..3]) 20

3. Memory System and Caches 21

3.1 Architecture of the Memory System 21
3.1.1 Address Space 22
3.1.2 Alignment 22
3.1.3 Cacheable and Noncacheable Memory Regions 23
3.1.4 Endianness 23
3.1.5 Erroneous Memory Transactions 26
3.2 Memory Protection 27
3.2.1 The Address-Checking Process 27
3.2.2 Memory Regions 28
3.2.3 Limits for Stack Growth 29
3.3 Cache Coherency 30
3.3.1 Coherency for Stack and Data Accesses 30
3.3.2 Coherency for Instruction Accesses 32
3.4 Instruction Cache 33
3.4.1 Configuration 33
3.4.2 Initialization 34
3.4.3 Operations 34
3.4.4 Modification of Instruction Space 35
3.5 Data Cache 36
3.5.1 Configuration 36

3.5.2 Initialization 37
3.5.3 Operations 37
3.6 Stack Cache 39
3.6.1 Configuration 39
3.6.2 Initialization 40
3.6.3 Dribbling 41
3.6.4 Flushing 44

4. Traps and Interrupts 45

4.1 Traps 45
4.1.1 Trap Table 46
4.1.2 The Process of Taking a Trap 46
4.1.3 Trap Types and Priorities 50
4.2 Instruction Emulation 54
4.3 Exceptions 54
4.4 Interrupts 56
4.4.1 Interrupt Control 57
4.4.2 Interrupt Latency 57
4.5 Context Switch 58

5. Data Types and Runtime Data Structures 61

5.1 Primitive Data Types 61
5.1.1 Integral Data Types 62
5.1.2 Floating-Point Data Types 62
5.2 Reference Types and Values 62
5.2.1 References and Headers 63
5.2.2 Object Storage 64
5.2.3 Array Storage 65
5.2.4 Layout of Array Data Structures 67
5.3 Essential Runtime Data Structures 71

5.3.1 Method Vector and Runtime Class Info Structure 71
5.3.2 Method Structure 72
5.3.3 Class Structure 73
5.3.4 Constant Pool 73

6. Instruction Set 75

Part II. Programming the picoJava-II Core

7. Java Method Invocation and Return 385

7.1 Allocating a New Frame 385
7.1.1 Incoming Arguments 386
7.1.2 Local Variables 387
7.1.3 Invoker's Method Context 387
7.1.4 Operand Stack 387
7.2 Invoking a Method 388
7.2.1 Resolving a Method Reference 388
7.2.2 Accessing a Method Structure 388
7.2.3 Allocating a New Method Frame 389
7.2.4 Saving the Invoker's Method Context 391
7.2.5 Passing Control to the Invoked Method 391
7.3 Invoking a Synchronized Method 391
7.4 Returning from a Method 393

8. Monitors 395

8.1 Structures 396
8.2 Hardware Synchronization 396
8.3 Software Support 397
8.3.1 `LockCountOverflow` Handler 397
8.3.2 `LockEnterMiss` Handler 397
8.3.3 `LockRelease` Handler 398
8.3.4 `LockExitMiss` Handler 399

8.3.5 Context Switch Support 400

9. Support of the C Programming Language 401

9.1 Register Conventions 402

9.2 Runtime Stack Architecture 402

9.2.1 Calling Convention for C-to-C Calls 404

9.2.2 Rules for Passing Arguments 404

9.2.3 Function Return Values 405

9.2.4 Function Prologue and Epilogue 405

9.2.5 Functions with Simple Parameters and Locals 406

9.2.6 Functions with Complex Parameters and Locals 408

9.2.7 Functions That Return Aggregate Values 412

9.2.8 Functions with Variable Number of Arguments 415

9.3 Calling Conventions for Java-to-C Calls 420

9.4 Optimizations 421

9.5 Function Tables 422

9.5.1 Structure 422

9.5.2 Properties 423

9.5.3 Provisions in the Operating System 423

9.5.4 `_init` and `_fini` 424

9.5.5 `OSGetNArgs` Algorithm 424

9.5.6 Extensions to Support `.so` (`.dll`) Files 425

9.6 Handling of Argument Mismatches 425

9.7 Object File Formats 426

10. Stack Chunking 429

10.1 Overview 429

10.2 `oplim_trap` Handler 430

10.3 Manual Updates of the VARS Register 431

10.4 Returns to Previously Saved Program States 432

10.5 Possible Write-After-Write Hazards 432

11. Support for Garbage Collection 433

11.1 Hardware Support 433

11.1.1 Support for Handles 434

11.1.2 Reserved Bits in References and Headers 434

11.2 Write Barriers 434

11.2.1 Instructions Subject to Write-Barrier Checks 436

11.2.2 Page-Based Write Barrier 436

11.2.3 Reference-Based Write Barrier 439

11.3 Examples 439

11.3.1 Train Algorithm-Based Collectors 440

11.3.2 Remembered Set-Based Generational Collector 440

11.4 References 441

12. System Management and Debugging 443

12.1 Power Management 443

12.2 Reset Management 444

12.2.1 Machine State After Reset 444

12.2.2 Enabling the Stack Cache 446

12.2.3 Enabling the Instruction and Data Caches 447

12.3 Breakpoints 450

12.3.1 Data Breakpoints 451

12.3.2 Instruction Breakpoints 451

12.3.3 Breakpoint Registers 451

12.3.4 Breakpoint Address Matching 454

12.3.5 Breakpoints and Halt Mode 454

12.4 Other Debug and Trace Features 454

Part III. Appendixes

A. Opcodes 457

Index 477

Figures

FIGURE 2-1 Program Counter Register (PC) 6

FIGURE 2-2 Local Variable Pointer Register (VARS) 6

FIGURE 2-3 FRAME Pointer Register (FRAME) 7

FIGURE 2-4 Top-of-Stack Pointer Register (OPTOP) 7

FIGURE 2-5 Minimum Value of Top of Stack Register (OPLIM) 8

FIGURE 2-6 Address of Deepest Stack Cache Entry Register (SC_BOTTOM) 9

FIGURE 2-7 Constant Pool Base Pointer Register (CONST_POOL) 9

FIGURE 2-8 USERRANGE Registers 10

FIGURE 2-9 Processor Status Register (PSR) 12

FIGURE 2-10 Trap Handler Address Register (TRAPBASE) 12

FIGURE 2-11 Lock Count Registers (LOCKCOUNT0 and LOCKCOUNT1) 13

FIGURE 2-12 Lock Address Registers (LOCKADDR0 and LOCKADDR1) 14

FIGURE 2-13 Garbage Collection Register (GC_CONFIG) 15

FIGURE 2-14 Breakpoint Register (BRK1A) 15

FIGURE 2-15 Breakpoint Register (BRK2A) 15

FIGURE 2-16 Breakpoint Control Register (BRK12C) 17

FIGURE 2-17 Version ID Register (VERSIONID) 18

FIGURE 2-18 Hardware Configuration Register (HCR) 19

FIGURE 2-19 Global (GLOBAL[0..3]) Registers 20

FIGURE 3-1 The Cacheable and Noncacheable Memory Regions 23
FIGURE 3-2 How a Range of Accessible Memory Is Formed from USERRANGE Register Values 29
FIGURE 3-3 Caching Operations During Instruction Execution 30
FIGURE 3-4 How the Stack Cache Caches Part of the Stack 40
FIGURE 4-1 Data Structure of the Trap Table 46
FIGURE 4-2 Invocation of a Trap 48
FIGURE 4-3 Return from a Trap 49
FIGURE 4-4 Interrupt Control Mechanism 57
FIGURE 5-1 Object or Array Reference Format 63
FIGURE 5-2 Object or Array Header Field with Reserved Bits 64
FIGURE 5-3 Object Format with Handle Bit Clear 64
FIGURE 5-4 Object Format with Handle Bit Set 65
FIGURE 5-5 Array Format with Handle Bit Clear 66
FIGURE 5-6 Array Format with Handle Bit Set 66
FIGURE 5-7 Array of Longs Structure 67
FIGURE 5-8 Array of Doubles Structure 67
FIGURE 5-9 Array of Objects Structure 68
FIGURE 5-10 Array of Arrays Structure 68
FIGURE 5-11 Array of Integers Structure 69
FIGURE 5-12 Array of Floats Structure 69
FIGURE 5-13 Array of Chars Structure 69
FIGURE 5-14 Array of Shorts Structure 70
FIGURE 5-15 Array of Bytes Structure 70
FIGURE 5-16 Array of Booleans Structure 70
FIGURE 5-17 Runtime Class Info Structure with Method Vector 71
FIGURE 5-18 Method Structure 72
FIGURE 5-19 Class Structure 73
FIGURE 5-20 Constant Pool 74

FIGURE 6-1 Format for Data Cache Data Address for 16-Kbyte Data Cache 325
FIGURE 6-2 Format for Data Cache Tag Address for 16-Kbyte Data Cache 327
FIGURE 6-3 Format for Data Cache Tag Data for 16-Kbyte Data Cache 328
FIGURE 6-4 Format for Instruction Cache Data Address for 16-Kbyte Instruction Cache 329
FIGURE 6-5 Format for 16-Kbyte Instruction Cache Tag Address 331
FIGURE 6-6 Format for Instruction Cache Tag Data for 16-Kbyte Instruction Cache 332
FIGURE 6-7 Format for Data Cache Data Address for 16-Kbyte Data Cache 338
FIGURE 6-8 Format for Data Cache Tag Address for 16-Kbyte Data Cache 340
FIGURE 6-9 Format for Data Cache Tag Data for 16-Kbyte Data Cache 341
FIGURE 6-10 Format for Instruction Cache Data Address for 16-Kbyte Instruction Cache 342
FIGURE 6-11 Format for 16-Kbyte Instruction Cache Tag Address 344
FIGURE 6-12 Format for Instruction Cache Tag Data for 16-Kbyte Instruction Cache 345
FIGURE 7-1 A Method Frame 386
FIGURE 7-2 Allocation of a New Frame 390
FIGURE 7-3 Return from a Method 394
FIGURE 9-1 Runtime Stack Allocation for a New Thread 403
FIGURE 9-2 Operand Stack Frame Layout for `zoo` and `zoo1` 406
FIGURE 9-3 Operand Stack Frame Layout Before `zoo1` Returns to `zoo` 407
FIGURE 9-4 Stack Frame Layout for `zoo` Calling `zoo1` 409
FIGURE 9-5 Stack Frame Layout for `zoo1` Before Returning 410
FIGURE 9-6 Stack Frame for Function `zoo1` Returning Aggregate Values 413
FIGURE 9-7 Stack Frame for Function `zoo` Calling `zoo1` with Variable Number of Arguments 417
FIGURE 9-8 Stack Frame of `zoo1` After Execution of `PROLOGUE` Code 418
FIGURE 9-9 Table Structures 422
FIGURE 10-1 Possible Stack States Before Entering `oplim_trap` Handler 430
FIGURE 11-1 Storing a Reference into a Field of an Object 435
FIGURE 11-2 Operation of Page-Based Write Barrier 438
FIGURE 11-3 Operation of Reference-Based Write Barrier 439

FIGURE 11-4 Generational Garbage Collection Using Remembered Sets 441

FIGURE 12-1 Breakpoint Register (BRK1A) 452

FIGURE 12-2 Breakpoint Register (BRK2A) 452

FIGURE 12-3 Breakpoint Control Register (BRK12C) 453

Tables

TABLE P-1 Typographic Conventions xix

TABLE 3-1 Instructions That Access Big-Endian Data by Default 24

TABLE 3-2 Instructions That Access Little-Endian Data by Default 24

TABLE 3-3 Byte Ordering and Sign Extension for Single-Byte Memory Operations 25

TABLE 3-4 Byte Ordering and Sign Extension for Multiple-Byte Memory Operations 26

TABLE 3-5 Instructions Subject to Memory Protection Checks When PSR.CAC = 0 28

TABLE 3-6 Instructions for Caching Operations 31

TABLE 3-7 Encoded Values of Watermarks 41

TABLE 3-8 0PT0P-Modifying Instructions That Can Cause a Stack Underflow 43

TABLE 4-1 Types and Priorities of Traps 50

TABLE 4-2 Instructions Subject to Memory Alignment Trap Checks 55

TABLE 4-3 Calculation of Worst-Case Interrupt Latency 58

TABLE 5-1 Primitive Data Types 61

TABLE 5-2 Method Structure Fields 72

TABLE 6-1 Basic Operations 75

TABLE 6-2 Type Casts and Conversions 76

TABLE 6-3 Cache Tag Accesses 76

TABLE 6-4 Memory Access-Related Operations 77

TABLE 6-5 Stack and Memory Access Operations 78

TABLE 7-1 Code Prepended to Synchronized Nonstatic Methods 392

TABLE 7-2 Code Prepended to Synchronized Static Methods 392

TABLE 9-1 Register Uses by C Calling Convention 402

TABLE 11-1 Instructions That Store References 436

TABLE 11-2 GC_CONFIG.REGION_MASK Values 437

TABLE 11-3 GC_CONFIG.CAR_MASK Values 437

TABLE 12-1 Machine State Changes on POR/SIR and Powerdown 444

TABLE A-1 picoJava-II 1-Byte Opcodes 458

TABLE A-2 picoJava-II 2-byte Opcodes 469

Code Samples

CODE EXAMPLE 4-1 Sample Context Switch Code 59

CODE EXAMPLE 9-1 Sample Code for a Function Call 404

CODE EXAMPLE 9-2 Function with Simple Parameters and Locals 406

CODE EXAMPLE 9-3 Compiled Code for a Function with Simple Parameters and Locals 407

CODE EXAMPLE 9-4 Functions with Aggregate Parameters and Locals 408

CODE EXAMPLE 9-5 Compiled Code for Function with Aggregate Parameters 410

CODE EXAMPLE 9-6 Function Returning Aggregate Values 412

CODE EXAMPLE 9-7 Compiled Code for a Function That Returns An Aggregate Value 413

CODE EXAMPLE 9-8 Optimized Code for Function Returning Aggregate Values 415

CODE EXAMPLE 9-9 Function with Variable Number of Arguments 415

CODE EXAMPLE 9-10 Sample Code In `stdarg.h` 416

CODE EXAMPLE 9-11 Compiled Code for Function with Variable Number of Arguments 418

CODE EXAMPLE 9-12 Rearranging the Function Frame for an Argument Mismatch 425

CODE EXAMPLE 11-1 Pseudocode for a Page-Based Write Barrier 438

CODE EXAMPLE 11-2 Pseudocode for a Reference-Based Write Barrier 439

CODE EXAMPLE 12-1 Enabling the Stack Cache 446

CODE EXAMPLE 12-2 Enabling the Instruction Cache 447

CODE EXAMPLE 12-3 Enabling the Data Cache 448

Preface

Java™ is an object-oriented programming language developed by Sun Microsystems, Inc., in the early 1990s. Modeled after C and C++, it is designed to be simple and platform-independent at both the source and binary levels. Java was initially developed to address the problems of building software for networked consumer devices.

The Java virtual machine is the cornerstone of Sun's Java programming language. It is the component of the Java technology responsible for Java's cross-platform delivery as well as for the small size of its compiled code. The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and uses various memory areas. The Java virtual machine understands only a particular file format—the class file format. A class file contains Java virtual machine instructions (or bytecodes), a symbol table, and other information. The Java virtual machine knows nothing of the Java programming language and does not require a specific underlying implementation.

Organization of This Book

The *picoJava-II Programmer's Reference Manual* is divided into two main parts: architecture and programming. A third part, Appendixes, provides supplemental information.

- *Part I: Understanding the picoJava-II Architecture* **contains these chapters.**
  - Chapter 1, *Overview*, offers background information that provides a context for the remaining chapters in Part I.
  - Chapter 2, *Registers*, begins our architectural exploration with a discussion of programmer-visible registers that are readable and writable.

- Chapter 3, *Memory System and Caches*, describes the structures for managing memory.
- Chapter 4, *Traps and Interrupts*, discusses the mechanism for transferring control to the supervisor state.
- Chapter 5, *Data Types and Runtime Data Structures*, provides details about the object data type, the array data type, primitive data types, and the floating-point data type.
- Chapter 6, *Instruction Set*, concludes our examination of the Java virtual machine architecture with complete descriptions of extensions and instructions.

- *Part II: Programming the picoJava-II Core* **contains these chapters.**
  - Chapter 7, *Java Method Invocation and Return*, discusses how to invoke methods—both synchronized and nonsynchronized—and return from a method. It also describes monitors and programming support for managing them.
  - Chapter 8, *Monitors*, describes the instructions that speed up common situations in which monitors are used.
  - Chapter 9, *Support of the C Programming Language*, describes the generation of C code for the picoJava-II core.
  - Chapter 10, *Stack Chunking*, tells how stack chunking works for the picoJava-II core.
  - Chapter 11, *Support for Garbage Collection*, defines picoJava-II support for various garbage collection (GC) schemes.
  - Chapter 12, *System Management and Debugging*, addresses external issues such as power management and reset, as well as debugging and tracing

- *Part III: Appendixes* **contains one appendix.**
  - Appendix A: *Opcodes*, lists all of the picoJava-II opcodes.

At the end of this book are an index and a quick reference guide.

Related Books and References

Three books form the documentation set for the picoJava-II release:

- *picoJava-II Programmers's Reference Manual* (this book)
- *picoJava-II Microarchitecture Guide*
- *picoJava-II Verification Guide*

The following publications are reference material for the subject matter:

- Lindholm, Tim, and Frank Yellin: *The Java Virtual Machine Specification*, Addison-Wesley, ISBN 0-201-63452-X
- *IEEE Standard Test Access Port and Boundary-Scan Architecture,* ANSI/IEEE Std. 1149.1-1990.
- *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Std. 754-1985.
- Ungar, David: *ACM SIGPLAN Notices*, 19(5):157-167: *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, April 1984.
- Wilson P., and T. Moher: *ACM SIGPLAN Notices*, 24(10):23-35: *A Card-marking Scheme For Controlling Intergenerational References In Generation-based Garbage Collection On Stock Hardware*, 1989.
- Steele, Guy L.: *Communications of the ACM*, 18(9): *Multiprocessing Compactifying Garbage Collection*, September 1975.
- Hudson, R., and J. E. B. Moss: *Proceedings of International Workshop on Memory Management*: *Incremental Garbage Collection For Mature Objects*, St. Malo, France, September 16–18, 1992.

Typographic Conventions

TABLE P-1 describes the typographic conventions used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, instructions, files, and directories; on-screen computer output; email addresses; URLs	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename.*
AaBbCc123	Book titles, section titles in cross-references, new words or terms, or emphasized words	Read Chapter 6 in *User's Guide.* These are called *class* options. You *must* be root to do this.
<>	A bit number or colon-separated range of bit numbers within a field; bit 0 is the least significant bit.	`WB_VECTOR`<15:0>

Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals by using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Sun Documentation Online

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`.

Disclaimer

The information in this manual is subject to change and will be revised from time to time. For up-to-date information, contact your Sun representative.

Feedback

Email your comments on this book to: `picojava-prm@eng.sun.com`.

Acknowledgment

Many people—picoJava-II licensees, engineers, programmers, marketers—contributed to this book. We thank them for their input, feedback, and support.

PART I Understanding the picoJava-II Architecture

CHAPTER 1

Overview

The picoJava-II core provides an optimized hardware environment for hosting a Java virtual machine. The core implements most of the Java virtual machine instructions directly in hardware while supporting a general-purpose instruction set capable of handling operating systems and code compiled from languages other than the Java language (such as C and C++).

This reference manual focuses on picoJava-II architecture and programming. To implement a complete Java runtime environment, a system based on the picoJava-II architecture needs supporting software, such as a class loader, a bytecode verifier, a thread manager, and a garbage collector, the details for which are beyond the scope of this manual. However, this manual does address some relevant implementation issues that are specific to the core.

This chapter provides an overview of the core in the following sections:

- *Purpose* on page 3
- *Relationship to the Java Virtual Machine* on page 4
- *Key Elements of the Core* on page 4

1.1 Purpose

The core is designed to enable high-performance Java implementations in a variety of embedded and system-on-a-chip applications. Optimized for situations in which the memory requirements for dynamic compilation are prohibitive, yet the performance of an interpreter is too low, the core can bring the benefits of Java technology to applications that cannot be addressed with other solutions.

1.2 Relationship to the Java Virtual Machine

The picoJava-II core is *not* the entire Java virtual machine in silicon. Rather, it is a microprocessor design optimized to run a small Java virtual machine implementation. As one step to achieve this goal, the core decodes all of the bytecodes defined by the Java virtual machine. Furthermore, it implements most of them directly in hardware.

The core can accelerate other aspects (for example, garbage collection) of a Java virtual machine with hardware. Bear in mind, however, that you must develop a significant amount of software to produce a compliant Java virtual machine that runs on the core.

1.3 Key Elements of the Core

The picoJava-II instruction set is stack-based: The core performs most operations on data from the stack by pushing data from memory and local variables onto the stack, from which instructions implicitly get their operands.

In addition, the core implements a number of registers, most of which control specific functions within the core, while others contain the addresses of various areas on the stack. Four general-purpose registers are also present.

The instruction set contains over 300 instructions. Most of them are similar to those in other microprocessors; some are unique to this core.

The remaining chapters of this manual discuss each of the above elements, as well as how they are commonly used.

CHAPTER 2

Registers

The core maintains several programmer-visible registers. These registers are 32 bits wide and perform various functions, as described in the following sections:

- *Program Counter Register (PC)* on page 5
- *Stack Management Registers* on page 6
- *Constant Pool Base Pointer Register (CONST_POOL)* on page 9
- *Memory Protection Registers (USERRANGE1 and USERRANGE2)* on page 10
- *Processor Status Register (PSR)* on page 10
- *Trap Handler Address Register (TRAPBASE)* on page 12
- *Monitor-Caching Registers* on page 12
- *Garbage Collection Register (GC_CONFIG)* on page 14
- *Breakpoint Registers* on page 15
- *Implementation Registers* on page 17
- *Global Registers (GLOBAL[0..3])* on page 20

2.1 Program Counter Register (PC)

The Program Counter Register, PC, contains the address of the first byte of the instruction that is executing.

The configuration of the PC register is listed below and illustrated in FIGURE 2-1.

Field	Type	Description
31:0	RW	Byte-aligned pointer to the instruction stream

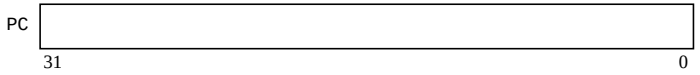


FIGURE 2-1 Program Counter Register (PC)

2.2 Stack Management Registers

This section describes the five registers that contain addresses that refer to locations in the stack.

2.2.1 Local Variable Pointer Register (VARS)

The Local Variable Pointer Register, VARS, contains the base address of the current local variables region in the stack. Typically, the local variables of a method or function are its parameters and any declared local variables. This register contains the address of the local variable zero of the method or function that is executing. Additional local variables are located at word-aligned offsets toward lower addresses. For example, local variable 1 is located at the current value of the VARS register minus 4 (the size of one word, in bytes).

The configuration of the VARS register is listed below and illustrated in FIGURE 2-2.

Field	Type	Description
31:2	RW	Word-aligned VARS pointer
1:0	RO	Always reads as 0x0.

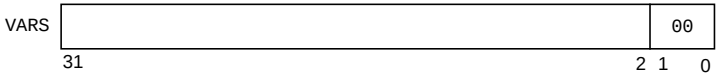


FIGURE 2-2 Local Variable Pointer Register (VARS)

2.2.2 FRAME Pointer Register (FRAME)

The FRAME Pointer Register, FRAME, contains the base address of the current call frame information on the stack for a Java method, that is, the address of the return PC for the method that is executing. Additional call frame information is located at word-aligned offsets toward lower addresses. Code compiled from other languages may not use the FRAME Pointer Register in this manner, however.

The configuration of the FRAME register is listed below and illustrated in FIGURE 2-3.

Field	Type	Description
31:2	RW	Word-aligned FRAME pointer
1:0	RO	Always reads as 0x0.

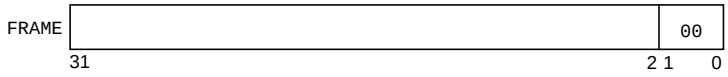


FIGURE 2-3 FRAME Pointer Register (FRAME)

2.2.3 Top-of-Stack Pointer Register (OPTOP)

The Top-of-Stack Pointer Register, OPTOP, contains the address of the current top-of-stack. The next entry to be pushed on to the stack is located at the address in the OPTOP, which is then decremented by 4 bytes. Consequently, the address of the "top" stack element that contains valid data is OPTOP + 4.

The configuration of the OPTOP register is listed below and illustrated in FIGURE 2-4.

Field	Type	Description
31:2	RW	Word-aligned OPTOP pointer
1:0	RO	Always reads as 0x0.

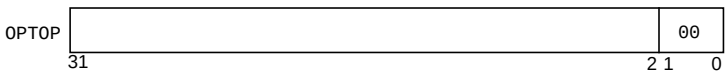


FIGURE 2-4 Top-of-Stack Pointer Register (OPTOP)

2.2.4 Minimum Value of Top-of-Stack Register (OPLIM)

The Minimum Value of Top-of-Stack Register, OPLIM, contains the minimum value that the OPTOP register can hold. This register limits stack growth to a certain memory region. For details on how the core uses OPLIM, see Chapter 10, *Stack Chunking*.

The configuration of the OPLIM register is listed below and illustrated in FIGURE 2-5.

Field	Type	Description
31	RW	OPLIM enable bit. Power On Reset (POR) clears this bit. When the core takes the oplim_trap, it clears the bit. If this bit is set, the core checks for OPLIM violations. Generally, software sets this bit; hardware resets it.
30:2	RW	Word-aligned OPLIM pointer. If OPLIM.ENABLE is set, the core checks bits 30:2 of OPTOP versus bits 30:2 of OPLIM. If OPTOP is less than OPLIM, the core generates an oplim_trap.
1:0	RO	Always reads as 0x0.

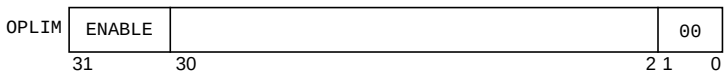


FIGURE 2-5 Minimum Value of Top of Stack Register (OPLIM)

2.2.5 Address of Deepest Stack Cache Entry Register (SC_BOTTOM)

The Address of Deepest Stack Cache Entry Register, SC_BOTTOM, indicates the current "deepest" entry of the operand stack that is valid in the stack cache. Do not write into this register if the stack cache dribbler is enabled (PSR.DRE = 1); otherwise, unpredictable behavior may result.

If the dribbler was on previously but is off now, writing to SC_BOTTOM and then enabling the dribbler can also cause unpredictable behavior. See *Enabling the Stack Cache* on page 446 regarding power-on writes of SC_BOTTOM.

The configuration of the SC_BOTTOM register is listed below and illustrated in FIGURE 2-6.

Field	Type	Description
31:2	RW	Word-aligned stack cache bottom pointer
1:0	RO	Always reads as 0x0.

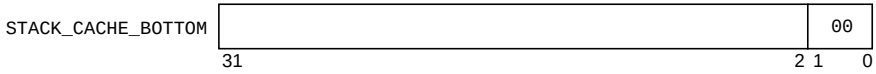


FIGURE 2-6 Address of Deepest Stack Cache Entry Register (SC_BOTTOM)

2.3 Constant Pool Base Pointer Register (CONST_POOL)

The Constant Pool Base Pointer Register, CONST_POOL, contains the base address—the address of element zero—of the current constant pool for a Java class. Additional elements of the constant pool are located at word-aligned offsets toward higher addresses. Code compiled from other languages may not use the CONST_POOL register in this manner, however. See *Constant Pool* on page 73 for more information.

The configuration of the CONST_POOL register is listed below and illustrated in FIGURE 2-7.

Field	Type	Description
31:2	RW	Word-aligned constant pool pointer
1:0	RO	Always reads as 0x0.

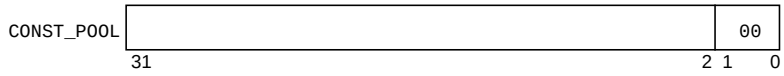


FIGURE 2-7 Constant Pool Base Pointer Register (CONST_POOL)

2.4 Memory Protection Registers (USERRANGE1 and USERRANGE2)

The USERRANGE1 and USERRANGE2 registers handle protection of memory. See *Memory Protection* on page 27 for details.

The configurations of the USERRANGE registers are listed below and illustrated in FIGURE 2-8.

Field	Type	Description
31:16	RW	USERHIGH: Bits 29:14 of the maximum address for the first restricted address range (16-Kbyte aligned)
15:0	RW	USERLOW: Bits 29:14 of the minimum address for the first restricted address range (16-Kbyte aligned)

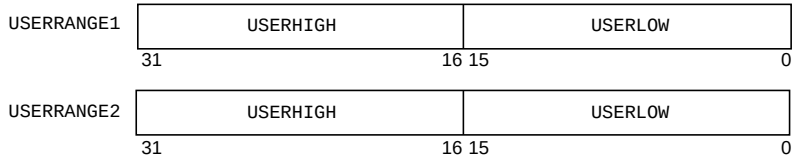


FIGURE 2-8 USERRANGE Registers

2.5 Processor Status Register (PSR)

The Processor Status Register, PSR, is a master register that controls which features are used at any given time and at what level.

The configuration of the PSR register is listed below and illustrated in FIGURE 2-9.

Field	Type	Description
31:23	Reserved	
21:19	RO	Dribble High-Watermark (DBH): This field specifies the most significant 3 bits of the high watermark for the dribbler.
18:16	RW	Dribble Low-Watermark (DBL): This field specifies the most significant 3 bits of the low watermark for the dribbler.
22	RW	Complete Address Check (CAC): When the core enables this bit along with the ACE bit, it checks all the addresses to the data cache with USERRANGEs in the protection registers. If the CAC bit is off, the core checks only the specific instructions against the USERRANGEs in the protection registers.
15	RW	drem Trap (DRT): The drem instruction causes an emulation trap even if the Floating Point Unit (FPU) is present. If this bit is 1 or if there is no FPU, then the core generates the emulation trap. This bit is 0 if the FPU handles the drem.
14	RW	Boot Mode 8 (BM8): If this bit is set, the instruction fetch size is 8 bits instead of the default 32 bits. The core initializes this bit from the pj_boot8 input signal at reset. Software can clear this bit but cannot set it to 1.
13	RW	Address Checking Enabled (ACE).
12	RW	Garbage Collection Page Check Enabled (GCE).
11	RW	Floating Point Unit Enabled (FPE).
10	RW	Data Cache Enabled (DCE).
9	RW	Instruction Cache Enabled (ICE).
8	RW	Asynchronous Store Error Mask bit (AEM).
7	RW	Dribbling Enabled (DRE): If this bit is set to 0, the stack cache contents and memory may mismatch. Always set this bit to 1, except during bootup and diagnostics.
6	RW	Folding Enabled (FLE).
5	RW	Supervisor (privileged) mode bit (SU): 1 sets this mode.
4	RW	Interrupt Enable (IE).
3:0	RW	Processor Interrupt Level (PIL).

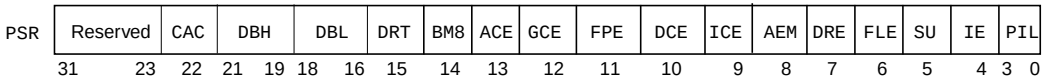


FIGURE 2-9 Processor Status Register (PSR)

2.6 Trap Handler Address Register (TRAPBASE)

When a trap occurs (except for external reset), the core writes a value that identifies the trap into the 8-bit TT field of the TRAPBASE register. Using the TRAPBASE address, the processor fetches the trap routine address and jumps to it. For more information on traps, see Chapter 4, *Traps and Interrupts*.

The configuration of the TRAPBASE register is listed below and illustrated in FIGURE 2-10.

Field	Type	Description
31:11	RW	TBA: Trap base address of the trap table.
10:3	RO	TT: Trap type; read-only to software but written by hardware when a trap occurs.
2:0	RO	Always reads as 0x0.

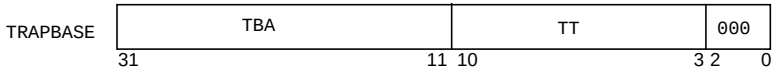


FIGURE 2-10 Trap Handler Address Register (TRAPBASE)

2.7 Monitor-Caching Registers

The picoJava-II core uses two lock count registers (`LOCKCOUNT0` and `LOCKCOUNT1`) and two lock address registers (`LOCKADDR0` and `LOCKADDR1`) to accelerate the `monitorenter` and `monitorexit` instructions. For details, see Chapter 8, *Monitors*.

2.7.1 Lock Count Registers (LOCKCOUNT[0..1])

The configurations of the two LOCKCOUNT registers are listed below and illustrated in FIGURE 2-11.

Field	Type	Description
31:16	Reserved	
15	RW	CO (Cached-Only): Set to 1 if only record of a lock is in these registers.
14	RW	LOCKWANT: Set to 1 if another thread is blocked on this monitor.
13:8	Reserved	
7:0	RW	COUNT: Number of times the current thread has entered the associated monitor.

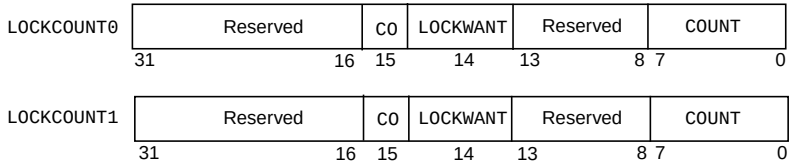


FIGURE 2-11 Lock Count Registers (LOCKCOUNT0 and LOCKCOUNT1)

2.7.2 Lock Address Registers (LOCKADDR[0..1])

The configurations of the two LOCKADDR registers are listed below and illustrated in FIGURE 2-12.

Field	Type	Description
31:2	RW	Word-aligned locked address pointer.
1:0	RO	Always reads as 0x0.

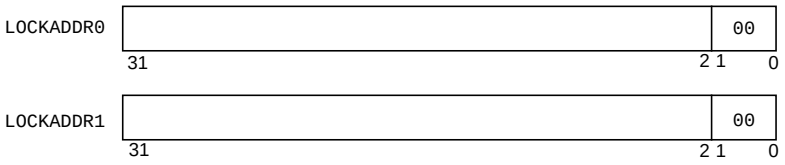


FIGURE 2-12 Lock Address Registers (LOCKADDR0 and LOCKADDR1)

Note – When an object reference is compared to the contents of either of these registers, only bits<29:2> are compared.

2.8 Garbage Collection Register (GC_CONFIG)

The GC_CONFIG register, writable in privileged mode, supports garbage collection by filtering stores to the heap. See Chapter 11, *Support for Garbage Collection* for more information.

The configuration of the GC_CONFIG register is listed below and illustrated in FIGURE 2-13.

Field	Type	Description
31:21	RW	REGION_MASK: This mask AND'ed with bits 28:18 in the reference and the stored data is used to decide if they belong to the same page.
20:16	RW	CAR_MASK: This mask AND'ed with bits 17:13 in the reference and the stored data is used to decide if they belong to the same car.
15:0	RW	WB_VECTOR (Write Barrier Vector): For a store of an object reference, the GC_TAG bits of the object reference being stored are concatenated with those of the destination object reference to form a 4-bit index into the WB_VECTOR. If the corresponding bit in the WB_VECTOR field is set, then the aputstatic_quick and aputfield_quick instructions signal a GC trap. For example, if GC_TAG{objref} = 00 and GC_TAG{store_data} = 01, then these instructions check bit 1 of WB_VECTOR.

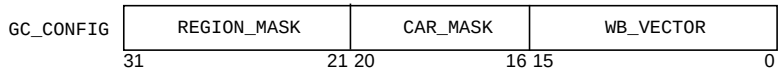


FIGURE 2-13 Garbage Collection Register (GC_CONFIG)

2.9 Breakpoint Registers

Two breakpoint registers (BRK1A and BRK2A) and a breakpoint control register (BRK12C) manipulate breakpoints.

Debugging and use of the breakpoint registers are discussed in Chapter 12, *System Management and Debugging*. For completeness and convenience, the register information is reprinted here.

The configuration of the BRK1A register is listed below and illustrated in FIGURE 2-14.

Field	Type	Description
31:00	RW	This is the breakpoint1 address against which to compare. This register is used along with BRK12C to set a breakpoint.

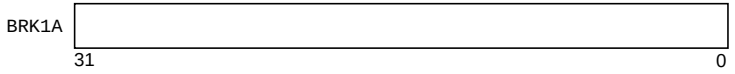


FIGURE 2-14 Breakpoint Register (BRK1A)

The configuration of the BRK2A register is listed below and illustrated in FIGURE 2-15.

Field	Type	Description
31:00	RW	This is the breakpoint2 address against which to compare. This register is used along with BRK12C to set a breakpoint. You can set a maximum of two breakpoints at one time.

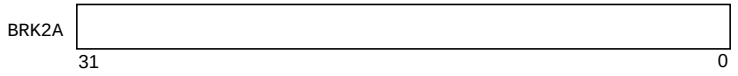


FIGURE 2-15 Breakpoint Register (BRK2A)

The configuration of the BRK12C register is listed below and illustrated in FIGURE 2-16.

Field	Type	Description
31	RW	HALT – Determines if a breakpoint halts or traps. At setting 0, the breakpoint traps (default); at setting 1, the core halts all transactions.
30:24	RW	BRKM2 – Mask bits for breakpoint2 <30> – Enable compare of BRK2A<31:13> <29> – Enable compare of BRK2A<12> <28> – Enable compare of BRK2A<11:4> <27> – Enable compare of BRK2A<3> <26> – Enable compare of BRK2A<2> <25> – Enable compare of BRK2A<1> <24> – Enable compare of BRK2A<0>
23	Reserved	
22:16	RW	BRKM1 – Mask bits for breakpoint1 <22> – Enable compare of BRK1A<31:13> <21> – Enable compare of BRK1A<12> <20> – Enable compare of BRK1A<11:4> <19> – Enable compare of BRK1A<3> <18> – Enable compare of BRK1A<2> <17> – Enable compare of BRK1A<1> <16> – Enable compare of BRK1A<0>
15:12	Reserved	
11	RW	SUBRK2 – Supervisor (privileged) or user access for breakpoint2
10:9	RW	SRCBRK2 – Source for breakpoint2 0x0 – Data cache read 0x1 – Data cache write 0x2 – Reserved 0x3 – Instruction cache fetch (folding disabled)
8	RW	BRKEN2 – Breakpoint2 trap enable bit 1 – The breakpoint is enabled 0 – The breakpoint is disabled
7:4	Reserved	

Field	Type	Description
3	RW	SUBRK1 – Supervisor (privileged) or user access for breakpoint1. If set to 1, then the core ignores the breakpoint in privileged mode.
2:1	RW	SRCBRK1 – Source for breakpoint1 0x0 – Data cache read 0x1 – Data cache write 0x2 – Reserved 0x3 – Instruction cache fetch (folding disabled)
0	RW	BRKEN1 – Breakpoint1 trap enable bit 1 – The breakpoint is enabled 0 – The breakpoint is disabled

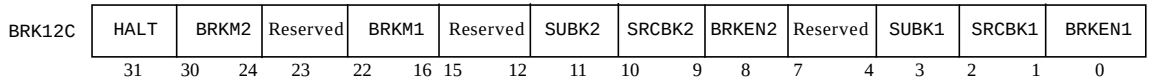


FIGURE 2-16 Breakpoint Control Register (BRK12C)

2.10 Implementation Registers

The Version ID Register (VERSIONID) and the Hardware Configuration Register (HCR) are two implementation registers that contain unique information about the core.

2.10.1 Version ID Register (VERSIONID)

The Version ID Register (VERSIONID) contains the licensee number that tracks closely to the JEDEC number that is assigned to a manufacturer. You can assign licensee revision and mask numbers to distinguish the features that have been added or deleted from the original core.

The configuration of the VERSIONID register is listed below and illustrated in FIGURE 2-17.

Field	Type	Description
31:16	RO	LN: Licencee number
15:8	RO	LMN: Licensee mask number
7:0	RO	LRN: Licensee revision number

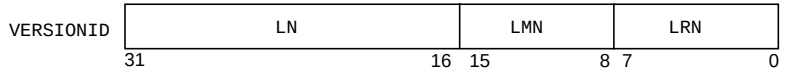


FIGURE 2-17 Version ID Register (VERSIONID)

2.10.2 Hardware Configuration Register (HCR)

The Hardware Configuration Register (HCR) contains hardwired, read-only fields that the operating system reads to determine the parameters of the core version.

The configuration of the HCR register is listed below and illustrated in FIGURE 2-18.

Field	Type	Description
31:30	RO	DCA – Data cache associativity 0x0 – Two-way set-associative (default) 0x1 – Four-way set-associative 0x2 – Direct-mapped 0x3 – Reserved
29:27	RO	DCL – Data cache line size 0x0 – 4 bytes 0x1 – 8 bytes 0x2 – 16 bytes (default) 0x3 – 32 bytes 0x4 – 64 bytes 0x5 – 128 bytes 0x6 – Reserved 0x7 – Reserved

Field	Type	Description
26:24	RO	ICL – Instruction cache line size 0x0 – 4 bytes 0x1 – 8 bytes 0x2 – 16 bytes (default) 0x3 – 32 bytes 0x4 – 64 bytes 0x5 – 128 bytes 0x6 – Reserved 0x7 – Reserved
23:21	RO	DCS – Data cache size 0x0 – 0 Kilobytes (no data cache present) 0x1 – 1 Kbytes 0x2 – 2 Kbytes 0x3 – 4 Kbytes 0x4 – 8 Kbytes 0x5 – 16 Kbytes 0x6 – 32 Kbytes 0x7 – Reserved
20:18	RO	ICS – Instruction cache size 0x0 – 0 Kbytes (no instruction cache present) 0x1 – 1 Kbytes 0x2 – 2 Kbytes 0x3 – 4 Kbytes 0x4 – 8 Kbytes 0x5 – 16 Kbytes 0x6 – 32 Kbytes 0x7 – Reserved
17	RO	FPP – Floating Point Unit (FPU) present This bit is set to 1 if the FPU is present in the hardware.
16	RO	ICA – Instruction cache associativity 0x0 – Direct-mapped (default) 0x1 – Two-way set-associative
15:8	Reserved[1]	
7:0	RO	SRN – Sun revision number[2]

[1]Reads as zeros.

[2]May vary among different versions of the picoJava core.

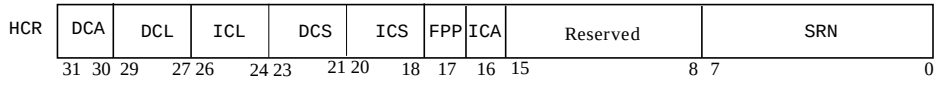


FIGURE 2-18 Hardware Configuration Register (HCR)

2.11 Global Registers (GLOBAL[0..3])

Four global registers (GLOBAL[0..3]) store global information in applications. Some operating systems or C language environments may reserve some or all of these registers for specific uses.

The configuration of a GLOBAL register is listed below and illustrated in FIGURE 2-19.

Field	Type	Description
31:00	RW	Any data

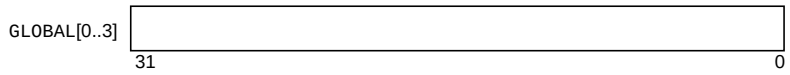


FIGURE 2-19 Global (GLOBAL[0..3]) Registers

CHAPTER 3

Memory System and Caches

This chapter describes the memory system architecture of the picoJava-II core as well as the properties and behavior of the caches. It contains the following sections:

- *Architecture of the Memory System* on page 21
- *Memory Protection* on page 27
- *Cache Coherency* on page 30
- *Instruction Cache* on page 33
- *Data Cache* on page 36
- *Stack Cache* on page 39

3.1 Architecture of the Memory System

Memory access by the core falls into one of the following categories:

- **Instruction access** —The core fetches instructions from memory to execute; any code thus fetched is an instruction access.
- **Stack access** — Most instructions derive their operands from or store their results to a special region of memory, known as the *operand stack*, which also contains the call frame, parameters, and local variables of a given method or function.
- **Data access** — Any memory access that is not one of the above two categories is a data access, which is usually the result of executing instructions that access the fields of objects, traverse data structures, or perform I/O. You can perform data accesses to examine or manipulate the stack or instructions. Be sure to take precautions for such data accesses, however.

System software often divides the address space into regions that correspond to the different access types. When the code in a program or Java class is loaded, the loader places the instructions into memory. Generally, the core uses instruction accesses to these locations during execution. System software also uses a region of memory to

hold the operand stack and another region—known as the *heap*—to allocate data for the programs that are running. Other data access regions may include the operating system data and I/O devices.

The core provides the following caches for memory access:

- An instruction cache, which holds instructions that were executed recently in local and fast memory (see *Instruction Cache* on page 33)
- A data cache, which holds data that was recently accessed in local and fast memory, including elements of the stack (see *Data Cache* on page 36)
- A stack cache, which holds data that is frequently accessed near the top of the stack (see *Stack Cache* on page 39)

  The stack cache allows multiple simultaneous accesses and *dribbling* of data into and out of the cache.

3.1.1 Address Space

The core can address up to 1 gigabyte (30 bits) of memory directly. This address space is organized as a flat and linear range. The core does not provide any virtual memory or address translation mechanisms. Any such mechanisms can only exist in logic outside the core.

The core supports 32-bit data elements. For an address that is larger than the 30-bit address space that it supports, the core ignores the two most significant bits <31:30> to determine the memory location that is referenced. The value of those two bits may, however, encode additional information about a memory access and thus change its behavior. *Therefore, do not assume that two accesses to addresses that differ only in bits <31:30> produce the same results.*

3.1.2 Alignment

The core does *not* support unaligned data accesses. Accesses to 2-byte and 4-byte data items must be to addresses that are aligned on 2-byte or 4-byte boundaries, respectively. The core performs accesses to 8-byte data items internally as two separate 4-byte accesses, which only need to be aligned on a 4-byte boundary.

The core interprets instructions as a stream of bytes and does not impose any restrictions for alignment, except for the `lookupswitch` and `tableswitch` instructions, which are followed by 4-byte-aligned jump tables. Methods or functions that use `lookupswitch` and `tableswitch` must start on a 4-byte boundary.

Failure to meet the above requirements causes the core to generate a `mem_address_not_aligned` trap.

3.1.3 Cacheable and Noncacheable Memory Regions

By default, the core caches data and instruction accesses to the memory region from address 0x00000000 to address 0x2fffffff. It does not cache accesses to the memory region from address 0x30000000 to address 0x3fffffff. See FIGURE 3-1.

Note – Explicitly noncacheable instructions, such as `ncload_word`, can access memory within the cacheable address region without bringing that data into the cache.

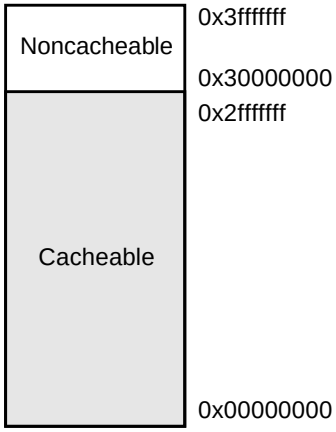


FIGURE 3-1 The Cacheable and Noncacheable Memory Regions

Caution – Do not allow the stack to reside in the noncacheable address region. The operand stack must be cacheable for proper operation of the core; otherwise, unpredictable behavior may result.

3.1.4 Endianness

In general, the core stores values in memory in big-endian byte order; by default, most instructions use a big-endian address space. However, some load and store instructions can access both big-endian and little-endian data.

TABLE 3-1 lists the instructions that access big-endian data by default.

TABLE 3-1 Instructions That Access Big-Endian Data by Default

load_char	load_word	ncload_word	store_short
load_char_index	load_word_index	ncstore_short	store_short_index
load_short	ncload_char	ncstore_word	store_word
load_short_index	ncload_short	nastore_word_index	store_word_index

These instructions use a big-endian byte order. However, if bit 30 of address <31:0> is set to 1, they convert the incoming loads or outgoing stores to access with a little-endian byte order.

TABLE 3-2 lists the instructions that access little-endian data by default.

TABLE 3-2 Instructions That Access Little-Endian Data by Default

load_char_oe	ncload_short_oe	ncstore_word_oe
load_short_oe	ncload_word_oe	store_short_oe
load_word_oe	ncstore_short_oe	store_word_oe
ncload_char_oe		

The _oe suffix in these instructions stands for *opposite endianness*—that is, they assume the opposite endianness of data when compared to the equivalent instructions that do not have that suffix in their names. They assume that the data is in little-endian order unless bit 30 of address <31:0> is set to 1; then, the core performs the access with a big-endian byte order.

The data cache stores the data in the same byte order as memory. The big-to-little-endian ordering swap occurs as the core reads data from memory (or the data cache) and pushes it onto the stack.

Data on the stack is in big-endian byte order. Thus, the core swaps little-endian data that is in memory into big-endian byte order while it is on the stack and then swaps it back to little-endian byte order as it returns the data to memory.

Note – Since data on the stack is stored in big-endian byte order, the core arranges items of the type long or double (64-bit values) that occupy two 32-bit stack entries such that the *most significant* 32 bits of each 64-bit value are the stack entry that is the *closest* to the top of the stack. When loading 64-bit data items onto the stack, you must ensure that the order in which the two 32-bit loads place the two halves of the 64-bit item on the stack preserves the correct data ordering. Take similar precautions when you store data back to memory.

TABLE 3-3 shows the behavior of single-byte load and store accesses as a result of byte ordering and sign extension. Big-endian and little-endian accesses of a single byte are identical.

TABLE 3-3 Byte Ordering and Sign Extension for Single-Byte Memory Operations

Operation	Endianness	Address <1:0>	Memory Contents at Address <1:0>				Top of Stack
			00	01	10	11	
Load(byte)	Big and little endian	00	0xdd	xxxx	xxxx	xxxx	0xffffffdd
Load(byte)	Big and little endian	01	xxxx	0xcc	xxxx	xxxx	0xffffffcc
Load(byte)	Big and little endian	10	xxxx	xxxx	0xbb	xxxx	0xffffffbb
Load(byte)	Big and little endian	11	xxxx	xxxx	xxxx	0xaa	0xffffffaa
Load(ubyte)	Big and little endian	00	0xdd	xxxx	xxxx	xxxx	0x000000dd
Load(ubyte)	Big and little endian	01	xxxx	0xcc	xxxx	xxxx	0x000000cc
Load(ubyte)	Big and little endian	10	xxxx	xxxx	0xbb	xxxx	0x000000bb
Load(ubyte)	Big and little endian	11	xxxx	xxxx	xxxx	0xaa	0x000000aa
Store(byte)	Big and little endian	00	0xdd	xxxx	xxxx	xxxx	0xaabbccdd
Store (byte)	Big and little endian	01	xxxx	0xdd	xxxx	xxxx	0xaabbccdd
Store(byte)	Big and little endian	10	xxxx	xxxx	0xdd	xxxx	0xaabbccdd
Store(byte)	Big and little endian	11	xxxx	xxxx	xxxx	0xdd	0xaabbccdd

TABLE 3-4 shows the behavior of multibyte load and store accesses as a result of byte ordering and sign extension.

TABLE 3-4 Byte Ordering and Sign Extension for Multiple-Byte Memory Operations

Operation	Endianness	Address <1:0>	Memory Contents at Address <1:0>				Top of Stack
			00	01	10	11	
Load(short)	Big endian	00	0xdd	0xcc	xxxx	xxxx	0xffffddcc
Load(short)	Big endian	10	xxxx	xxxx	0xbb	0xaa	0xffffbbaa
Load(short)	Little endian	00	0xdd	0xcc	xxxx	xxxx	0xffffccdd
Load(short)	Little endian	10	xxxx	xxxx	0xbb	0xaa	0xffffaabb
Load(char)	Big endian	00	0xdd	0xcc	xxxx	xxxx	0x0000ddcc
Load(char)	Big endian	10	xxxx	xxxx	0xbb	0xaa	0x0000bbaa
Load(char)	Little endian	00	0xdd	0xcc	xxxx	xxxx	0x0000ccdd
Load(char)	Little endian	10	xxxx	xxxx	0xbb	0xaa	0x0000aabb
Load(word)	Big endian	00	0xdd	0xcc	0xbb	0xaa	0xddccbbaa
Load (word)	Little endian	00	0xdd	0xcc	0xbb	0xaa	0xaabbccdd
Store(short)	Big endian	00	0xcc	0xdd	xxxx	xxxx	0xaabbccdd
Store(short)	Big endian	10	xxxx	xxxx	0xcc	0xdd	0xaabbccdd
Store(short)	Little endian	00	0xdd	0xcc	xxxx	xxxx	0xaabbccdd
Store(short)	Little endian	10	xxxx	xxxx	0xdd	0xcc	0xaabbccdd
Store(word)	Big endian	00	0xaa	0xbb	0xcc	0xdd	0xaabbccdd
Store(word)	Little endian	00	0xdd	0xcc	0xbb	0xaa	0xaabbccdd

Recall that the core interprets instructions as a stream of bytes; it stores constants in the instruction stream in big-endian byte order. Like the data cache, the instruction cache has the same byte order as memory.

3.1.5 Erroneous Memory Transactions

In response to bus requests, error acknowledgment codes signal to the core bus transactions that fail because of memory or system errors. The scenarios are as follows:

- If an error occurs during an instruction fetch, the core generates a fault bit and associates it with the instruction word. Subsequently, it does *not* write the line into the instruction cache even if the cache is enabled. Instead, it propagates the

faulty instruction through the core along with the fault bit; when execution reaches the faulty instruction, the core takes an `instruction_access_error` trap.

- If an error occurs during a data read, then, depending on the error acknowledgment code the core receives, the core takes a `data_access_mem_error` or `data_access_io_error` trap. It does not write the data into the data cache even if the cache is enabled—it leaves the state of the machine exactly as it was before execution started. All instructions perform all data reads before they modify the state of the processor or memory.
- If an error occurs on a data store, the core takes an `asynchronous_error` trap. Since stored data can reside in the data cache for some time before it is written to memory, the error acknowledgment may signal after the instruction that caused the error has completed its execution. In that case, the state of the machine may be inconsistent since the instruction or store that failed is unknown.

For more information on traps and trap types, see Chapter 4, *Traps and Interrupts*.

3.2 Memory Protection

The core contains an address-checking mechanism that can prevent accesses to locations outside specified memory regions. Hence, you can create a “safe” execution area for C-based programs that can then run in their own areas of memory but cannot read or write to memory outside of them, thereby having no effect on other programs. Similarly, this mechanism can support multiple and independent Java virtual machines by ensuring that an instantiation of the Java virtual machine affects itself only.

The protection mechanism checks data and stack accesses only. Since instruction accesses are read-only and, therefore, cannot change memory, they are not subject to those checks.

3.2.1 The Address-Checking Process

The ACE and CAC bits in the PSR register determine the circumstances under which a memory check takes place, as follows.

Note – Memory checks occur regardless of the value of the PSR.SU bit.

- If PSR.ACE = 1 and PSR.CAC = 1, the core checks all data and stack memory references by any instruction to verify that the address is in the range designated by the USERLOW and USERHIGH fields of either of the USERRANGE registers.

  The core checks only the first access of a stack location, but not the stack accesses that hit in the stack cache.

Caution – The stack cache can access within 256 bytes of the top of stack even without an explicit reference to that memory location. *Do not allocate the stack within 256 bytes of the boundary of a memory region.*

- If PSR.ACE = 1 and PSR.CAC = 0, the core checks the data accesses of instructions that are not also a part of the instruction set of the Java virtual machine, as listed in TABLE 3-5. It does *not* check any stack accesses.

TABLE 3-5 Instructions Subject to Memory Protection Checks When PSR.CAC = 0

cache_flush	load_short_oe	ncload_short	store_byte
cache_index_flush	load_ubyte	ncload_short_oe	store_short
cache_invalidate	load_ubyte_index	ncload_ubyte	store_short_index
load_byte	load_word	ncload_word	store_short_oe
load_byte_index	load_word_index	ncload_word_oe	store_word
load_char	load_word_oe	ncstore_byte	store_word_index
load_char_index	nastore_word_index	ncstore_short	store_word_oe
load_char_oe	ncload_byte	ncstore_short_oe	zero_line
load_short	ncload_char	ncstore_word	
load_short_index	ncload_char_oe	ncstore_word_oe	

- If PSR.ACE = 0, then the core does not perform any memory checks.

3.2.2 Memory Regions

The USERRANGE1 and USERRANGE2 registers specify two memory regions that can be accessed by the program that is running, as follows:

- The 16-bit USERLOW and 16-bit USERHIGH fields of each register form 16-Kbyte-aligned addresses.
- The 16 bits of each field correspond to bits <29:14> of the addresses.

The resulting addresses then form the low and high limits of each region of allowable memory accesses. See FIGURE 3-2.

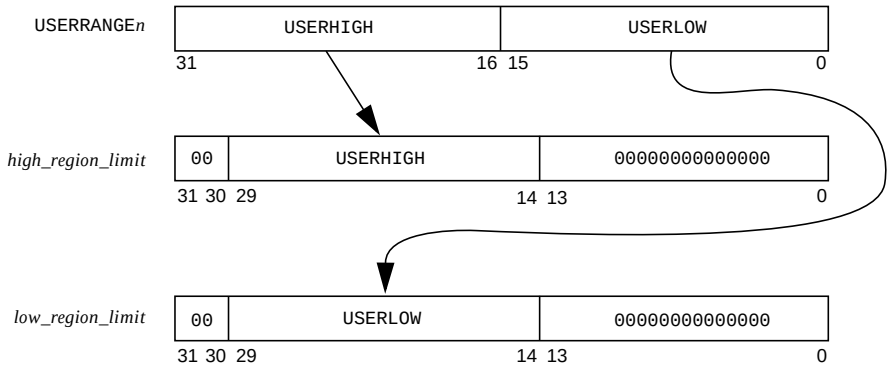


FIGURE 3-2 How a Range of Accessible Memory Is Formed from USERRANGE Register Values

A memory address that is checked must be greater than or equal to the address that corresponds to USERLOW and less than the address that corresponds to USERHIGH for either (or both) of the USERRANGE registers.

If USERLOW is greater than USERHIGH, then the resulting memory range is of zero size, which allows no access.

If the checked memory access is outside both address ranges, then a mem_protection_error trap (trap type 0x02) occurs.

3.2.3 Limits for Stack Growth

You can use another mechanism to limit the extent to which the operand stack can grow. The OPLIM register specifies the minimum value that OPTOP (the top of stack pointer) can hold. If bit 31 of OPLIM is set to 1 and OPTOP < OPLIM, then the core signals oplim_trap (trap type 0x0c). This way, the stack region cannot grow beyond its allocated area.

You can also use this check for a technique called *stack chunking*, which is described in Chapter 10, *Stack Chunking*.

3.3 Cache Coherency

On occasion, the caches in the core may contain different data that corresponds to the same logical memory location. When two caches or a cache and memory contain the same contents that are stored at specific memory locations, they are said to be *coherent*.

3.3.1 Coherency for Stack and Data Accesses

FIGURE 3-3 illustrates the three types of caching operations that take place as a result of data or stack accesses during execution of instructions. Different types of accesses bypass levels of the cache hierarchy. For example, memory operations bypass the stack cache and access the data cache first if it is present.

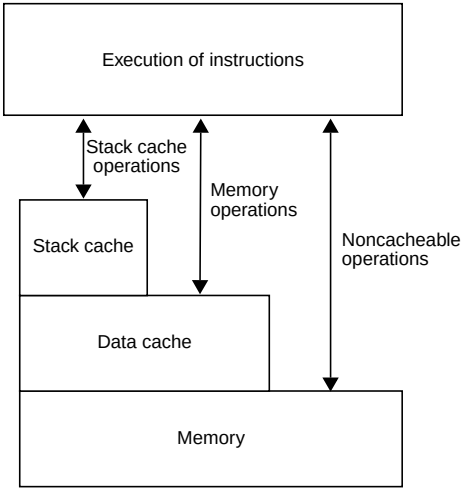


FIGURE 3-3 Caching Operations During Instruction Execution

TABLE 3-6 lists some of the instructions that trigger each of these operations.

TABLE 3-6 Instructions for Caching Operations

Stack cache operations		
`iload` and others	`istore` and others	`bipush` and others
Data cache operations		
`cache_flush`	`load_short_index`	`store_short_index`
`cache_index_flush`	`load_short_oe`	`store_short_oe`
`cache_invalidate`	`load_word`	`store_word`
`load_char`	`load_word_index`	`store_word_index`
`load_char_index`	`nastore_word_index`	`store_word_oe`
`load_char_oe`	`store_short`	`zero_line`
`load_short`		
Noncacheable operations		
`ncload_char_oe`	`ncload_word`	`ncstore_short_oe`
`ncload_short`	`ncstore_char`	`ncstore_word`
`ncload_short_oe`	`ncstore_short`	`ncstore_word_oe`

When using these instructions, take into account the distinctions between each class of instructions and which caches they bypass; otherwise, you may cause inconsistent or incorrect program behavior.

Coherency of the Stack Cache and the Data Cache

The stack cache and the data cache are *not* coherent. If the stack cache contains data that corresponds to a given memory location, the data cache does not reflect updates to that location immediately. Because the stack cache is effectively a writeback cache, a memory access that bypasses the stack cache (via `load_word`, for example) does not access the stack cache and may access the “stale” data in the data cache instead.

Similarly, stores that bypass the stack cache do not automatically update the stack cache. A memory store (via `store_word`, for example) does not access the stack cache and updates data directly in the data cache instead. Because the stack cache is effectively a writeback cache, the core may sometimes write back stack cache data (which is no longer consistent with the more recently updated contents in the data cache) to the data cache and overwrite the more current contents in error.

Therefore, do not perform direct memory accesses that bypass the stack cache to locations that can also be present in the stack cache, that is, addresses from `OPTOP` *through* `OPTOP` *– 256, inclusive.*

To access a location that may be present in the stack cache with a memory operation that bypasses the stack cache, flush the contents explicitly out of the stack cache back to the data cache and relocate the stack so that the region to be accessed does *not* exist in the stack cache. For details, see *Flushing* on page 44.

Note – The preceding rules pertain to coherency between the stack cache and memory when no data cache is present.

Coherency of the Data Cache and Memory

The data cache and memory are *not* coherent. If the data cache contains data that corresponds to a given memory location, memory does not reflect updates to that location. Because the data cache is a writeback cache, a noncacheable memory access (via `ncload_word`, for example) does not access the data cache and may access the stale data in memory instead.

Similarly, direct stores to a given memory location do not automatically update the data cache. A noncacheable memory store (via `ncstore_word`, for example) does not access the data cache and updates data directly in memory instead. Because the data cache is a writeback cache, the core may sometimes write back data cache data (which is no longer consistent with the more recently updated contents in memory) and overwrite the more current contents in error.

Therefore, use caution when you perform direct noncacheable accesses to memory locations that can also be present in the data cache. We recommend strongly that you access all addresses that correspond to a given data cache line in a similar manner; that is, once you access a given address as cacheable, always access it as cacheable.

To access a previously cacheable location as noncacheable (to disable the previously enabled data cache, for example), flush the contents out of the data cache back to memory. For details, see *Flushing* on page 37.

3.3.2 Coherency for Instruction Accesses

Only when you modify the instruction space of a program that is running must you take care to ensure the coherency of the instruction cache as it relates to the data cache and memory.

Coherency of the Instruction Cache and the Data Cache

The instruction cache and the data cache are *not* coherent, that is, the instruction cache does not automatically reflect updates to the data cache. As a result, software that modifies instructions must ensure that the instructions being modified are not present in the instruction cache.

For specific steps, see *Modification of Instruction Space* on page 35.

Coherency of the Instruction Cache and Memory

The instruction cache and memory are *not* coherent, that is, the instruction cache does not automatically reflect updates to memory. As a result, software that modifies instructions must ensure that the instructions being modified are not present in the instruction cache.

For specific steps, see *Modification of Instruction Space* on page 35.

3.4 Instruction Cache

The instruction cache holds instructions that were accessed recently in local and fast memory because it is probable that they will be accessed again soon.

3.4.1 Configuration

The instruction cache is a direct-mapped cache with a line size of 16 bytes. The core supports different size configurations of the instruction cache—it can have a size of 0 Kbytes, 1 Kbyte, 2 Kbytes, 4 Kbytes, 8 Kbytes, or 16 Kbytes.

All implementations of the core have a fixed 16-byte line size for the instruction cache. However, in the Hardware Configuration Register (`HCR`):

- The `ICS` field specifies the size of the instruction cache.
- The `ICL` field specifies the line size of the instruction cache. This field facilitates software portability between different generations of the picoJava core family.
- The `ICA` field specifies the associativity of the instruction cache. This field facilitates software portability between different generations of the picoJava core family.

You can enable or disable the instruction cache by setting the ICE field of the Processor Status Register (PSR). An implementation with no instruction cache (a 0-Kbyte cache size) must have a setting of PSR.ICE = 0.

For details on the PSR, see *Constant Pool Base Pointer Register (CONST_POOL)* on page 9. For details on the HCR, see *Hardware Configuration Register (HCR)* on page 18.

3.4.2 Initialization

The instruction cache is disabled at reset, after which the contents of the tag, data, and status RAMs in the cache are undefined. Reset code must use diagnostic cache writes to initialize the cache prior to enabling it. See *Reset Management* on page 444 for details.

3.4.3 Operations

Several instructions affect the operation of the instruction cache.

Flushing

The cache_flush, cache_index_flush, and cache_invalidate instructions selectively invalidate the contents of the instruction cache, as follows:

- Both cache_flush and cache_index_flush perform the same function for the instruction cache—they invalidate the line in the instruction cache that corresponds to the specified address regardless of whether that line is actually in the instruction cache.
- cache_invalidate invalidates the line that contains the specified address only if it is present in the instruction cache.

These instructions also affect the data cache.

Diagnostic Accesses

Four instructions (priv_read_icache_tag, priv_write_icache_tag, priv_read_icache_data, and priv_write_icache_data) allow diagnostic accesses to the contents of the instruction cache.

By accessing the instruction cache tags with priv_read_icache_tag and priv_write_icache_tag, you determine:

- Whether the contents of a given instruction cache line are valid

- Which memory addresses are cached by that line

By accessing the instruction cache data with `priv_read_icache_data` and `priv_write_icache_data`, you examine or update the contents of instructions cached in a given instruction cache line.

Chapter 6, *Instruction Set* contains a more detailed description of these instructions for diagnostic accesses.

Caution – Execute diagnostic accesses to the instruction cache with care; diagnostic writes during normal operations with the caches enabled can result in unpredictable behavior. Also, because the instruction cache handles read-only data only and diagnostic writes to that cache are not written back to memory, inconsistencies between the cache and memory may occur.

3.4.4 Modification of Instruction Space

Software that modifies instruction space, such as self-modifying code and some trap code, must perform the following steps to ensure correct functionality when the modified code resides in cacheable space and the data cache is on:

1. **Execute a store instruction to store to the modified code address.**
2. **Execute the `cache_flush` instruction to the modified code address.**
3. **Perform other operations, if any.**
4. **Execute a branch to the modified code address.**

If the modified code resides at a noncacheable memory address or if the data cache is off, add an additional step after step 2. The steps then become:

1. **Execute an extended store instruction to store to the modified code address.**
2. **Execute a `cache_flush` instruction to the modified code address.**
3. **Execute a load instruction to perform a load from the same address.**

   This additional step is necessary because `cache_flush` does *not* flush pending writes to memory if the address is noncacheable or if the data cache is disabled. Adopting this protocol may allow an instruction load to overtake a store to instruction space and cause the program to use a code address that has been modified by old data. The load instruction serializes these operations by forcing the store to be written out to memory.
4. **Perform any other operations, if any.**
5. **Execute a branch to the modified code address.**

Note – `cache_flush` and `cache_index_flush` do not flush the contents of the instruction buffer or the instruction pipeline. Therefore, software must ensure that the code being changed does *not* already exist in the instruction buffer or the pipeline by ensuring that the location for the modified code lies *outside* the 32 bytes that follow the flush instruction.

3.5 Data Cache

The data cache holds data that was accessed recently in local and fast memory because it is probable that such data will be accessed soon.

3.5.1 Configuration

The data cache is a two-way set-associative, writeback, write-allocate cache and uses a pseudo-Least Recently Used (LRU) replacement policy. The data cache can have a size of 0 Kbytes, 1 Kbyte, 2 Kbytes, 4 Kbytes, 8 Kbytes, or 16 Kbytes.

All implementations of the core have a fixed 16-byte line size for the data cache. However, in the Hardware Configuration Register (HCR):

- The DCS field specifies the size of the cache.
- The DCL field specifies the line size of the data cache. This field facilitates software portability between different generations of the picoJava core family.
- The DCA field specifies the associativity of the data cache. This field facilitates software portability between different generations of the picoJava core family.

You can enable or disable the data cache by setting the DCE field of the Processor Status Register (PSR). An implementation with no data cache (a 0-Kbyte cache size) must have a setting of PSR.DCE = 0.

For details on the PSR, see *Constant Pool Base Pointer Register (CONST_POOL)* on page 9. For details on the HCR, see *Hardware Configuration Register (HCR)* on page 18.

Caution – Because the data cache is a writeback cache, data in memory may not be consistent with the data that corresponds to its address in the data cache. See *Coherency of the Data Cache and Memory* on page 32 for details.

3.5.2 Initialization

The data cache is disabled at reset, after which the contents of the tag, data, and status RAMs in the cache are undefined. Reset code must use diagnostic cache writes to initialize the cache prior to enabling it. See *Reset Management* on page 444 for details.

3.5.3 Operations

Several instructions affect the operation of the data cache.

Flushing

The `cache_flush`, `cache_index_flush`, and `cache_invalidate` instructions selectively flush or invalidate the contents of the data cache, as follows:

- `cache_flush` evicts the line that contains the specified address if it is present in the data cache, causing the data to be written back to memory if the line has been modified.
- `cache_index_flush` flushes the line that corresponds to the specified address regardless of whether that line is actually in the data cache. For the two-way set-associative data cache, bit 31 of the address indicates the set to be flushed.

  If the cache line is modified, `cache_index_flush` writes it back to memory.

  Use `cache_index_flush` to flush the entire contents of the data cache to memory.
- `cache_invalidate` invalidates the line that contains the specified address if it is present in the data cache. The core performs no writebacks even if the line is modified.

Diagnostic Accesses

Four instructions (`priv_read_dcache_tag`, `priv_write_dcache_tag`, `priv_read_dcache_data`, and `priv_write_dcache_data`) allow diagnostic accesses to the contents of the data cache.

By accessing the data cache tags with `priv_read_dcache_tag` and `priv_write_dcache_tag`, you determine the following information about a given data line:

- Whether the contents of a given data line are valid or have been modified

- Whether the contents are the least recently used when compared to the corresponding data cache line in the other set (since the data cache is two-way set associative)
- Which memory addresses are cached by that line

By accessing the data cache data with `priv_read_dcache_data` and `priv_write_dcache_data`, you examine or update the contents of data cached in a given data cache line.

Chapter 6, *Instruction Set* contains a more detailed description of these instructions for diagnostic accesses.

Caution – Execute diagnostic accesses to the data cache with care; diagnostic writes during normal operations with the caches enabled can result in unpredictable behavior. Writebacks of diagnostic writes to the data cache to memory may not occur since they do not necessarily set the modified bit associated with the line. Without an explicit setting of the modified bit, inconsistencies between the cache and memory may occur.

Special Operations

Two instructions perform special operations that affect the data cache:

- The `zero_line` instruction performs a quick zeroing of a line in the data cache. If the line is not in the cache, `zero_line` allocates it there (possibly evicting another line) and, instead of fetching the data in the newly allocated line from memory, the core fills that line with zeroes.

  If the data cache is disabled (`PSR.DCE` = 0) when `zero_line` executes, then the core takes a `zero_line` emulation trap.

  For more information on `zero_line`, see page 380.
- The `nastore_word_index` instruction performs a nonallocating word write to a memory address. If the line that contains the memory address is not in the cache, however, this operation does *not* fetch it into the cache but stores the data directly to memory.

  No instructions perform a nonallocating store to any data size other than one word.

3.6 Stack Cache

The stack cache holds the top several values on the operand stack in fast, local memory. Those items are the most frequently accessed memory locations because of the stack-oriented nature of the picoJava-II instruction set. Thus, the core contains a special structure that is optimized for caching and providing access to the stack.

All memory accesses generated as offsets from OPTOP, VARS, or FRAME are stack accesses. The core checks first to see if the location required exists in the stack cache. All other memory accesses go directly to the data cache.

Through dribbling, the core enforces the requirements for minimum and maximum numbers of valid entries in the stack cache. It also maintains the number of valid entries in the stack cache near an optimum that you can specify.

3.6.1 Configuration

The stack cache is a 64-entry cache of a single contiguous range of addresses, which is the interval of OPTOP + 4 through SC_BOTTOM, inclusive. Its dribbler manipulates the value of SC_BOTTOM to keep the cached address range within the required number of entries.

You cannot disable the stack cache, but you can enable or disable its dribbler by setting the DRE field of the Processor Status Register (PSR). For details on the PSR, see *Constant Pool Base Pointer Register (CONST_POOL)* on page 9.

The dribbler must be enabled to ensure correct program behavior for arbitrary code. For details, see *Dribbling* on page 41.

Whenever you disable the dribbler (including during a reset), your program must handle the operation of the stack cache and you must tailor your code to manage that cache explicitly. Moreover, if the dribbler is disabled, you *must and can only* run code that tracks the contents of the 64 elements in the stack cache and that transfers data explicitly between the stack cache and memory as necessary.

Caution – Because the stack cache is a writeback cache, data in the data cache or memory may not be consistent with the data that corresponds to its address in the stack cache. A data access (via load_word, for example) does not access the stack cache and may access the stale data in memory instead. Therefore, as mentioned previously, use caution when you direct data accesses to memory locations that can be present in the stack cache, that is, with an address that is in the range of OPTOP to OPTOP – 256, inclusive.

FIGURE 3-4 shows how the stack cache caches part of the stack.

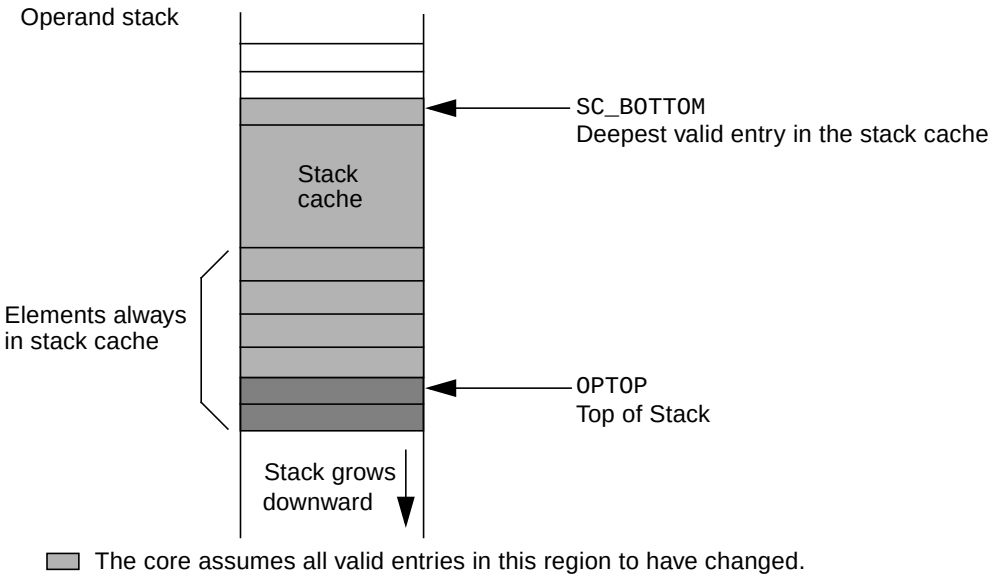


FIGURE 3-4 How the Stack Cache Caches Part of the Stack

3.6.2 Initialization

At reset (see *Reset Management* on page 444), the dribbler is disabled and the values are undefined. After reset, reset code must initialize the stack registers for proper operation of the stack cache. To initialize the stack registers:

1. **Set `SC_BOTTOM` and `OPTOP` to the same value.**
2. **Set `PSR.DBH` and `PSR.DBL` to the appropriate watermarks.**

   High and Low Watermarks on page 41 lists the values. `PSR.DBH` must be greater than `PSR.DBL`; both must be nonzero.
3. **Set `PSR.DRE` to 1 to enable the dribbler.**

Caution – Once the dribbler is enabled, the hardware maintains the `SC_BOTTOM` register. Writes to `SC_BOTTOM` may cause unpredictable behavior.

3.6.3 Dribbling

The dribbler may trigger either a spill or a fill transaction that takes place in the background to transfer data into and out of the stack cache; meantime, the core can continue to execute instructions.

Minimum Requirement of Entries

Memory accesses generated as offsets from OPTOP, VARS, or FRAME are stack accesses. The core checks if the location required exists in the stack cache. Generally, an access that misses the stack cache accesses the data cache and possibly memory. An address is a “hit” in the stack cache if the address is in the interval of OPTOP + 4 through SC_BOTTOM, inclusive.

Memory accesses generated to addresses from OPTOP + 16 through OPTOP – 4 must *always* hit the stack cache. As a result, the locations that correspond to the four top words in the stack and the two empty words beyond the top of the stack are present in the stack cache.

Since there are only 64 positions in the stack cache, the dribbler ensures that at least 4 valid entries are at the top of the stack and that the required 2 free entries are available, making a maximum of 62 valid entries. If these conditions are not present, the core suspends execution of instructions and takes corrective action to ensure enough free space on the stack cache, as described in *Spill and Fill Transactions* on page 42 and *Stack Overflows* on page 43.

If the dribbler is disabled, you must write your code to maintain the above requirements in the stack cache explicitly; otherwise, the required conditions are not guaranteed to be met.

High and Low Watermarks

In each cycle, the core compares the total available entries to a high watermark and a low watermark, which are determined by the 3-bit PSR.DBH and PSR.DBL fields, respectively. TABLE 3-7 lists the watermark values encoded by these fields.

TABLE 3-7 Encoded Values of Watermarks

PSR.DBH **or** PSR.DBL	**Watermark Value**
000	Reserved
001	8
010	16
011	24

TABLE 3-7 Encoded Values of Watermarks *(Continued)*

PSR.DBH **or** PSR.DBL	**Watermark Value**
100	32
101	40
110	48
111	56

The following rules for watermark values apply:

- The high watermark and the low watermark must not match.
- The low watermark must be less than the high watermark.
- The watermarks must remain unchanged while the dribbler is enabled; otherwise, unpredictable behavior may result.

Spill and Fill Transactions

In each cycle, the core determines the total available entries by calculating the difference between SC_BOTTOM and OPTOP. The result determines whether a spill or a fill transaction follows, as described below:

- If the number of valid entries is greater than the high watermark, then the core starts a spill transaction to memory by writing the value in the location at SC_BOTTOM to the data cache and updating SC_BOTTOM to point to the next valid entry in the stack cache. This process may repeat several times until the number of valid entries drops to equal the high watermark.

  When the stack cache is writing data from the stack cache into the data cache, the core allocates the line in the data cache if it is not already present, but does not fetch data for the allocated line from memory because it will be overwritten by data from the stack cache.

- If the number of valid entries is less than the low watermark, the core starts a fill transaction from memory by requesting the value in the location at SC_BOTTOM + 4 from the data cache and, once the data is available, updating SC_BOTTOM to point to the new entry in the stack cache. This process may repeat several times until the number of valid entries increases to reach the low watermark.

Remember that execution of instructions can continue during spill and fill transactions.

Note – The core assumes that all valid entries in the stack cache have changed and must be saved back to memory by a spill or stack overflow.

Stack Overflows

The stack overflows when OPTOP ≤ SC_BOTTOM – 60, causing OPTOP to "overflow" the region currently cached in the stack cache and requiring more than 64 elements in the stack cache to accommodate the growth. These actions ensue:

1. The core stops execution of instructions and spills the valid contents of the stack cache to the data cache or memory.
2. The dribbler verifies that the required topmost entries of the stack (based on the new OPTOP location) are present in the stack cache by reading them from the data cache or memory.

As soon as the stack cache contains the required entries, execution continues.

Note – The stack also overflows if, after write_optop or priv_update_optop writes to OPTOP, the new OPTOP value (OPTOP') is greater than the SC_BOTTOM value. Whenever OPTOP' meets the condition OPTOP < OPTOP' ≤ OPTOP + 64, depending on the SC_BOTTOM value, the stack may not overflow, in which case the core may not write the contents of the stack cache between OPTOP and OPTOP' back to memory.

Stack Underflows

The stack underflows when OPTOP > SC_BOTTOM, causing OPTOP to fall off the top of the stack cache. An underflow can occur when a method returns, as caused by the instructions listed in TABLE 3-8.

TABLE 3-8 OPTOP-Modifying Instructions That Can Cause a Stack Underflow

return	ireturn	areturn	freturn	dreturn
lreturn	return0	return1	return2	priv_ret_from_trap

These actions ensue:

1. The core stops execution of instructions.

   The core has popped the data that was in the stack cache and does not need to write this data back to the data cache or memory.

2. The dribbler ensures that the required topmost entries of the stack (based on the new OPTOP location) are present in the stack cache by reading them from the data cache or memory.

As soon as the stack cache contains the required entries, execution continues.

Note – If you use the `prev_ret_from_trap` instruction to facilitate context switching, other software must guarantee a writeback of the original context's stack cache contents to the data cache.

3.6.4 Flushing

At times, you may need to flush the contents of the stack cache to memory. To do so, use either of the following techniques:

- **Change to a new stack location.**

  If you use the `write_optop` or `priv_update_optop` instructions and the new `OPTOP` value causes a stack overflow condition, then the core saves the contents in the old stack and execution continues with the stack in the new location.

- **Push 64 entries worth of padding on to the stack to force the contents of the stack cache to be dribbled out.**

  A sequence of 32 `lconst_0` instructions forces all the previous stack contents to be dribbled out of the stack cache.

CHAPTER 4

Traps and Interrupts

At times, events in the picoJava-II core cause normal program execution to be suspended and control transferred to a service routine. These control transfers are called *traps*; the corresponding service routines, the *trap handlers*. Three types of traps can occur in the core:

- **Instruction Emulation** — A subset of the Java virtual machine instructions that must be emulated in software (see Chapter 6, *Instruction Set*)
- **Exceptions** — Conditions, such as runtime errors, exceptions, and hardware errors, generated during execution of instructions
- **Interrupts** — Signals generated by devices external to the core

This chapter describes the trap mechanism in the core in the following sections:

- *Traps* on page 45
- *Instruction Emulation* on page 54
- *Exceptions* on page 54
- *Interrupts* on page 56
- *Context Switch* on page 58

4.1 Traps

Traps are vectored transfers of control to the privileged state through a trap table. When the core takes a trap, it creates a trap frame in which to save its current state. It then branches to the location of the trap handler and continues execution there. Trap execution continues on the stack and, when complete, returns to the interrupted method.

4.1.1 Trap Table

The trap table base address resides in the trap base address (TBA) field of the TRAPBASE register (see *Trap Handler Address Register (TRAPBASE)* on page 12.) Software should initialize the TBA field of the TRAPBASE register to the upper 21 bits of the trap table address and must align the trap vector table on a 2-Kbyte boundary.

Note – Each entry in the trap table is 8 bytes, the first 4 of which contain the address of the trap handler. The core does not use the second 4 bytes, which software can use.

FIGURE 4-1 depicts the data structure of the trap table.

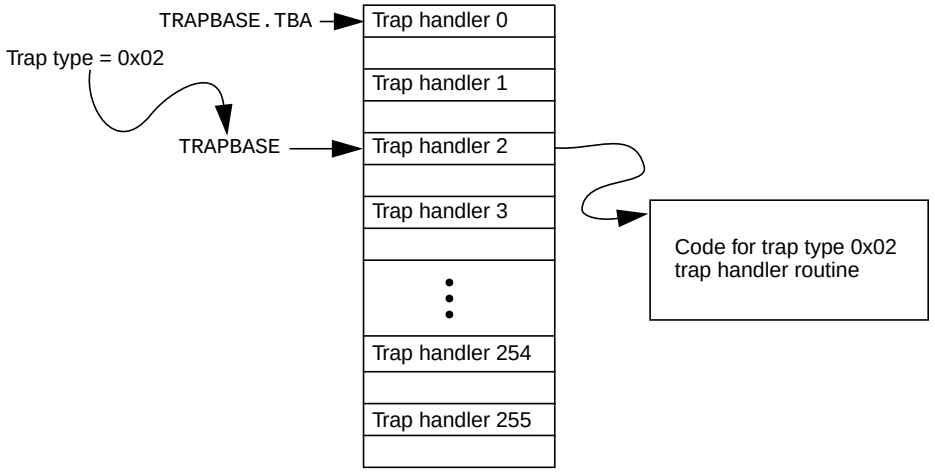


FIGURE 4-1 Data Structure of the Trap Table

For more information on trap types, see TABLE 4-1 on page 50.

4.1.2 The Process of Taking a Trap

As each instruction executes, the core checks for conditions that cause a trap. If multiple trap conditions occur simultaneously, the core takes only the highest priority trap. (See TABLE 4-1 on page 50 for the priorities of each trap condition.) The highest priority is 1; trap types with equal priority never occur simultaneously.

Since the stack mechanism provides a clean interface for trap invocations, the core allows nested trap levels. There is no hardware limit for the number of nested levels.

When it takes a trap or an interrupt, the core takes the following actions:

1. Push registers onto the stack in the following order:
   a. PSR
   b. PC
   c. VARS
   d. FRAME

   This step is uninterruptible. The PC pushed onto the stack is the address of the instruction that caused the trap or, in the case of interrupts or asynchronous errors, the next instruction to be executed.

2. Disable interrupts (PSR.IE ⇐ 0).

   The trap handler can enable interrupts during its execution by setting PSR.IE back to 1.

3. Enter privileged mode (PSR.SU ⇐ 1).

4. Update FRAME to point to the location of the saved PC value on the stack.

5. Write a value that identifies the trap into the 8-bit TT field of the TRAPBASE register.

Note – The TT field of the TRAPBASE register is set by hardware to the value of the trap type; it retains that value until the next trap or interrupt. The TT field does not revert to its previous value when it returns from a trap or interrupt handler. Therefore, use the TT value only under controlled circumstances, for example, when no traps or interrupts that may unexpectedly change that value can occur.

---

6. Determine the trap handler address by reading the memory location at the trap vector address, which is the value of the TRAPBASE register.

   For information on how the address is formed, see *Trap Handler Address Register (TRAPBASE)* on page 12.

7. Initialize the PC to the address of the trap handler and continue execution from that address.

FIGURE 4-2 illustrates how a trap is invoked.

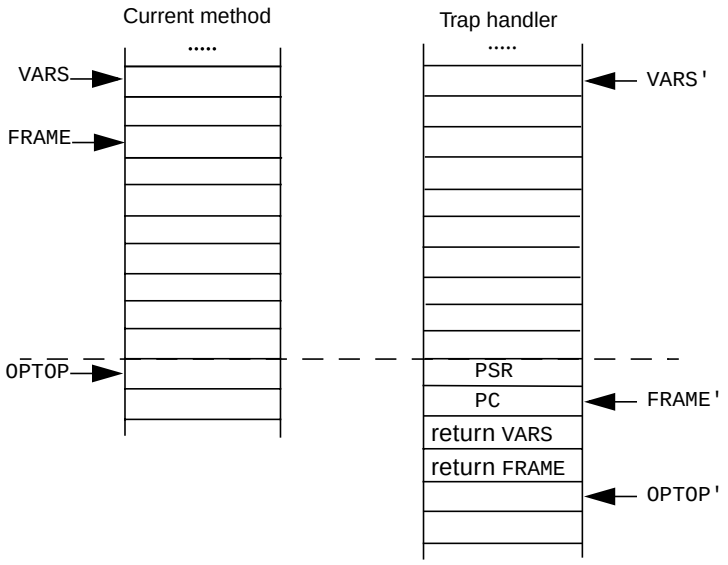


FIGURE 4-2 Invocation of a Trap

While taking a trap, VARS does not change. Instead, OPTOP moves by four words to accommodate the trap frame. The trap handler can set VARS and OPTOP to any value: For example, it can change OPTOP to allocate space for temporaries to be used during trap execution.

A trap frame is different from a normal method-call frame: It does not contain the constant pool or method vector words but includes one word for the PSR. As a result, a trap handler must return from a trap handler using the `priv_ret_from_trap` instruction.

When the trap handler completes execution, it may need to return to the trapped instruction or to the instruction that follows it. If the latter, the trap handler must calculate the return address and store it in the stack at the address in FRAME so that `priv_ret_from_trap` causes a branch to the intended instruction.

FIGURE 4-3 illustrates a return from a trap because of `priv_ret_from_trap`.

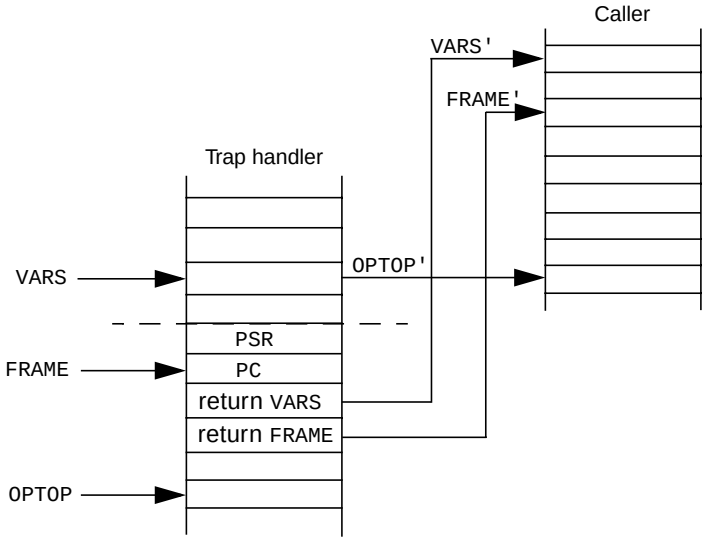


FIGURE 4-3 Return from a Trap

`priv_ret_from_trap` performs these steps:

1. Update `OPTOP` with the current value of `VARS`.
2. Restore `PC` from the trap frame (`FRAME` + 0).
3. Restore `VARS` from the trap frame (`FRAME` – 4).
4. Restore `PSR` from the trap frame (`FRAME` + 4).
5. Restore `FRAME` from the trap frame (`FRAME` – 8).

Since `OPTOP` is set to be equal to the value of `VARS` upon a `priv_ret_from_trap`, the trap handler must set up `VARS` prior to returning from the trap. Any updates to `VARS` must take into account the limitations introduced by stack chunking (see *Manual Updates of the VARS Register* on page 431).

4.1.3 Trap Types and Priorities

TABLE 4-1 lists the types of picoJava traps in order of priority. Recall that 1 is the highest priority.

TABLE 4-1 Types and Priorities of Traps

Trap	Priority	Trap Type
asynchronous_error	1	0x01
mem_protection_error	2	0x02
breakpoint1	3	0x07
breakpoint2	4	0x08
instruction_access_error	5	0x04
illegal_instruction	6	0x06
privileged_instruction	7	0x05
oplim_trap	8	0x0c
mem_address_not_aligned	9	0x09
data_access_mem_error	10	0x03
data_access_io_error	11	0x0a
fadd[1]	12	0x62
dadd[1]	12	0x63
fsub[1]	12	0x66
dsub[1]	12	0x67
fmul[1]	12	0x6a
dmul[1]	12	0x6b
fdiv[1]	12	0x6e
ddiv[1]	12	0x6f
frem[1]	12	0x72
drem[2]	12	0x73
i2f[1]	12	0x86
i2d[1]	12	0x87
l2f[1]	12	0x89
l2d[1]	12	0x8a
f2i[1]	12	0x8b

TABLE 4-1 Types and Priorities of Traps *(Continued)*

Trap	Priority	Trap Type
f2l[1]	12	0x8c
d2i[1]	12	0x8e
d2l[1]	12	0x8f
f2d[1]	12	0x8d
d2f[1]	12	0x90
fcmpg[1]	12	0x96
fcmpl[1]	12	0x95
dcmpg[1]	12	0x98
dcmpl[1]	12	0x97
soft_trap	12	0x0d
ldiv	12	0x6d
lmul	12	0x69
lrem	12	0x71
ldc	12	0x12
ldc_w	12	0x13
ldc2_w	12	0x14
getstatic	12	0xb2
putstatic	12	0xb3
getfield	12	0xb4
putfield	12	0xb5
new	12	0xbb
newarray	12	0xbc
anewarray	12	0xbd
checkcast	12	0xc0
instanceof	12	0xc1
multianewarray	12	0xc5
new_quick	12	0xdd
anewarray_quick	12	0xde
checkcast_quick	12	0xe0
instanceof_quick	12	0xe1

TABLE 4-1 Types and Priorities of Traps *(Continued)*

Trap	Priority	Trap Type
multianewarray_quick	12	0xdf
invokevirtual	12	0xb6
invokespecial	12	0xb7
invokestatic	12	0xb8
invokeinterface	12	0xb9
invokeinterface_quick	12	0xda
putfield_quick_w	12	0xe4
getfield_quick_w	12	0xe3
aastore	15	0x53
athrow	12	0xbf
breakpoint	12	0xca
lookupswitch	12	0xab
wide	12	0xc4
zero_line[3]	12	0x29
unimplemented_instr_0xba	12	0xba
unimplemented_instr_0xdb	12	0xdb
unimplemented_instr_0xf7	12	0xf7
unimplemented_instr_0xf8	12	0xf8
unimplemented_instr_0xf9	12	0xf9
unimplemented_instr_0xfa	12	0xfa
unimplemented_instr_0xfb	12	0xfb
unimplemented_instr_0xfc	12	0xfc
unimplemented_instr_0xfd	12	0xfd
unimplemented_instr_0xfe	12	0xfe
ArithmeticException	13	0x16
ArrayIndexOutOfBounds	13	0x19
NullPointer	13	0x1B
LockCountOverflow	13	0x23
LockEnterMiss	13	0x24
LockRelease	13	0x25

TABLE 4-1 Types and Priorities of Traps *(Continued)*

Trap	Priority	Trap Type
LockExitMiss	13	0x26
gc_notify	14	0x27
nmi	15	0x30
Interrupt_level_15	16	0x3f
Interrupt_level_14	17	0x3e
Interrupt_level_13	18	0x3d
Interrupt_level_12	19	0x3c
Interrupt_level_11	20	0x3b
Interrupt_level_10	21	0x3a
Interrupt_level_9	22	0x39
Interrupt_level_8	23	0x38
Interrupt_level_7	24	0x37
Interrupt_level_6	25	0x36
Interrupt_level_5	26	0x35
Interrupt_level_4	27	0x34
Interrupt_level_3	28	0x33
Interrupt_level_2	29	0x32
Interrupt_level_1	30	0x31
Implementation-dependent (reserved)	Depends on implementation	All trap types not listed above

1. Applies only if PSR.FPE is clear.
2. Applies only if PSR.FPE is clear or PSR.DRT is set.
3. Applies only if PSR.DCE is clear or address is noncacheable.

Note – Undefined trap type values are reserved for future use.

Note – Some unimplemented_instr_0x*XX* traps may not be part of future picoJava cores. However, a picoJava-II system can remain compatible by emulating new instructions that are defined to use those opcodes in the appropriate trap handler.

4.2 Instruction Emulation

Some Java virtual machine instructions in Chapter 6, *Instruction Set* are emulated in software through trap handlers.

When the core encounters an instruction that must be emulated, it generates a trap with a trap type corresponding to that instruction, then jumps to an emulation trap handler that emulates the instruction in software. For these instructions, the trap table is defined so that the trap types are the same as the instruction opcodes.

Certain optimizations in the core can reduce the overhead of subsequent executions of an emulated instruction. For example, when an `invokestatic` instruction is emulated, the trap handler can replace the instruction with an equivalent `invokestatic_quick` version after resolution of the constant pool. The `_quick` version is much faster because it is executed in hardware in subsequent executions of that instruction.

Floating-point instructions trigger traps in the processor if either `HCR.FPP` = 0 (the FPU is not included in the core) or `PSR.FPE` = 0 (the FPU is disabled). In addition, a `drem` instruction traps if `PSR.DRT` = 1 even if `PSR.FPE` = 1 and `HCR.FPP` = 1. The `fneg` and `dneg` instructions are considered integer operations, not floating-point operations, even though they operate on floating-point data.

The `zero_line` instruction triggers an emulation trap in the event that `PSR.DCE` = 0 (the data cache is disabled or not present) or the address specified is noncacheable. The trap handler can zero the specified memory locations and continue execution.

4.3 Exceptions

Another cause for traps are exceptional events that result from the execution of certain instructions. The core detects the following exceptions, causing the corresponding trap:

- `asynchronous_error` — The core takes this trap after an error acknowledgment is returned to the core as a result of a store. The point at which this exception is taken has no relationship to the instruction that caused the memory transaction that triggered the exception.

  The core also takes an asynchronous error trap if the `PSR.CAC` bit is set to 1 and a stack access occurs outside the user memory address ranges.

- mem_protection_error — The core takes this trap if a memory access contains an address that is out of the range of both USERRANGE registers. See *Memory Protection* on page 27 for details about memory protection.
- breakpoint1 and breakpoint2 — The core takes these traps if it detects a breakpoint as specified by the BRK12C, BRK1A, and BRK2A registers. See *Breakpoint Registers* on page 15 for details about breakpoint functionality.
- instruction_access_error — The core takes this trap when an error acknowledgment is returned to the core on an instruction access to memory or I/O. Such an instruction byte is marked as invalid, and once execution reaches the invalid instruction, an instruction access error occurs.

  Some instructions (for example, the tableswitch instructions and instructions emulated in software via traps) access their own operands as data. If an error occurs during such an access, then the core flags it as a data access error even if the access is to instruction space.
- privileged_instruction — The core takes this trap if it attempts to execute a privileged instruction when PSR.SU = 0.
- oplim_trap — The core takes this trap if an instruction causes OPTOP to be less than OPLIM. The trap handler must either grow the current stack area or allocate a new "chunk." See Chapter 10, *Stack Chunking* for more information.

Note – The core clears the OPLIM.ENABLE bit as soon as the trap is taken to ensure that repeated traps do not occur.

- mem_address_not_aligned — The core takes this trap if a load or store instruction in TABLE 4-2 generates an address that is not properly aligned; that is, a word address is not 32-bit aligned or a short address is not 16-bit aligned.

TABLE 4-2 Instructions Subject to Memory Alignment Trap Checks

load_char	load_word_index	ncload_word	store_short_index
load_char_index	load_word_oe	ncload_word_oe	store_short_oe
load_char_oe	nastore_word_index	ncstore_short	store_word
load_short	ncload_char	ncstore_short_oe	store_word_index
load_short_index	ncload_char_oe	ncstore_word	store_word_oe
load_short_oe	ncload_short	ncstore_word_oe	
load_word	ncload_short_oe	store_short	

If a misaligned access occurs for other reasons, the instruction completes without an exception but returns an undefined value.

- `data_access_mem_error` and `data_access_io_error` — The core takes these traps when a data request to memory or I/O space results in an error acknowledgment. The external system determines whether a given access is to memory or I/O space and, subsequently, which error acknowledgment code to return to the core.
- `illegal_instruction` — The core takes this trap if it executes an opcode that is not a valid instruction. Only 2-byte opcodes (first instruction byte = 0xff) can trigger this exception because all other bytecodes are either valid instructions or cause a unique type of trap (`unimplemented_instruction_0x`*opcode*).
- `ArithmeticException` — The core takes this trap if the hardware attempts to execute an integer division or remainder operation with a denominator of zero.
- `ArrayIndexOutOfBounds` — The core takes this trap if an array load or store instruction accesses an element that is outside the legal bounds for the array.
- `NullPointer` — The core takes this trap if such instructions as `getfield_quick` and `invokevirtual_quick` (which expect a non-null object reference) encounter a null object reference (0x00000000).
- `LockCountOverflow` — The core takes this trap if the `LOCKCOUNT` register overflows or underflows when the core increments or decrements it while entering or exiting a monitor. See Chapter 8, *Monitors*.
- `LockEnterMiss` — The core takes this trap if the object reference for the `monitorenter` instruction is not present in any of the `LOCKADDR` registers. See Chapter 8, *Monitors*.
- `LockExitMiss` — The core takes this trap if the monitor being exited is absent in a `LOCKADDR` register. See Chapter 8, *Monitors*.
- `LockRelease` — The core takes this trap if `LOCKCOUNT` equals zero and the corresponding `LOCKWANT` bit is set. See Chapter 8, *Monitors*.
- `gc_notify` — The core takes this trap when garbage collection events occur. See Chapter 11, *Garbage Collection*, for details.

4.4 Interrupts

Traps are also caused by interrupts that are signalled by external devices. The core can receive two types of interrupt signals, as follows:

- **The nonmaskable interrupt (NMI)** — A single bit that is asserted when a high-priority interrupt is required
- **The maskable interrupt** — A signal that can be asserted with 15 different priorities

4.4.1 Interrupt Control

The values of the Interrupt Enable (IE) and the Processor Interrupt Level (PIL) bits in the PSR determine if an interrupt causes a trap.

A trap occurs when an interrupt is signalled by an external device only if both of the following conditions are met:

- The PSR.IE bit is set to 1.

  When this bit is set to 0, the core ignores all interrupts, including the NMI.
- Either the interrupt request level (IRL) of the external interrupt is greater than the value in the PSR.PIL field or the NMI signal is asserted.

FIGURE 4-4 illustrates the mechanism for interrupt control.

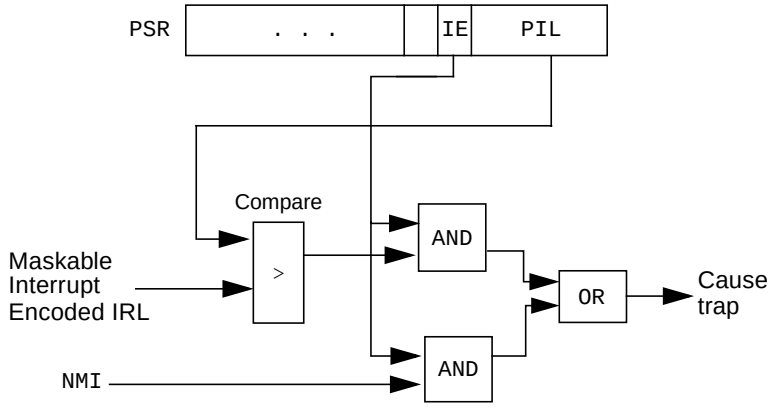


FIGURE 4-4 Interrupt Control Mechanism

When the core takes a trap, it sets the PSR.IE bit to 0, disabling further interrupts. You can reenable interrupts by setting PSR.IE to 1 in the trap handler after it has completed any critical and uninterruptible tasks. You can also set the PSR.PIL field to the current interrupt level to enable only higher-priority interrupts.

Details of interrupt servicing are a function of the software-defined interrupt service routines.

4.4.2 Interrupt Latency

In the best case, assuming that an interrupt occurs at an instruction boundary, the latency from the arrival of an interrupt to when the interrupt handler starts execution is approximately six cycles.

The worst-case interrupt latency, assuming that PSR.DRT is set and that a cache line fill or writeback takes 30 core clocks, can be calculated as shown in TABLE 4-3.

TABLE 4-3 Calculation of Worst-Case Interrupt Latency

$IL_{worst\ case}$ =	
<200	Longest uninterruptible instruction (frem)
+(6 × 30)	Cache misses during its execution, causing 6 writebacks
+(6 × 30)	Time to flush out the stack cache—worst case, 16 writebacks
+ ~6	Time to set up a trap frame
+(1 × 30)	Data cache miss, causing writeback, during trap frame setup
+30	Time to get to the reset handler, with instruction cache miss
926 clocks	

Note – If PSR.IE = 0, then the interrupt latency is extended by the time PSR.IE remains 0.

4.5 Context Switch

Once control has transferred to a trap or interrupt handler routine, you may wish to switch execution contexts. For example, a LockRelease trap will want to start executing the thread that was awaiting the release of the lock.

It is often convenient to use the stack of the outgoing thread to store relevant context state, restoring the data from the stack of the incoming execution context.

CODE EXAMPLE 4-1 is an example of context switch code, assuming:

- GLOBAL and USERRANGE registers are part of each context.
- GC_CONFIG, TRAPBASE, and breakpoints are global.
- GLOBAL3 holds a pointer to the address for the current thread data structure.
- The trap into the context switch routine saves the PC and PSR.

CODE EXAMPLE 4-1 Sample Context Switch Code

```
 read_vars     // Push state registers onto the stack.
  read_frame
   read_const_pool
    read_global0
     read_global1
      read_global2
       read_userrange1
        read_userrange2

// Monitor handling code occurs here if the LOCKADDR and LOCKCOUNT
// registers are used. See Chapter 8, Monitors.

         read_optop// Save OPTOP and OPLIM into the thread.
          read_global3 // Data structure pointed to by global3.
           bipush optop_offset
          iadd
        store_word
         read_oplim
          read_global3
           bipush oplim_offset
          iadd
        store_word

// Determine the next thread to run and put a pointer to its
// data structure in GLOBAL3. How this selection is made depends
// on the kernel and is not shown here.

         read_global3 // Now we have new thread; load new OPLIM.
          bipush oplim_offset
         iadd
         load_word
          read_global3// Also, load new OPTOP.
           bipush optop_offset
          iadd
          load_word
        update_optop// Update both registers.

// Once OPTOP and OPLIM have changed, the rest of the state
// is on the new stack and must be restored.

         iconst_0
        write_lockaddr1 // Zero lockaddr1 contents.
         iconst_0
        write_lockaddr0 // Zero lockaddr0 contents.
```

CODE EXAMPLE 4-1 Sample Context Switch Code *(Continued)*

```
        write_userrange2 // Restore the remaining state registers.
       write_userrange1
      write_global2
     write_global1
    write_global0
   write_const_pool
  write_frame
 write_vars
 ret_from_trap  // Restore the program counter
                // and processor state registers.
```

CHAPTER 5

Data Types and Runtime Data Structures

The picoJava-II data types fall into two categories: primitive types and reference types. This chapter describes the different types and runtime data structures that are used by the core:

- *Primitive Data Types* on page 61
- *Reference Types and Values* on page 62
- *Essential Runtime Data Structures* on page 71

See Chapter 6, *Instruction Set*, for details on specific instructions.

5.1 Primitive Data Types

The picoJava-II architecture supports all primitive data types of the Java virtual machine with the addition of an unsigned byte type.

TABLE 5-1 lists and describes the primitive data types.

TABLE 5-1 Primitive Data Types

Data Type	Description
Unsigned byte	8-bit unsigned integers
Byte	8-bit signed two's-complement integers
Char	16-bit unsigned integers
Short	16-bit signed two's-complement integers
Integer	32-bit signed two's-complement integers

TABLE 5-1 Primitive Data Types *(Continued)*

Data Type	Description
Long	64-bit signed two's-complement integers
Float	32-bit single-precision IEEE 754 floating-point numbers
Double	64-bit double-precision IEEE 754 floating-point numbers

Note – Items of type long or double (64-bit values) occupy two 32-bit stack entries and are arranged such that the *most significant* 32 bits of each 64-bit value are the stack entry *closest* to the top of the stack.

5.1.1 Integral Data Types

The range of values for the unsigned byte type is from 0 to 255 inclusive. All remaining integral types of the core have the same values as those of the integral types of the Java virtual machine. For details, see section 3.3, "Primitive Types and Values" of *The Java Virtual Machine Specification*.

5.1.2 Floating-Point Data Types

The Java virtual machine floating-point specification requires implementations to support denormalized floating-point numbers and gradual underflow, as defined by IEEE 754.

For details on the float and double data types, see section 3.2.2 of *The Java Virtual Machine Specification*.

5.2 Reference Types and Values

The core has the same reference types as the Java virtual machine, as follows:

- Object references are references to instances of a class or references to class instances or arrays that implement an interface.
- Array references are references to arrays or primitive types or arrays of references.

A null reference has the value 0x00000000.

5.2.1 References and Headers

A reference is a pointer to storage that represents the object or array being targeted for the current instruction. A Java compiler generates an instruction to push this pointer onto the operand stack before generating the instruction to operate on it. For details of the various instructions that operate on references, see Chapter 6.

The following table describes the 4 reserved bits in the reference, all of which are masked out before the reference is used as an address and in comparison instructions, such as `if_acmpeq` and `if_acmpne`.

`GC_TAG`	These 2 bits form an index into the `GC_CONFIG.WB_VECTOR` field to determine whether to signal a write barrier GC trap. See Chapter 11.
`H`	This handle bit indicates if the object is referenced directly or indirectly through a handle. If set to 0, this bit indicates a direct reference; a value of 1 indicates an indirect reference.
X	This bit can be used by software for various purposes, for example, to indicate whether the object is an array type.

Figure 5-1 shows the format of a reference.

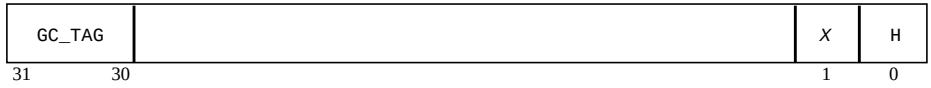


FIGURE 5-1 Object or Array Reference Format

Note – When using a reference directly as an address, you must mask off at least at bit 30 of the object reference in software because this bit also indicates little endian accesses for these instructions. We strongly recommend that you also mask off bits 31 and 1:0.

The word-aligned address that is obtained from masking out the `GC_TAG`, `H`, and *X* bits from a reference points to the header for the object or array.

The header is one 32-bit word that contains the method vector base of the correct class. Four bits are reserved for such information as garbage collection and synchronization. Bit 0 is reserved as the `LOCK` bit (see *Monitors* on page 395 for more details).

The reserved bits in the header are masked off to obtain the method vector base address. A reference always points to the location of the header, as in FIGURE 5-2.

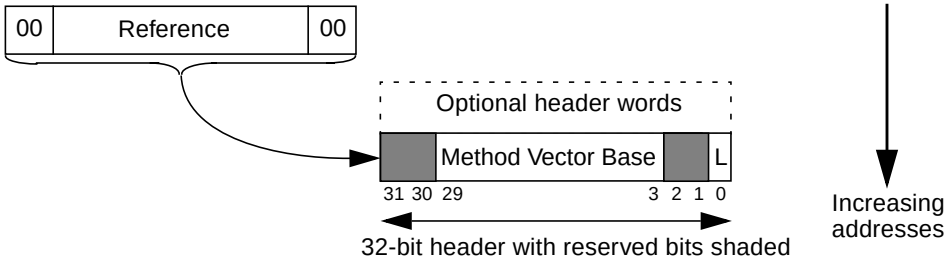


FIGURE 5-2 Object or Array Header Field with Reserved Bits

Note – The class loader must ensure that the method vector to which the object header points is double-word aligned.

You can design your implementation of the Java virtual machine to maintain additional header words at contiguous lower addresses to hold other data.

5.2.2 Object Storage

If the handle (H) bit in the object reference is not set, the instance variable storage starts one word after the object header and contains the primitive data or reference types allocated for the object. See FIGURE 5-3.

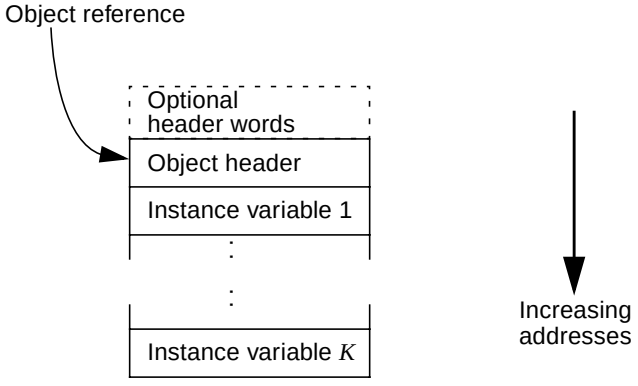


FIGURE 5-3 Object Format with Handle Bit Clear

If the H bit in an object reference is set, references must then go through the handle to access the object, as illustrated in FIGURE 5-4.

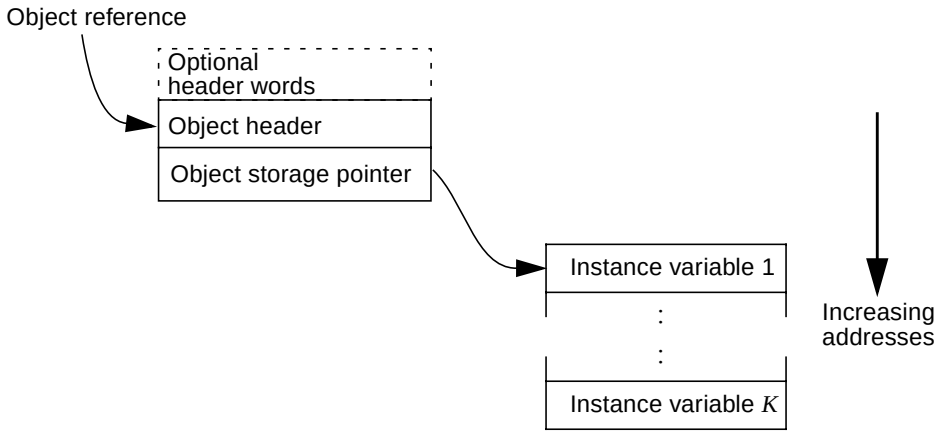


FIGURE 5-4 Object Format with Handle Bit Set

Instance variables of type double or long require two words of storage, with the most significant word at the lower address in memory. All other instance variable types require only one word of storage.

5.2.3 Array Storage

The array format is similar to the object format, except that array size information is stored into the first word of the data storage. You can use some of the reserved bits in the array header to encode the size of each element. FIGURE 5-5 illustrates the array format.

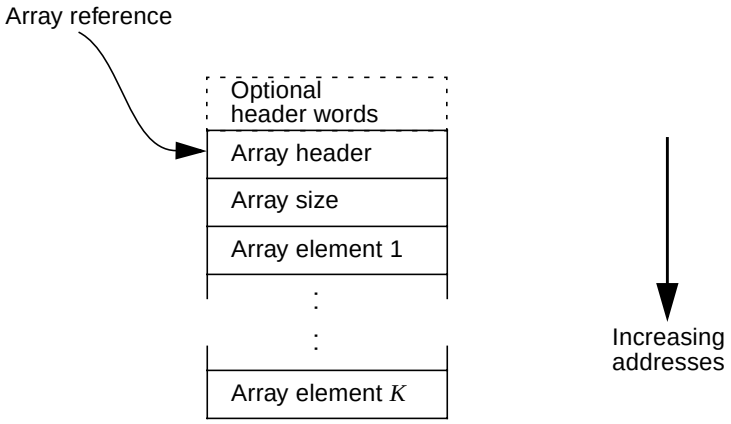


FIGURE 5-5 Array Format with Handle Bit Clear

If the H bit in the array reference is set, references must then go through the handle to access the array, as illustrated in FIGURE 5-6.

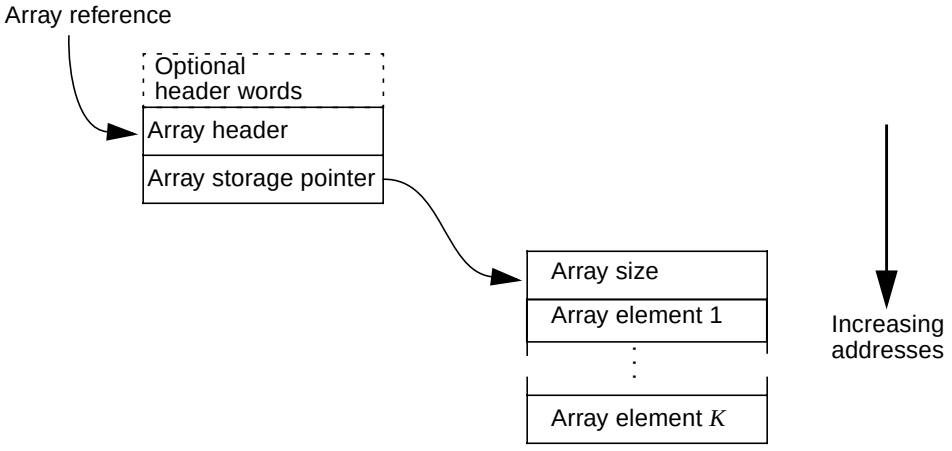


FIGURE 5-6 Array Format with Handle Bit Set

5.2.4 Layout of Array Data Structures

This section describes the layout of array data structures for each array type. The examples show the scenario where the array reference is *not* a handle. The array header is always allocated word-aligned; however, short/char (16 bits) and byte/boolean (8 bits) types are packed.

Array of Longs Structure

FIGURE 5-7 defines the structure of an array of longs accessed by the `laload` and `lastore` instructions.

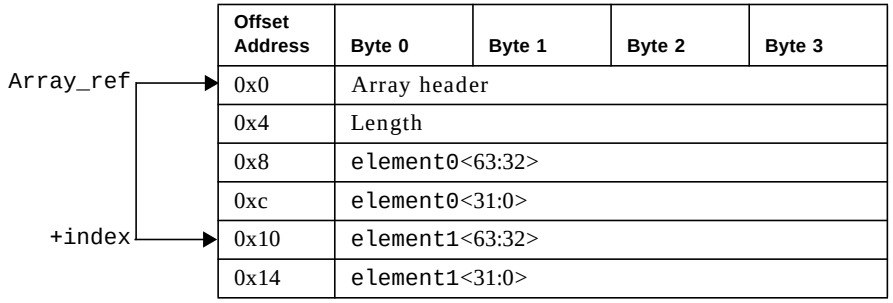


FIGURE 5-7 Array of Longs Structure

Array of Doubles Structure

FIGURE 5-8 defines the structure of an array of longs accessed by the `daload` and `dastore` instructions.

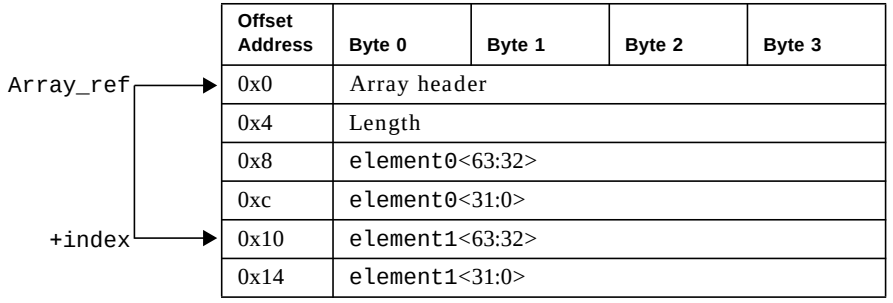


FIGURE 5-8 Array of Doubles Structure

Array of Objects Structure

FIGURE 5-9 defines the structure of an array of objects accessed by the `aaload` and `aastore` instructions.

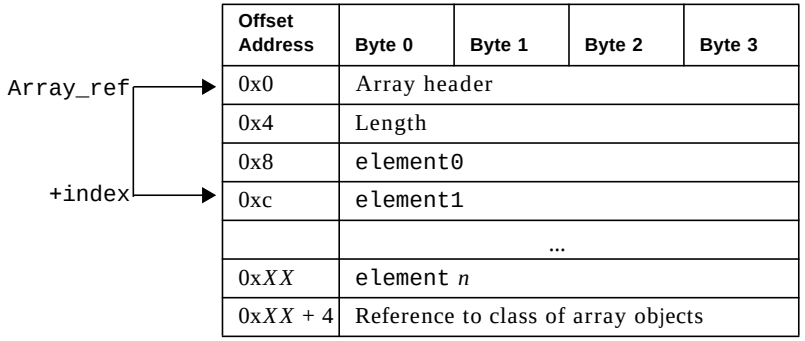


FIGURE 5-9 Array of Objects Structure

Array of Arrays Structure

FIGURE 5-10 defines the structure of an array of arrays accessed by the `aaload` and `aastore` instructions.

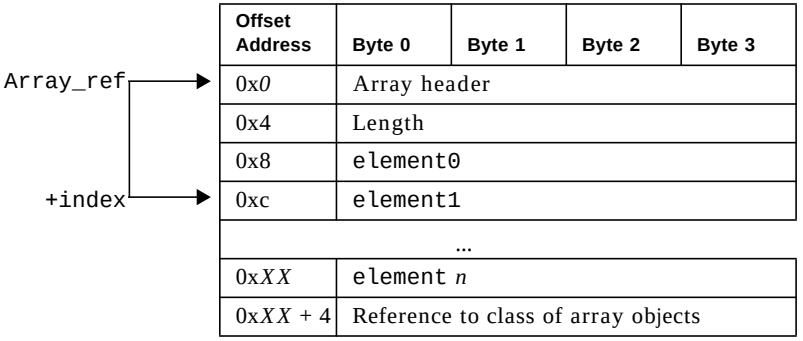


FIGURE 5-10 Array of Arrays Structure

Array of Integers Structure

FIGURE 5-11 defines the structure of an array of integers accessed by the `iaload` and `iastore` instructions.

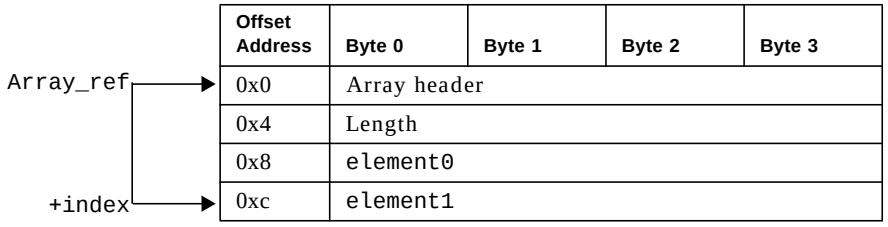


FIGURE 5-11 Array of Integers Structure

Array of Floats Structure

FIGURE 5-12 defines the structure of an array of floats accessed by the `faload` and `fastore` instructions.

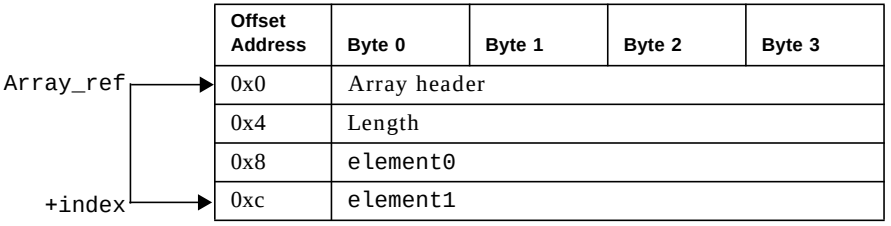


FIGURE 5-12 Array of Floats Structure

Array of Chars Structure

FIGURE 5-13 defines the structure of an array of chars accessed by the `caload` and `castore` instructions.

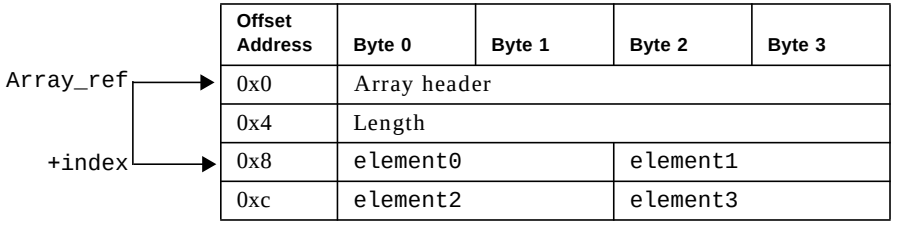


FIGURE 5-13 Array of Chars Structure

Array of Shorts Structure

FIGURE 5-14 defines the structure of an array of shorts accessed by the `saload` and `sastore` instructions.

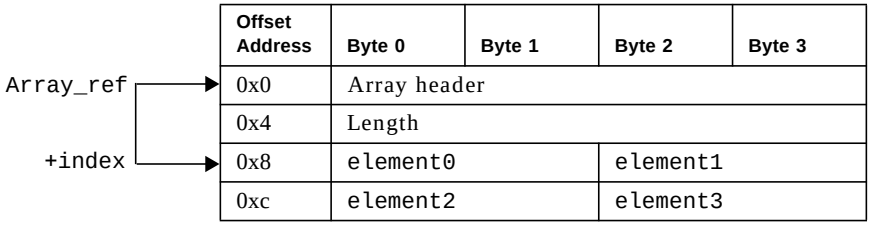


FIGURE 5-14 Array of Shorts Structure

Array of Bytes Structure

FIGURE 5-15 defines the structure of an array of bytes accessed by the `baload` and `bastore` instructions.

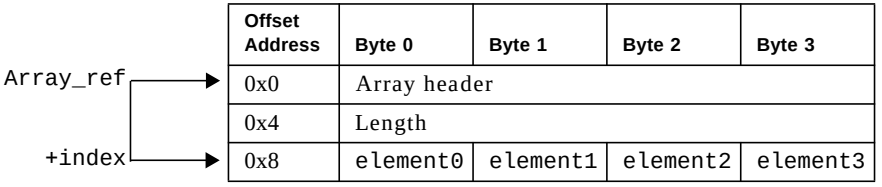


FIGURE 5-15 Array of Bytes Structure

Array of Booleans Structure

FIGURE 5-16 defines the structure of an array of booleans accessed by the `baload` and `bastore` instructions.

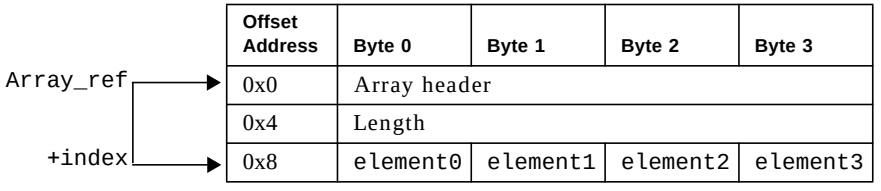


FIGURE 5-16 Array of Booleans Structure

5.3 Essential Runtime Data Structures

This section discusses the essential runtime data structures that are required by some instructions. Many of these structures contain fields unused by instructions implemented in hardware. You can use these fields in software.

5.3.1 Method Vector and Runtime Class Info Structure

The method vector base in an object or array header points to the base of a table of method structure pointers, which can be invoked on a reference, and provides a mechanism for overriding methods in superclasses.

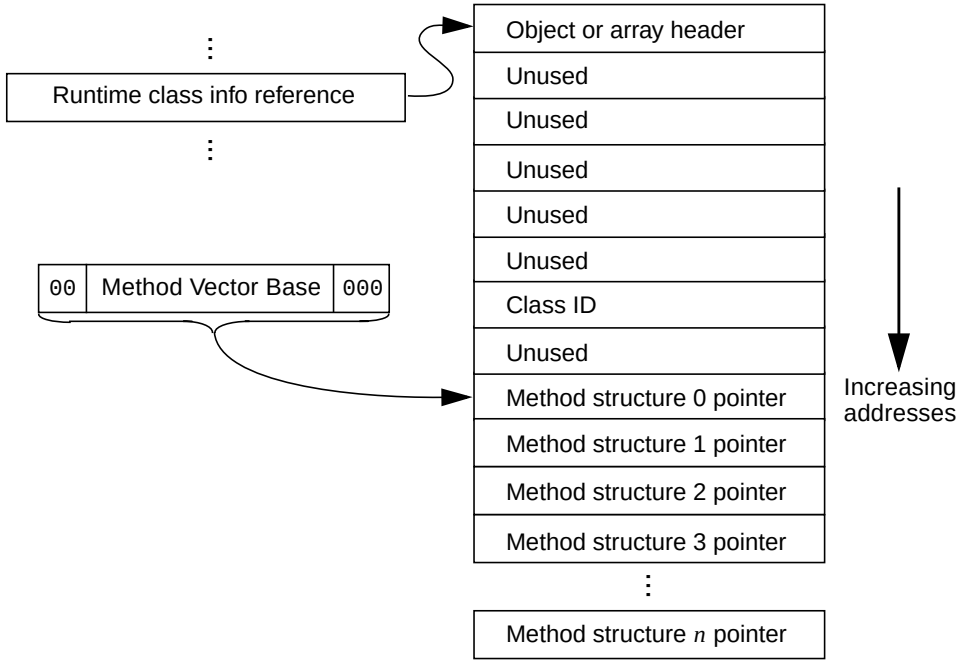


FIGURE 5-17 Runtime Class Info Structure with Method Vector

The method vector is a part of the larger runtime class information structure. No references to this structure can have the `H` bit set. The hardware ignores the unused fields, which are required to maintain the relative offsets of the other fields. The `checkcast_quick` and `instanceof_quick` instructions use the Class ID field, a unique 32-bit identifier for the class associated with this structure.

5.3.2 Method Structure

The hardware expects each method pointer to point to a method structure, as shown in FIGURE 5-18, and ignores the unused fields, which maintain the relative offsets of the other fields. The fields are 32 bits long, unless otherwise specified.

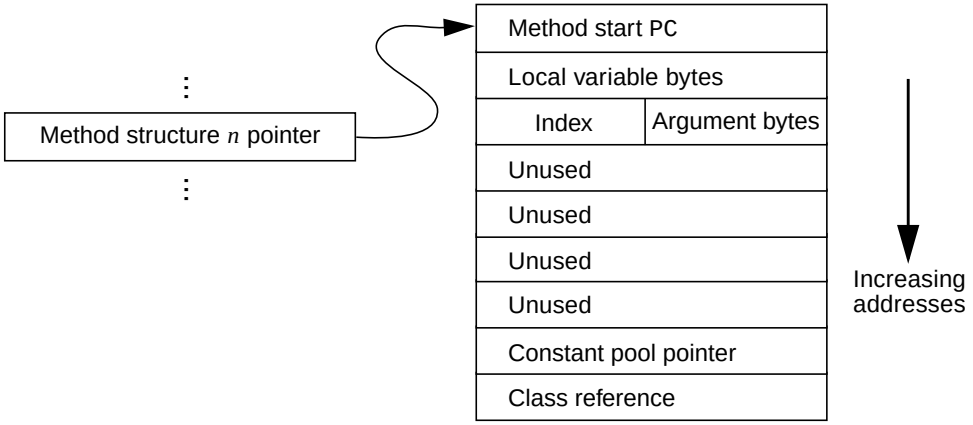


FIGURE 5-18 Method Structure

TABLE 5-2 describes the fields the core uses.

TABLE 5-2 Method Structure Fields

Field	Description
Method start PC	The address of the first instruction in the method
Local variable bytes	Number of local variables in bytes, excluding arguments
Argument bytes	16-bit value for the number of argument bytes
Index	16-bit index into the array header table
Constant pool pointer	Pointer to the constant pool table (see *Constant Pool* on page 73)
Class reference	Reference to the class structure of the method's class; the handle bit must be 0.

5.3.3 Class Structure

The class structure holds the data that describes a class that has been loaded into memory. No references to this structure can have the handle bit set. Hence, following the header for the object is the instance variable storage for the object, as shown in FIGURE 5-19.

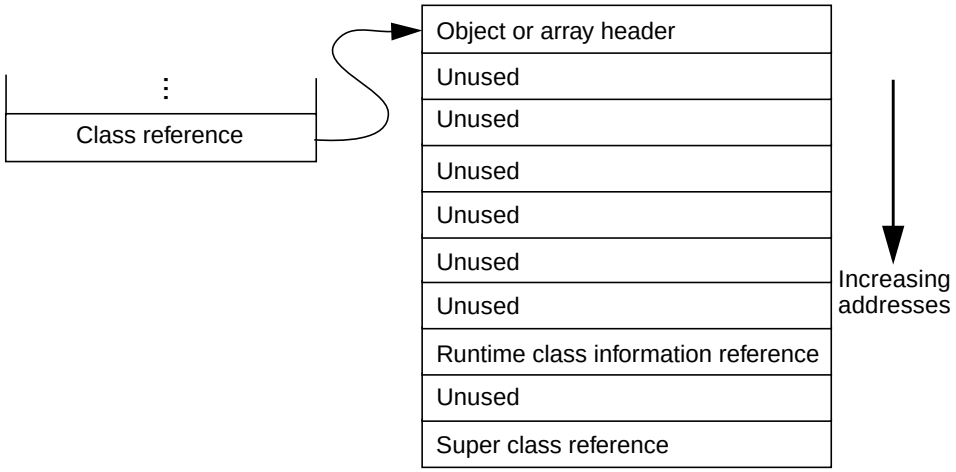


FIGURE 5-19 Class Structure

The hardware ignores the unused instance variable fields, which maintain the relative offsets of the other fields.

The core uses two fields in the class structure:

- A reference to the runtime class information structure that corresponds to this class (see *Method Vector and Runtime Class Info Structure* on page 71)
- A reference to the class structure of the superclass of the current class

Neither of these references can have the H bit set.

5.3.4 Constant Pool

The constant pool provides an indexed mapping mechanism for the compiled Java methods in a class file. Every class has an associated constant pool table, which stores the mapping information for the indexed references embedded in the class's methods.

The size of each element in the constant pool table is 32 bits. The first element of the constant pool is not used and can point to an array that contains the type of each subsequent element. (See section 4.4 in *The Java Virtual Machine Specification* for details on constant pool tags). All other elements contain information specific to the type of element as defined by the class file format.

During program execution, the Java virtual machine resolves certain element types and replaces the original information with new data to be used directly by hardware instructions. See `agetstatic_quick`, `aldc_quick`, `aldc_w_quick`, `aputstatic_quick`, `checkcast_quick`, `getstatic_quick`, `getstatic2_quick`, `instanceof_quick`, `invokenonvirtual_quick`, `invokestatic_quick`, `invokevirtual_quick_w`, `ldc_quick`, `ldc_w_quick`, `ldc2_w_quick`, `putstatic_quick`, and `putstatic2_quick` in Chapter 6 for more details.

FIGURE 5-20 illustrates the relationship between the constant pool pointer and the elements.

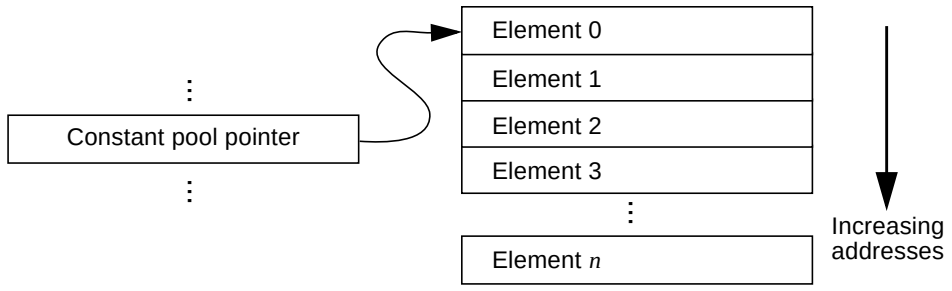


FIGURE 5-20 Constant Pool

CHAPTER 6

Instruction Set

This chapter describes the picoJava-II instruction set. The descriptions in the chapter pertain to hardware functionalities only. For additional semantics that are required for a Java virtual machine implementation, see *The Java Virtual Machine Specification*.

Every instruction description provides pseudocode detailing its operation. The operations used within the pseudocode are described in the tables below.

TABLE 6-1 Basic Operations

Operation	Description
value1 + *value2*	Add *value1* and *value2*.
value1 - *value2*	Subtract *value2* from *value1*.
value1 × *value2*	Multiply *value1* by *value2*.
value1 ÷ *value2*	Divide *value1* by *value2*.
value1 & *value2*	Bitwise AND of *value1* and *value2*.
value1 | *value2*	Bitwise OR of *value1* and *value2*.
value1 ^ *value2*	Bitwise XOR of *value1* and *value2*.
value1 << *value2*	Left-shift of *value1* by *value2* bits.
value1 >> *value2*	Right-shift of *value1* by *value2* bits.
value1 >>> *value2*	Unsigned right-shift of *value1* by *value2* bits.

TABLE 6-2 Type Casts and Conversions

Operation	Description
`double(`*msw, lsw*`)`	Treat the two 32-bit words, *msw* and *lsw*, as the most significant and least significant words of a 64-bit double-precision floating-point value. Operations from TABLE 6-1 are treated as double-precision floating-point operations if their operands are treated as doubles.
`float(`*value*`)`	Treat the 32-bit word, *value*, as a 32-bit single precision floating-point value. Operations from TABLE 6-1 are treated as single-precision floating-point operations if their operands are treated as floats.
`long(`*msw, lsw*`)`	Treat the two 32-bit words, *msw* and *lsw*, as the most significant and least significant words of a 64-bit long integer. Operations from TABLE 6-1 are treated as 64-bit integer operations if their operands are treated as longs.
$\texttt{convert}_{\texttt{d2f}}$`(`*value*`)`	Convert the double-precision floating-point *value* to a single-precision floating-point result.
$\texttt{convert}_{\texttt{d2i}}$`(`*value*`)`	Convert the double-precision floating-point *value* to a 32-bit integer result.
$\texttt{convert}_{\texttt{d2l}}$`(`*value*`)`	Convert the double precision floating-point *value* to a 64-bit integer result.
$\texttt{convert}_{\texttt{f2d}}$`(`*value*`)`	Convert the single-precision floating-point *value* to a double-precision floating-point result.
$\texttt{convert}_{\texttt{f2i}}$`(`*value*`)`	Convert the single-precision floating-point *value* to a 32-bit integer result.
$\texttt{convert}_{\texttt{f2l}}$`(`*value*`)`	Convert the single-precision floating-point *value* to a 64-bit integer result.
$\texttt{convert}_{\texttt{i2d}}$`(`*value*`)`	Convert the 32-bit integer *value* to a double-precision floating-point result.
$\texttt{convert}_{\texttt{i2f}}$`(`*value*`)`	Convert the 32-bit integer *value* to a single-precision floating-point result.
$\texttt{convert}_{\texttt{l2d}}$`(`*value*`)`	Convert the 64-bit integer *value* to a double-precision floating-point result.
$\texttt{convert}_{\texttt{l2f}}$`(`*value*`)`	Convert the 64-bit integer *value* to a single-precision floating-point result.

TABLE 6-3 Cache Tag Accesses

Operation	Description
⇐ `icache_data[`*addr*`]`	Read the instruction cache data array, using address as described by `priv_read_icache_data`.
`icache_data[`*addr*`]` ⇐ *data*	Write the instruction cache data array, using address as described by `priv_write_icache_data`.
⇐ `icache_tag[`*addr*`]`	Read the instruction cache tag array, using address as described by `priv_read_icache_tag`.

TABLE 6-3 Cache Tag Accesses *(Continued)*

Operation	Description
icache_tag[*addr*] ⇐ *data*	Write the instruction cache tag array, using address as described by priv_write_icache_tag.
⇐ dcache_data[*addr*]	Read the data cache data array, using address as described by priv_read_dcache_data.
dcache_data[*addr*] ⇐ *data*	Write the data cache data array, using address as described by priv_write_dcache_data.
⇐ dcache_tag[*addr*]	Read the data cache tag array, using address as described by priv_read_dcache_tag.
dcache_tag[*addr*] ⇐ *data*	Write the data cache tag array, using address as described by priv_write_dcache_tag.

TABLE 6-4 Memory Access-Related Operations

Operation	Description
addr_out_of_range(*addr*)	Check whether an address is outside of both USERRANGE regions.

	masked_addr ⇐ *addr* & 0x7fffffff
	low1 ⇐ USERRANGE1.USERLOW << 14
	high1 ⇐ USERRANGE1.USERHIGH << 14
	low2 ⇐ USERRANGE2.USERLOW << 14
	high2 ⇐ USERRANGE2.USERHIGH << 14
	if ((masked_addr ≥ low1) AND (masked_addr < high1)) then
	  return FALSE
	if ((masked_addr ≥ low2) AND (masked_addr < high2)) then
	  return FALSE
	return TRUE

endian_swap(*word*)	Convert a 32-bit word from big-endian byte order to little-endian byte order, or vice versa.
	byte1 ⇐ *word* & 0x000000ff byte2 ⇐ (*word* >> 8) & 0x000000ff byte3 ⇐ (*word* >> 16) & 0x000000ff byte4 ⇐ (*word* >> 24) & 0x000000ff result ⇐ (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4 return result

TABLE 6-4 Memory Access-Related Operations *(Continued)*

Operation	Description
endian_swap$_{16}$(*word*)	Convert the low 16-bits of a 32-bit word from big-endian byte order to little-endian byte order, or vice versa.
	byte1 ⇐ *word* & 0x000000ff byte2 ⇐ (*word* >> 8) & 0x000000ff result ⇐ (byte1 << 8) | byte2 return result
sign_ext$_{16}$(*word*)	Sign-extend a 16-bit value to a 32-bit word.

	sign ⇐ *word* & 0x00008000
	if (sign = 0) then
	result ⇐ *word* & 0x0000ffff
	else
	result ⇐ *word* | 0xffff0000
	return result

sign_ext$_{8}$(*word*)	Sign-extend an 8-bit value to a 32-bit word.

	sign ⇐ *word* & 0x00000080
	if (sign = 0) then
	result ⇐ *word* & 0x000000ff
	else
	result ⇐ *word* | 0xffffff00
	return result

TABLE 6-5 Stack and Memory Access Operations

Operation	Description
⇐ stack[*addr*]	Read a 32-bit stack location.

	if (*addr* > OPTOP) AND (*addr* ≤ SC_BOTTOM) then
	return read_stack_cache(*addr* & 0x0000003f)
	else
	return read_memory(*addr*, cacheable, 32 bits)

stack[*addr*] ⇐ *data*	Write a 32-bit stack location.

	if (*addr* > OPTOP) AND (*addr* ≤ SC_BOTTOM) then
	write_stack_cache((*addr* & 0x0000003f), *data*)
	else
	write_memory(*addr*, *data*, cacheable, 32 bits)

TABLE 6-5 Stack and Memory Access Operations *(Continued)*

Operation	Description
⇐ mem[*addr*]	Read a 32-bit memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x00000003) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`if ((`*addr* `& 0x30000000) = 0x30000000) then`
	`  return read_memory(`*addr*`, noncacheable, 32 bits)`
	`else`
	`  return read_memory(`*addr*`, cacheable, 32 bits)`

mem[*addr*] ⇐ *data*	Write a 32-bit memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x00000003) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`if ((`*addr* `& 0x30000000) = 0x30000000) then`
	`  write_memory(`*addr*`, `*data*`, noncacheable, 32 bits)`
	`else`
	`  write_memory(`*addr*`, `*data*`, cacheable, 32 bits)`

⇐ mem_{NC}[*addr*]	Read a 32-bit noncacheable memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x00000003) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`return read_memory(`*addr*`, noncacheable, 32 bits)`

mem_{NC}[*addr*] ⇐ *data*	Write a 32-bit noncacheable memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x00000003) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`write_memory(`*addr*`, `*data*`, noncacheable, 32 bits`

TABLE 6-5 Stack and Memory Access Operations *(Continued)*

Operation	Description
$mem_{NA}[addr] \Leftarrow data$	Write a 32-bit memory location, nonallocating

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(addr)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((addr & 0x00000003) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`if ((addr & 0x30000000) = 0x30000000) then`
	`  write_memory(addr, data, noncacheable, 32 bits)`
	`else if ((is_present_in_data_cache(addr))`
	`  write_memory(addr, data, cacheable, 32 bits)`
	`else`
	`  write_memory(addr, data, noncacheable, 32 bits)`

$\Leftarrow mem_{16}[addr]$	Read a 16-bit memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(addr)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((addr & 0x00000001) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`if ((addr & 0x30000000) = 0x30000000) then`
	`  return read_memory(addr, noncacheable, 16 bits)`
	`else`
	`  return read_memory(addr, cacheable, 16 bits)`

$mem_{16}[addr] \Leftarrow data$	Write a 16-bit memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(addr)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((addr & 0x00000001) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`if ((addr & 0x30000000) = 0x30000000) then`
	`  write_memory(addr, data, noncacheable, 16 bits)`
	`else`
	`  write_memory(addr, data, cacheable, 16 bits)`

$\Leftarrow mem_{16,NC}[addr]$	Read a 16-bit noncacheable memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(addr)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((addr & 0x00000001) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`return read_memory(addr, noncacheable, 16 bits)`

TABLE 6-5 Stack and Memory Access Operations *(Continued)*

Operation	Description
$mem_{16,NC}[addr] \Leftarrow data$	Write a 16-bit noncacheable memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x00000001) ≠ 0) then`
	`  trap mem_address_not_aligned (type 0x09)`
	`write_memory(`*addr*`, `*data*`, noncacheable, 16 bits)`

$\Leftarrow mem_8[addr]$	Read an 8-bit memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x30000000) = 0x30000000) then`
	`  return read_memory(`*addr*`, noncacheable, 8 bits)`
	`else`
	`  return read_memory(`*addr*`, cacheable, 8 bits)`

$mem_8[addr] \Leftarrow data$	Write an 8-bit memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`if ((`*addr* `& 0x30000000) = 0x30000000) then`
	`  write_memory(`*addr*`, `*data*`, noncacheable, 8 bits)`
	`else`
	`  write_memory(`*addr*`, `*data*`, cacheable, 8 bits)`

$\Leftarrow mem_{8,NC}[addr]$	Read an 8-bit noncacheable memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`return read_memory(`*addr*`, noncacheable, 8 bits)`

$mem_{8,NC}[addr] \Leftarrow data$	Write an 8-bit noncacheable memory location.

	`if ((PSR.ACE = 1) AND (PSR.CAC = 1)) then`
	`  if (addr_out_of_range(`*addr*`)) then`
	`  trap mem_protection_error (type 0x02)`
	`write_memory(`*addr*`, `*data*`, noncacheable, 8 bits)`

Load a reference from an array.

Format

aaload

Forms

aaload = 50 (0x32)

Stack

…, arrayref, index ⇒
…, value

Description

aaload treats the stack entry *arrayref* as a reference to an array of word-sized elements; the stack entry, *index*, is a signed 32-bit integer. It returns the element at *index* of the array.

If *arrayref* is null, then aaload takes a NullPointer trap. If *index* is not within the bounds of the array referenced by *arrayref*, then aaload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ mem[addr_of_length + 4 + (index × 4)]
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, aaload is identical to iaload and faload.

Trap to emulation routine that stores a reference into an array of references.

Format

aastore

Forms

`aastore` = 83 (0x53)

Stack

…, arrayref, index, value ⇒
…

Description

`aastore` traps to the emulation routine referenced by entry 0x53 in the trap table.

Operation

```
trap aastore (type = 0x53)
```

Recommendations

The trap handler should perform the following tasks:

- Emulate `aastore`, as defined in *The Java Virtual Machine Specification*.
- Perform the garbage collection checks that are described in *Write Barriers* on page 434.

Trap to emulation routine that stores a reference into an array of references without type checks.

Format

aastore_quick

Forms

aastore_quick = 220 (0xdc)

Stack

…, arrayref, index, value ⇒
…

Description

aastore_quick treats *arrayref* as a reference to an array of one-word elements. It stores the reference *value* on the stack to the one-word element at *index* of the array.

If *arrayref* is null, then aastore_quick takes a NullPointer trap. If *index* is not within the bounds of the array referenced by *arrayref*, then aastore_quick takes an ArrayIndexOutOfBounds trap. Otherwise, it performs the garbage collection checks described in *Write Barriers* on page 434, possibly generating a gc_notify trap.

Operation

```
arrayref ⇐ stack[OPTOP + 12]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 8]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
gc_index ⇐ ((arrayref & 0xc0000000) >> 28) | (value >> 30)
write_barrier_bit = (GC_CONFIG >> gc_index) & 0x00000001
if (write_barrier_bit = 1) then
            trap gc_notify (type = 0x27)
object_region = (arrayref >> 18) & (GC_CONFIG >> 21)
object_car = ((arrayref >> 13) & (GC_CONFIG >> 16)) & 0x0000001f
value_region = (value >> 18) & (GC_CONFIG >> 21)
value_car = ((value >> 13) & (GC_CONFIG >> 16)) & 0x0000001f
if ((PSR.GCE = 1) AND
    (object_region = value_region) AND
    (object_car ≠ value_car)) then
            trap gc_notify (type = 0x27)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
```

```
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem[addr_of_length + 4 + (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 12
```

Notes

Use this instruction when you can ensure that the type check that is required for `aastore` succeeds, for example, when the array has elements of type, `java.lang.Object`.

Push a `null` reference onto the stack.

Format

aconst_null

Forms

`aconst_null` = 1 (0x01)

Stack

… ⇒
…, `0`

Description

`aconst_null` pushes the reference value `null` onto the stack.

Operation

```
stack[OPTOP] ⇐ 0
OPTOP ⇐ OPTOP - 4
```

Notes

The `null` value is a 32-bit word with a numerical value of 0. As a result, `aconst_null` is identical to `iconst_0` and `fconst_0` in the picoJava-II core.

Read a reference field in an object.

Format

agetfield_quick
indexbyte1
indexbyte2

Forms

`agetfield_quick` = 230 (0xe6)

Stack

…, *objectref* ⇒
…, *value*

Description

`agetfield_quick` treats *objectref* on the stack as an object reference. It then reads a one-word *value* from the offset ((*indexbyte1* << 8) | *indexbyte2)* into the class instance referenced by *objectref* and pushes it onto the stack.

If *objectref* is `null`, then `agetfield_quick` takes a `NullPointer` trap.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ ((indexbyte1 << 8) | indexbyte2)
handle_bit ⇐ objectref & 0x00000001
if (handle_bit = 1) then
            addr_of_fields ⇐ mem[(objectref & 0x7ffffffc) + 4]
else
            addr_of_fields ⇐ (objectref & 0x7ffffffc) + 4
stack[OPTOP + 4] ⇐ mem[addr_of_fields + (index × 4)]
```

Read a static reference field in a class.

Format

agetstatic_quick
indexbyte1
indexbyte2

Forms

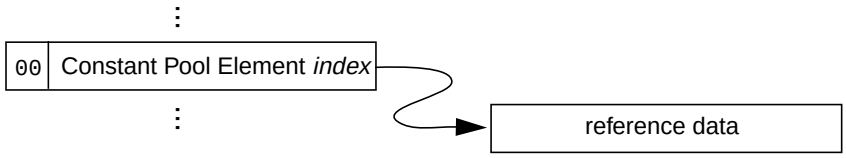
`agetstatic_quick` = 232 (0xe8)

Stack

… ⇒
…, *value*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to contain the address of the static field. `agetstatic_quick` reads the value of that class field and pushes it onto the stack as *value*.



Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
addr_of_static ⇐ mem[CONST_POOL + (index × 4)] & 0x7fffffff
stack[OPTOP] ⇐ mem[addr_of_static]
OPTOP ⇐ OPTOP – 4
```

Notes

In the picoJava-II core, `agetstatic_quick` is identical to `getstatic_quick`. The distinction allows future implementations of `agetstatic_quick` to differ in garbage collection events.

Push a reference item from constant pool.

Format

aldc_quick
index

Forms

aldc_quick = 234 (0xea)

Stack

… ⇒
…, *value*

Description

The unsigned *index* byte is an index into the constant pool of the current class. The constant pool item should have been resolved to contain the reference to the constant object. aldc_quick reads the *value* from the constant pool and pushes it onto the stack.

Operation

```
stack[OPTOP] ⇐ mem[CONST_POOL + (index × 4)]
OPTOP ⇐ OPTOP – 4
```

Notes

In the picoJava-II core, aldc_quick is identical to ldc_quick. The distinction allows future implementations of aldc_quick to differ in garbage collection events.

Push a reference item from constant pool.

Format

aldc_w_quick
indexbyte1
indexbyte2

Forms

aldc_w_quick = 235 (0xeb)

Stack

… ⇒
…, *value*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to contain the reference to the constant object. aldc_w_quick reads the value from the constant pool and pushes it onto the stack.

Operation

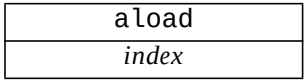
```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
stack[OPTOP] ⇐ mem[CONST_POOL + (index × 4)]
OPTOP ⇐ OPTOP – 4
```

Notes

In the picoJava-II core, aldc_w_quick is identical to ldc_w_quick. The distinction allows future implementations of aldc_w_quick to differ in garbage collection events.

Load a reference from a local variable.

Format



Forms

`aload` = 25 (0x19)

Stack

… ⇒
…, *value*

Description

`aload` pushes a one-word local variable, which is at *index* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS -(index × 4)]
OPTOP ⇐ OPTOP – 4
```

Notes

In the picoJava-II core, `aload` is identical to `iload` and `fload`.

Load a reference from a local variable.

Format

aload_*n*

Forms

aload_0 = 42 (0x2a)
aload_1 = 43 (0x2b)
aload_2 = 44 (0x2c)
aload_3 = 45 (0x2d)

Stack

… ⇒
…, *value*

Description

aload_*n* pushes a one-word local variable, which is at *n* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

stack[OPTOP] ⇐ stack[VARS – (*n* × 4)]
OPTOP ⇐ OPTOP – 4

Notes

In the picoJava-II core, aload_*n* is identical to iload_*n* and fload_*n*.

Trap to emulation routine that resolves constant pool entry and creates a new array of references.

Format

anewarray
indexbyte1
indexbyte2

Forms

anewarray = 189 (0xbd)

Stack

…, *count* ⇒
…, *objectref*

Description

anewarray traps to the emulation routine referenced by entry 0xbd in the trap table.

Operation

```
trap anewarray (type = 0xbd)
```

Recommendations

The trap handler should emulate anewarray, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the anewarray instruction with the anewarray_quick instruction.

Trap to emulation routine that creates a new array of references.

Format

anewarray_quick
indexbyte1
indexbyte2

Forms

anewarray_quick = 222 (0xde)

Stack

…, *integer* ⇒
…, *objectref*

Description

anewarray_quick traps to the emulation routine referenced by entry 0xde in the trap table.

Operation

```
trap anewarray_quick (type = 0xde)
```

Recommendations

The trap handler should emulate anewarray_quick, as described in *The Java Virtual Machine Specification*. The constant pool entry referenced by anewarray_quick should have been resolved, allowing the emulation trap handler to bypass this operation.

Set a reference field in an object with garbage collection checks.

Format

aputfield_quick
indexbyte1
indexbyte2

Forms

aputfield_quick = 231 (0xe7)

Stack

…, *objectref*, *value* ⇒
…

Description

aputfield_quick pops *objectref* and *value*, which it treats as object references, from the operand stack. It then stores *value* at the offset ((*indexbyte1* << 8) | *indexbyte2*) into the class instance referenced by *objectref*.

If *objectref* is null, then aputfield_quick takes a NullPointer trap. Otherwise, it performs the garbage collection checks described in *Write Barriers* on page 434, possibly generating a gc_notify trap.

Operation

```
objectref ⇐ stack[OPTOP + 8]
if (objectref = 0) then
            trap NullPointer (type = 0x1b)
gc_index ⇐ ((objectref & 0xc0000000) >> 28) | (value >> 30)
write_barrier_bit = (GC_CONFIG >> gc_index) & 0x00000001
if (write_barrier_bit = 1) then
            trap gc_notify (type = 0x27)
object_region = (objectref >> 18) & (GC_CONFIG >> 21)
object_car = ((objectref >> 13) & (GC_CONFIG >> 16)) & 0x0000001f
value_region = (value >> 18) & (GC_CONFIG >> 21)
value_car = ((value >> 13) & (GC_CONFIG >> 16)) & 0x0000001f
if ((PSR.GCE = 1) AND
    (object_region = value_region) AND
    (object_car ≠ value_car)) then
            trap gc_notify (type = 0x27)
index ⇐ ((indexbyte1 << 8) | indexbyte2)
handle_bit ⇐ objectref & 0x00000001
```

```
if (handle_bit = 1) then
            addr_of_fields ⇐ mem[(objectref & 0x7ffffffc) + 4]
else
            addr_of_fields ⇐ (objectref & 0x7ffffffc) + 4
mem[addr_of_fields + (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
```

Set a static reference field in a class with garbage collection checks.

Format

aputstatic_quick
indexbyte1
indexbyte2

Forms

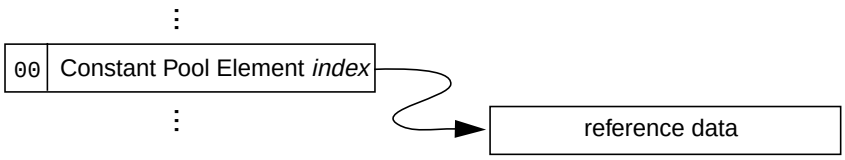
aputstatic_quick = 233 (0xe9)

Stack

…, *value* ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item must already have been resolved and must contain the address of the static reference type. aputstatic_quick pops *value* from the operand stack and sets that static field to *value*.



aputstatic_quick treats the address of the static variable storage that is held in the constant pool as an object reference for the purpose of performing the garbage collection checks, as described in *Write Barriers* on page 434. As a result of these garbage collection checks, aputstatic_quick may generate a gc_notify trap.

Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
addr_of_static ⇐ mem[CONST_POOL + (index × 4)]
gc_index ⇐ ((addr_of_static & 0xc0000000) >> 28) | (value >> 30)
write_barrier_bit = (GC_CONFIG >> gc_index) & 0x00000001
if (write_barrier_bit = 1) then
            trap gc_notify (type = 0x27)
```

```
object_region = (addr_of_static >> 18) & (GC_CONFIG >> 21)
object_car = ((addr_of_static >> 13) & (GC_CONFIG >> 16)) & 0x0000001f
value_region = (value >> 18) & (GC_CONFIG >> 21)
value_car = ((value >> 13) & (GC_CONFIG >> 16)) & 0x0000001f
if ((PSR.GCE = 1) AND
    (object_region = value_region) AND
    (object_car ≠ value_car)) then
            trap gc_notify (type = 0x27)
mem[addr_of_static & 0x7ffffffc] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Return a reference from a method.

Format

```
areturn
```

Forms

`areturn` = 176 (0xb0)

Stack

…, *value* ⇒
[empty]

Description

`areturn` returns to the caller of this Java method, popping all the arguments to the current method and pushing the reference that is at the top of the operand stack onto the top of the caller's operand stack.

Operation

```
PC ⇐ stack[FRAME]
CONST_POOL ⇐ stack[FRAME – 12]
stack[VARS] ⇐ stack[OPTOP + 4]
VARS ⇐ stack[FRAME – 4]
FRAME ⇐ stack[FRAME – 8]
OPTOP ⇐ VARS + 4
```

Notes

In the picoJava-II core, `areturn` is identical to `ireturn` and `freturn`.

Return the number of elements in an array.

Format

arraylength

Forms

`arraylength` = 190 (0xbe)

Stack

…, *arrayref* ⇒
…, *length*

Description

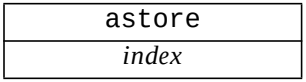
`arraylength` pushes the length of the array referenced by the array reference, *arrayref*, onto the stack.

Operation

```
arrayref ⇐ stack[OPTOP + 4]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
stack[OPTOP + 4] ⇐ mem[addr_of_length]
```

Store a reference to a local variable.

Format



Forms

`astore` = 58 (0x3a)

Stack

…, *value* ⇒
…

Description

`astore` stores the top entry of the operand stack into a one-word local variable, which is at *index* stack entries offset from the start of the current local variables.

Operation

`stack[VARS-(`*index* × `4)] ⇐ stack[OPTOP + 4]`
`OPTOP ⇐ OPTOP + 4`

Notes

In the picoJava-II core, `astore` is identical to `istore` and `fstore`.

Store a reference to a local variable.

Format

astore_*n*

Forms

astore_0 = 75 (0x4b)
astore_1 = 76 (0x4c)
astore_2 = 77 (0x4d)
astore_3 = 78 (0x4e)

Stack

…, *value* ⇒
…

Description

astore_*n* stores the top entry of the operand stack into a one-word local variable, which is at *n* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS -(n × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, astore_*n* is identical to istore_*n* and fstore_*n*.

Trap to emulation routine that throws an exception or error.

Format

athrow

Forms

`athrow` = 191 (0xbf)

Stack

…, objectref ⇒
objectref

Description

`athrow` traps to the emulation routine referenced by entry 0xbf in the trap table.

Operation

```
trap athrow (type=0xbf)
```

Recommendations

The trap handler should emulate `athrow`, as described in *The Java Virtual Machine Specification*.

Load a byte from an array.

Format

baload

Forms

baload = 51 (0x33)

Stack

…, arrayref, index ⇒
…, value

Description

baload treats *arrayref* as a reference to an array of bytes. It loads and sign-extends the one-byte element at *index* and pushes it onto the stack as *value*.

If *arrayref* is null, then baload takes a NullPointer trap. If *index* is not within the bounds of the array referenced by *arrayref*, then baload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ sign_ext8(mem8[addr_of_length + 4 + index])
OPTOP ⇐ OPTOP + 4
```

Store a byte to an array.

Format

bastore

Forms

`bastore` = 84 (0x54)

Stack

…, arrayref, index, value ⇒
…

Description

`bastore` treats *arrayref* as a reference to an array of bytes. It truncates the integer *value* on the stack to the low 8 bits and stores it to the one-byte element at *index* of the array.

If *arrayref* is `null`, then `bastore` takes a `NullPointer` trap. If *index* is not within the bounds of the array referenced by *arrayref*, then `bastore` takes an `ArrayIndexOutOfBounds` trap.

Operation

```
arrayref ⇐ stack[OPTOP + 12]
if (arrayref = 0) then
            trap NullPointer (type=0x1b)
index ⇐ stack[OPTOP + 8]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem8[addr_of_length + 4 + index] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 12
```

Push a byte onto the operand stack.

Format

bipush
byte

Forms

bipush = 16 (0x10)

Stack

… ⇒
…, *value*

Description

bipush sign-extends the *byte* constant and pushes it onto the operand stack.

Operation

stack[OPTOP] ⇐ $sign\_ext_8$(*byte*)
OPTOP ⇐ OPTOP – 4

Flush a cache line and possibly invalidate it if it is present in the cache.

Format

extend
cache_flush

Forms

`extend` = 255 (0xff)
`cache_flush` = 30 (0x1e)

Stack

…, *increment*, *address* ⇒
…, *increment*, *result_address*

Description

`cache_flush` checks both the instruction and data caches for *address*, which specifies a line in the data and instruction caches that is to be flushed or invalidated.

If the line is present in either or both caches, then `cache_flush` invalidates it. If the cache line is dirty—that is, has been modified—then `cache_flush` writes it back to memory. If a cache is off (`PSR.DCE` = 0 or `PSR.ICE` = 0), then that cache ignores the flush request.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

result_address is *address* plus *increment* as if they had been summed with `iadd`.

Operation

```
address ⇐ stack[OPTOP + 4] & 0x7ffffff0
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if (PSR.ICE = 1) then
            iindex_mask ⇐ ((1 << (HCR.ICS + 9)) - 1)
            itag_address ⇐ address & iindex_mask
            itag ⇐ icache_tag[itag_address] & 0xfffffffe
            icache_tag[itag_address] ⇐ itag
if (PSR.DCE = 1) then
            dindex_mask ⇐ ((1 << (HCR.DCS + 8)) - 1)
            dtag_mask ⇐ dindex_mask ^ 0x7fffffff
            dtag_address0 ⇐ address & dindex_mask
            dtag_address1 ⇐ dtag_address0 | 0x80000000
```

```
        dtag0 ⇐ dcache_tag[dtag_address0] & dtag_mask
        dtag1 ⇐ dcache_tag[dtag_address1] & dtag_mask
        dtag_to_match ⇐ address & dtag_mask
        if (dtag0 = dtag_to_match) then
                        dtag_address ⇐ dtag_address0
                        dtag ⇐ dcache_tag[dtag_address]
                        dirty_valid_bits ⇐ dtag & 0x0000003
                        dcache_tag[dtag_address] ⇐ dtag & 0xfffffff8
        else if (dtag1 = dtag_to_match) then
                        dtag_address ⇐ dtag_address1
                        dtag ⇐ dcache_tag[dtag_address]
                        dirty_valid_bits ⇐ dtag & 0x0000003
                        dcache_tag[dtag_address] ⇐ dtag & 0xfffffff8
        else
                        dirty_valid_bits ⇐ 0
        if (dirty_valid_bits = 0x3) then
                        mem_addr ⇐ (dtag | dtag_address) & 0x7ffffff0
                        mem_NC[mem_addr] ⇐ dcache_data[dtag_address]
                        mem_NC[mem_addr + 4] ⇐ dcache_data[dtag_address + 4]
                        mem_NC[mem_addr + 8] ⇐ dcache_data[dtag_address + 8]
                        mem_NC[mem_addr + 12] ⇐ dcache_data[dtag_address + 12]
stack[OPTOP + 4] ⇐ address + stack[OPTOP + 8]
```

Notes

Although the picoJava-II core has 16-byte cache lines for both the instruction and data caches, you should rely on the values in the DCL and ICL fields of the Hardware Configuration Register (HCR) to facilitate porting software between implementations—the HCR.DCL field indicates the number of bytes in a data cache line; the HCR.ICL field indicates the number of bytes in an instruction cache line.

Flush a cache line and possibly invalidate it (with no tag checks).

Format

extend
cache_index_flush

Forms

extend = 255 (0xff)
cache_index_flush = 31 (0x1f)

Stack

…, *increment*, *address* ⇒
…, *increment*, *result_address*

Description

cache_index_flush invalidates the line that corresponds to the index indicated by *address* in both the instruction and data caches. For the two-way set-associative data cache, bit 31 indicates the set to be flushed.

If the cache line is dirty—that is, has been modified—then cache_index_flush writes it back to memory. If a cache is off (PSR.DCE = 0 or PSR.ICE = 0), then that cache ignores the invalidation request.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

result_address is *address* plus *increment* as if they had been summed with iadd.

Operation

```
address ⇐ stack[OPTOP + 4] & 0xfffffff0
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if (PSR.ICE = 1) then
            iindex_mask ⇐ ((1 << (HCR.ICS + 9)) - 1)
            itag_address ⇐ address & iindex_mask
            itag ⇐ icache_tag[itag_address] & 0xfffffffe
            icache_tag[itag_address] ⇐ itag
if (PSR.DCE = 1) then
            dindex_mask ⇐ ((1 << (HCR.DCS + 8)) - 1) | 0x80000000
            dtag_address ⇐ address & dindex_mask
            dtag ⇐ dcache_tag[dtag_address]
```

```
            dirty_valid_bits ⇐ dtag & 0x0000003
            if (dirty_valid_bits = 0x3) then
                        mem_addr ⇐ (dtag | dtag_address) & 0x7ffffff0
                        mem_NC[mem_addr] ⇐ dcache_data[dtag_address]
                        mem_NC[mem_addr + 4] ⇐ dcache_data[dtag_address + 4]
                        mem_NC[mem_addr + 8] ⇐ dcache_data[dtag_address + 8]
                        mem_NC[mem_addr + 12] ⇐ dcache_data[dtag_address + 12]
            dcache_tag[dtag_address] ⇐ dtag & 0xfffffff8
stack[OPTOP + 4] ⇐ address + stack[OPTOP + 8]
```

Notes

Although the picoJava-II core has 16-byte cache lines for both the instruction and data caches, you should rely on the values in the DCL and ICL fields of the Hardware Configuration Register (HCR) to facilitate porting software between implementations—the HCR.DCL field indicates the number of bytes in a data cache line; the HCR.ICL field indicates the number of bytes in an instruction cache line.

Invalidate a cache line if it is present in the cache.

Format

extend
cache_invalidate

Forms

`extend` = 255 (0xff)
`cache_invalidate` = 23 (0x17)

Stack

…, *increment*, *address* ⇒
…, *increment*, *result_address*

Description

`cache_invalidate` checks both the instruction and data caches for *address*, which specifies a line in the data and instruction caches that is to be invalidated.

If the line is present in either or both of the caches, then `cache_invalidate` invalidates it. If a cache is off (`PSR.DCE` = 0 or `PSR.ICE` = 0), then that cache ignores the invalidation request.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

result_address is *address* plus *increment* as if they had been summed with `iadd`.

Operation

```
address ⇐ stack[OPTOP + 4] & 0x7ffffff0
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if (PSR.ICE = 1) then
            iindex_mask ⇐ ((1 << (HCR.ICS + 9)) - 1)
            itag_mask ⇐ iindex_mask ^ 0x7fffffff
            itag_address ⇐ address & iindex_mask
            itag ⇐ icache_tag[itag_address] & itag_mask
            itag_to_match ⇐ address & itag_mask
            if (itag = itag_to_match) then
                        icache_tag[itag_address] ⇐ itag
if (PSR.DCE = 1) then
            dindex_mask ⇐ ((1 << (HCR.DCS + 8)) - 1)
            dtag_mask ⇐ dindex_mask ^ 0x7fffffff
```

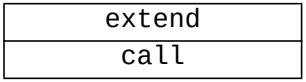
```
            dtag_address0 ⇐ address & dindex_mask
            dtag_address1 ⇐ dtag_address0 | 0x80000000
            dtag0 ⇐ dcache_tag[dtag_address0] & dtag_mask
            dtag1 ⇐ dcache_tag[dtag_address1] & dtag_mask
            dtag_to_match ⇐ address & dtag_mask
            if (dtag0 = dtag_to_match) then
                        dcache_tag[dtag_address0] ⇐ dtag0
            else if (dtag1 = dtag_to_match) then
                        dcache_tag[dtag_address1] ⇐ dtag1
stack[OPTOP + 4] ⇐ address + stack[OPTOP + 8]
```

Notes

Although the picoJava-II core has 16-byte cache lines for both the instruction and data caches, you should rely on the values in the DCL and ICL fields of the Hardware Configuration Register (HCR) to facilitate porting software between implementations—the HCR.DCL field indicates the number of bytes in a data cache line; the HCR.ICL field indicates the number of bytes in an instruction cache line.

Call a subroutine with the specified number of arguments.

Format



Forms

`extend` = 255 (0xff)
`call` = 61 (0x3d)

Stack

…, arg_0, arg_1, ..., arg_n, *targetPC*, *nargs* ⇒
…, arg_0, arg_1, ..., arg_n, *returnVARS*, *returnPC*

Description

`call` takes the target PC of a subroutine and the number of argument words (including the *targetPC* and *nargs* values) from the stack. It then transfers control to the code at *targetPC*, updates the VARS registers to point to arg_0, and saves the original contents of the VARS register and the return PC in the stack positions that held the *targetPC* and *nargs* values before the call.

Operation

```
return_PC ⇐ PC + 2
PC ⇐ stack[OPTOP + 8]
stack[OPTOP + 8] ⇐ VARS
VARS ⇐ OPTOP + (stack[OPTOP + 4] × 4)
stack[OPTOP + 4] ⇐ return_PC
```

Load a character from an array.

Format

caload

Forms

caload = 52 (0x34)

Stack

…, arrayref, index ⇒
…, value

Description

caload treats *arrayref* as a reference to an array of characters. It loads the two-byte unsigned element at *index* and pushes it onto the stack as *value*.

If *arrayref* is null, then caload takes a NullPointer trap. If *index* is not within the bounds of the array referenced by *arrayref*, then caload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ mem16[addr_of_length + 4 + (index × 2)]
OPTOP ⇐ OPTOP + 4
```

Store a character to an array.

Format

castore

Forms

`castore` = 85 (0x55)

Stack

…, arrayref, index, value ⇒
…

Description

`castore` treats *arrayref* as a reference to an array of characters. It truncates the integer *value* on the stack to the low 16 bits and stores it to the two-byte element at *index* of the array.

If *arrayref* is `null`, then `castore` takes a `NullPointer` trap. If *index* is not within the bounds of the array referenced by *arrayref*, then `castore` takes an `ArrayIndexOutOfBounds` trap.

Operation

```
arrayref ⇐ stack[OPTOP + 12]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 8]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem16[addr_of_length + 4 + (index × 2)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 12
```

Notes

In the picoJava-II core, `castore` is identical to `sastore`.

Trap to emulation routine that resolves the constant pool entry and checks whether an object is of the given type.

Format

checkcast
indexbyte1
indexbyte2

Forms

`checkcast` = 192 (0xc0)

Stack

…, *objectref* ⇒
…, *objectref*

Description

`checkcast` traps to the emulation routine referenced by entry 0xc0 in the trap table.

Operation

```
trap checkcast (type=0xc0)
```

Recommendations

The trap handler should emulate `checkcast`, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the `checkcast` instruction with the `checkcast_quick` instruction.

Check whether an object is of the given type.

Format

checkcast_quick
indexbyte1
indexbyte2

Forms

checkcast_quick = 224 (0xe0)

Stack

…, *objectref* ⇒
…, *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item must have already been resolved and must contain a class ID. The word on the top of the stack, *objectref*, is treated as a reference.

If *objectref* is not null, then the class ID of *objectref* is compared with the class ID from the constant pool. If the two class IDs are not equal, then the core generates a checkcast_quick emulation trap. Otherwise, checkast_quick completes without further action.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref ≠ 0) then
            object_header ⇐ mem[objectref & 0x7ffffffc]
            object_class_addr ⇐ (object_header & 0x7ffffff8) - 8
            object_class ⇐ mem[object_class_addr]
            index ⇐ (indexbyte1 << 8) | indexbyte2
            constant_class ⇐ mem[CONST_POOL + (index × 4)]
            if (object_class ≠ constant_class)
                        trap checkcast_quick (type=0xe0)
```

Recommendations

The trap handler should emulate checkcast_quick, as defined in *The Java Virtual Machine Specification*. The trap handler, however, should skip the initial checks performed by the core.

Convert a double to a float.

Format

d2f

Forms

d2f = 144 (0x90)

Stack

…, value<31:0>, value<63:32> ⇒
…, result

Description

d2f treats the value on the top of the operand stack as the type double, then pops it from the operand stack and converts it to a float with the IEEE 754 round-to-nearest mode, which it pushes onto the operand stack.

The conversion rules are:

- d2f converts a finite value that is too small to be represented as a float to a zero of the same sign; d2f converts a finite value that is too large to be represented as a float to an infinity of the same sign.
- d2f converts a double NaN to a float NaN.

If the floating point unit is not enabled (PSR.FPE = 0), d2f traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            stack[OPTOP + 8] ⇐
                        convert_d2f(double(stack[OPTOP + 4], stack[OPTOP + 8]))
            OPTOP ⇐ OPTOP + 4
else
            trap d2f (type = 0x90)
```

Notes

result may lose precision and information about the overall magnitude of *value.*

Convert a double to an integer.

Format

d2i

Forms

`d2i` = 142 (0x8e)

Stack

…, value<31:0>, value<63:32> ⇒
…, result

Description

`d2i` treats the value on the top of the operand stack (*value*) as the type double, pops it from the operand stack, and converts it to an integer. It then pushes the result (*result)* onto the operand stack.

If *value* is NaN, then *result* is the integer 0. If *value* is not an infinity, then `d2i` rounds it to an integer toward zero with the IEEE 754 round-toward-zero mode. If that integer is a 32-bit integer, then it becomes *result*. Otherwise, either of the following conditions applies:

- *value* must be too small (a negative value of large magnitude or negative infinity) and *result* is –2147483648 (0x8000000)
- *value* must be too large (a positive value of large magnitude or positive infinity) and *result* is 2147483647 (0x7fffffff).

If the floating point unit is not enabled (`PSR.FPE` = 0), then `d2i` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            stack[OPTOP + 8] ⇐
                              convert_d2i(double(stack[OPTOP + 4], stack[OPTOP + 8]))
            OPTOP ⇐ OPTOP + 4
else
            trap d2i (type = 0x8e)
```

Notes

result may lose precision and information about the overall magnitude of *value*.

Convert a double to a long.

Format

d2l

Forms

d2l = 143 (0x8f)

Stack

…, value<31:0>, value<63:32> ⇒
…, result<31:0>, result<63:32>

Description

d2l treats the value on the top of the operand stack (*value*) as the type double, pops it from the operand stack, and converts it to a long integer. It then pushes the result (*result)* onto the operand stack.

If *value* is NaN, then *result* is the long integer 0. If *value* is not an infinity, then d2l rounds it to a long integer toward zero with the IEEE 754 round-toward-zero mode. If that long integer is a 64-bit long integer, then it becomes *result*. Otherwise, either of the following conditions applies:

- *value* must be too small (a negative value of large magnitude or negative infinity) and *result* is –9223372036854775808 (0x800000000000000)
- *value* must be too large (a positive value of large magnitude or positive infinity) and *result* is 9223372036854775807 (0x7fffffffffffffff).

If the floating point unit is not enabled (PSR.FPE = 0), then d2l traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            long(stack[OPTOP + 4],stack[OPTOP + 8]) ⇐
                        convert_d2l(double(stack[OPTOP + 4], stack[OPTOP + 8]))
else
            trap d2l (type = 0x8f)
```

Notes

result may lose precision and information about the overall magnitude of *value*.

Add two doubles.

Format

dadd

Forms

dadd = 99 (0x63)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

dadd treats both *value1* and *value2* as the type double and pops them from the operand stack. It then pushes *result*, which is a double and the sum of *value1* + *value2*, onto the operand stack.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not, then *result* is *value2* (NaN) with a positive sign.
- The sum of two infinities of opposite signs is NaN (0x7fffe00000000000).
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite signs is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite signs is positive zero.
- In the remaining cases, where an infinity, a zero, or NaN is not involved and where the values have the same sign or different magnitudes, the sum is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a double, the operation overflows; *result* is then an infinity of the appropriate sign. If the magnitude is too small to be represented as a double, the operation underflows; result is then a zero of the appropriate sign.

If the Floating Point Unit (FPU) is not enabled (PSR.FPE = 0), dadd traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
                                double(stack[OPTOP + 12], stack[OPTOP + 16]) +
                                double(stack[OPTOP + 4], stack[OPTOP + 8])
            OPTOP ⇐ OPTOP + 8
else
            trap dadd (type = 0x63)
```

Load a double from an array.

Format

daload

Forms

daload = 49 (0x31)

Stack

…, arrayref, index ⇒
…, value<31:0>, value<63:32>

Description

daload treats *arrayref* as a reference to an array of doubles. It loads the two-word element at *index* and pushes it onto the stack as *value*.

If *arrayref* is null, then daload takes a NullPointer trap. If *index* is not within the bounds of the array that *arrayref* references, then daload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ mem[addr_of_length + 8 + (index × 8)]
stack[OPTOP + 4] ⇐ mem[addr_of_length + 4 + (index × 8)]
```

Notes

In the picoJava-II core, daload is identical to laload.

Store a double to an array.

Format

dastore

Forms

dastore = 82 (0x52)

Stack

…, arrayref, index, value<31:0>, value<63:32>⇒
…

Description

dastore treats *arrayref* as a reference to an array of doubles. It stores the double *value* on the stack to the two-word element at *index* of the array.

If *arrayref* is null, then dastore takes a NullPointer trap. If *index* is not within the bounds of the array that *arrayref* references, then dastore takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 16]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 12]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)

mem[addr_of_length + 4 + (index × 8)] ⇐ stack[OPTOP + 4]
mem[addr_of_length + 8 + (index × 8)] ⇐ stack[OPTOP + 8]
OPTOP ⇐ OPTOP + 16
```

Notes

In the picoJava-II core, dastore is identical to lastore.

Compare two doubles with greater than on NaN.

Format

dcmpg

Forms

dcmpg = 152 (0x98)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result

Description

dcmpg treats both *value1* and *value2* as the type double, pops them from the operand stack, then performs a floating-point comparison and executes as follows:

- If *value1* is greater than *value2*, then dcmpg pushes the integer value 1 onto the operand stack.
- If *value1* is equal to *value2*, then dcmpg pushes the integer value 0 onto the operand stack.
- If *value1* is less than *value2*, then dcmpg pushes the integer value –1 onto the operand stack.
- If either *value1* or *value2* is a NaN, then dcmpg pushes the integer value 1 onto the operand stack.

dcmpg performs floating-point comparisons according to IEEE 754 rules: It orders all values other than NaN with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

If the floating point unit is not enabled (PSR.FPE = 0), then dcmpg traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            diff = double(stack[OPTOP + 16], stack[OPTOP + 12]) -
                        double(stack[OPTOP + 8], stack[OPTOP + 4])
            if (diff < 0.0) then
                        stack[OPTOP + 16] ⇐ -1
            else if ((diff > 0.0) OR (diff = NaN)) then
                        stack[OPTOP + 16] ⇐ 1
            else
                        stack[OPTOP + 16] ⇐ 0
            OPTOP ⇐ OPTOP + 12
else
            trap dcmpg (type = 0x98)
```

Notes

dcmpg and dcmpl differ only in how they compare values that involve NaN.

Compare two doubles with less than on NaN.

Format

dcmpl

Forms

dcmpl = 151 (0x97)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result

Description

dcmpl treats both *value1* and *value2* as the type double, pops them from the operand stack, then performs a floating-point comparison and executes as follows:

- If *value1* is greater than *value2*, then dcmpl pushes the integer value 1 onto the operand stack.
- If *value1* is equal to *value2*, then dcmpl pushes the integer value 0 onto the operand stack.
- If *value1* is less than *value2*, then dcmpl pushes the integer value –1 onto the operand stack.
- If either *value1* or *value2* is a NaN, then dcmpl pushes the integer value –1 onto the operand stack.

dcmpl performs floating-point comparisons according to IEEE 754 rules: It orders all values other than NaN with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

If the floating point unit is not enabled (PSR.FPE = 0), then dcmpl traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            diff = double(stack[OPTOP + 16], stack[OPTOP + 12]) -
                           double(stack[OPTOP + 8], stack[OPTOP + 4])
            if ((diff < 0.0) OR (diff = NaN)) then
                           stack[OPTOP + 16] ⇐ -1
            else if (diff > 0.0) then
                           stack[OPTOP + 16] ⇐ 1
            else
                           stack[OPTOP + 16] ⇐ 0
            OPTOP ⇐ OPTOP + 12
else
            trap dcmpl (type = 0x97)
```

Notes

dcmpg and dcmpl differ only in how they compare values that involve NaN.

Push the double constant 0.0.

Format

dconst_0

Forms

dconst_0 = 14 (0x0e)

Stack

… ⇒
…, 0, 0

Description

dconst_0 pushes the double constant 0.0 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 0
stack[OPTOP - 4] ⇐ 0
OPTOP ⇐ OPTOP - 8
```

Notes

dconst_0 is identical to lconst_0.

Push the double constant 1.0.

Format

dconst_1

Forms

dconst_1 = 15 (0x0f)

Stack

… ⇒
…, 0, 0x3ff00000

Description

dconst_1 pushes the double constant 1.0 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 0
stack[OPTOP - 4] ⇐ 0x3ff00000
OPTOP ⇐ OPTOP - 8
```

Divide two doubles.

Format

ddiv

Forms

`ddiv` = 111 (0x6f)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

`ddiv` treats both *value1* and *value2* as the type double, pops them from the operand stack, and pushes the double *result*, which is *value1* ÷ *value2*, onto the operand stack.

result is governed by the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not, *result* is *value2* (NaN) with a positive sign.
- The sum of two infinities of opposite signs is NaN (0x7fffe00000000000).
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite signs is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite signs is positive zero.
- In the remaining cases, where an infinity, a zero, or NaN is not involved and where the values have the same sign or different magnitudes, the sum is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a double, the operation overflows; *result* is then an infinity of the appropriate sign. If the magnitude is too small to be represented as a double, the operation underflows; *result* is then a zero of the appropriate sign.

If the floating point unit is not enabled (`PSR.FPE` = 0), then `ddiv` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
                                 double(stack[OPTOP + 12], stack[OPTOP + 16]) ÷
                                 double(stack[OPTOP + 4], stack[OPTOP + 8])
            OPTOP ⇐ OPTOP + 8
else
            trap ddiv (type = 0x6f)
```

Load a double from a local variable.

Format

dload
index

Forms

dload = 24 (0x18)

Stack

… ⇒
…, *value<31:0>*, *value<63:32>*

Description

dload pushes a two-word local variable, which is at *index* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS - (index × 4)]
stack[OPTOP - 4] ⇐ stack[VARS - 4 - (index × 4)]
OPTOP ⇐ OPTOP - 8
```

Notes

In the picoJava-II core, dload is identical to lload.

Load a double from a local variable.

Format

dload_*n*

Forms

dload_0 = 38 (0x26)
dload_1 = 39 (0x27)
dload_2 = 40 (0x28)
dload_3 = 41 (0x29)

Stack

… ⇒
…, *value<31:0>*, *value<63:32>*

Description

dload_*n* pushes a two-word local variable, which is at *n* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS -(n × 4)]
stack[OPTOP - 4] ⇐ stack[VARS - 4 - (n × 4)]
OPTOP ⇐ OPTOP - 8
```

Notes

In the picoJava-II core, dload_*n* is identical to lload_*n*.

Multiply two doubles.

Format

dmul

Forms

dmul = 107 (0x6b)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

dmul treats both *value1* and *value2* as the type double and pops them from the operand stack. It then pushes the double *result*, which is *value1* × *value2*, onto the operand stack.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not NaN, then *result* is *value2* (NaN) with a positive sign.
- If neither value is NaN, then the sign of *result* is positive if both values have the same sign, negative if the values have different signs.
- Multiplication of an infinity by a finite value results in a signed infinity according to the sign-producing rule above.
- Multiplication of an infinity by a zero results in NaN (0x7fffe00000000000).
- In the remaining cases, where neither an infinity nor NaN is involved, the product is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a double, the operation overflows; *result* is then an infinity of appropriate sign. If the magnitude is too small to be represented as a double, the operation underflows; *result* is then a zero of appropriate sign.

If the floating point unit is not enabled (PSR.FPE = 0), then dmul traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
                                double(stack[OPTOP + 12], stack[OPTOP + 16]) ×
                                double(stack[OPTOP + 4], stack[OPTOP + 8])
            OPTOP ⇐ OPTOP + 8
else
            trap dmul (type = 0x6b)
```

Negate a double.

Format

dneg

Forms

dneg = 119 (0x77)

Stack

…, value<31:0>, value<63:32> ⇒
…, result<31:0>, result<63:32>

Description

dneg treats *value* as the type double and pops it from the operand stack. The double *result* is *–value,* which is the arithmetic negation of *value*. dneg then pushes *result* onto the operand stack.

For double values, negation is different from subtraction from zero. If *x* is +0.0, then 0.0 – *x* equals +0.0, but *–x* equals –0.0. Unary minus merely inverts the sign of a double.

Also note the following rules:

- If the operand is NaN, then *result* is NaN; NaN has no sign.
- If the operand is an infinity, then *result* is the infinity of the opposite sign.
- If the operand is a zero, then *result* is the zero of the opposite sign.

In practice, negating a double-precision value is simply inverting the sign bit – bit 31 of the most significant word.

Operation

```
stack[OPTOP + 4] ⇐ stack[OPTOP + 4] ^ 0x80000000
```

Notes

In the picoJava-II core, dneg is identical to fneg.

Compute the remainder of two doubles.

Format

drem

Forms

drem = 115 (0x73)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

drem treats both *value1* and *value2* as the type double and pops them from the operand stack. It then computes *result* and pushes it onto the operand stack as a double.

result is not the same as that of the remainder operation defined by IEEE 754, which computes the remainder from a rounding division, *not* a truncating division, causing a different behavior from that of the usual integer remainder operator. drem operates in the manner specified by the Java virtual machine.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not NaN, then *result* is *value2* (NaN) with a positive sign.
- If neither value is NaN, then the sign of *result* equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, then *result* is NaN (0x7fffe00000000000).
- If the dividend is finite and the divisor is an infinity, then the result equals the dividend.
- If the dividend is a zero and the divisor is finite, then the result equals the dividend.
- In the remaining cases, where an infinity, a zero, or NaN is not involved, the floating-point remainder *result* from a dividend *value1* and a divisor *value2* is defined by the mathematical relation *result* = *value1* - (*value2* × *q*), where *q* is an integer that is negative only if *value1* ÷ *value2* is negative and positive only if *value1* ÷ *value2* is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of *value1* and *value2*.

If the floating point unit is not enabled (PSR.FPE = 0) or the drem trap bit is set (PSR.DRT = 1), then dadd traps to an emulation routine.

Operation

```
if ((PSR.DRT = 0) AND (PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
                        double(stack[OPTOP + 12], stack[OPTOP + 16]) %
                        double(stack[OPTOP + 4], stack[OPTOP + 8])
            OPTOP ⇐ OPTOP + 8
else
            trap drem (type = 0x73)
```

Notes

`drem`, when executed by the floating point unit, may take in excess of 2,000 uninterruptible cycles to complete. You can prevent these long-running, uninterruptible instructions by setting the `PSR.DRT` bit and emulating them in software.

Return a double from a method.

Format

dreturn

Forms

`dreturn` = 175 (0xaf)

Stack

…, value<31:0>, value<63:32>⇒
[empty]

Description

`dreturn` returns to the caller of this method, popping all the arguments to the current method and pushing the double that is at the top of the operand stack onto the top of the caller's operand stack.

Operation

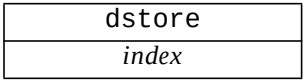
```
PC ⇐ stack[FRAME]
CONST_POOL ⇐ stack[FRAME - 12]
ret_value_word1 ⇐ stack[OPTOP + 4]
ret_value_word2 ⇐ stack[OPTOP + 8]
VARS ⇐ stack[FRAME - 4]
FRAME ⇐ stack[FRAME - 8]
OPTOP ⇐ VARS + 8
stack[OPTOP + 4] ⇐ ret_value_word1
stack[OPTOP + 8] ⇐ ret_value_word2
```

Notes

In the picoJava-II core, `dreturn` is identical to `lreturn`.

Store a double to a local variable.

Format



Forms

`dstore` = 57 (0x39)

Stack

…, value<31:0>, value<63:32> ⇒
…

Description

`dstore` stores the double on the top of the operand stack into a two-word local variable, which is at *index* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS - (index × 4)] ⇐ stack[OPTOP + 8]
stack[VARS - 4 - (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
```

Notes

In the picoJava-II core, `dstore` is identical to `lstore`.

Store a double to a local variable.

Format

dstore_*n*

Forms

dstore_0 = 71 (0x47)
dstore_1 = 72 (0x48)
dstore_2 = 73 (0x49)
dstore_3 = 74 (0x4a)

Stack

…, *value<31:0>*, *value<63:32>* ⇒
…

Description

dstore_*n* stores the double on the top of the operand stack into a two-word local variable, which is at *n* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS - (n × 4)] ⇐ stack[OPTOP + 8]
stack[VARS - 4 - (n × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
```

Notes

In the picoJava-II core, dstore_*n* is identical to lstore_*n*.

Subtract two doubles.

Format

dsub

Forms

dsub = 103 (0x67)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

dsub treats both *value1* and *value2* as the type double and pops them from the operand stack. It then pushes the double *result*, which is *value1 – value2*, onto the operand stack.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not NaN, then *result* is *value2* (NaN) with a positive sign.
- The difference of two infinities of the same sign is NaN (0x7fffe00000000000).
- The difference of two infinities of opposite signs is *value1*.
- The difference of an infinity and any finite value is equal to the infinity.
- The difference of two zeroes of opposite signs is *value1*.
- The difference of two zeroes of the same sign is positive zero.
- If *value1* is a zero and *value2* is a nonzero finite value, then *result* is *value2* with the opposite sign.
- If *value1* is a nonzero finite value and *value2* is a zero, then *result* is *value1*.
- The difference of two nonzero finite values of the same magnitude and same sign is positive zero.
- In the remaining cases, where an infinity, a zero, or NaN is not involved and where the values have the same sign or different magnitudes, the sum is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a double, the operation overflows; *result* is then an infinity of the appropriate sign. If the magnitude is too small to be represented as a double, the operation underflows; *result* is then a zero of the appropriate sign.

If the Floating Point Unit (FPU) is not enabled (PSR.FPE = 0), then dsub traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
                                    double(stack[OPTOP + 12],stack[OPTOP + 16]) –
                                    double(stack[OPTOP + 4], stack[OPTOP + 8])
            OPTOP ⇐ OPTOP + 8
else
            trap dsub (type = 0x67)
```

Duplicate the top operand stack word.

Format

dup

Forms

dup = 89 (0x59)

Stack

…, value ⇒
…, value, value

Description

dup duplicates the word on the top of the stack and pushes it onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP - 4
```

Duplicate top operand stack word and put two words down in the stack.

Format

```
dup_x1
```

Forms

dup_x1 = 90 (0x5a)

Stack

…, value2, value1⇒
…, value1, value2, value1

Description

dup_x1 duplicates the word on the top of the stack and inserts it two words down from the top of the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[OPTOP + 4]
stack[OPTOP + 4] ⇐ stack[OPTOP + 8]
stack[OPTOP + 8] ⇐ stack[OPTOP]
OPTOP ⇐ OPTOP - 4
```

Duplicate the top operand stack word and put three words down in the stack.

Format

dup_x1

Forms

dup_x2 = 91 (0x5b)

Stack

…, value3, value2, value1⇒
…, value1, value3, value2, value1

Description

dup_x2 duplicates the word on the top of the stack and inserts it three words down from the top of the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[OPTOP + 4]
stack[OPTOP + 4] ⇐ stack[OPTOP + 8]
stack[OPTOP + 8] ⇐ stack[OPTOP + 12]
stack[OPTOP + 12] ⇐ stack[OPTOP]
OPTOP ⇐ OPTOP - 4
```

Duplicate the top two operand stack words.

Format

dup2

Forms

dup2 = 92 (0x5c)

Stack

…, value2, value1⇒
…, value2, value1, value2, value1

Description

dup2 duplicates the two words on the top of the stack and pushes them onto the operand stack.

Operation

```
stack[OPTOP-4] ⇐ stack[OPTOP + 4]
stack[OPTOP] ⇐ stack[OPTOP + 8]
OPTOP ⇐ OPTOP - 8
```

Duplicate the top two operand stack words and put three words down in the stack.

Format

dup2_x1

Forms

dup2_x1 = 93 (0x5d)

Stack

…, value3, value2, value1 ⇒
…, value2, value1, value3, value2, value1

Description

dup2_x1 duplicates the two words on the top of the stack and inserts them three words down from the top of the operand stack.

Operation

```
stack[OPTOP - 4] ⇐ stack[OPTOP + 4]
stack[OPTOP] ⇐ stack[OPTOP + 8]
stack[OPTOP + 4] ⇐ stack[OPTOP + 12]
stack[OPTOP + 8] ⇐ stack[OPTOP - 4]
stack[OPTOP + 12] ⇐ stack[OPTOP]
OPTOP ⇐ OPTOP - 8
```

Duplicate the top two operand stack words and put four words down in the stack.

Format

dup2_x2

Forms

dup2_x2 = 94 (0x5e)

Stack

…, value4, value3, value2, value1 ⇒
…, value2, value1, value4, value3, value2, value1

Description

dup2_x2 duplicates the two words on the top of the stack and inserts them four words down from the top of the operand stack.

Operation

```
stack[OPTOP - 4] ⇐ stack[OPTOP + 4]
stack[OPTOP] ⇐ stack[OPTOP + 8]
stack[OPTOP + 4] ⇐ stack[OPTOP + 12]
stack[OPTOP + 8] ⇐ stack[OPTOP + 16]
stack[OPTOP + 12] ⇐ stack[OPTOP - 4]
stack[OPTOP + 16] ⇐ stack[OPTOP]
OPTOP ⇐ OPTOP - 8
```

Jump to the return code for a synchronized method.

Format

```
exit_sync_method
```

Forms

`exit_sync_method` = 236 (0xec)

Stack

… ⇒
…

Description

`exit_sync_method` branches to the address stored in the stack location at FRAME – 20. It does not modify the stack.

Operation

```
PC ⇐ stack[FRAME - 20]
```

Notes

`exit_sync_method` supports synchronized methods on the picoJava-II core. For details, see *Invoking a Synchronized Method* on page 391.

Convert a float to a double.

Format

f2d

Forms

f2d = 141 (0x8d)

Stack

…, value ⇒
…, result<31:0>, result<63:32>

Description

f2d treats the value on the top of the operand stack as the type float, then pops it from the operand stack and converts it to a double, which it pushes onto the operand stack.

If the floating point unit is not enabled (PSR.FPE = 0), f2d traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP], stack[OPTOP + 4]) ⇐
                        convert_f2d(float(stack[OPTOP + 4]))
            OPTOP ⇐ OPTOP - 4
else
            trap f2d (type = 0x8d)
```

Convert a float to an integer.

Format

f2i

Forms

`f2i` = 139 (0x8b)

Stack

…, value ⇒
…, result

Description

`f2i` treats the value on the top of the operand stack (*value*) as the type float, pops it from the operand stack, and converts it to an integer. It then pushes the result (*result)* onto the operand stack.

If *value* is NaN, then *result* is the integer 0. If *value* is not an infinity, then `f2i` rounds it to an integer toward zero with the IEEE 754 round-toward-zero mode. If that integer is a 32-bit integer, then it becomes *result*. Otherwise, either of the following conditions applies:

- *value* must be too small (a negative value of large magnitude or negative infinity) and *result* is –2147483648 (0x8000000)
- *value* must be too large (a positive value of large magnitude or positive infinity) and *result* is 2147483647 (0x7fffffff).

If the floating point unit is not enabled (`PSR.FPE` = 0), then `f2i` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            stack[OPTOP + 4] ⇐ convert_f2i(float(stack[OPTOP + 4]))
else
            trap f2i (type = 0x8b)
```

Notes

result may lose precision and information about the overall magnitude of *value*.

Convert a float to a long.

Format

f2l

Forms

`f2l` = 140 (0x8c)

Stack

…, value ⇒
…, result<31:0>, result<63:32>

Description

`f2l` treats the value on the top of the operand stack (*value*) as the type float, pops it from the operand stack, and converts it to a long integer. It then pushes the result (*result)* onto the operand stack.

If *value* is NaN, then *result* is the long integer 0. If *value* is not an infinity, then `f2l` rounds it to a long integer toward zero with the IEEE 754 round-toward-zero mode. If that long integer is a 64-bit long integer, then it becomes *result*. Otherwise, either of the following conditions applies:

- *value* must be too small (a negative value of large magnitude or negative infinity) and *result* is –9223372036854775808 (0x800000000000000)
- *value* must be too large (a positive value of large magnitude or positive infinity) and *result* is 9223372036854775807 (0x7fffffffffffffff).

If the floating point unit is not enabled (`PSR.FPE` = 0), `f2l` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            long(stack[OPTOP],stack[OPTOP + 4]) ⇐
                              convert_f2l(float(stack[OPTOP + 4])
else
            trap f2l (type = 0x8c)
```

Notes

result may lose precision and information about the overall magnitude of *value*.

Add two floats.

Format

fadd

Forms

fadd = 98 (0x62)

Stack

…, value1, value2 ⇒
…, result

Description

fadd treats both *value1* and *value2* as the type float and pops them from the operand stack. It then pushes *result*, which is a float and the sum of *value1* + *value2*, onto the operand stack.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not, then *result* is *value2* (NaN) with a positive sign.
- The sum of two infinities of opposite signs is NaN (0x7fff0000).
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite signs is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite signs is positive zero.
- In the remaining cases, where an infinity, a zero, or NaN is not involved and where the values have the same sign or different magnitudes, the sum is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a float, the operation overflows; *result* is then an infinity of the appropriate sign. If the magnitude is too small to be represented as a float, the operation underflows; *result* is then a zero of the appropriate sign.

If the floating point unit is not enabled (PSR.FPE = 0), then fadd traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            float(stack[OPTOP + 8]) ⇐
                                float(stack[OPTOP + 8]) + float(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP + 4
else
            trap fadd (type = 0x62)
```

Load a float from an array.

Format

faload

Forms

faload = 48 (0x30)

Stack

…, *arrayref*, *index* ⇒
…, *value*

Description

faload treats *arrayref* as a reference to an array of floats. It loads the one-word element at *index* and pushes it onto the stack as *value*.

If *arrayref* is null, then faload takes a NullPointer trap. If *index* is not within the bounds of the array that *arrayref* references, then faload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ mem[addr_of_length + 4 + (index × 4)]
OPTOP ⇐ OPTOP + 4
```

Notes

faload is identical to iaload and aaload.

Store a float to an array.

Format

fastore

Forms

`fastore` = 81 (0x51)

Stack

…, arrayref, index, value⇒
…

Description

`fastore` treats *arrayref* as a reference to an array of floats. It stores the float *value* on the stack to the one-word element at *index* of the array.

If *arrayref* is `null`, then `fastore` takes a `NullPointer` trap. If *index* is not within the bounds of the array that *arrayref* references, then `fastore` takes an `ArrayIndexOutOfBounds` trap.

Operation

```
arrayref ⇐ stack[OPTOP + 12]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 8]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem[addr_of_length + 4 + (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 12
```

Notes

`fastore` is identical to `iastore`.

Compare two floats with greater than on NaN.

Format

fcmpg

Forms

fcmpg = 150 (0x96)

Stack

…, value1, value2 ⇒
…, result

Description

fcmpg treats both *value1* and *value2* as the type float, pops them from the operand stack, then performs a floating-point comparison and executes as follows:

- If *value1* is greater than *value2*, then fcmpg pushes the integer value 1 onto the operand stack.
- If *value1* is equal to *value2*, then fcmpg pushes the integer value 0 onto the operand stack.
- If *value1* is less than *value2*, then fcmpg pushes the integer value –1 onto the operand stack.
- If either *value1* or *value2* is a NaN, then fcmpg pushes the integer value 1 onto the operand stack.

fcmpg performs floating-point comparisons according to IEEE 754 rules: It orders all values other than NaN with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

If the floating point unit is not enabled (PSR.FPE = 0), fcmpg traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            diff = float(stack[OPTOP + 8]) - float(stack[OPTOP + 4])
            if (diff < 0.0) then
                        stack[OPTOP + 8] ⇐ -1
            else if ((diff > 0.0) OR (diff = NaN)) then
                        stack[OPTOP + 8] ⇐ 1
            else
                        stack[OPTOP + 8] ⇐ 0
            OPTOP ⇐ OPTOP + 4
else
            trap fcmpg (type = 0x96)
```

Notes

fcmpg and fcmpl differ only in how they compare values that involve NaN.

Compare two floats with less than on NaN.

Format

fcmpl

Forms

`fcmpl` = 149 (0x95)

Stack

…, value1, value2 ⇒
…, result

Description

`fcmpl` treats both *value1* and *value2* as the type float, pops them from the operand stack, then performs a floating-point comparison and executes as follows:

- If *value1* is greater than *value2*, then `fcmpl` pushes the integer value 1 onto the operand stack.
- If *value1* is equal to *value2*, then `fcmpl` pushes the integer value 0 onto the operand stack.
- If *value1* is less than *value2*, then `fcmpl` pushes the integer value –1 onto the operand stack.
- If either *value1* or *value2* is a NaN, then `fcmpl` pushes the integer value –1 onto the operand stack.

`fcmpl` performs floating-point comparisons according to IEEE 754 rules: It orders all values other than NaN with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

If the floating point unit is not enabled (`PSR.FPE` = 0), `fcmpl` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            diff = float(stack[OPTOP + 8]) - float(stack[OPTOP + 4])
            if ((diff < 0.0) OR (diff = NaN)) then
                        stack[OPTOP + 8] ⇐ -1
            else if (diff > 0.0) then
                        stack[OPTOP + 8] ⇐ 1
            else
                        stack[OPTOP + 8] ⇐ 0
            OPTOP ⇐ OPTOP + 4
else
            trap fcmpl (type = 0x95)
```

Notes

`fcmpg` and `fcmpl` differ only in how they compare values that involve NaN.

Push the floating-point constant 0.0.

Format

fconst_0

Forms

`fconst_0` = 11 (0x0b)

Stack

…⇒
…, `0`

Description

`fconst_0` pushes the float constant 0.0 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 0
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, `fconst_0` is equivalent to `iconst_0` and `aconst_null`.

Push the float constant 1.0.

Format

```
fconst_1
```

Forms

`fconst_1` = 12 (0x0c)

Stack

… ⇒
…, 0x3f800000

Description

`fconst_1` pushes the float constant 1.0 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 0x3f800000
OPTOP ⇐ OPTOP - 4
```

Push the float constant 2.0.

Format

fconst_2

Forms

fconst_2 = 13 (0x0d)

Stack

… ⇒
…, 0x40000000

Description

fconst_2 pushes the float constant 2.0 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 0x40000000
OPTOP ⇐ OPTOP - 4
```

Divide two floats.

Format

fdiv

Forms

`fdiv` = 110 (0x6e)

Stack

…, value1, value2 ⇒
…, result

Description

`fdiv` treats both *value1* and *value2* as the type float, pops them from the operand stack, and pushes the float *result*, which is *value1* ÷ *value2*, onto the operand stack.

result is governed by the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not, then *result* is *value2* (NaN) with a positive sign.
- The sum of two infinities of opposite signs is NaN (0x7fff0000).
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite signs is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite signs is positive zero.
- In the remaining cases, where an infinity, a zero, or NaN is not involved and where the values have the same sign or different magnitudes, the sum is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a float, the operation overflows; *result* is then an infinity of the appropriate sign. If the magnitude is too small to be represented as a float, the operation underflows; *result* is then a zero of the appropriate sign.

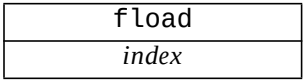
If the Floating Point Unit (FPU) is not enabled (`PSR.FPE` = 0), `fdiv` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            float(stack[OPTOP + 8]) ⇐
                                  float(stack[OPTOP + 8]) ÷ float(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP + 4
else
            trap fdiv (type = 0x6e)
```

Load a float from a local variable.

Format



Forms

`fload` = 23 (0x17)

Stack

… ⇒
…, *value*

Description

`fload` pushes a one-word local variable, which is at *index* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS - (index × 4)]
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, `fload` is identical to `aload` and `iload`.

Load a float from a local variable.

Format

fload_*n*

Forms

fload_0 = 34 (0x22)
fload_1 = 35 (0x23)
fload_2 = 36 (0x24)
fload_3 = 37 (0x25)

Stack

… ⇒
…, *value*

Description

fload_*n* pushes a one-word local variable, which is at *n* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS -(n × 4)]
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, fload_*n* is identical to aload_*n* and iload_*n*.

Multiply two floats.

Format

fmul

Forms

fmul = 106 (0x6a)

Stack

…, value1, value2 ⇒
…, result

Description

fmul treats both *value1* and *value2* as the type float and pops them from the operand stack. It then pushes the float *result*, which is *value1* × *value2*, onto the operand stack.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not NaN, then *result* is *value2* (NaN) with a positive sign.
- If neither value is NaN, then the sign of *result* is positive if both values have the same sign, negative if the values have different signs.
- Multiplication of an infinity by a finite value results in a signed infinity according to the sign-producing rule (see two bullets above).
- Multiplication of an infinity by a zero results in NaN (0x7fff0000).
- In the remaining cases, where neither an infinity nor NaN is involved, the product is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a float, the operation overflows; *result* is then an infinity of appropriate sign. If the magnitude is too small to be represented as a float, the operation underflows; *result* is then a zero of appropriate sign.

If the Floating Point Unit (FPU) is not enabled (PSR.FPE = 0), fmul traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            float(stack[OPTOP + 8]) ⇐
                                 float(stack[OPTOP + 8]) × float(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP + 4
else
            trap fmul (type = 0x6a)
```

Negate a float.

Format

fneg

Forms

fneg = 118 (0x76)

Stack

…, *value* ⇒
…, *result*

Description

dneg treats *value* as the type float and pops it from the operand stack. The float *result* is –*value*, which is the arithmetic negation of *value*. fneg then pushes *result* onto the operand stack.

For float values, negation is different from subtraction from zero. If *x* is +0.0, then 0.0 – *x* equals +0.0, but –*x* equals –0.0. Unary minus merely inverts the sign of a float.

Also note the following rules:

- If the operand is NaN, then *result* is NaN; NaN has no sign.
- If the operand is an infinity, then *result* is the infinity of the opposite sign.
- If the operand is a zero, then *result* is the zero of the opposite sign.

In practice, negating a float precision value is simply inverting the sign bit – bit 31.

Operation

```
stack[OPTOP + 4] ⇐ stack[OPTOP + 4] XOR 0x80000000
```

Notes

In the picoJava-II core, fneg is identical to dneg.

Compute the remainder of two floats.

Format

frem

Forms

`frem` = 114 (0x72)

Stack

…, value1, value2 ⇒
…, result

Description

`frem` treats both *value1* and *value2* as the type float and pops them from the operand stack. It then computes *result* and pushes it onto the operand stack as a float.

result is not the same as that of the remainder operation defined by IEEE 754, which computes the remainder from a rounding division, *not* a truncating division, causing a different behavior from that of the usual integer remainder operator. `frem` operates in the manner specified by the Java virtual machine.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not NaN, then *result* is *value2* (NaN) with a positive sign.
- If neither value is NaN, then the sign of *result* equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, then *result* is NaN (0x7fff0000).
- If the dividend is finite and the divisor is an infinity, then the result equals the dividend.
- If the dividend is a zero and the divisor is finite, then the result equals the dividend.
- In the remaining cases, where an infinity, a zero, or NaN is not involved, the floating-point remainder *result* from a dividend *value1* and a divisor *value2* is defined by the mathematical relation *result* = *value1* − (*value2* × *q*), where *q* is an integer that is negative only if *value1* ÷ *value2* is negative and positive only if *value1* ÷ *value2* is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of *value1* and *value2*.

If the floAting Point Unit (FPU) is not enabled (`PSR.FPE` = 0), `frem` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            float(stack[OPTOP + 8]) ⇐
                        float(stack[OPTOP + 8]) % float(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP + 4
else
            trap frem (type = 0x72)
```

Return a float from a method.

Format

freturn

Forms

freturn = 174 (0xae)

Stack

…, value ⇒
[empty]

Description

freturn returns to the caller of this method, popping all the arguments to the current method and pushing the float that is at the top of the operand stack onto the top of the caller's operand stack.

Operation

```
PC ⇐ stack[FRAME]
CONST_POOL ⇐ stack[FRAME - 12]
ret_value_word1 ⇐ stack[OPTOP + 4]
VARS ⇐ stack[FRAME - 4]
FRAME ⇐ stack[FRAME - 8]
OPTOP ⇐ VARS + 4
stack[OPTOP + 4] ⇐ ret_value_word1
```

Notes

In the picoJava-II core, freturn is identical to ireturn and areturn.

Store a float to a local variable.

Format

fstore
index

Forms

`fstore` = 56 (0x38)

Stack

…, *value* ⇒
…

Description

`fstore` stores the float on the top of the operand stack into a one-word local variable, which is at *index* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS - (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, `fstore` is identical to `istore` and `astore`.

Store a float to a local variable.

Format

fstore_*n*

Forms

fstore_0 = 67 (0x43)
fstore_1 = 68 (0x44)
fstore_2 = 69 (0x45)
fstore_3 = 70 (0x46)

Stack

…, *value* ⇒
…

Description

fstore_*n* stores the float on the top of the operand stack into a one-word local variable, which is at *n* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS - (n × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, fstore_*n* is identical to istore_*n* and astore_*n*.

Subtract two floats.

Format

fsub

Forms

fsub = 102 (0x66)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

fsub treats both *value1* and *value2* as the type float and pops them from the operand stack. It then pushes the float *result*, which is *value1* – *value2*, onto the operand stack.

result is subject to the rules of IEEE arithmetic, as follows:

- If *value1* is NaN, then *result* is *value1* (NaN) with a positive sign.
- If *value2* is NaN but *value1* is not NaN, then *result* is *value2* (NaN) with a positive sign.
- The difference of two infinities of the same sign is NaN (0x7fff0000).
- The difference of two infinities of opposite signs is *value1*.
- The difference of an infinity and any finite value is equal to the infinity.
- The difference of two zeroes of opposite signs is *value1*.
- The difference of two zeroes of the same sign is positive zero.
- If *value1* is a zero and *value2* is a nonzero finite value, then *result* is *value2* with the opposite sign.
- If *value1* is a nonzero finite value and *value2* is a zero, then *result* is *value1*.
- The difference of two nonzero finite values of the same magnitude and same sign is positive zero.
- In the remaining cases, where an infinity, a zero, or NaN is not involved and where the values have the same sign or different magnitudes, the sum is rounded to the nearest representable value with the IEEE 754 round-to-nearest mode. If the magnitude is too large to be represented as a float, the operation overflows; *result* is then an infinity of the appropriate sign. If the magnitude is too small to be represented as a float, the operation underflows; *result* is then a zero of the appropriate sign.

If the Floating Point Unit (FPU) is not enabled (PSR.FPE = 0), fsub traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            float(stack[OPTOP + 8]) ⇐
                                    float(stack[OPTOP + 8]) – float(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP + 4
else
            trap fsub (type = 0x66)
```

Push the class pointer for the current method.

Format

extend
get_current_class

Forms

`extend` = 255 (0xff)
`get_current_class` = 55 (0x37)

Stack

… ⇒
…, *classref*

Description

`get_current_class` returns the class reference for the current method. For example, it determines which object to lock while entering a static synchronized method.

Operation

```
curr_method ⇐ stack[FRAME - 16]
stack[OPTOP] ⇐ mem[curr_method + 32]
OPTOP ⇐ OPTOP - 4
```

Notes

`get_current_class` supports synchronized methods on the picoJava-II core. For details, see *Invoking a Synchronized Method* on page 391.

Trap to emulation routine that resolves the constant pool and reads a field in an object.

Format

getfield
indexbyte1
indexbyte2

Forms

`getfield` = 180 (0xb4)

Stack

…, *objectref* ⇒
…, *value* or …, *value<31:0>*, *value<63:32>*

Description

`getfield` traps to the emulation routine referenced by entry 0xb4 in the trap table.

Operation

```
trap getfield (type = 0xb4)
```

Recommendations

The trap handler should emulate `getfield`, as defined in *The Java Virtual Machine Specification*.

When the constant pool entry referenced by `getfield` is resolved, the `getfield` trap handler computes the offset for the field it references and determines the field type which, along with the size of the offset, in turn determines whether a `getfield_quick`, `getfield_quick_w`, `getfield2_quick`, or `agetfield_quick` opcode byte should replace the original `getfield` opcode byte.

If the `getfield` operates on a field determined dynamically to have an offset into the class instance data that corresponds to a one-word field that is of the type reference, the `getfield` trap handler should replace the `getfield` instruction with `agetfield_quick`. Otherwise, if the offset into the object is less than or equal to 255 words, the `getfield` instruction should be replaced with `getfield_quick` or `getfield2_quick` if the field is one or two words in size, respectively. Finally, if the offset is larger than 255 words, then the `getfield` trap handler should replace the `getfield` with `getfield_quick_w`.

Read a one-word field from an object.

Format

getfield_quick
index
<*unused*>

Forms

`getfield_quick` = 206 (0xce)

Stack

…, *objectref* ⇒
…, *value*

Description

`getfield_quick` pops *objectref*, which must be a reference type, from the operand stack. It then reads *value*, which must be one word in size, from the offset *index* into the class instance referenced by *objectref* and pushes it onto the stack.

If *objectref* is `null`, then `getfield_quick` throws `NullPointerException`.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref = 0) then
           trap NullPointer (type = 0x1b)
handle_bit ⇐ objectref & 0x00000001
if (handle_bit = 1) then
           addr_of_fields ⇐ mem[(objectref & 0x7ffffffc) + 4]
else
           addr_of_fields ⇐ (objectref & 0x7ffffffc) + 4
stack[OPTOP + 4] ⇐ mem[addr_of_fields + (index × 4)]
```

Trap to emulation routine that reads a field in an object, with a wide index.

Format

getfield_quick_w
indexbyte1
indexbyte2

Forms

getfield_quick_w = 227 (0xe3)

Stack

…, *objectref* ⇒
…, *value* or …, *value<31:0>*, *value<63:32>*

Description

getfield_quick_w traps to the emulation routine referenced by entry 0xe3 in the trap table.

Operation

```
trap getfield_quick_w (type = 0xe3)
```

Recommendations

The trap handler should emulate getfield_quick_w, as defined in *The Java Virtual Machine Specification*. The getfield_quick_w trap handler can perform the required load quickly because the constant pool entry should already have been resolved by the getfield trap handler.

Read a two-word field from an object.

Format

getfield2_quick
index
<*unused*>

Forms

`getfield2_quick` = 208 (0xd0)

Stack

…, objectref ⇒
…, value<31:0>, value<63:32>

Description

`getfield2_quick` pops *objectref*, which must be of the type `reference`, from the operand stack. It then reads *value*, which must be two words in size, from the offset *index* into the class instance referenced by *objectref* and pushes it onto the stack.

If *objectref* is `null`, then `getfield2_quick` throws `NullPointerException`.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref = 0) then
           trap NullPointer (type = 0x1b)
handle_bit ⇐ objectref & 0x00000001
if (handle_bit = 1) then
           addr_of_fields ⇐ mem[(objectref & 0x7ffffffc) + 4]
else
           addr_of_fields ⇐ (objectref & 0x7ffffffc) + 4
stack[OPTOP] ⇐ mem[addr_of_fields + (index × 4)]
stack[OPTOP + 4] ⇐ mem[addr_of_fields + (index × 4) + 4]
```

Trap to emulation routine that resolves constant pool entry and reads a static field in a class.

Format

getstatic
indexbyte1
indexbyte2

Forms

`getstatic` = 178 (0xb2)

Stack

… ⇒
…, *value* or …, *value<31:0>*, *value<63:32>*

Description

`getstatic` traps to the emulation routine referenced by entry 0xb2 in the trap table.

Operation

```
trap getstatic (type = 0xb2)
```

Recommendations

The trap handler should emulate `getstatic`, as defined in *The Java Virtual Machine Specification*.

When the constant pool entry referenced by `getstatic` is resolved, the `getstatic` trap handler stores the address for the field it references into the constant pool. Depending on the type of the static field, a `getstatic_quick`, `getstatic2_quick`, or `agetstatic_quick` opcode byte should replace the original `getstatic` opcode byte.

If `getstatic` operates on a field determined dynamically to correspond to a one-word field that is of the type reference, the `getstatic` trap handler should replace the `getstatic` instruction with `agetstatic_quick`. Otherwise, the `getstatic` instruction should be replaced with `getstatic_quick` or `getstatic2_quick` if the field is one or two words in size, respectively.

Read a one-word static field in a class.

Format

getstatic_quick
indexbyte1
indexbyte2

Forms

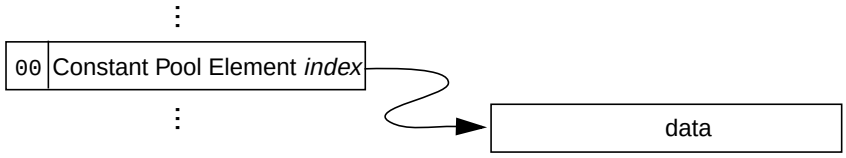
`getstatic_quick` = 210 (0xd2)

Stack

… ⇒
…, *value*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a class (`static`) field. `getstatic_quick` reads the value of that class field and pushes it onto the stack as *value*.



Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
addr_of_static ⇐ mem[CONST_POOL + (index × 4)] & 0x7fffffff
stack[OPTOP] ⇐ mem[addr_of_static]
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, `getstatic_quick` is identical to `agetstatic_quick`. The distinction allows future implementations of `agetstatic_quick` to differ in garbage collection events.

Read a two-word `static` field in a class.

Format

`getstatic2_quick`
indexbyte1
indexbyte2

Forms

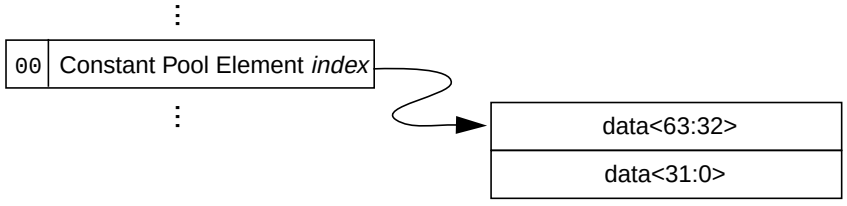
`getstatic2_quick` = 212 (0xd4)

Stack

… ⇒
…, *value.word1*, *value.word2*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a two-word class (`static`) field. `getstatic2_quick` reads the value of that class field and pushes it onto the stack as *value*.



Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
addr_of_static ⇐ mem[CONST_POOL + (index × 4)] & 0x7fffffff
stack[OPTOP] ⇐ mem[addr_of_static + 4]
stack[OPTOP - 4] ⇐ mem[addr_of_static]
OPTOP ⇐ OPTOP - 8
```

Perform an unconditional branch to a 16-bit signed offset.

Format

goto
offsetbyte1
offsetbyte2

Forms

goto = 167 (0xa7)

Stack

… ⇒
…

Description

goto branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*. It does not modify the stack.

Operation

offset ⇐ $sign\_ext_{16}$((*offsetbyte1* << 8) | *offsetbyte2)*
PC ⇐ PC + offset

Perform an unconditional branch to a 32-bit signed offset.

Format

goto_w
offsetbyte1
offsetbyte2
offsetbyte3
offsetbyte4

Forms

goto_w = 200 (0xc8)

Stack

… ⇒
…

Description

goto_w branches to the signed 32-bit offset formed by *offsetbyte1*, *offsetbyte2*, *offsetbyte3*, and *offsetbyte4*. It does not modify the stack.

Operation

```
offset ⇐ (offsetbyte1 << 24) | (offsetbyte2 << 16) | (offsetbyte3 << 8) | offsetbyte4
PC ⇐ PC + offset
```

Convert an integer to a byte.

Format

i2b

Forms

`i2b` = 145 (0x91)

Stack

…, *value* ⇒
…, *result*

Description

`i2b` truncates the word on the top of the stack to its least significant 8 bits, then sign-extends it back to a 32-bit value.

Operation

`stack[OPTOP + 4]` ⇐ $\texttt{sign\_ext}_8$`(stack[OPTOP + 4] & 0x000000ff)`

Notes

result may lose information that pertains to the overall magnitude of *value*. Also, it may not have the same sign as *value*.

Convert an integer to an unsigned short (char).

Format

i2c

Forms

`i2c` = 146 (0x92)

Stack

…, value ⇒
…, result

Description

`i2c` truncates the word on the top of the stack to its least significant 16 bits.

Operation

```
stack[OPTOP + 4] ⇐ stack[OPTOP + 4] & 0x0000ffff
```

Notes

result may lose information that pertains to the overall magnitude of *value*. Also, it may not have the same sign as *value*.

Convert an integer to a double.

Format

i2d

Forms

i2d = 135 (0x87)

Stack

…, *value* ⇒
…, *result*<31:0>, *result*<63:32>

Description

The integer, *value*, on the top of the stack is converted to a double-precision, floating-point *result*, using IEEE 754 round-to-nearest mode.

If the Floating Point Unit (FPU) is not enabled (PSR.FPE = 0), then i2d traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP], stack[OPTOP + 4]) ⇐
                              convert_i2d(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP - 4
else
            trap i2d (type = 0x87)
```

Convert an integer to a float.

Format

i2f

Forms

`i2f` = 134 (0x86)

Stack

…, value ⇒
…, result

Description

The integer, *value*, on the top of the stack is converted to a single-precision, floating-point *result*, using IEEE 754 round-to-nearest mode.

If the Floating Point Unit (FPU) is not enabled (`PSR.FPE` = 0), then `i2f` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            float(stack[OPTOP + 4]) ⇐
                                convert_i2f(stack[OPTOP + 4])
            OPTOP ⇐ OPTOP - 4
else
            trap i2f (type = 0x86)
```

Notes

result may lose precision.

Convert an integer to a long integer.

Format

i2l

Forms

`i2l` = 133 (0x85)

Stack

…, value ⇒
…, result<31:0>, result<63:32>

Description

`i2l` sign-extends the word on the top of the stack to a long integer.

Operation

```
if (stack[OPTOP + 4] < 0) then
            stack[OPTOP] ⇐ 0xffffffff
else
            stack[OPTOP] ⇐ 0x00000000
OPTOP ⇐ OPTOP - 4
```

Convert an integer to a signed short.

Format

i2s

Forms

`i2s` = 147 (0x93)

Stack

…, *value* ⇒
…, *result*

Description

`i2s` truncates the word on the top of the stack to its least significant 16 bits and sign-extends it back to a 32-bit value.

Operation

`stack[OPTOP + 4]` ⇐ $\texttt{sign\_ext}_{16}$`(stack[OPTOP + 4] & 0x0000ffff)`

Notes

result may lose information that pertains to the overall magnitude of *value*. Also, it may not have the same sign as *value*.

Add two integers.

Format

iadd

Forms

`iadd` = 96 (0x60)

Stack

…, value1, value2 ⇒
…, result

Description

`iadd` treats both *value1* and *value2* as integers and pops them from the operand stack. It then pushes *result*, which is an integer and the result of *value1* + *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] + stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Load an integer from an array.

Format

iaload

Forms

iaload = 46 (0x2e)

Stack

…, arrayref, index ⇒
…, value

Description

iaload treats the stack entry *arrayref* as a reference to an array of one-word elements, and it treats the stack entry *index* as a signed 32-bit integer. It returns the element at *index* of the array.

If *arrayref* is null, then iaload takes a NullPointer trap. If *index* is not within the bounds of the array referenced by *arrayref*, then iaload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ mem[addr_of_length + 4 + (index × 4)]
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, iaload is identical to aaload and faload.

Compute the bitwise AND of two integers.

Format

iand

Forms

`iand` = 126 (0x7e)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

`iand` treats both *value1* and *value2* as integers and pops them from the operand stack. It then pushes *result*, which is the bitwise AND of *value1* and *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] & stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Store an integer to an array.

Format

iastore

Forms

`iastore` = 79 (0x4f)

Stack

…, arrayref, index, value ⇒
…

Description

`iastore` treats *arrayref* as a reference to an array of one-word elements. It stores the integer *value* on the stack to the one-word element at *index* of the array.

If *arrayref* is `null`, then `iastore` takes a `NullPointer` trap. If *index* is not within the bounds of the array referenced by *arrayref*, then `iastore` takes an `ArrayIndexOutOfBounds` trap.

Operation

```
arrayref ⇐ stack[OPTOP + 12]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 8]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem[addr_of_length + 4 + (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 12
```

Notes

In the picoJava-II core, `iastore` is identical to `fastore`.

Push the integer constant –1.

Format

iconst_m1

Forms

`iconst_m1` = 2 (0x02)

Stack

…⇒
…, `-1`

Description

`iconst_m1` pushes the integer constant –1 onto the stack.

Operation

```
stack[OPTOP] ⇐ -1
OPTOP ⇐ OPTOP - 4
```

Push the integer constant 0.

Format

iconst_0

Forms

`iconst_0` = 3 (0x03)

Stack

…⇒
…, `0`

Description

`iconst_0` pushes the integer constant 0 onto the stack.

Operation

```
stack[OPTOP] ⇐ 0
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, `iconst_0` is equivalent to `fconst_0` and `aconst_null`.

Push the integer constant 1.

Format

iconst_1

Forms

`iconst_1` = 4 (0x04)

Stack

…⇒
…, `1`

Description

`iconst_1` pushes the integer constant 1 onto the stack.

Operation

```
stack[OPTOP] ⇐ 1
OPTOP ⇐ OPTOP - 4
```

Push the integer constant 2.

Format

```
iconst_2
```

Forms

`iconst_2` = 5 (0x05)

Stack

…⇒
…, 2

Description

`iconst_2` pushes the integer constant 2 onto the stack.

Operation

```
stack[OPTOP] ⇐ 2
OPTOP ⇐ OPTOP - 4
```

Push the integer constant 3.

Format

iconst_3

Forms

`iconst_3` = 6 (0x06)

Stack

…⇒
…, 3

Description

`iconst_3` pushes the integer constant 3 onto the stack.

Operation

```
stack[OPTOP] ⇐ 3
OPTOP ⇐ OPTOP - 4
```

Push the integer constant 4.

Format

iconst_4

Forms

iconst_4 = 7 (0x07)

Stack

…⇒
…, 4

Description

iconst_4 pushes the integer constant 4 onto the stack.

Operation

```
stack[OPTOP] ⇐ 4
OPTOP ⇐ OPTOP - 4
```

Push the integer constant 5.

Format

iconst_5

Forms

`iconst_5` = 8 (0x08)

Stack

…⇒
…, 5

Description

`iconst_5` pushes the integer constant 5 onto the stack.

Operation

```
stack[OPTOP] ⇐ 5
OPTOP ⇐ OPTOP - 4
```

Divide two integers.

Format

idiv

Forms

`idiv` = 108 (0x6c)

Stack

…, value1, value2 ⇒
…, result

Description

`idiv` treats both *value1* and *value2* as integers and pops them from the operand stack. It then pushes *result*, which is an integer and the result of the expression of *value1* ÷ *value2*, onto the operand stack.

If *value2* is 0, then `idiv` gene`rates an ArithmeticException` trap.

Operation

```
if (stack[OPTOP + 4] = 0)
            trap ArithmeticException (type 0x16)
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] ÷ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Compare two references and branch to the 16-bit signed offset if they are equal.

Format

if_acmpeq
offsetbyte1
offsetbyte2

Forms

if_acmpeq = 165 (0xa5)

Stack

…, *value1*, *value2* ⇒
…

Description

if_acmpeq compares the object reference *value1* and *value2* for equality, considering only bits <29:2>. If they are equal, then if_acmpeq branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8] & 0x7ffffffc
value2 ⇐ stack[OPTOP + 4] & 0x7ffffffc
OPTOP ⇐ OPTOP + 8
if (value1 = value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare two references and branch to the 16-bit signed offset if they are not equal.

Format

if_acmpne
offsetbyte1
offsetbyte2

Forms

`if_acmpne` = 166 (0xa6)

Stack

…, value1, value2 ⇒
…

Description

`if_acmpne` compares the object reference *value1* and *value2* for equality, considering only bits <29:2>. If they are not equal, then `if_acmpne` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8] & 0x7ffffffc
value2 ⇐ stack[OPTOP + 4] & 0x7ffffffc
OPTOP ⇐ OPTOP + 8
if (value1 ≠ value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare two integers and branch to the 16-bit signed offset if they are equal.

Format

if_icmpeq
offsetbyte1
offsetbyte2

Forms

`if_icmpeq` = 159 (0x9f)

Stack

…, *value1*, *value2* ⇒
…

Description

`if_icmpeq` compares the integers *value1* and *value2* for equality. If they are equal, then `if_icmpeq` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8]
value2 ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
if (value1 = value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare two integers and branch to the 16-bit signed offset if greater than or equal.

Format

if_icmpge
offsetbyte1
offsetbyte2

Forms

if_icmpge = 162 (0xa2)

Stack

…, value1, value2 ⇒
…

Description

if_icmpge compares the integers *value1* and *value2*. If *value1* is greater than or equal to *value2*, then if_icmpge branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8]
value2 ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
if (value1 ≥ value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare two integers and branch to the 16-bit signed offset if greater than.

Format

if_icmpgt
offsetbyte1
offsetbyte2

Forms

`if_icmpgt` = 163 (0xa3)

Stack

…, *value1*, *value2* ⇒
…

Description

`if_icmpgt` compares the integers *value1* and *value2*. If *value1* is greater than *value2*, then `if_icmpgt` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8]
value2 ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
if (value1 > value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare two integers and branch to the 16-bit signed offset if less than or equal.

Format

if_icmple
offsetbyte1
offsetbyte2

Forms

`if_icmple` = 164 (0xa4)

Stack

…, *value1*, *value2* ⇒
…

Description

`if_icmple` compares the integers *value1* and *value2*. If *value1* is less than or equal to *value2*, then `if_icmple` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8]
value2 ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
if (value1   value2) then
           offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
           PC ⇐ PC + offset
```

Compare two integers and branch to the 16-bit signed offset if less than.

Format

if_icmplt
offsetbyte1
offsetbyte2

Forms

`if_icmplt` = 161 (0xa1)

Stack

…, *value1*, *value2* ⇒
…

Description

`if_icmpgt` compares the integers *value1* and *value2*. If *value1* is less than *value2*, then `if_icmpgt` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8]
value2 ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
if (value1 < value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare two integers and branch to the 16-bit signed offset if they are not equal.

Format

if_icmpne
offsetbyte1
offsetbyte2

Forms

`if_icmpne` = 160 (0xa0)

Stack

…, value1, value2 ⇒
…

Description

`if_icmpne` compares the integers *value1* and *value2* for equality. If they are not equal, then `if_icmpne` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value1 ⇐ stack[OPTOP + 8]
value2 ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
if (value1 ≠ value2) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare an integer to zero and branch to the 16-bit signed offset if they are equal.

Format

ifeq
offsetbyte1
offsetbyte2

Forms

`ifeq` = 153 (0x99)

Stack

…, *value* ⇒
…

Description

`ifeq` compares the integer *value* to zero. If they are equal, then `ifeq` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (value = 0) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Notes

In the picoJava-II core, `ifeq` is identical to `ifnull`.

Compare an integer to zero and branch to the 16-bit signed offset if greater than or equal.

Format

ifge
offsetbyte1
offsetbyte2

Forms

`ifge` = 156 (0x9c)

Stack

…, *value* ⇒
…

Description

`ifge` compares the integer *value* to zero. If *value* is greater than or equal to zero, then `ifge` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (value ≥ 0) then
           offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
           PC ⇐ PC + offset
```

Compare an integer to zero and branch to the 16-bit signed offset if greater than.

Format

ifgt
offsetbyte1
offsetbyte2

Forms

`ifgt` = 157 (0x9d)

Stack

…, *value* ⇒
…

Description

`ifgt` compares the integer *value* to zero. If *value* is greater than zero, then `ifgt` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (value > 0) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare an integer to zero and branch to the 16-bit signed offset if less than or equal.

Format

ifle
offsetbyte1
offsetbyte2

Forms

`ifle` = 158 (0x9e)

Stack

…, *value* ⇒
…

Description

`ifle` compares the integer *value* to zero. If *value* is less than or equal to zero, then `ifle` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (value ≤ 0) then
           offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
           PC ⇐ PC + offset
```

Compare an integer to zero and branch to the 16-bit signed offset if less than.

Format

iflt
offsetbyte1
offsetbyte2

Forms

`iflt` = 155 (0x9b)

Stack

…, *value* ⇒
…

Description

`iflt` compares the integer *value* to zero. If *value* is less than zero, then `iflt` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (value < 0) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Compare an integer to zero and branch to the 16-bit signed offset if they are not equal.

Format

ifne
offsetbyte1
offsetbyte2

Forms

ifne = 154 (0x9a)

Stack

…, *value* ⇒
…

Description

ifne compares the integer *value* to zero. If they are not equal, then ifne branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
value ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (value ≠ 0) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Notes

In the picoJava-II core, ifne is identical to ifnonnull.

Compare an object reference to `null` and branch to the 16-bit signed offset if they are not equal.

Format

ifnonnull
offsetbyte1
offsetbyte2

Forms

`ifnonnull` = 199 (0xc7)

Stack

…, *value* ⇒
…

Description

`ifnonnull` compares the object reference *value* to `null`. If they are not equal, then `ifnonnull` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
objectref ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (objectref ≠ 0) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Notes

In the picoJava-II core, `ifnonnull` is identical to `ifne`.

Compare an object reference to `null` and branch to the 16-bit signed offset if they are equal.

Format

ifnull
offsetbyte1
offsetbyte2

Forms

`ifnull` = 198 (0xc6)

Stack

…, *value* ⇒
…

Description

`ifnull` compares the reference *value* to `null`. If *value* is `null`, then `ifnull` branches to the signed 16-bit offset, which is formed by *offsetbyte1* and *offsetbyte2*.

Operation

```
objectref ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
if (objectref ≠ 0) then
            offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
            PC ⇐ PC + offset
```

Notes

In the picoJava-II core, `ifnull` is identical to `ifeq`.

Increment an integer in a local variable.

Format

iinc
index
const

Forms

iinc = 132 (0x84)

Stack

… ⇒
…

Description

iinc adds an 8-bit signed *const* to a one-word local variable, which is at *index* stack entries offset from the start of the current local variables.

Operation

stack[VARS - (*index* × 4)] ⇐ stack[VARS - (*index* × 4)] + sign_ext_8(*const*)

Load an integer from a local variable.

Format

iload
index

Forms

`iload` = 21 (0x15)

Stack

… ⇒
…, *value*

Description

`iload` pushes a one-word local variable, which is at *index* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS - (index × 4)]
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, `iload` is identical to `aload` and `fload`.

Load an integer from a local variable.

Format

iload_*n*

Forms

`iload_0` = 26 (0x1a)
`iload_1` = 27 (0x1b)
`iload_2` = 28 (0x1c)
`iload_3` = 29 (0x1d)

Stack

… ⇒
…, *value*

Description

`iload_`*n* pushes a one-word local variable, which is at *n* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS -(n × 4)]
OPTOP ⇐ OPTOP - 4
```

Notes

In the picoJava-II core, `iload_`*n* is identical to `aload_`*n* and `fload_`*n*.

Multiply two integers.

Format

imul

Forms

`imul` = 104 (0x68)

Stack

…, value1, value2 ⇒
…, result

Description

`imul` treats both *value1* and *value2* as the type integer and pops them from the operand stack. It then pushes the integer *result*, which is the low 32-bits of *value1* × *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] × stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Negate an integer.

Format

ineg

Forms

`ineg` = 116 (0x74)

Stack

…, *value* ⇒
…, *result*

Description

`ineg` treats *value* as the type integer and pops it from the operand stack. It then pushes *result*, which is also an integer and is the result of 0 – *value*, onto the operand stack.

Operation

```
stack[OPTOP + 4] ⇐ 0 - stack[OPTOP + 4]
```

Trap to emulation routine that resolves the constant pool entry and checks whether an object is of the given type.

Format

instanceof
indexbyte1
indexbyte2

Forms

`instanceof` = 193 (0xc1)

Stack

…, *objectref* ⇒
…, *result*

Description

`instanceof` traps to the emulation routine referenced by entry 0xc1 in the trap table.

Operation

```
trap instanceof (type=0xc1)
```

Recommendations

The trap handler should emulate `instanceof`, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the `instanceof` instruction with the `instanceof_quick` instruction.

Check whether an object is of the given type.

Format

instanceof_quick
indexbyte1
indexbyte2

Forms

`instanceof_quick` = 225 (0xe1)

Stack

…, *objectref* ⇒
…, *result*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item must have already been resolved and must contain a class ID. The word on the top of the stack, *objectref*, is popped from the stack and treated as a reference.

If *objectref* is `null`, then `instanceof_quick` completes by pushing a value of 0 on the stack. If *objectref* is not `null`, then the class ID of *objectref* is compared with the class ID from the constant pool. If the two class IDs are not equal, then the core generates an `instanceof_quick` emulation trap. Otherwise, `instanceof_quick` completes by pushing a value of 1 on the stack.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref ≠ 0) then
            object_header ⇐ mem[objectref & 0x7ffffffc]
            object_class_addr ⇐ (object_header & 0x7ffffff8) - 8
            object_class ⇐ mem[object_class_addr]
            index ⇐ (indexbyte1 << 8) | indexbyte2
            constant_class ⇐ mem[CONST_POOL + (index × 4)]
            if (object_class = constant_class) then
                        stack[OPTOP + 4] ⇐ 1
            else
                        trap instanceof_quick (type=0xe1)
else
            stack[OPTOP + 4] ⇐ 0
```

Recommendations

The trap handler should emulate `instanceof_quick`, as defined in *The Java Virtual Machine Specification*. The trap handler, however, should skip the initial checks performed by the core.

Trap to emulation routine that resolves the constant pool entry and invokes an interface method.

Format

invokeinterface
indexbyte1
indexbyte2
nargs
0

Forms

`invokeinterface` = 185 (0xb9)

Stack

…, *objectref* ⇒
…, *result*

Description

`invokeinterface` traps to the emulation routine referenced by entry 0xb9 in the trap table.

Operation

```
trap invokeinterface (type=0xb9)
```

Recommendations

The trap handler should emulate `invokeinterface`, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the `invokeinterface` instruction with `invokeinterface_quick`.

Trap to emulation routine that invokes an interface method.

Format

invokeinterface_quick
idbyte1
idbyte2
nargs
guess

Forms

`invokeinterface_quick` = 218 (0xda)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

`invokeinterface_quick` traps to the emulation routine referenced by entry 0xda in the trap table.

Operation

```
trap invokeinterface_quick (type=0xda)
```

Recommendations

The trap handler should emulate `invokeinterface_quick`, as defined in *The Java Virtual Machine Specification*.

invokenonvirtual_quick invokenonvirtual_quick

Invoke a method based on a compile time type.

Format

invokenonvirtual_quick
indexbyte1
indexbyte2

Forms

`invokenonvirtual_quick` = 215 (0xd7)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a method structure (see *Method Structure* on page 72). `invokenonvirtual_quick` invokes the method referenced by the method structure pointer, as described in *Invoking a Method* on page 388.

If, based on the number of argument words to be passed to the method, `invokenonvirtual_quick` determines that the object reference forming the first argument is `null`, then `invokenonvirtual_quick` generates a `NullPointer` trap.

Operation

```
index ⇐ (indexbyte1 << 8) | indexbyte2
method_structure ⇐ mem[CONST_POOL + (index × 4)]
newVARS ⇐ OPTOP + mem16[method_structure + 10]
objectref ⇐ stack[newVARS]
if (objectref = 0) then
            trap NullPointer (type=0x1b)
newFRAME ⇐ OPTOP - mem[method_structure + 4]
OPTOP ⇐ newFRAME - 20
stack[newFRAME] ⇐ PC + 3
stack[newFRAME - 4] ⇐ VARS
stack[newFRAME - 8] ⇐ FRAME
stack[newFRAME - 12] ⇐ CONST_POOL
stack[newFRAME - 16] ⇐ method_structure
PC ⇐ mem[method_structure]
VARS ⇐ newVARS
FRAME ⇐ newFRAME
CONST_POOL ⇐ mem[method_structure + 28]
```

Notes

This opcode used to be `invokespecial`, which was dynamically determined to refer to a method dispatched on the basis of compile time type, such as to an initialization method or a private method.

Trap to emulation routine that resolves a constant pool entry and invokes a method based on a compile time type.

Format

invokespecial
indexbyte1
indexbyte2

Forms

`invokespecial` = 183 (0xb7)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

`invokespecial` traps to the emulation routine referenced by entry 0xb7 in the trap table.

Operation

```
trap invokespecial (type=0xb7)
```

Recommendations

The trap handler should emulate `invokespecial`, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the `invokespecial` instruction with either `invokenonvirtual_quick` or `invokesuper_quick`, as appropriate.

Trap to emulation routine that resolves the constant pool entry and invokes a static method.

Format

invokestatic
indexbyte1
indexbyte2

Forms

`invokestatic` = 184 (0xb8)

Stack

…, [*arg1*, ...] ⇒
…

Description

`invokestatic` traps to the emulation routine referenced by entry 0xb8 in the trap table.

Operation

```
trap invokestatic (type=0xb8)
```

Recommendations

The trap handler should emulate `invokestatic`, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the `invokestatic` instruction with `invokestatic_quick`.

Invoke a static method.

Format

invokestatic_quick
indexbyte1
indexbyte2

Forms

`invokestatic_quick` = 217 (0xd9)

Stack

…, [*arg1*, ...] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a method structure (see *Method Structure* on page 72). `invokestatic_quick` invokes the method referenced by the method structure pointer, as described in *Invoking a Method* on page 388.

Operation

```
index ⇐ (indexbyte1 << 8) | indexbyte2
method_descriptor ⇐ mem[CONST_POOL + (index × 4)]
newVARS ⇐ OPTOP + mem16[method_descriptor + 10]
newFRAME ⇐ OPTOP - mem[method_descriptor + 4]
OPTOP ⇐ newFRAME - 20
stack[newFRAME] ⇐ PC + 3
stack[newFRAME - 4] ⇐ VARS
stack[newFRAME - 8] ⇐ FRAME
stack[newFRAME - 12] ⇐ CONST_POOL
stack[newFRAME - 16] ⇐ method_descriptor
PC ⇐ mem[method_descriptor]
VARS ⇐ newVARS
FRAME ⇐ newFRAME
CONST_POOL ⇐ mem[method_descriptor + 28]
```

Invoke a superclass method based on a compile time type.

Format

invokesuper_quick
indexbyte1
indexbyte2

Forms

`invokesuper_quick` = 216 (0xd8)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct a 16-bit virtual method index, where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The core uses this virtual method index as an index into the method vector of the superclass of the currently active method to load a pointer to a method structure (see *Method Structure* on page 72). From the method structure obtained from the superclass, the number of argument bytes are loaded.

Based on the number of argument bytes to be passed to the method, `invokesuper_quick` determines the stack entry that forms the first argument. This stack entry, *objectref*, is treated as a reference type. If *objectref* is `null`, then `invokesuper_quick` generates a `NullPointer` trap. Otherwise, `invokesuper_quick` then invokes the method referenced by the method structure pointer, as described in *Invoking a Method* on page 388.

Operation

```
method_block ⇐ stack[FRAME - 16]
current_class ⇐ mem[method_block + 32]
super_class ⇐ mem[current_class + 36]
super_obj_hint_blk ⇐ mem[super_class + 28]
index ⇐ (indexbyte1 << 8) | indexbyte2
method_descriptor ⇐ mem[(super_obj_hint_blk & 0x3ffffffc) + 32 + (index × 4)]
newVARS ⇐ OPTOP + mem16[method_descriptor + 10]
objectref ⇐ stack[newVARS]
if (objectref = 0) then
            trap NullPointer (type=0x1b)
newFRAME ⇐ OPTOP - mem[method_descriptor + 4]
OPTOP ⇐ newFRAME - 20
stack[newFRAME] ⇐ PC + 3
```

```
stack[newFRAME - 4] ⇐ VARS
stack[newFRAME - 8] ⇐ FRAME
stack[newFRAME - 12] ⇐ CONST_POOL
stack[newFRAME - 16] ⇐ method_descriptor
PC ⇐ mem[method_descriptor]
VARS ⇐ newVARS
FRAME ⇐ newFRAME
CONST_POOL ⇐ mem[method_descriptor + 28]
```

Notes

This opcode used to be `invokespecial`, which was dynamically determined to refer to a superclass method.

Trap to emulation routine that resolves the constant pool entry and invokes a method based on a runtime type.

Format

invokevirtual
indexbyte1
indexbyte2

Forms

`invokevirtual`= 182 (0xb6)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

`invokevirtual` traps to the emulation routine referenced by entry 0xb6 in the trap table.

Operation

```
trap invokevirtual (type=0xb6)
```

Recommendations

The trap handler should emulate `invokevirtual`, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the `invokevirtual` instruction with `invokevirtual_quick` or, if the virtual method index is greater than 255, with `invokevirtual_quick_w`.

Invoke a method based on a runtime type.

Format

invokevirtual_quick
index
nargs

Forms

`invokevirtual_quick` = 214 (0xd6)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

The *index* byte in the instruction is treated as a virtual method index, and *nargs* is treated as the number of argument words to be passed to the method. Based on *nargs*, `invokevirtual_quick` determines the stack entry that forms the first argument. This stack entry, *objectref*, is treated as a reference type. If *objectref* is `null`, then `invokevirtual_quick` generates a `NullPointer` trap. Otherwise, the method vector pointed to by the header of *objectref* is accessed (see *Method Vector and Runtime Class Info Structure* on page 71). The core loads the method structure pointer at the virtual method index in the method vector. `invokevirtual_quick` then invokes the method referenced by this method structure pointer, as described in *Invoking a Method* on page 388.

Operation

```
index ⇐ (indexbyte1 << 8) | indexbyte2
newVARS ⇐ OPTOP + (nargs × 4)
objectref ⇐ stack[newVARS]
if (objectref = 0) then
            trap NullPointer (type=0x1b)
method_descriptor ⇐ mem[(objectref & 0x7ffffffc) + (index × 4)]
newFRAME ⇐ OPTOP - mem[method_descriptor + 4]
OPTOP ⇐ newFRAME - 20
stack[newFRAME] ⇐ PC + 3
stack[newFRAME - 4] ⇐ VARS
stack[newFRAME - 8] ⇐ FRAME
stack[newFRAME - 12] ⇐ CONST_POOL
stack[newFRAME - 16] ⇐ method_descriptor
PC ⇐ mem[method_descriptor]
VARS ⇐ newVARS
FRAME ⇐ newFRAME
CONST_POOL ⇐ mem[method_descriptor + 28]
```

Invoke a method based on a runtime type with wide index.

Format

invokevirtual_quick_w
indexbyte1
indexbyte2

Forms

`invokevirtual_quick_w` = 226 (0xe2)

Stack

…, *objectref*, [*arg1*, ...] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a method structure (see *Method Structure* on page 72). From the method structure obtained from the constant pool, the 16-bit virtual method index and the number of argument bytes are loaded.

Based on the number of argument bytes to be passed to the method, `invokevirtual_quick_w` determines the stack entry that forms the first argument. This stack entry, *objectref*, is treated as a reference type. If *objectref* is null, then `invokevirtual_quick_w` generates a `NullPointer` trap. Otherwise, the method vector pointed to by the header of *objectref* is accessed (see *Method Vector and Runtime Class Info Structure* on page 71). The core loads the method structure pointer at the virtual method index in the method vector. `invokevirtual_quick_w` then invokes the method referenced by this method structure pointer, as described in *Invoking a Method* on page 388.

Operation

```
descr_index ⇐ (indexbyte1 << 8) | indexbyte2
descr ⇐ mem[CONST_POOL + (descr_index × 4)]
index ⇐ OPTOP + mem16[descr + 8]
newVARS ⇐ OPTOP + mem16[descr + 10]
objectref ⇐ stack[newVARS]
if (objectref = 0) then
            trap NullPointer (type=0x1b)
method_descriptor ⇐ mem[(objectref & 0x7ffffffc) + (index × 4)]
newFRAME ⇐ OPTOP - mem[method_descriptor + 4]
OPTOP ⇐ newFRAME - 20
```

```
stack[newFRAME] ⇐ PC + 3
stack[newFRAME - 4] ⇐ VARS
stack[newFRAME - 8] ⇐ FRAME
stack[newFRAME - 12] ⇐ CONST_POOL
stack[newFRAME - 16] ⇐ method_descriptor
PC ⇐ mem[method_descriptor]
VARS ⇐ newVARS
FRAME ⇐ newFRAME
CONST_POOL ⇐ mem[method_descriptor + 28]
```

Bitwise OR of two integers.

Format

ior

Forms

`ior` = 128 (0x80)

Stack

..., value1, value2 ⇒
..., result

Description

`ior` treats both *value1* and *value2* as the type integer and pops them from the operand stack. It then pushes *result*, which is the bitwise OR of *value1* and *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] | stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Remainder of two integers.

Format

irem

Forms

`irem` = 112 (0x70)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

`irem` treats both *value1* and *value2* as the type integer and pops them from the operand stack. It then pushes *result*, which is an integer and remainder of the expression of *value1* ÷ *value2*, onto the operand stack.

If *value2* is 0, then `irem` generates an `ArithmeticException` trap.

Operation

```
if (stack[OPTOP + 4] = 0)
           trap ArithmeticException (type 0x16)
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] % stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Return an integer from a method.

Format

ireturn

Forms

`ireturn` = 172 (0xac)

Stack

…, value ⇒
[empty]

Description

`ireturn` returns to the caller of this method, popping all the arguments to the current method and pushing the integer that is at the top of the operand stack onto the top of the caller's operand stack.

Operation

```
PC ⇐ stack[FRAME]
CONST_POOL ⇐ stack[FRAME - 12]
ret_value_word1 ⇐ stack[OPTOP + 4]
VARS ⇐ stack[FRAME - 4]
FRAME ⇐ stack[FRAME - 8]
OPTOP ⇐ VARS + 4
stack[OPTOP + 4] ⇐ ret_value_word1
```

Notes

In the picoJava-II core, `ireturn` is identical to `freturn` and `areturn`.

Shift-left of an integer.

Format

ishl

Forms

`ishl` = 120 (0x78)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

`ishl` treats both *value1* and *value2* as integers and pops them from the operand stack. It then pushes *result*, which is also an integer and is the result of shifting *value1* left by the number of positions equal to the low 5 bits of *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] << (stack[OPTOP + 4] & 0x1f)
OPTOP ⇐ OPTOP + 4
```

Shift-right of an integer.

Format

ishr

Forms

`ishr` = 122 (0x7a)

Stack

…, value1, value2 ⇒
…, result

Description

`ishr` treats both *value1* and *value2* as integers and pops them from the operand stack. It then pushes *result*, which is also an integer and is the result of shifting *value1* right (with sign extension) by the number of positions equal to the low 5 bits of *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] >> (stack[OPTOP + 4] & 0x1f)
OPTOP ⇐ OPTOP + 4
```

Store an integer to a local variable.

Format

Forms

`istore` = 54 (0x36)

Stack

…, *value* ⇒
…

Description

`istore` stores the integer on the top of the operand stack into a one-word local variable, which is at *index* stack entries offset from the start of the current local variables.

Operation

`stack[VARS - (`*index* × `4)] ⇐ stack[OPTOP + 4]`
`OPTOP ⇐ OPTOP + 4`

Notes

In the picoJava-II core, `istore` is identical to `fstore` and `astore`.

Store an integer to a local variable.

Format

istore_*n*

Forms

istore_0 = 59 (0x3b)
istore_1 = 60 (0x3c)
istore_2 = 61 (0x3d)
istore_3 = 72 (0x3e)

Stack

…, *value* ⇒
…

Description

istore_*n* stores the integer on the top of the operand stack into a one-word local variable, which is at *n* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS - (n × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, istore_*n* is identical to fstore_*n* and astore_*n*.

Subtract two integers.

Format

isub

Forms

`isub` = 100 (0x64)

Stack

…, value1, value2 ⇒
…, result

Description

`isub` treats both *value1* and *value2* as the type integer and pops them from the operand stack. It then pushes *result*, which is an integer and the result of *value1 – value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] - stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Compare unsigned integers.

Format

extend
iucmp

Forms

`extend` = 255 (0xff)
`iucmp` = 21 (0x15)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

`iucmp` compares *value1* and *value2*, the top two stack elements, treating them as unsigned 32-bit integers. It then pops their values from the stack and pushes *result* onto the stack.

If *value1* is greater than *value2*, then the result is 1. If *value1* equals *value2*, then the result is 0. If *value1* is less than *value2*, then the result is –1.

Operation

```
if ((stack[OPTOP+4] > 0) AND (stack[OPTOP+8] < 0)) then
            stack[OPTOP+8] ⇐ 1
else if ((stack[OPTOP+4] < 0) AND (stack[OPTOP+8] > 0)) then
            stack[OPTOP+8] ⇐ -1
else
            diff ⇐ stack[OPTOP + 8] - stack[OPTOP + 4]
            if (diff > 0) then
                        stack[OPTOP+8] ⇐ 1
            else if (diff < 0) then
                        stack[OPTOP+8] ⇐ -1
            else
                        stack[OPTOP+8] ⇐ 0
OPTOP ⇐ OPTOP + 4
```

Unsigned shift-right of an integer.

Format

iushr

Forms

`iushr` = 124 (0x7c)

Stack

…, value1, value2 ⇒
…, result

Description

`ishr` treats both *value1* and *value2* as integers and pops them from the operand stack. It then pushes *result*, which is also an integer and is the result of shifting *value1* right (with zero extension) by the number of positions equal to the low 5 bits of *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] >>> (stack[OPTOP + 4] & 0x1f)
OPTOP ⇐ OPTOP + 4
```

Bitwise XOR of two integers.

Format

```
ixor
```

Forms

`ixor` = 130 (0x82)

Stack

…, value1, value2 ⇒
…, result

Description

`ixor` treats both *value1* and *value2* as the type integer and pops them from the operand stack. It then pushes *result*, which is the bitwise XOR of *value1* and *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 8] ⇐ stack[OPTOP + 8] ^ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Jump to subroutine.

Format

jsr
offsetbyte1
offsetbyte2

Forms

`jsr` = 168 (0xa8)

Stack

… ⇒
…, *return_address*

Description

`jsr` branches to the signed 16-bit offset formed by *offsetbyte1* and *offsetbyte2*. It pushes the PC of the instruction following the `jsr` onto the stack.

Operation

```
stack[OPTOP] ⇐ PC + 3
offset ⇐ sign_ext16((offsetbyte1 << 8) | offsetbyte2)
PC ⇐ PC + offset
OPTOP ⇐ OPTOP - 4
```

Jump to subroutine, with wide offset.

Format

jsr_w
offsetbyte1
offsetbyte2
offsetbyte3
offsetbyte4

Forms

jsr_w = 201 (0xc9)

Stack

… ⇒
…, *return_address*

Description

jsr_w branches to the signed 32-bit offset formed by *offsetbyte1*, *offsetbyte2*, *offsetbyte3*, and *offsetbyte4*. It pushes the PC of the instruction following the jsr_w onto the stack.

Operation

```
stack[OPTOP] ⇐ PC + 5
offset ⇐ (offsetbyte1 << 24) | (offsetbyte2 << 16) | (offsetbyte3 << 8) | offsetbyte4
PC ⇐ PC + offset
OPTOP ⇐ OPTOP - 4
```

Convert a long to a double.

Format

l2d

Forms

l2d = 138 (0x8a)

Stack

…, value<31:0>, value<63:32> ⇒
…, result<31:0>, result<63:32>

Description

l2d treats the value on the top of the operand stack (*value*) as the type long, pops it from the operand stack, and converts it to a double, using the IEEE 754 round-to-nearest mode. l2d then pushes the result (*result)* onto the operand stack.

If the Floating Point Unit (FPU) is not enabled (PSR.FPE = 0), then l2d traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            double(stack[OPTOP + 4],stack[OPTOP + 8]) ⇐
                        convert_l2d(long(stack[OPTOP + 4], stack[OPTOP + 8]))
else
            trap l2d (type = 0x8a)
```

Notes

result may lose precision.

Convert a long to a float.

Format

l2f

Forms

`l2f` = 137 (0x89)

Stack

…, value<31:0>, value<63:32> ⇒
…, result

Description

`l2f` treats the value on the top of the operand stack as the type long, pops it from the operand stack, converts it to a float with the IEEE 754 round-to-nearest mode, then pushes it onto the operand stack.

If the Floating Point Unit (FPU) is not enabled (`PSR.FPE` = 0), then `l2f` traps to an emulation routine.

Operation

```
if ((PSR.FPE = 1) AND (HCR.FPP = 1)) then
            stack[OPTOP + 8] ⇐
                        convert_l2f(long(stack[OPTOP + 4], stack[OPTOP + 8]))
            OPTOP ⇐ OPTOP + 4
else
            trap l2f (type = 0x89)
```

Notes

result may lose precision.

Convert a long to an integer.

Format

l2i

Forms

`l2i` = 136 (0x88)

Stack

…, value<31:0>, value<63:32> ⇒
…, result

Description

`l2i` treats the value on the top of the operand stack (*value*) as the type long, pops it from the operand stack, and converts it to an integer by discarding the upper 32 bits. `l2i` then pushes the result (*result)* onto the operand stack.

Operation

```
OPTOP ⇐ OPTOP + 4
```

Notes

result may lose information about the overall magnitude of *value*. Also, the *result* may not have the same sign as *value*.

In the picoJava-II core, `l2i` is identical to `pop`.

Add two longs.

Format

ladd

Forms

ladd = 97 (0x61)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

ladd treats both *value1* and *value2* as the type long and pops them from the operand stack. It then pushes *result*, which is a long and the sum of *value1* + *value2*, onto the operand stack.

Operation

```
long(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
            long(stack[OPTOP + 12], stack[OPTOP + 16]) +
            long(stack[OPTOP + 4], stack[OPTOP + 8])
OPTOP ⇐ OPTOP + 8
```

Load a long from an array.

Format

laload

Forms

laload = 47 (0x2f)

Stack

…, arrayref, index ⇒
…, value<31:0>, value<63:32>

Description

laload treats *arrayref* as a reference to an array of longs. It loads the two-word element at *index* and pushes it onto the stack as *value*.

If *arrayref* is null, then daload takes a NullPointer trap. If *index* is not within the bounds of the array that *arrayref* references, then daload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ mem[addr_of_length + 8 + (index × 8)]
stack[OPTOP + 4] ⇐ mem[addr_of_length + 4 + (index × 8)]
```

Notes

In the picoJava-II core, laload is identical to daload.

Bitwise AND of two longs.

Format

```
land
```

Forms

`land` = 127 (0x7f)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

`land` treats both *value1* and *value2* as the type long and pops them from the operand stack. It then pushes *result*, which is a long and the bitwise AND of *value1* and *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 12] ⇐ stack[OPTOP + 12] & stack[OPTOP + 4]
stack[OPTOP + 16] ⇐ stack[OPTOP + 16] & stack[OPTOP + 8]
OPTOP ⇐ OPTOP + 8
```

Store a long to an array.

Format

lastore

Forms

lastore = 80 (0x50)

Stack

…, arrayref, index, value<31:0>, value<63:32> ⇒
…

Description

lastore treats *arrayref* as a reference to an array of longs. It stores the double *value* on the stack to the two-word element at *index* of the array.

If *arrayref* is null, then dastore takes a NullPointer trap. If *index* is not within the bounds of the array that *arrayref* references, then dastore takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 16]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 12]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem[addr_of_length + 4 + (index × 8)] ⇐ stack[OPTOP + 4]
mem[addr_of_length + 8 + (index × 8)] ⇐ stack[OPTOP + 8]
OPTOP ⇐ OPTOP + 16
```

Notes

In the picoJava-II core, lastore is identical to dastore.

Compare two longs.

Format

lcmp

Forms

lcmp = 127 (0x7f)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result

Description

lcmp treats both *value1* and *value2* as the type long, pops them from the operand stack, then performs a comparison and executes as follows:

- If *value1* is greater than *value2*, then lcmp pushes the integer value 1 onto the operand stack.
- If *value1* is equal to *value2*, then lcmp pushes the integer value 0 onto the operand stack.
- If *value1* is less than *value2*, then lcmp pushes the integer value –1 onto the operand stack.

Operation

```
diff = long (stack[OPTOP + 16], stack[OPTOP + 12]) -
            long(stack[OPTOP + 8], stack[OPTOP + 4])
if (diff < 0) then
            stack[OPTOP + 16] ⇐ -1
else if (diff > 0) then
            stack[OPTOP + 16] ⇐ 1
else
            stack[OPTOP + 16] ⇐ 0
OPTOP ⇐ OPTOP + 12
```

Push the long constant 0.

Format

lconst_0

Forms

`lconst_0` = 9 (0x09)

Stack

… ⇒
…, 0, 0

Description

Push the long constant 0 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 0
stack[OPTOP - 4] ⇐ 0
OPTOP ⇐ OPTOP - 8
```

Notes

`lconst_0` is identical to `dconst_0`.

Push the long constant 1.

Format

lconst_1

Forms

`lconst_1` = 10 (0x0a)

Stack

… ⇒
…, 1, 0

Description

Push the long constant 1 onto the operand stack.

Operation

```
stack[OPTOP] ⇐ 1
stack[OPTOP - 4] ⇐ 0
OPTOP ⇐ OPTOP - 8
```

Trap to emulation routine that resolves a constant pool item and pushes it onto the stack.

Format

ldc
index

Forms

ldc = 18 (0x12)

Stack

… ⇒
…, *value*

Description

ldc traps to the emulation routine referenced by entry 0x12 in the trap table.

Operation

```
trap ldc (type 0x12)
```

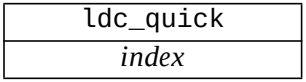
Recommendations

The trap handler should emulate ldc, as defined in *The Java Virtual Machine Specification*.

Once the constant pool entry is resolved, the type of the constant indexed determines whether a ldc_quick or aldc_quick byte should replace the original ldc byte. If the constant is a reference type, the aldc_quick instruction should replace the ldc opcode. Otherwise, the ldc_quick instruction should replace the ldc opcode.

Push a one-word item from the constant pool.

Format



Forms

`ldc_quick` = 203 (0xcb)

Stack

… ⇒
…, *value*

Description

The unsigned *index* byte is an index into the constant pool of the current class. The constant pool item should be a constant that has already been resolved. `ldc_quick` reads the value from the constant pool and pushes it onto the stack.

Operation

```
stack[OPTOP] ⇐ mem[CONST_POOL + (index × 4)]
OPTOP ⇐ OPTOP – 4
```

Notes

In the picoJava-II core, `ldc_quick` is identical to `aldc_quick`. The distinction allows future implementations of `ldc_quick` to differ in garbage collection events.

Trap to emulation routine that resolves a constant pool item, with a wide index, and pushes it onto the stack.

Format

ldc_w
indexbyte1
indexbyte2

Forms

ldc_w = 19 (0x13)

Stack

… ⇒
…, *value*

Description

ldc_w traps to the emulation routine referenced by entry 0x13 in the trap table.

Operation

```
trap ldc_w (type 0x13)
```

Recommendations

The trap handler should emulate ldc_w, as defined in *The Java Virtual Machine Specification*.

Once the constant pool entry is resolved, the type of the constant indexed determines whether a ldc_w_quick or aldc_w_quick byte replaces the original ldc_w byte. If the constant is a reference type, the aldc_w_quick instruction should replace the ldc_w opcode. Otherwise, the ldc_w_quick instruction should replace the ldc_w opcode.

Push an item from constant pool, with wide index.

Format

ldc_w_quick
indexbyte1
indexbyte2

Forms

`ldc_w_quick` = 204 (0xcc)

Stack

… ⇒
…, *value*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class where the value of each index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item should be a constant that has already been resolved. `ldc_w_quick` reads the value from the constant pool and pushes it onto the stack.

Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
stack[OPTOP] ⇐ mem[CONST_POOL + (index × 4)]
OPTOP ⇐ OPTOP – 4
```

Notes

In the picoJava-II core, `ldc_w_quick` is identical to `aldc_w_quick`. The distinction allows future implementations of `ldc_w_quick` to differ in garbage collection events.

Trap to emulation routine that resolves a two-word constant pool item, with a wide index, and pushes it onto the stack.

Format

ldc2_w
indexbyte1
indexbyte2

Forms

ldc2_w = 20 (0x14)

Stack

… ⇒
…, *value<31:0>*, *value<63:32>*

Description

ldc2_w traps to the emulation routine referenced by entry 0x14 in the trap table.

Operation

```
trap ldc2_w (type 0x14)
```

Recommendations

The trap handler should emulate ldc2_w, as defined in *The Java Virtual Machine Specification*.

Once the constant pool entry is resolved, the ldc2_w_quick byte should replace the original ldc2_w byte.

Push a two-word item from constant pool, with wide index.

Format

ldc2_w_quick
indexbyte1
indexbyte2

Forms

`ldc2_w_quick` = 205 (0xcd)

Stack

… ⇒
…, *value<31:0>*, *value<63:32>*

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class where the value of each index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item should be a constant that has already been resolved. `ldc2_w_quick` reads the two-word value from the constant pool and pushes it onto the stack.

Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
stack[OPTOP] ⇐ mem[CONST_POOL + (index × 4) + 4]
stack[OPTOP - 4] ⇐ mem[CONST_POOL + (index × 4)]
OPTOP ⇐ OPTOP – 8
```

Trap to emulation routine that divides two longs.

Format

ldiv

Forms

`ldiv` = 109 (0x6d)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

`ldiv` traps to the emulation routine referenced by entry 0x6d in the trap table.

Operation

```
trap ldiv (type 0x6d)
```

Recommendations

The trap handler should emulate `ldiv`, as defined in *The Java Virtual Machine Specification*.

Load a long integer from a local variable.

Format

lload
index

Forms

lload = 22 (0x16)

Stack

… ⇒
…, *value<31:0>*, *value<63:32>*

Description

lload pushes a two-word local variable, which is at *index* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS – (index × 4)]
stack[OPTOP – 4] ⇐ stack[VARS – 4 – (index × 4)]
OPTOP ⇐ OPTOP – 8
```

Notes

In the picoJava-II core, lload is identical to dload.

Load a long integer from a local variable.

Format

lload_*n*

Forms

lload_0 = 30 (0x1e)
lload_1 = 31 (0x1f)
lload_2 = 32 (0x20)
lload_3 = 33 (0x21)

Stack

… ⇒
…, *value<31:0>*, *value<63:32>*

Description

lload_*n* pushes a two-word local variable, which is at *n* stack entries offset from the start of the current local variables, onto the operand stack.

Operation

```
stack[OPTOP] ⇐ stack[VARS -(n × 4)]
stack[OPTOP – 4] ⇐ stack[VARS – 4 – (n × 4)]
OPTOP ⇐ OPTOP – 8
```

Notes

In the picoJava-II core, lload_*n* is identical to dload_*n*.

Trap to emulation routine that multiplies two longs.

Format

lmul

Forms

`lmul` = 105 (0x69)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

`lmul` traps to the emulation routine referenced by entry 0x69 in the trap table.

Operation

```
trap lmul (type 0x69)
```

Recommendations

The trap handler should emulate `lmul`, as defined in *The Java Virtual Machine Specification*.

Negate a long.

Format

lneg

Forms

lneg = 117 (0x75)

Stack

…, *value<31:0>*, *value<63:32>* ⇒
…, *result<31:0>*, *result<63:32>*

Description

lneg treats *value* as the type long and pops it from the operand stack. lneg then pushes *result*, which is a long and the result of 0 - *value*, onto the operand stack.

Operation

```
long(stack[OPTOP + 4], stack[OPTOP + 8]) ⇐
            long(0, 0) - long(stack[OPTOP + 4], stack[OPTOP + 8])
```

Load a signed byte from memory.

Format

extend
load_byte

Forms

extend = 255 (0xff)
load_byte = 1 (0x01)

Stack

…, *address* ⇒
…, *value*

Description

load_byte loads and sign-extends the signed 8-bit value at the memory location at *address*, then pushes it onto the top of the stack.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ sign_ext8(mem8[address])
```

Load a signed 8-bit value at a fixed offset from the address in a local variable from memory.

Format

load_byte_index
local_var
offset

Forms

load_byte_index = 241 (0xf1)

Stack

…⇒
…, *value*

Description

load_byte_index loads and sign-extends the 8-bit value at the memory location at the effective address, then pushes it onto the top of the stack. load_byte_index computes the effective address by loading the contents of the local variable, *local_var*, and adding the signed value of *offset*.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + sign_ext8(offset)
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP] ⇐ sign_ext8(mem8[eff_addr])
OPTOP ⇐ OPTOP – 4
```

Notes

load_byte_index is equivalent to the following sequence of instructions:

```
iload local_var; bipush offset; iadd; load_byte
```

Load unsigned `short (char)` from memory.

Format

extend
load_char

Forms

`extend` = 255 (0xff)
`load_char` = 2 (0x02)

Stack

…, *address* ⇒
…, *value*

Description

`load_char` loads the unsigned 16-bit value at the memory location at *address* and pushes it onto the top of the stack. If bit 30 of *address* is set to 1, then the data loaded is treated as if it were stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
```

Load an unsigned 16-bit value at a fixed offset from an address in a local variable from memory.

Format

load_char_index
local_var
offset

Forms

load_char_index = 240 (0xf0)

Stack

…⇒
…, *value*

Description

load_char_index loads the 16-bit value at the memory location at the effective address and pushes it onto the top of the stack. load_char_index computes the effective address by loading the contents of the local variable, *local_var*, and adding the signed value of *offset* << 1. If bit 30 of the effective address is set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The effective address must be aligned on a 16-bit boundary.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + (sign_ext8(offset) × 2)
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP] ⇐ mem16[eff_addr]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP] ⇐ endian_swap16(stack[OPTOP])
OPTOP ⇐ OPTOP – 4
```

Notes

load_char_index is equivalent to the following sequence of instructions:

iload *local_var*; bipush *offset*; iconst_1; ishl; iadd; load_char

Use opposite endianness to load unsigned `short (char)` from memory.

Format

extend
load_char_oe

Forms

`extend` = 255 (0xff)
`load_char_oe` = 10 (0x0a)

Stack

…, *address* ⇒
…, *value*

Description

`load_char_oe` loads the unsigned 16-bit value at the memory location at *address*, then pushes it onto the top of the stack. If bit 30 of *address* is not set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16[address]
if ((address & 0x40000000) = 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
```

Load signed short from memory.

Format

extend
load_short

Forms

extend = 255 (0xff)
load_short = 3 (0x03)

Stack

…, *address* ⇒
…, *value*

Description

load_short loads and sign-extends the signed 16-bit value at the memory location at *address*, then pushes it onto the top of the stack. If bit 30 of *address* is set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
stack[OPTOP + 4] ⇐ sign_ext16(stack[OPTOP + 4])
```

Load a signed 16-bit value at a fixed offset from an address in a local variable from memory.

Format

load_short_index
local_var
offset

Forms

load_short_index = 239 (0xef)

Stack

…⇒
…, *value*

Description

load_short_index loads and sign-extends the 16-bit value at the memory location at the effective address, then pushes it onto the top of the stack. load_short_index computes the effective address by loading the contents of the local variable, *local_var.* and adding the signed value of *offset* × 2. If bit 30 of the effective address is set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The effective address must be aligned on a 16-bit boundary.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + (sign_ext8(offset) × 2)
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP] ⇐ mem16[eff_addr]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP] ⇐ endian_swap16(stack[OPTOP])
stack[OPTOP] ⇐ sign_ext16(stack[OPTOP])
OPTOP ⇐ OPTOP – 4
```

Notes

load_short_index is equivalent to the following sequence of instructions:

```
iload local_var; bipush offset; iconst_1; ishl; iadd; load_short
```

Use opposite endianness to load signed `short` from memory.

Format

extend
load_short_oe

Forms

`extend` = 255 (0xff)
`load_short_oe` = 11 (0x0b)

Stack

…, *address* ⇒
…, *value*

Description

`load_short_oe` loads and sign-extends the signed 16-bit value at the memory location at *address* onto the top of the stack. If bit 30 of *address* is not set to 1, then the data loaded is treated as if it were stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

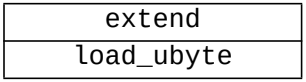
The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
stack[OPTOP + 4] ⇐ sign_ext16(stack[OPTOP + 4])
```

Load unsigned byte from memory.

Format



Forms

extend = 255 (0xff)
load_ubyte = 0 (0x00)

Stack

…, *address* ⇒
…, *value*

Description

load_ubyte loads the unsigned 8-bit value at the memory location at *address* and pushes it onto the top of the stack.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem8[address]
```

Notes

load_ubyte supports the compilation of C or C++ code that contains an unsigned byte data type.

Load an unsigned 8-bit value at a fixed offset from an address in a local variable from memory.

Format

load_byte_index
local_var
offset

Forms

load_ubyte_index = 242 (0xf2)

Stack

…⇒
…, *value*

Description

load_ubyte_index loads the 8-bit value at the memory location at the effective address and pushes it onto the top of the stack. load_ubyte_index computes the effective address by loading the contents of the local variable, *local_var*, and adding the signed value of *offset*.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + sign_ext8(offset)
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP] ⇐ mem8[eff_addr]
OPTOP ⇐ OPTOP – 4
```

Notes

load_ubyte_index is equivalent to the following sequence of instructions:

iload *local_var*; bipush *offset*; iadd; load_ubyte

Load `integer` from memory.

Format

extend
load_word

Forms

`extend` = 255 (0xff)
`load_word` = 4 (0x04)

Stack

…, *address* ⇒
…, *value*

Description

`load_word` loads the 32-bit value at the memory location at *address* and pushes it onto the top of the stack. If bit 30 of *address* is set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap(stack[OPTOP + 4])
```

Load a word at a fixed offset from an address in a local variable from memory.

Format

load_word_index
local_var
offset

Forms

load_word_index = 238 (0xee)

Stack

…⇒
…, *value*

Description

load_word_index loads the 32-bit value at the memory location at the effective address and pushes it onto the top of the stack. load_word_index computes the effective address by loading the contents of the local variable, *local_var*, and adding the signed value of *offset* × 4. If bit 30 of the effective address is set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The effective address must be aligned on a 32-bit boundary.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + (sign_ext8(offset) × 4)
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP] ⇐ mem[eff_addr]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP] ⇐ endian_swap(stack[OPTOP])
OPTOP ⇐ OPTOP - 4
```

Notes

load_word_index is equivalent to the following sequence of instructions:

```
iload local_var; bipush offset; iconst_2; ishl; iadd; load_word
```

Use opposite endianness to load an integer from memory.

Format

extend
load_word_oe

Forms

extend = 255 (0xff)
load_word_oe = 12 (0x0c)

Stack

…, *address* ⇒
…, *value*

Description

load_word_oe loads the 32-bit value at the memory location at *address*, then pushes it onto the top of the stack. If bit 30 of *address* is not set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem[address]
if ((address & 0x40000000) = 0) then
            stack[OPTOP + 4] ⇐ endian_swap(stack[OPTOP + 4])
```

Trap to emulation routine that accesses jump table by key and jumps.

Format

lookupswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
npairs1
npairs2
npairs3
npairs4
match-offset pairs

Forms

`lookupswitch` = 171 (0xab)

Stack

…, *key* ⇒
…

Description

`lookupswitch` traps to the emulation routine referenced by entry 0xab in the trap table.

Operation

```
trap lookupswitch (type 0xab)
```

Recommendations

The trap handler should emulate `lookupswitch`, as defined in *The Java Virtual Machine Specification*.

Bitwise OR of two longs.

Format

lor

Forms

`lor` = 129 (0x81)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

`lor` treats both *value1* and *value2* as the type long and pops them from the operand stack. It then pushes *result*, which is a long and the bitwise OR of *value1* and *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 12] ⇐ stack[OPTOP + 12] | stack[OPTOP + 4]
stack[OPTOP + 16] ⇐ stack[OPTOP + 16] | stack[OPTOP + 8]
OPTOP ⇐ OPTOP + 8
```

Trap to emulation routine that computes the remainder of two longs.

Format

lrem

Forms

`lrem` = 113 (0x71)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32>2 ⇒
…, result<31:0>, result<63:32>

Description

`lrem` traps to the emulation routine referenced by entry 0x71 in the trap table.

Operation

```
trap lrem (type 0x71)
```

Recommendations

The trap handler should emulate `lrem`, as defined in *The Java Virtual Machine Specification*.

Return a long integer from a method.

Format

lreturn

Forms

lreturn = 173 (0xad)

Stack

…, value<31:0>, value<63:32> ⇒
[empty]

Description

lreturn returns to the caller of this method, popping all the arguments to the current method and pushing the long integer that is on the top of the operand stack onto the top of the caller's operand stack.

Operation

```
PC ⇐ stack[FRAME]
CONST_POOL ⇐ stack[FRAME – 12]
ret_value_word1 ⇐ stack[OPTOP + 4]
ret_value_word2 ⇐ stack[OPTOP + 8]
VARS ⇐ stack[FRAME - 4]
FRAME ⇐ stack[FRAME - 8]
OPTOP ⇐ VARS + 8
stack[OPTOP + 4] ⇐ ret_value_word1
stack[OPTOP + 8] ⇐ ret_value_word2
```

Notes

In the picoJava-II core, lreturn is identical to dreturn.

Shift-left of a long.

Format

lshl

Forms

`lshl` = 121 (0x79)

Stack

…, value1<31:0>, value1<63:32>, value2 ⇒
…, result<31:0>, result<63:32>

Description

`lshl` treats *value1* as the type long and *value2* as an integer and pops them from the operand stack. It then pushes *result*, which is a long and the result of shifting *value1* left by the number of positions equal to the low 6 bits of *value2*, onto the operand stack.

Operation

```
long(stack[OPTOP + 8], stack[OPTOP + 12]) ⇐
            long(stack[OPTOP + 8], stack[OPTOP + 12]) << (stack[OPTOP + 4] & 0x3f)
OPTOP ⇐ OPTOP + 4
```

Shift-right of a long.

Format

lshr

Forms

`lshr` = 123 (0x7b)

Stack

…, value1<31:0>, value1<63:32>, value2 ⇒
…, result<31:0>, result<63:32>

Description

`lshr` treats *value1* as the type long and *value2* as an integer and pops them from the operand stack. It then pushes *result*, which is a long and the result of shifting *value1* right (with sign extension) by the number of positions equal to the low 6 bits of *value2*, onto the operand stack.

Operation

```
long(stack[OPTOP + 8], stack[OPTOP + 12]) ⇐
            long(stack[OPTOP + 8], stack[OPTOP + 12]) >> (stack[OPTOP + 4] & 0x3f)
OPTOP ⇐ OPTOP + 4
```

Store a long integer to a local variable.

Format

Forms

`lstore` = 55 (0x37)

Stack

…, *value<31:0>*, *value<63:32>* ⇒
…

Description

`lstore` stores the long integer on the top of the operand stack into a two-word local variable, which is at *index* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS – (index × 4)] ⇐ stack[OPTOP]
stack[VARS – 4 -(index × 4)] ⇐ stack[OPTOP – 4]
OPTOP ⇐ OPTOP + 8
```

Notes

In the picoJava-II core, `lstore` is identical to `dstore`.

Store a long integer to a local variable.

Format

lstore_*n*

Forms

`lstore_0` = 63 (0x3f)
`lstore_1` = 64 (0x40)
`lstore_2` = 65 (0x41)
`lstore_3` = 66 (0x42)

Stack

…, *value<31:0>*, *value<63:32>* ⇒
…

Description

`lstore_`*n* stores the long integer on the top of the operand stack into a two-word local variable, which is at *n* stack entries offset from the start of the current local variables.

Operation

```
stack[VARS – (n × 4)] ⇐ stack[OPTOP]
stack[VARS – 4 - (n × 4)] ⇐ stack[OPTOP – 4]
OPTOP ⇐ OPTOP + 8
```

Notes

In the picoJava-II core, `lstore_`*n* is identical to `dstore_`*n*.

Subtract two longs.

Format

lsub

Forms

lsub = 101 (0x65)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

lsub treats both *value1* and *value2* as the type long and pops them from the operand stack. It then pushes *result*, which is a long and the result of *value1* - *value2*, onto the operand stack.

Operation

```
long(stack[OPTOP + 12], stack[OPTOP + 16]) ⇐
            long(stack[OPTOP + 12], stack[OPTOP + 16]) -
            long(stack[OPTOP + 4], stack[OPTOP + 8])
OPTOP ⇐ OPTOP + 8
```

Unsigned shift-right of a long.

Format

lushr

Forms

`lushr` = 125 (0x7d)

Stack

…, value1<31:0>, value1<63:32>, value2 ⇒
…, result<31:0>, result<63:32>

Description

`lushr` treats *value1* as the type long and *value2* as an integer and pops them from the operand stack. It then pushes *result*, which is a long and the result of shifting *value1* right (with zero extension) by the number of positions equal to the low 6 bits of *value2*, onto the operand stack.

Operation

```
long(stack[OPTOP + 8], stack[OPTOP + 12]) ⇐
            long(stack[OPTOP + 8], stack[OPTOP + 12]) >>>
            (stack[OPTOP + 4] & 0x3f)
OPTOP ⇐ OPTOP + 4
```

Bitwise XOR of two longs.

Format

lxor

Forms

`lxor` = 131 (0x83)

Stack

…, value1<31:0>, value1<63:32>, value2<31:0>, value2<63:32> ⇒
…, result<31:0>, result<63:32>

Description

`lxor` treats both *value1* and *value2* as the type long and pops them from the operand stack. It then pushes *result*, which is a long and the bitwise XOR of *value1* and *value2*, onto the operand stack.

Operation

```
stack[OPTOP + 12] ⇐ stack[OPTOP + 12] ^ stack[OPTOP + 4]
stack[OPTOP + 16] ⇐ stack[OPTOP + 16] ^ stack[OPTOP + 8]
OPTOP ⇐ OPTOP + 8
```

Enter monitor for an object.

Format

monitorenter

Forms

monitorenter = 194 (0xc2)

Stack

…, objectref ⇒
…

Description

The top entry on the stack, *objectref*, is treated as a reference. If *objectref* is null, then monitorenter takes a NullPointer trap. Otherwise, if *objectref* matches either of the LOCKADDR registers, then monitorenter increments the count of the number of times a monitor associated with *objectref* has been entered. If incrementing the count would cause an overflow, then monitorenter generates a LockCountOverflow trap. If *objectref* matches neither of the LOCKADDR registers at least one of the LOCKADDR registers is 0, then the LOCKADDR register is assigned the value of objectref and the LOCKCOUNT register is initialized with the COUNT field set to 1 and the CO bit set to 1. If *objectref* matches neither of the LOCKADDR registers and neither of the LOCKADDR registers is 0, then monitorenter generates a LockEnterMiss trap.

See *Monitors* on page 395 for more information.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref = 0) then
            trap NullPointer (type = 0x1b)
masked_objref ⇐ objectref & 0x7ffffffc
masked_lockaddr0 ⇐ LOCKADDR0 & 0x7ffffffc
masked_lockaddr1 ⇐ LOCKADDR1 & 0x7ffffffc
if (masked_lockaddr0 = masked_objref) then
            if (LOCKCOUNT0.COUNT = 255) then
                        trap LockCountOverflow (type = 0x23)
            LOCKCOUNT0.COUNT ⇐ LOCKCOUNT0.COUNT + 1
else if (masked_lockaddr1 = masked_objref) then
            if (LOCKCOUNT1.COUNT = 255) then
                        trap LockCountOverflow (type = 0x23)
            LOCKCOUNT1.COUNT ⇐ LOCKCOUNT1.COUNT + 1
else if ((LOCKADDR0 = 0) OR (LOCKADDR1 = 0)) then
```

```
            lockbit ⇐ mem[masked_objref] & 0x00000001
            if ((lockbit = 0) AND (LOCKADDR0 = 0)) then
                           LOCKADDR0 ⇐ objectref
                           LOCKCOUNT0 ⇐ 0x00008001
            else if ((lockbit = 0) AND (LOCKADDR1 = 0)) then
                           LOCKADDR1 ⇐ objectref
                           LOCKCOUNT1 ⇐ 0x00008001
            else
                           trap LockEnterMiss (type = 0x24)
else
            trap LockEnterMiss (type = 0x24)
OPTOP ⇐ OPTOP + 4
```

Exit monitor for an object.

Format

monitorexit

Forms

monitorexit = 195 (0xc3)

Stack

…, objectref ⇒
…

Description

The top entry on the stack, *objectref*, is treated as a reference. If *objectref* is null, then monitorexit takes a NullPointer trap. Otherwise, if *objectref* matches either of the LOCKADDR registers, then monitorexit decrements the count of the number of times a monitor associated with *objectref* has been entered. If decrementing the count causes an underflow, then monitorexit generates a LockCountOverflow trap. If decrementing the count causes the monitor to be released and another thread is waiting to enter, then monitorexit causes a LockRelease trap. If decrementing the count causes the monitor to be released and the CO bit is set, then the associated LOCKADDR register is set to 0. If *objectref* matches neither of the LOCKADDR registers, then monitorexit generates a LockExitMiss trap.

See *Monitors* on page 395 for more information.

Operation

```
objectref ⇐ stack[OPTOP + 4]
if (objectref = 0) then
            trap NullPointer (type = 0x1b)
masked_objref ⇐ objectref & 0x7ffffffc
masked_lockaddr0 ⇐ LOCKADDR0 & 0x7ffffffc
masked_lockaddr1 ⇐ LOCKADDR1 & 0x7ffffffc
if (masked_lockaddr0 = masked_objref) then
            LOCKCOUNT0.COUNT ⇐ LOCKCOUNT0.COUNT - 1
            if ((LOCKCOUNT0.COUNT = 0) AND (LOCKCOUNT0.LOCKWANT = 1)) then
                        trap LockRelease (type = 0x25)
            else if ((LOCKCOUNT0.COUNT = 0) AND (LOCKCOUNT0.CO = 1)) then
                        LOCKADDR0 ⇐ 0
                        LOCKCOUNT0 ⇐ 0
            else if (LOCKCOUNT0.COUNT = 0xff) then
                        trap LockCountOverflow (type = 0x23)
else if (masked_lockaddr1 = masked_objref) then
            LOCKCOUNT1.COUNT ⇐ LOCKCOUNT1.COUNT - 1
            else if ((LOCKCOUNT1.COUNT = 0) AND (LOCKCOUNT1.LOCKWANT = 1)) then
```

```
                        trap LockRelease (type = 0x25)
            else if ((LOCKCOUNT1.COUNT = 0) AND (LOCKCOUNT1.CO = 1)) then
                        LOCKADDR1 ⇐ 0
                        LOCKCOUNT1 ⇐ 0
            else if (LOCKCOUNT1.COUNT = 0xff) then
                        trap LockCountOverflow (type = 0x23)
else
            trap LockExitMiss (type = 0x26)
OPTOP ⇐ OPTOP + 4
```

Trap to emulation routine that resolves a constant pool item and creates a new multidimensional array.

Format

multianewarray
indexbyte1
indexbyte2
dimensions

Forms

multianewarray = 197 (0xc5)

Stack

…, *count1*, [*count2*, ...] ⇒
…, *arrayref*

Description

multianewarray traps to the emulation routine referenced by entry 0xc5 in the trap table.

Operation

```
trap multianewarray (type = 0xc5)
```

Recommendations

The trap handler should emulate multianewarray, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the multianewarray instruction with the multianewarray_quick instruction.

multianewarray_quick multianewarray_quick

Trap to emulation routine that creates a new multidimensional array.

Format

multianewarray_quick
indexbyte1
indexbyte2
dimensions

Forms

multianewarray_quick = 223 (0xdf)

Stack

…, *count1*, [*count2*, ...] ⇒
…, *arrayref*

Description

multianewarray traps to the emulation routine referenced by entry 0xdf in the trap table.

Operation

```
trap multianewarray_quick (type = 0xdf)
```

Recommendations

The trap handler should emulate multianewarray_quick, as defined in *The Java Virtual Machine Specification*.

Nonallocating store of a word at a fixed offset from an address in a local variable to memory.

Format

nastore_word_index
local_var
offset

Forms

nastore_word_index = 244 (0xf4)

Stack

…, *value* ⇒
…

Description

nastore_word_index stores the 32-bit value on the stack to the memory location at the effective address. It computes the effective address by loading the contents of the local variable, *local_var*, and adding the signed value of *offset* × 4. If bit 30 of the effective address is set to 1, then the data is stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The effective address must be aligned on a 32-bit boundary.

The store is nonallocating: If the data is not present in the data cache, then nastore_word_index stores it into memory without allocating a line in the cache.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + (sign_ext8(offset) × 4)
data ⇐ stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
if ((eff_addr & 0x40000000) ≠ 0) then
            data ⇐ endian_swap(data)
memNA[eff_addr] ⇐ data
OPTOP ⇐ OPTOP + 4
```

Use opposite endianness to load an integer from memory.

Format

extend
ncload_byte

Forms

`extend` = 255 (0xff)
`ncload_byte` = 17 (0x11)

Stack

…, *address* ⇒
…, *value*

Description

`ncload_byte` loads and sign-extends the signed 8-bit value at the memory location at *address*, then pushes it onto the top of the stack. `ncload_byte` bypasses the data cache and sends the request directly to memory.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ sign_ext8(mem8,NC[address])
```

Noncacheable load unsigned `short (char)` from memory.

Format

extend
ncload_char

Forms

`extend` = 255 (0xff)
`ncload_char` = 18 (0x12)

Stack

…, *address* ⇒
…, *value*

Description

`ncload_char` loads the unsigned 16-bit value at the memory location at *address* and pushes it onto the top of the stack. `ncload_char` bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is set to 1, then the data loaded is treated as if it were stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16,NC[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
```

Use opposite endianness to perform a noncacheable load of unsigned `short (char)` from memory.

Format

extend
ncload_char_oe

Forms

`extend` = 255 (0xff)
`ncload_char_oe` = 26 (0x1a)

Stack

…, *address* ⇒
…, *value*

Description

`ncload_char_oe` loads the unsigned 16-bit value at the memory location at *address*, then pushes it onto the top of the stack. `ncload_char_oe` bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is not set to 1, then the data loaded is treated as if it were stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16,NC[address]
if ((address & 0x40000000) = 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
```

Noncacheable load signed `short` from memory.

Format

extend
ncload_short

Forms

`extend` = 255 (0xff)
`ncload_short` = 19 (0x13)

Stack

…, *address* ⇒
…, *value*

Description

`ncload_short` loads and sign-extends the signed 16-bit value at the memory location at *address*, then pushes it onto the top of the stack. `ncload_short` bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is set to 1, then the data loaded is treated as if it were stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16,NC[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
stack[OPTOP + 4] ⇐ sign_ext16(stack[OPTOP + 4])
```

Use opposite endianness to perform a noncacheable load of signed `short` from memory.

Format

extend
ncload_short_oe

Forms

`extend` = 255 (0xff)
`ncload_short_oe` = 27 (0x1b)

Stack

…, *address* ⇒
…, *value*

Description

`ncload_short_oe` loads and sign-extends the signed 16-bit value at the memory location at *address*, then pushes it onto the top of the stack. `ncload_short_oe` bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is not set to 1, then the data loaded is treated as if it were stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem16,NC[address]
if ((address & 0x40000000) = 0) then
            stack[OPTOP + 4] ⇐ endian_swap16(stack[OPTOP + 4])
stack[OPTOP + 4] ⇐ sign_ext16(stack[OPTOP + 4])
```

Noncacheable load unsigned byte from memory.

Format

extend
ncload_ubyte

Forms

extend = 255 (0xff)
ncload_ubyte = 16 (0x10)

Stack

…, *address* ⇒
…, *value*

Description

ncload_ubyte loads the unsigned 8-bit value at the memory location at *address*, then pushes it onto the top of the stack. ncload_ubyte bypasses the data cache and sends the request directly to memory.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem8,NC[address]
```

Noncacheable load an integer from memory.

Format

extend
ncload_word

Forms

extend = 255 (0xff)
ncload_word = 20 (0x14)

Stack

…, *address* ⇒
…, *value*

Description

ncload_word loads the 32-bit value at the memory location at *address*, then pushes it onto the top of the stack. ncload_word bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem_NC[address]
if ((address & 0x40000000) ≠ 0) then
            stack[OPTOP + 4] ⇐ endian_swap(stack[OPTOP + 4])
```

Use opposite endianness to perform a noncacheable load of an integer from memory.

Format

extend
ncload_word_oe

Forms

extend = 255 (0xff)
ncload_word_oe = 28 (0x1c)

Stack

…, *address* ⇒
…, *value*

Description

ncload_word_oe loads the 32-bit value at the memory location at *address*, then pushes it onto the top of the stack. ncload_word_oe bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is not set to 1, then the data loaded is treated as if it were stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address = stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
stack[OPTOP + 4] ⇐ mem_NC[address]
if ((address & 0x40000000) = 0) then
            stack[OPTOP + 4] ⇐ endian_swap(stack[OPTOP + 4])
```

Noncacheable store byte to memory.

Format

extend
ncstore_byte

Forms

extend = 255 (0xff)
ncstore_byte = 48 (0x30)

Stack

…, *value*, *address* ⇒
…

Description

ncstore_byte stores the low 8 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack. It bypasses the data cache and sends the request directly to memory.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
mem8,NC[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Noncacheable store `short` or `char` to memory.

Format

extend
ncstore_short

Forms

`extend` = 255 (0xff)
`ncstore_short` = 50 (0x32)

Stack

…, *value*, *address* ⇒
…

Description

`ncstore_short` stores the low 16 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack. It bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is set to 1, then the data is stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if ((address & 0x40000000) ≠ 0) then
            data ⇐ endian_swap16(data)
mem16,NC[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Use opposite endianness to perform a noncacheable store of `short` to memory.

Format

extend
ncstore_short_oe

Forms

`extend` = 255 (0xff)
`ncstore_short_oe` = 58 (0x3a)

Stack

…, *value*, *address* ⇒
…

Description

`ncstore_short_oe` stores the low 16 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack. It bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is not set to 1, then the data is stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
           if (addr_out_of_range(address)) then
                      trap mem_protection_error (type 0x02)
if ((address & 0x40000000) = 0) then
           data ⇐ endian_swap16(data)
mem16,NC[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Perform a noncacheable store of `integer` to memory.

Format

extend
ncstore_word

Forms

`extend` = 255 (0xff)
`ncstore_word` = 52 (0x34)

Stack

…, *value*, *address* ⇒
…

Description

`ncstore_word` stores the 32 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack. It bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is set to 1, then the data is stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if ((address & 0x40000000) ≠ 0) then
            data ⇐ endian_swap(data)
mem_NC[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Use opposite endianness to perform a noncacheable store of `integer` to memory.

Format

extend
ncstore_word_oe

Forms

`extend` = 255 (0xff)
`ncstore_word_oe` = 60 (0x3c)

Stack

…, *value*, *address* ⇒
…

Description

`ncstore_word_oe` stores the 32 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack. It bypasses the data cache and sends the request directly to memory. If bit 30 of *address* is not set to 1, then the data is stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

`ncstore_word_oe` aligns the address on a 32-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
           if (addr_out_of_range(address)) then
                      trap mem_protection_error (type 0x02)
if ((address & 0x40000000) = 0) then
           data ⇐ endian_swap(data)
mem_NC[address] ⇐ eata
OPTOP ⇐ OPTOP + 8
```

Trap to emulation routine that resolves a constant pool item and creates a new object.

Format

new
indexbyte1
indexbyte2

Forms

new = 187 (0xbb)

Stack

…, ⇒
…, *objectref*

Description

new traps to the emulation routine referenced by entry 0xbb in the trap table.

Operation

```
trap new (type = 0xbb)
```

Recommendations

The trap handler should emulate new, as defined in *The Java Virtual Machine Specification*. After the trap handler resolves the constant pool entry, it should replace the new instruction with the new_quick instruction.

Trap to emulation routine that creates a new object.

Format

new_quick
indexbyte1
indexbyte2

Forms

new = 221 (0xdd)

Stack

…, ⇒
…, *objectref*

Description

new_quick traps to the emulation routine referenced by entry 0xdd in the trap table.

Operation

```
trap new_quick (type = 0xdd)
```

Recommendations

The trap handler should emulate new_quick, as defined in *The Java Virtual Machine Specification*.

Trap to emulation routine that creates a new array.

Format

newarray
atype

Forms

newarray = 188 (0xbc)

Stack

…, *count* ⇒
…, *objectref*

Description

newarray traps to the emulation routine referenced by entry 0xbc in the trap table.

Operation

```
trap newarray (type = 0xbc)
```

Recommendations

The trap handler should emulate newarray, as defined in *The Java Virtual Machine Specification*.

Read a reference on the stack and generate an exception if it is `null`.

Format

nonnull_quick

Forms

`nonnull_quick` = 229 (0xe5)

Stack

…, *objectref* ⇒
…

Description

`nonnull_quick` pops *objectref*, which is treated as a reference, from the operand stack.

If *objectref* is `null`, then `nonnull_quick` signals a `NullPointer` trap.

Operation

```
if (stack[OPTOP + 4] = 0) then
            trap NullPointer (type = 0x1b)
OPTOP ⇐ OPTOP + 4
```

Do nothing.

Format

nop

Forms

nop = 0 (0x00)

Stack

… ⇒
…

Description

nop does nothing.

Operation

None.

Pop the top word off the stack.

Format

pop

Forms

pop = 87 (0x57)

Stack

…, *word* ⇒
…

Description

pop removes the one-word element at the top of the stack.

Operation

```
OPTOP ⇐ OPTOP + 4
```

Notes

In the picoJava-II core, pop is identical to `l2i`.

Pop the top two words off the stack.

Format

pop2

Forms

pop2 = 88 (0x58)

Stack

…, *word1, word2*⇒
…

Description

pop2 removes the two one-word elements or one two-word element at the top of the stack.

Operation

```
OPTOP ⇐ OPTOP + 8
```

Introduce a privileged and software-initiated entry into low-power standby mode.

Format

extend
powerdown

Forms

`extend` = 255 (0xff)
`priv_powerdown` = 22 (0x16)

Stack

… ⇒
…

Description

When `priv_powerdown` executes, the picoJava-II core enters a low-power standby mode. The core remains in the low-power standby mode, not executing any instructions, until an external interrupt is signalled. Upon receiving the interrupt signal, the core resumes normal execution, transferring control to the appropriate interrupt handler routine. The interrupt handler routine, after completing, returns to the instruction immediately following `priv_powerdown`.

If `PSR.SU` = 0, then `priv_powerdown` generates a `privileged_instruction` trap.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
Enter powerdown mode
```

Perform a privileged diagnostic read of the data cache data array.

Format

extend
read_dcache_data

Forms

`extend` = 255 (0xff)
`priv_read_dcache_data` = 7 (0x07)

Stack

…, *data_address* ⇒
…, *data*

Description

`priv_read_dcache_data` reads a word directly from the data cache data array. The word to be read from the data cache data array is specified by the value on the top of the stack, *data_address*. *data_address* is decoded as shown in FIGURE 6-1.

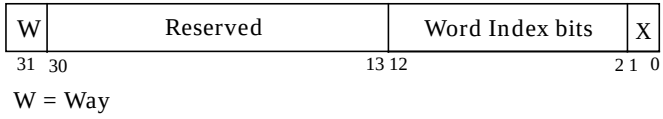


FIGURE 6-1 Format for Data Cache Data Address for 16-Kbyte Data Cache

Bit 31 of *data_address* specifies which “way” of the data cache is to be accessed. Depending on the size of the data cache, the Word Index bits, below, of *data_address* specify which word in the way is read.

Data Cache Size	Word Index Bits
1 Kbytes	<8:2>
2 Kbytes	<9:2>
4 Kbytes	<10:2>
8 Kbytes	<11:2>
16 Kbytes	<12:2>

If PSR.SU = 0, then priv_read_dcache_data generates a privileged_instruction trap.

If no data cache is present, then priv_read_dcache_data pops the word on the top of the stack and pushes 0.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.DCS = 0) then
            stack[OPTOP + 4] ⇐ 0
else
            stack[OPTOP + 4] ⇐ dcache_data[stack[OPTOP + 4]]
```

Perform a privileged diagnostic read of data cache tags.

Format

extend
read_dcache_tag

Forms

`extend` = 255 (0xff)
`priv_read_dcache_tag` = 6 (0x06)

Stack

…, *tag_address* ⇒
…, *tag_data*

Description

`priv_read_dcache_tag` reads a word directly from the data cache tag array. The word to be read from the data cache tag array is specified by the value on the top of the stack, *tag_address*. *tag_address* is decoded as shown in FIGURE 6-2.

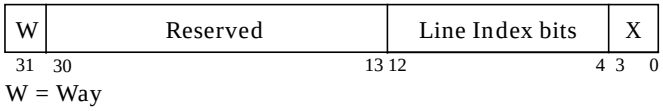


FIGURE 6-2 Format for Data Cache Tag Address for 16-Kbyte Data Cache

Bit 31 of *tag_address* specifies which “way” of the data cache is to be accessed. Depending on the size of the data cache, the Line Index bits, below, of *tag_address* specify which data cache line tag is read.

Data Cache Size	Line Index Bits	Data Cache Tag Bits
1 Kbytes	<8:4>	<29:9>
2 Kbytes	<9:4>	<29:10>
4 Kbytes	<10:4>	<29:11>
8 Kbytes	<11:4>	<29:12>
16 Kbytes	<12:4>	<29:13>

The *tag_data* read is of the format shown in FIGURE 6-3. The Data Cache Tag bits vary, based on the size of the data cache as above. The Valid bit of *tag_data* specifies whether this data cache line contains valid data. The Dirty bit specifies whether the corresponding cache line needs to be written back to memory when it is replaced or flushed. Finally, the LRU bit of the *tag_data* specifies whether way 0 or way 1 is the least recently used.

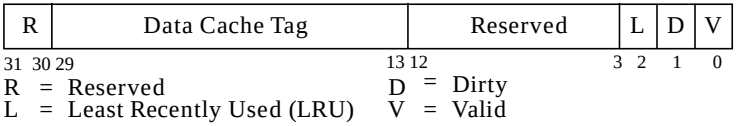


FIGURE 6-3 Format for Data Cache Tag Data for 16-Kbyte Data Cache

If PSR.SU = 0, then `priv_read_dcache_tag` generates a `privileged_instruction` trap.

If no data cache is present, then `priv_read_dcache_tag` pops the word on the top of the stack and pushes 0.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.DCS = 0) then
            stack[OPTOP + 4] ⇐ 0
else
            stack[OPTOP + 4] ⇐ dcache_tag[stack[OPTOP + 4]]
```

Perform a privileged diagnostic read of the instruction cache data array.

Format

extend
read_icache_data

Forms

`extend` = 255 (0xff)
`priv_read_icache_data` = 15 (0x0f)

Stack

…, *data_address* ⇒
…, *data*

Description

`priv_read_icache_data` reads a word directly from the instruction cache data array. The word to be read from the instruction cache data array is specified by the value on the top of the stack, *data_address*. *data_address* is decoded as shown in FIGURE 6-4.

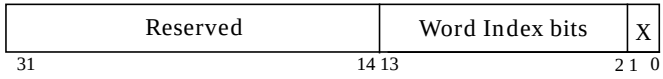


FIGURE 6-4 Format for Instruction Cache Data Address for 16-Kbyte Instruction Cache

Depending on the size of the instruction cache, the Word Index bits, below, of *data_address* specify which word in the instruction cache is read.

Instruction Cache Size	Word Index Bits
1 Kbytes	<9:2>
2 Kbytes	<10:2>
4 Kbytes	<11:2>
8 Kbytes	<12:2>
16 Kbytes	<13:2>

If PSR.SU = 0, then priv_read_icache_data generates a privileged_instruction trap.

If no instruction cache is present, then priv_read_icache_data pops the word on the top of the stack and pushes 0.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.ICS = 0) then
            stack[OPTOP + 4] ⇐ 0
else
            stack[OPTOP + 4] ⇐ icache_data[stack[OPTOP + 4]]
```

Perform a privileged diagnostic read of the instruction cache tags.

Format

extend
read_icache_tag

Forms

`extend` = 255 (0xff)
`priv_read_icache_tag` = 14 (0x0e)

Stack

…, *tag_address* ⇒
…, *tag_data*

Description

`priv_read_icache_tag` reads a word directly from the instruction cache tag array. The word to be read from the instruction cache tag array is specified by the value on the top of the stack, *tag_address*. *tag_address* is decoded as shown in FIGURE 6-5.

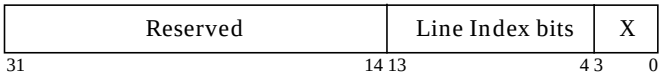


FIGURE 6-5 Format for 16-Kbyte Instruction Cache Tag Address

Depending on the size of the instruction cache, the Line Index bits, below, of *tag_address* specify which instruction cache line tag is read.

Instruction Cache Size	Line Index Bits	Instruction Cache Tag Bits
1 Kbytes	<9:4>	<29:10>
2 Kbytes	<10:4>	<29:11>
4 Kbytes	<11:4>	<29:12>
8 Kbytes	<12:4>	<29:13>
16 Kbytes	<13:4>	<29:14>

The *tag_data* to be read is of the format shown in FIGURE 6-6. The Instruction Cache Tag bits vary, based on the size of the instruction cache as above. The Valid bit of *tag_data* specifies whether this instruction cache line contains valid data.

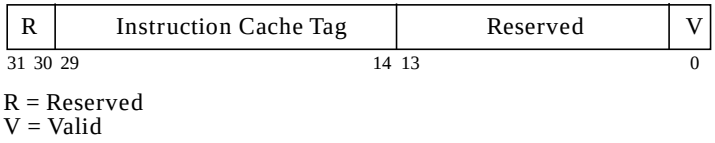


FIGURE 6-6 Format for Instruction Cache Tag Data for 16-Kbyte Instruction Cache

If PSR.SU = 0, then `priv_read_icache_tag` generates a `privileged_instruction` trap.

If no instruction cache is present, then `priv_read_icache_tag` pops the word on the top of the stack and pushes 0.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.ICS = 0) then
            stack[OPTOP + 4] ⇐ 0
else
            stack[OPTOP + 4] ⇐ icache_tag[stack[OPTOP + 4]]
```

Read a machine register.

Format

extend
read_*reg*

Forms

extend = 255 (0xff)
priv_read_*reg* = 64 (0x40) through 89 (0x59) (See table below.)

Stack

… ⇒
…, *value*

Description

priv_read_*reg* pushes the contents of a machine register onto the stack. The following table tabulates which priv_read_*reg* pushes which register, along with the opcodes.

Instruction	Machine Register	Opcode
priv_read_oplim	OPLIM	0x44
priv_read_psr	PSR	0x46
priv_read_trapbase	TRAPBASE	0x47
priv_read_lockcount0	LOCKCOUNT0	0x48
priv_read_lockcount1	LOCKCOUNT1	0x49
priv_read_lockaddr0	LOCKADDR0	0x4c
priv_read_lockaddr1	LOCKADDR1	0x4d
priv_read_userrange1	USERRANGE1	0x50
priv_read_userrange2	USERRANGE2	0x55
priv_read_gc_config	GC_CONFIG	0x51
priv_read_brk1a	BRK1A	0x52
priv_read_brk2a	BRK2A	0x53
priv_read_brk12c	BRK12C	0x54

Instruction	Machine Register	Opcode
priv_read_versionid	VERSIONID	0x57
priv_read_hcr	HCR	0x58
priv_read_sc_bottom	SC_BOTTOM	0x59

If PSR.SU = 0, then priv_read_*reg* generates a privileged_instruction trap.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
stack[OPTOP] ⇐ reg
OPTOP ⇐ OPTOP - 4
```

Perform a privileged and software-initiated reset.

Format

extend
reset

Forms

`extend` = 255 (0xff)
`priv_reset` = 54 (0x36)

Stack

… ⇒
…

Description

When `priv_reset` executes, the picoJava-II core enters the same state as if a power-on reset had occurred. Execution starts from address 0x00000000.

If `PSR.SU` = 0, then `priv_reset` generates a `privileged_instruction` trap.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
reset
```

Perform a privileged return from a trap instruction.

Format

extend
ret_from_trap

Forms

extend = 255 (0xff)
priv_ret_from_trap = 5 (0x05)

Stack

… ⇒
…

Description

priv_ret_from_trap pops off a trap call frame and restores the PSR register. The value of OPTOP after ret_from_trap is equal to VARS when the ret_from_trap starts execution. Care must be taken prior to ret_from_trap to ensure that VARS has an appropriate value.

If PSR.SU = 0, then priv_ret_from_trap generates a privileged_instruction trap.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
returnOPTOP ⇐ VARS
PC ⇐ stack[FRAME]
VARS ⇐ stack[FRAME - 4]
PSR ⇐ stack[FRAME + 4]
FRAME ⇐ stack[FRAME - 8]
OPTOP ⇐ returnOPTOP
```

Perform a privileged and atomic update of the OPTOP and OPLIM registers.

Format

extend
update_optop

Forms

extend = 255 (0xff)
priv_update_optop = 63 (0x3f)

Stack

…, new_oplim, new_optop ⇒
…

Description

priv_update_optop updates the OPTOP and OPLIM registers atomically. priv_update_optop never allows the inconsistent state when either OPTOP or OPLIM obtains the new value, while the other register retains the original value. This property is particularly useful during context switches or while changing stack chunks.

If PSR.SU = 0, then priv_update_optop generates a privileged_instruction trap.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
OPLIM ⇐ stack[OPTOP + 8]
OPTOP ⇐ stack[OPTOP + 4]
```

priv_write_dcache_data priv_write_dcache_data

Perform a privileged diagnostic write of the data cache data array.

Format

extend
write_dcache_data

Forms

`extend` = 255 (0xff)
`priv_write_dcache_data` = 39 (0x27)

Stack

…, *data*, *data_address* ⇒
…

Description

`priv_write_dcache_data` writes a word directly into the data cache data array. The word to be written in the data cache data array is specified by the value on the top of the stack, *data_address*. *data_address* is decoded as shown in FIGURE 6-7.

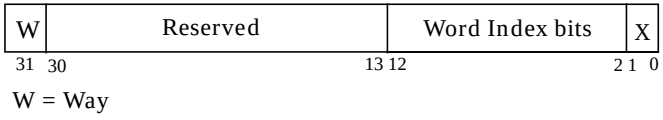


FIGURE 6-7 Format for Data Cache Data Address for 16-Kbyte Data Cache

Bit 31 of *data_address* specifies which “way” of the data cache is to be accessed. Depending on the size of the data cache, the Word Index bits, below, of *data_address* specify which word in the way is written.

Data Cache Size	Word Index Bits
1 Kbytes	<8:2>
2 Kbytes	<9:2>
4 Kbytes	<10:2>
8 Kbytes	<11:2>
16 Kbytes	<12:2>

If PSR.SU = 0, then priv_write_dcache_data generates a privileged_instruction trap.

If no data cache is present, then priv_write_dcache_data pops the two words on the top of the stack and does nothing.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.DCS ≠ 0) then
            dcache_data[stack[OPTOP + 4]] ⇐ stack[OPTOP + 8]]
OPTOP ⇐ OPTOP + 8
```

Perform a privileged diagnostic write of data cache tags.

Format

extend
write_dcache_tag

Forms

`extend` = 255 (0xff)
`priv_write_dcache_tag` = 38 (0x26)

Stack

…, *tag_data*, *tag_address* ⇒
…

Description

`priv_write_dcache_tag` writes a word directly into the data cache tag array. The word to be written in the data cache tag array is specified by the value on the top of the stack, *tag_address*. *tag_address* is decoded as shown in FIGURE 6-8.

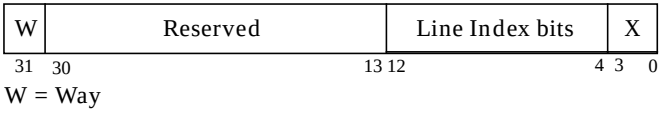


FIGURE 6-8 Format for Data Cache Tag Address for 16-Kbyte Data Cache

Bit 31 of *tag_address* specifies which "way" of the data cache is to be accessed. Depending on the size of the data cache, the Line Index bits, below, of *tag_address* specify which data cache line tag is written.

Data Cache Size	Line Index Bits	Data Cache Tag Bits
1 Kbytes	<8:4>	<29:9>
2 Kbytes	<9:4>	<29:10>
4 Kbytes	<10:4>	<29:11>
8 Kbytes	<11:4>	<29:12>
16 Kbytes	<12:4>	<29:13>

The *tag_data* to be written is of the format shown in FIGURE 6-9. The Data Cache Tag bits vary, based on the size of the data cache as above. The Valid bit of *tag_data* specifies whether this data cache line contains valid data. The Dirty bit specifies whether the corresponding cache line needs to be written back to memory when it is replaced or flushed. Finally, the LRU bit of the *tag_data* specifies whether way 0 or way 1 is the least recently used.

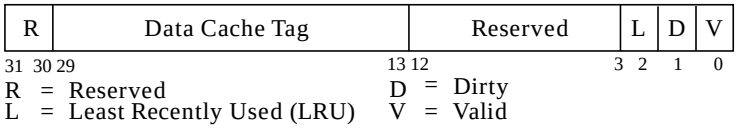


FIGURE 6-9 Format for Data Cache Tag Data for 16-Kbyte Data Cache

If PSR.SU = 0, then `priv_write_dcache_tag` generates a `privileged_instruction` trap.

If no data cache is present, then `priv_write_dcache_tag` pops the two words on the top of the stack and does nothing.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.DCS ≠ 0) then
            dcache_tag[stack[OPTOP + 4]] ⇐ stack[OPTOP + 8]]
OPTOP ⇐ OPTOP + 8
```

priv_write_icache_data priv_write_icache_data

Perform a privileged diagnostic write of the instruction cache data array.

Format

extend
write_icache_data

Forms

`extend` = 255 (0xff)
`priv_write_icache_data` = 47 (0x2f)

Stack

…, *data*, *data_address* ⇒
…

Description

`priv_write_icache_data` writes a word directly into the instruction cache data array. The word to be written in the instruction cache data array is specified by the value on the top of the stack, *data_address*. *data_address* is decoded as shown in FIGURE 6-10.

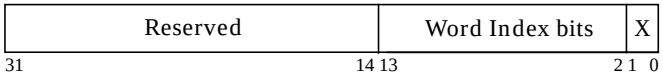


FIGURE 6-10 Format for Instruction Cache Data Address for 16-Kbyte Instruction Cache

Depending on the size of the instruction cache, the Word Index bits, below, of *data_address* specify which word in the instruction cache is written.

Instruction Cache Size	Word Index Bits
1 Kbytes	<9:2>
2 Kbytes	<10:2>
4 Kbytes	<11:2>
8 Kbytes	<12:2>
16 Kbytes	<13:2>

If `PSR.SU` = 0, then `priv_write_icache_data` generates a `privileged_instruction` trap.

If no instruction cache is present, then `priv_write_icache_data` pops the two words on the top of the stack and does nothing.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.ICS ≠ 0) then
            icache_data[stack[OPTOP + 4]] ⇐ stack[OPTOP + 8]]
OPTOP ⇐ OPTOP + 8
```

Perform a privileged diagnostic write of instruction cache tags.

Format

extend
write_icache_tag

Forms

`extend` = 255 (0xff)
`priv_write_icache_tag` = 46 (0x2e)

Stack

…, *tag_data*, *tag_address* ⇒
…

Description

`priv_write_icache_tag` writes a word directly into the instruction cache tag array. The word to be written in the instruction cache tag array is specified by the value on the top of the stack, *tag_address*. *tag_address* is decoded as shown in FIGURE 6-11.

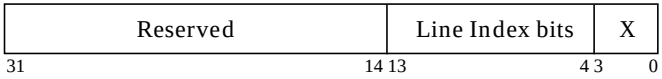


FIGURE 6-11 Format for 16-Kbyte Instruction Cache Tag Address

Depending on the size of the instruction cache, the Line Index bits, below, of *tag_address* specify which instruction cache line tag is written.

Instruction Cache Size	Line Index Bits	Instruction Cache Tag Bits
1 Kbytes	<9:4>	<29:10>
2 Kbytes	<10:4>	<29:11>
4 Kbytes	<11:4>	<29:12>
8 Kbytes	<12:4>	<29:13>
16 Kbytes	<13:4>	<29:14>

The *tag_data* to be written is of the format shown in FIGURE 6-12. The Instruction Cache Tag bits vary, based on the size of the instruction cache as above. The Valid bit of *tag_data* specifies whether this instruction cache line contains valid data.

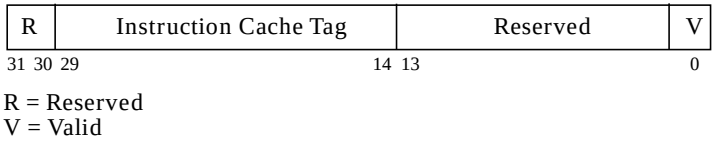


FIGURE 6-12 Format for Instruction Cache Tag Data for 16-Kbyte Instruction Cache

If PSR.SU = 0, then `priv_write_icache_tag` generates a `privileged_instruction` trap.

If no instruction cache is present, then `priv_write_icache_tag` pops the two words on the top of the stack and does nothing.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
if (HCR.ICS ≠ 0) then
            icache_tag[stack[OPTOP + 4]] ⇐ stack[OPTOP + 8]]
OPTOP ⇐ OPTOP + 8
```

Perform a privileged write of a machine register.

Format

extend
write_*reg*

Forms

`extend` = 255 (0xff)
`priv_write_`*reg* = 96 (0x60) through 121 (0x79) (see table below)

Stack

…, *value* ⇒
…

Description

`priv_write_`*reg* writes the contents of one of the machine registers with the value from the stack. The following table tabulates which `priv_write_`*reg* pushes which register, along with the opcodes.

Instruction	Machine Register	Opcode
priv_write_oplim	OPLIM	0x64
priv_write_psr	PSR	0x66
priv_write_trapbase	TRAPBASE	0x67
priv_write_lockcount0	LOCKCOUNT0	0x68
priv_write_lockcount1	LOCKCOUNT1	0x69
priv_write_lockaddr0	LOCKADDR0	0x6c
priv_write_lockaddr1	LOCKADDR1	0x6d
priv_write_userrange1	USERRANGE1	0x70
priv_write_userrange2	USERRANGE2	0x75
priv_write_gc_config	GC_CONFIG	0x71
priv_write_brk1a	BRK1A	0x72
priv_write_brk2a	BRK2A	0x73
priv_write_brk12c	BRK12C	0x74
priv_write_sc_bottom	SC_BOTTOM	0x79

If PSR.SU = 0, then priv_write_*reg* generates a privileged_instruction trap.

Operation

```
if (PSR.SU = 0) then
            trap privileged_instruction (type = 0x05)
reg ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Trap to emulation routine that resolves a constant pool item and writes a field in an object.

Format

putfield
indexbyte1
indexbyte2

Forms

putfield = 181 (0xb5)

Stack

…, *objectref*, *value* or …, *objectref*, *value<31:0>*, *value<63:32>* ⇒
…

Description

putfield traps to the emulation routine referenced by entry 0xb5 in the trap table.

Operation

```
trap putfield (type = 0xb5)
```

Recommendations

The trap handler should emulate putfield, as defined in *The Java Virtual Machine Specification*.

When the constant pool entry referenced by putfield is resolved, the putfield trap handler computes the offset for the field it references and determines the field type which, along with the size of the offset, in turn determines whether a putfield_quick, putfield_quick_w, putfield2_quick, or aputfield_quick opcode byte should replace the original putfield opcode byte.

If the putfield operates on a field determined dynamically to have an offset into the class instance data that corresponds to a one-word field that is of the type reference, then the putfield trap handler should replace the putfield instruction with aputfield_quick. Otherwise, if the offset into the object is less than or equal to 255 words, then the putfield instruction should be replaced with putfield_quick or putfield2_quick if the field is one or two words in size, respectively. Finally, if the offset is larger than 255 words, then the putfield trap handler should replace the putfield with putfield_quick_w.

Write a one-word field from an object.

Format

putfield_quick
index
<*unused*>

Forms

putfield_quick = 207 (0xcf)

Stack

…, *objectref*, *value* ⇒
…

Description

putfield_quick pops *objectref*, which must be of the type reference, and *value*, which must be one word in size, from the operand stack. It then writes *value* to the field at the offset *index* into the class instance referenced by *objectref*.

If *objectref* is null, then putfield_quick signals a NullPointer trap.

Operation

```
objectref ⇐ stack[OPTOP + 8]
if (objectref = 0) then
            trap NullPointer (type = 0x1b)
handle_bit ⇐ objectref & 0x00000001
if (handle_bit = 1) then
            addr_of_fields ⇐ mem[(objectref & 0x7ffffffc) + 4]
else
            addr_of_fields ⇐ (objectref & 0x7ffffffc) + 4
mem[addr_of_fields + (index × 4)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
```

Trap to emulation routine that writes a field in an object, with a wide index.

Format

putfield_quick_w
indexbyte1
indexbyte2

Forms

`putfield_quick_w` = 228 (0xe4)

Stack

…, *objectref*, *value* or …, *objectref*, *value<31:0>*, *value<63:32>* ⇒
…

Description

`putfield_quick_w` traps to the emulation routine referenced by entry 0xe4 in the trap table.

Operation

```
trap putfield_quick_w (type = 0xe4)
```

Recommendations

The trap handler should emulate `putfield_quick_w`, as defined in *The Java Virtual Machine Specification*. The `putfield_quick_w` trap handler can perform the required store quickly because the constant pool entry has already been resolved.

Write a two-word field from an object.

Format

putfield2_quick
index
<*unused*>

Forms

putfield2_quick = 209 (0xd1)

Stack

…, *objectref*, *value<31:0>*, *value<63:32>* ⇒
…

Description

putfield2_quick pops *objectref*, which must be of the type reference, and *value*, which must be two words in size, from the operand stack. It then writes *value* to the field at offset *index* into the class instance referenced by *objectref* and pushes it onto the stack.

If *objectref* is null, then putfield2_quick generates a NullPointer trap.

Operation

```
objectref ⇐ stack[OPTOP + 12]
if (objectref = 0) then
            trap NullPointer (type = 0x1b)
handle_bit ⇐ objectref & 0x00000001
if (handle_bit = 1) then
            addr_of_fields ⇐ mem[(objectref & 0x7ffffffc) + 4]
else
            addr_of_fields ⇐ (objectref & 0x7ffffffc) + 4
mem[addr_of_fields + (index × 4)] ⇐ stack[OPTOP + 4]
mem[addr_of_fields + (index × 4) + 4] ⇐ stack[OPTOP + 8]
OPTOP ⇐ OPTOP + 12
```

Trap to emulation routine that resolves constant pool item and writes a static field in a class.

Format

putstatic
indexbyte1
indexbyte2

Forms

putstatic = 179 (0xb3)

Stack

…, *value* or …, *value<31:0>*, *value<63:32>* ⇒
…

Description

putstatic traps to the emulation routine referenced by entry 0xb3 in the trap table.

Operation

```
trap putstatic (type = 0xb3)
```

Recommendations

The trap handler should emulate putstatic, as defined in *The Java Virtual Machine Specification*.

When the constant pool entry referenced by putstatic is resolved, the putstatic trap handler stores the address for the field it references into the constant pool. Depending on the type of the static field, a putstatic_quick, putstatic2_quick, or aputstatic_quick opcode byte should replace the original putstatic opcode byte.

If the putstatic operates on a field determined dynamically to correspond to a one-word field that is of the type reference, then the putstatic trap handler should replace the putstatic instruction with aputstatic_quick. Otherwise, if the field is one or two words in size, the putstatic instruction should be replaced with putstatic_quick or putstatic2_quick, respectively.

Write a static field in a class.

Format

putstatic_quick
indexbyte1
indexbyte2

Forms

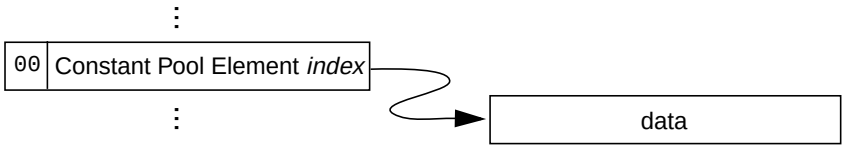
putstatic_quick = 211 (0xd3)

Stack

…, *value* ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a class (static) field. putstatic_quick pops *value* from the stack and writes it into this class field.



Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
addr_of_static ⇐ mem[CONST_POOL + (index × 4)]
mem[addr_of_static] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Write a two-word `static` field in a class.

Format

`putstatic2_quick`
indexbyte1
indexbyte2

Forms

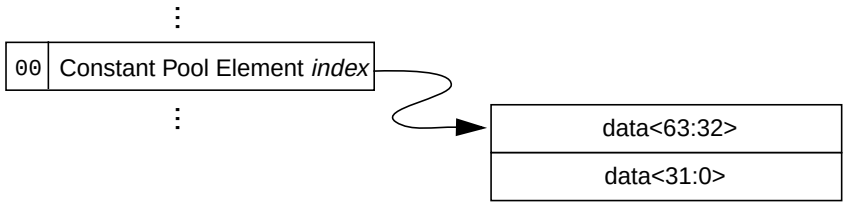
`putstatic2_quick` = 213 (0xd5)

Stack

…, *value<31:0>*, *value<63:32>* ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* construct an index into the constant pool of the current class, where the value of each index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item should have been resolved to be a pointer to a two-word class (`static`) field. `putstatic2_quick` pops the two words of *value* from the stack and writes it into this class field.



Operation

```
index ⇐ ((indexbyte1 << 8) | indexbyte2)
addr_of_static ⇐ mem[CONST_POOL + (index × 4)]
mem[addr_of_static + 4] ⇐ stack[OPTOP + 8]
mem[addr_of_static] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 8
```

Read a machine register.

Format

extend
read_*reg*

Forms

extend = 255 (0xff)
read_*reg* = 64 (0x40) through 89 (0x59) (See table below.)

Stack

… ⇒
…, *value*

Description

read_*reg* pushes the contents of one of the machine registers onto the stack. The following table tabulates which read_*reg* pushes which register, along with the opcodes.

Instruction	Machine Register	Opcode
read_pc	PC	0x40
read_vars	VARS	0x41
read_frame	FRAME	0x42
read_optop	OPTOP	0x43
read_const_pool	CONST_POOL	0x45
read_global0	GLOBAL0	0x5a
read_global1	GLOBAL1	0x5b
read_global2	GLOBAL2	0x5c
read_global3	GLOBAL3	0x5d

Operation

stack[OPTOP] ⇐ *reg*
OPTOP ⇐ OPTOP - 4

Return from a subroutine.

Format

ret
index

Forms

`ret` = 169 (0xa9)

Stack

… ⇒
…

Description

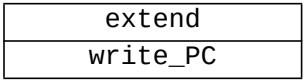
Return control to the PC stored in local variable *index*.

Operation

`PC ⇐ stack[VARS – (`*index* × `4)]`

Return from a subroutine with a return address from the stack.

Format



Forms

`extend` = 255 (0xff)
`write_PC` = 96 (0x60)

Stack

…, *return_address* ⇒
…

Description

`ret_from_sub`, an assembler synonym of `write_PC`, transfers control to the address from the top of the stack.

Operation

```
PC ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4
```

Return from a method.

Format

return

Forms

`return` = 177 (0xb1)

Stack

… ⇒
[empty]

Description

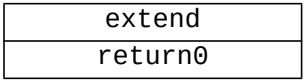
`return` returns to the caller of this method, popping all the arguments to the current method.

Operation

```
PC ⇐ stack[FRAME]
CONST_POOL ⇐ stack[FRAME - 12]
VARS ⇐ stack[FRAME - 4]
FRAME ⇐ stack[FRAME - 8]
OPTOP ⇐ VARS
```

Return with no value from a routine entered via `call`.

Format



Forms

`extend` = 255 (0xff)
`return0` = 13 (0x0d)

Stack

…, *returnVARS*, *return_address* ⇒
…

Description

Assuming that the return PC and the return VARS addresses are on the top of the stack, `return0` transfers control to the specified return address and updates the VARS register with the specified return value. It then adjusts the top of the stack to point to the original VARS value.

Thus, `return0` effects a return from a routine that is entered via `call` and pops all the arguments that are passed into the call from the caller's stack.

Operation

```
returnOPTOP ⇐ VARS
PC ⇐ stack[OPTOP + 4]
VARS ⇐ stack[OPTOP + 8]
OPTOP ⇐ returnOPTOP
```

Return with one-word value from a routine entered via `call`.

Format

extend
return1

Forms

`extend` = 255 (0xff)
`return1` = 29 (0x1d)

Stack

…, returnVARS, return_address, return_value ⇒
…, return_value

Description

Assuming that a one-word return value, the return PC, and the return VARS address are on the top of the stack, `return1` transfers control to the specified PC address and updates the VARS register with the specified value. It then adjusts the top of the stack to point to the original VARS value and pushes the return value.

Thus, `return1` effects a return from a routine that is entered via `call`, pops all the arguments that are passed into the call from the caller's stack, and pushes a one-word result.

Operation

```
returnOPTOP ⇐ VARS - 4
returnVal ⇐ stack[OPTOP + 4]
PC ⇐ stack[OPTOP + 8]
VARS ⇐ stack[OPTOP + 12]
OPTOP ⇐ returnOPTOP
stack[OPTOP + 4] ⇐ returnVal
```

Return with a two-word value from a routine entered via `call`.

Format

extend
return2

Forms

`extend` = 255 (0xff)
`return2` = 45 (0x2d)

Stack

…, returnVARS, return_address, return_value1, return_value2 ⇒
…, return_value1, return_value2

Description

Assuming that a two-word return value, the return PC, and the return VARS address are on the top of the stack, `return2` transfers control to the specified PC address and updates the VARS register with the specified value. It then adjusts the top of the stack to point to the original VARS value and pushes the return value.

Thus, `return2` effects a return from a routine that is entered via `call`, pops all the arguments that are passed into the call from the caller's stack, and pushes a two-word result.

Operation

```
returnOPTOP ⇐ VARS - 8
returnVal1 ⇐ stack[OPTOP + 8]
returnVal2 ⇐ stack[OPTOP + 4]
PC ⇐ stack[OPTOP + 12]
VARS ⇐ stack[OPTOP + 16]
OPTOP ⇐ returnOPTOP
stack[OPTOP + 8] ⇐ returnVal1
stack[OPTOP + 4] ⇐ returnVal2
```

Load a short from an array.

Format

saload

Forms

saload = 53 (0x35)

Stack

…, *arrayref*, *index* ⇒
…, *value*

Description

saload treats *arrayref* as a reference to an array of shorts. It loads the two-byte element at *index*, sign-extends the result, and pushes it onto the stack as *value*.

If *arrayref* is null, then saload takes a NullPointer trap. If *index* is not within the bounds of the array referenced by *arrayref*, then saload takes an ArrayIndexOutOfBounds trap.

Operation

```
arrayref ⇐ stack[OPTOP + 8]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 4]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
stack[OPTOP + 8] ⇐ sign_ext16(mem16[addr_of_length + 4 + (index × 2)])
OPTOP ⇐ OPTOP + 4
```

Store a short to an array.

Format

sastore

Forms

`sastore` = 86 (0x56)

Stack

…, arrayref, index, value ⇒
…

Description

`sastore` treats *arrayref* as a reference to an array of shorts. It truncates the integer *value* on the stack to the low 16 bits and stores it to the two-byte element at *index* of the array.

If *arrayref* is `null`, then `sastore` takes a `NullPointer` trap. If *index* is not within the bounds of the array referenced by *arrayref*, then `sastore` takes an `ArrayIndexOutOfBounds` trap.

Operation

```
arrayref ⇐ stack[OPTOP + 12]
if (arrayref = 0) then
            trap NullPointer (type = 0x1b)
index ⇐ stack[OPTOP + 8]
if (index < 0) then
            trap ArrayIndexOutOfBounds (type = 0x19)
handle_bit ⇐ arrayref & 0x00000001
if (handle_bit = 1) then
            addr_of_length ⇐ mem[(arrayref & 0x7ffffffc) + 4]
else
            addr_of_length ⇐ (arrayref & 0x7ffffffc) + 4
length ⇐ mem[addr_of_length]
if (index ≥ length) then
            trap ArrayIndexOutOfBounds (type = 0x19)
mem16[addr_of_length + 4 + (index × 2)] ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 12
```

Notes

In the picoJava-II core, `sastore` is identical to `castore`.

Set the upper 16 bits of the top element of the stack.

Format

sethi
byte1
byte2

Forms

`sethi` = 237 (0xed)

Stack

…, *value* ⇒
…, *result*

Description

`sethi` sets the upper 16 bits of *value* to the value of the 16-bit operand of `sethi`.

Operation

```
highbits ⇐ ((byte1 << 8) | byte2) << 16
stack[OPTOP + 4] ⇐ highbits | (stack[OPTOP + 4] & 0x0000ffff)
```

Notes

Use `sethi` with `sipush` to create 32-bit constants.

Push signed 16-bit constant.

Format

sipush
byte1
byte2

Forms

`sipush` = 17 (0x11)

Stack

… ⇒
…, *value*

Description

`sipush` sign-extends the constant with value (*byte1* << 8) | *byte2* and pushes it onto the operand stack.

Operation

`stack[OPTOP]` ⇐ $\texttt{sign\_ext}_{16}$`((`*byte1* `<< 8) |` *byte2*`)`
`OPTOP` ⇐ `OPTOP – 4`

Initiate a software trap.

Format

extend
soft_trap

Forms

`extend` = 255 (0xff)
`soft_trap` = 37 (0x25)

Stack

… ⇒
…

Description

`soft_trap` causes a trap of the type `soft_trap` (0x0d).

Operation

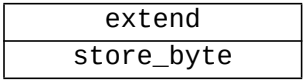
```
trap soft_trap (type = 0x0d)
```

Notes

`soft_trap` is generally used to initiate calls into the underlying operating system.

Store byte to memory.

Format



Forms

extend = 255 (0xff)
store_byte = 32 (0x20)

Stack

…, *value*, *address*⇒
…

Description

store_byte stores the low 8 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
mem8[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Store an 8-bit value at a fixed offset from the address in a local variable to memory.

Format

store_byte_index
local_var
offset

Forms

store_byte_index = 246 (0xf6)

Stack

…, *value* ⇒
…

Description

store_byte_index pops and stores the 8-bit *value* on the stack at the memory location at the effective address. It computes the effective address by loading the contents of the local variable, *local_var,* and adding the signed value of *offset*.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + sign_ext8(offset)
data ⇐ stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
mem8[eff_addr] ⇐ data
OPTOP ⇐ OPTOP + 4
```

Notes

store_byte_index is equivalent to the following sequence of instructions:

iload *local_var*; bipush *offset*; iadd; store_byte

Store `short` or `char` to memory.

Format

extend
store_short

Forms

`extend` = 255 (0xff)
`store_short` = 34 (0x22)

Stack

…, *value*, *address* ⇒
…

Description

`store_short` stores the low 16 bits of *value* at the memory location at *address*, then pops both *value* and *address* from the stack. If bit 30 of *address* is set to 1, then the data is stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if ((address & 0x40000000) ≠ 0) then
            data ⇐ endian_swap16(data)
mem16[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Store a 16-bit value at a fixed offset from the address in a local variable to memory.

Format

store_short_index
local_var
offset

Forms

store_short_index = 245 (0xf5)

Stack

…, *value* ⇒
…

Description

store_short_index pops the 16-bit *value* on the stack and stores it at the memory location at the effective address, which it computes by loading the contents of the local variable, *local_var*, and adding the signed value of *offset* × 2. If bit 30 of the effective address is set to 1, then the data is stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The resulting address must be aligned on a 16-bit boundary.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + (sign_ext8(offset) × 2)
data ⇐ stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
if ((eff_addr & 0x40000000) ≠ 0) then
            data ⇐ endian_swap16(data)
mem16[eff_addr] ⇐ data
OPTOP ⇐ OPTOP + 4
```

Notes

store_short_index is equivalent to the following sequence of instructions:

iload *local_var*; bipush *offset*; iconst_1; ishl; iadd; store_short

Use opposite endianness to store `short` to memory.

Format

extend
store_short_oe

Forms

`extend` = 255 (0xff)
`store_short_oe` = 42 (0x2a)

Stack

…, *value*, *address* ⇒
…

Description

`store_short_oe` stores the low 16 bits of *value* at the memory location at *address*. It then pops both *value* and *address* from the stack. If bit 30 of *address* is not set to 1, then the data is stored in little endian order.

If `PSR.ACE` is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the `PSR.CAC` bit.

The address must be aligned on a 16-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if ((address & 0x40000000) = 0) then
            data ⇐ endian_swap16(data)
mem16[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Store `integer` to memory.

Format

extend
store_word

Forms

`extend` = 255 (0xff)
`store_word` = 36 (0x24)

Stack

…, *value*, *address* ⇒
…

Description

`store_word` stores the 32 bits of *value* at the memory location at *address*. It then pops both *value* and *address* from the stack. If bit 30 of *address* is set to 1, then the data is stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if ((address & 0x40000000) ≠ 0) then
            data ⇐ endian_swap16(data)
mem[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Store a word at a fixed offset from the address in a local variable to memory.

Format

store_word_index
local_var
offset

Forms

store_word_index = 243 (0xf3)

Stack

…, *value* ⇒
…

Description

store_word_index pops and stores the 32-bit *value* on the stack at the memory location at the effective address, which it computes by loading the contents of the local variable, *local_var*, and adding the signed value of *offset* × 4. If bit 30 of the effective address is set to 1, then the data is stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The resulting address must be aligned on a 32-bit boundary.

Operation

```
eff_addr ⇐ stack[VARS - (local_var × 4)] + (sign_ext8(offset) × 4)
data ⇐ stack[OPTOP + 4]
if (PSR.ACE = 1) then
            if (addr_out_of_range(eff_addr)) then
                        trap mem_protection_error (type 0x02)
if ((eff_addr & 0x40000000) ≠ 0) then
            data ⇐ endian_swap16(data)
mem[eff_addr] ⇐ data
OPTOP ⇐ OPTOP + 4
```

Notes

store_word_index is equivalent to the following sequence of instructions:

```
iload local_var; bipush offset; iconst_2; ishl; iadd; store_word
```

Use opposite endianness to store an integer to memory.

Format

extend
store_word_oe

Forms

extend = 255 (0xff)
store_word_oe = 44 (0x2c)

Stack

…, value, address ⇒
…

Description

store_word_oe stores the 32 bits of *value* at the memory location at *address*. It then pops both *value* and *address* from the stack. If bit 30 of *address* is not set to 1, then the data is stored in little endian order.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

The address must be aligned on a 32-bit boundary.

Operation

```
address ⇐ stack[OPTOP + 4]
data ⇐ stack[OPTOP + 8]
if (PSR.ACE = 1) then
           if (addr_out_of_range(address)) then
                      trap mem_protection_error (type 0x02)
if ((address & 0x40000000) = 0) then
           data ⇐ endian_swap16(data)
mem[address] ⇐ data
OPTOP ⇐ OPTOP + 8
```

Swap two words on the top of the stack.

Format

```
swap
```

Forms

swap = 95 (0x5f)

Stack

…, *word1*, *word2* ⇒
…, *word2*, *word1*

Description

swap interchanges the top two words on the stack.

Operation

```
temp ⇐ stack[OPTOP + 8]
stack[OPTOP + 8] ⇐ stack[OPTOP + 4]
stack[OPTOP + 4] ⇐ temp
```

Access jump table by index and jump.

Format

tableswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
lowbyte1
lowbyte2
lowbyte3
lowbyte4
highbyte1
highbyte2
highbyte3
highbyte4
jump offsets...

Forms

`tableswitch` = 170 (0xaa)

Stack

…, *index* ⇒
…

Description

The top of the stack is treated as an integer *index* into the word-aligned jump table that follows the `tableswitch` opcode. The first three words of the jump table are a default jump offset, a *low* index, and a *high* index. After the three initial words, *high* – *low* + 1, further signed 4-byte jump offsets complete the table. If the *index* from the top of the stack is less than the *low* index or greater than the *high* index, then control is transferred to the address at the default jump offset from the PC of the `tableswitch` opcode. Otherwise, the core uses the jump offset word 3 + *index* – *low* entries into the jump table to add to the PC of the `tableswitch` opcode to compute the next execution address.

Operation

```
tablestart ⇐ (PC + 4) & 0xfffffffc
low ⇐ mem[tablestart + 4]
high ⇐ mem[tablestart + 8]
index ⇐ stack[OPTOP + 4]
if ((index < low) OR (index > high) then
            PC ⇐ mem[tablestart]
else
            PC ⇐ mem[(tablestart + 12 + ((index - low) × 4))]
```

Trap to emulation routine that performs local variable accesses or updates with extended index.

Format

wide
opcode
...

Forms

`wide` = 196 (0xc4)

Stack

…, <*varies*> ⇒
…, <*varies*>

Description

`wide` traps to the emulation routine referenced by entry 0xc4 in the trap table.

Operation

```
trap wide (type = 0xc4)
```

Recommendations

The trap handler should emulate `wide`, as defined in *The Java Virtual Machine Specification*.

The operation depends on the opcode byte that follows the `wide` opcode byte.

Write a machine register.

Format

extend
write_*reg*

Forms

extend = 255 (0xff)
write_*reg* = 96 (0x60) through 121 (0x79) (see table below)

Stack

…, *value* ⇒
…

Description

write_*reg* writes the contents of one of the machine registers with the value from the stack. The following table tabulates which write_*reg* writes which register, along with the opcodes.

Instruction	Machine Register	Opcode
write_pc	PC	0x60
write_vars	VARS	0x61
write_frame	FRAME	0x62
write_optop	OPTOP	0x63
write_const_pool	CONST_POOL	0x65
write_global0	GLOBAL0	0x7a
write_global1	GLOBAL1	0x7b
write_global2	GLOBAL2	0x7c
write_global3	GLOBAL3	0x7d

Operation

reg ⇐ stack[OPTOP + 4]
OPTOP ⇐ OPTOP + 4

Set a cache line to valid, dirty, and zero all the data.

Format

extend
zero_line

Forms

extend = 255 (0xff)
zero_line = 62 (0x3e)

Stack

…, *address* ⇒
…

Description

address specifies a line in the data cache that is to be zeroed. zero_line allocates the cache line that contains this address in the data cache, without fetching the corresponding data from memory, and initializes it to 0. Because all the bytes in the data cache line are to be written as zeroes, zero_line does not need to read the corresponding line from main memory in the event of a cache miss.

If the data cache is off (PSR.DCE = 0), then zero_line traps to an emulation routine.

If PSR.ACE is set to 1, then the address checking process described in *Memory Protection* on page 27 is performed, regardless of the state of the PSR.CAC bit.

Operation

```
address ⇐ stack[OPTOP + 4] & 0x7ffffff0
if (PSR.ACE = 1) then
            if (addr_out_of_range(address)) then
                        trap mem_protection_error (type 0x02)
if (PSR.DCE = 0) then
            trap zero_line (type 0x29)
else
            dindex_mask ⇐ ((1 << (HCR.DCS + 8)) - 1)
            dtag_mask ⇐ dindex_mask ^ 0x7fffffff
            dtag_address0 ⇐ address & dindex_mask
            dtag_address1 ⇐ dtag_address0 | 0x80000000
            dtag0 ⇐ dcache_tag[dtag_address0] & dtag_mask
            dtag1 ⇐ dcache_tag[dtag_address1] & dtag_mask
            lru_way ⇐ (dcache_tag[dtag_address0] & 0x00000004) >> 2
            dtag_to_match ⇐ address & dtag_mask
            dirty_valid_bits ⇐ 0
```

```
            if (dtag0 = dtag_to_match) then
                        dtag_address ⇐ dtag_address0
            else if (dtag1 = dtag_to_match) then
                        dtag_address ⇐ dtag_address1
            else if (lru_way = 1) then
                        dtag_address ⇐ dtag_address0
                        dtag ⇐ dcache_tag[dtag_address]
                        dirty_valid_bits ⇐ dtag & 0x0000003
            else
                        dtag_address ⇐ dtag_address1
                        dtag ⇐ dcache_tag[dtag_address]
                        dirty_valid_bits ⇐ dtag & 0x0000003
            if (dirty_valid_bits = 0x3) then
                        mem_addr ⇐ (dtag | dtag_address) & 0x7ffffff0
                        mem_NC[mem_addr] ⇐ dcache_data[dtag_address]
                        mem_NC[mem_addr + 4] ⇐ dcache_data[dtag_address + 4]
                        mem_NC[mem_addr + 8] ⇐ dcache_data[dtag_address + 8]
                        mem_NC[mem_addr + 12] ⇐ dcache_data[dtag_address + 12]
            dtag ⇐ dtag_to_match | (dtag_address >> 29) | 0x00000003
            dcache_tag[dtag_address] ⇐ dtag
            dcache_data[dtag_address] ⇐ 0
            dcache_data[dtag_address + 4] ⇐ 0
            dcache_data[dtag_address + 8] ⇐ 0
            dcache_data[dtag_address + 12] ⇐ 0
OPTOP ⇐ OPTOP + 4
```

Notes

Although the picoJava-II core has 16-byte cache lines for the data cache, you should rely on the value in the DCL field of the Hardware Configuration Register (HCR) to facilitate porting software between implementations—the HCR.DCL field indicates the number of bytes in a data cache line, which will be the number of bytes written to zero by `zero_line`.

Recommendations

The trap handler should write the appropriate number of zeroes to main memory as if the cache is on.

PART II Programming the picoJava-II Core

CHAPTER 7

Java Method Invocation and Return

The picoJava-II core allocates and deallocates a new method frame upon each Java method invocation and return, respectively.

A frame provides storage both for local variables and for an operand stack to support the execution of the invoked method. The frame may also contain incoming arguments passed to the invoked method. Similarly, the operand stack may include outgoing arguments passed to another method to be invoked.

The core also saves method context information in each new frame for use in later restoration of the invoker's frame. This information includes the return PC, VARS, FRAME, and CONST_POOL registers. When a method returns, the frame can forward a return value to the invoker.

This chapter describes these actions in the following sections:

- *Allocating a New Frame* on page 385
- *Invoking a Method* on page 388
- *Invoking a Synchronized Method* on page 391
- *Returning from a Method* on page 393

7.1 Allocating a New Frame

All method invocations use the same process of creating a method frame. When the core invokes a method, it allocates a new method frame, which then becomes the current frame. Depending on the situation, the frame may contain some or all of the following entities:

- Object reference
- Incoming arguments
- Local variables
- Invoker's method context (always present)

- Operand stack (always present)
- Return value from a method invocation

FIGURE 7-1 illustrates the method frame.

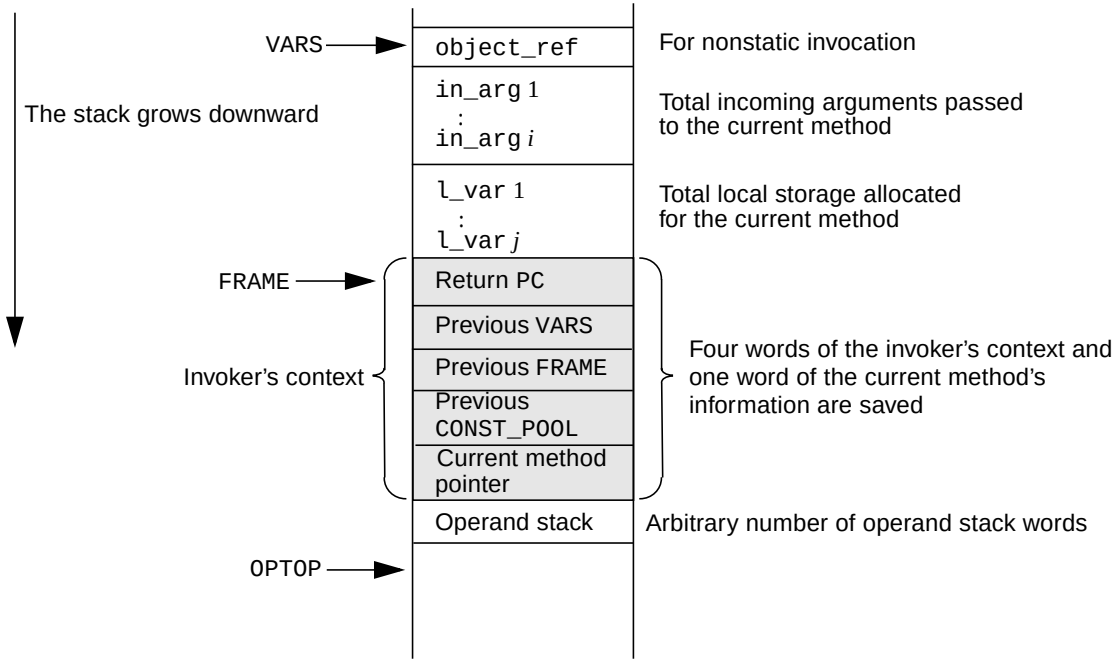


FIGURE 7-1 A Method Frame

7.1.1 Incoming Arguments

Incoming arguments transfer information from an invoker to an invoked method. Similar to an object reference, arguments are pushed onto the operand stack by compiler-generated instructions by the caller and can be accessed as local variables by the invoked method.

A Java compiler statically produces a method structure containing the number of arguments:

- For a nonstatic method invocation, the object reference and the first argument are accessible as local variable 0 and local variable 1, respectively.
- For a static method invocation, the first argument becomes local variable 0.

Note – A 64-bit value is pushed onto the stack such that it appears that the least significant 32 bits of the value are pushed onto the stack, followed by the most significant 32 bits. This convention is consistent with the picoJava-II model of a downward-growing stack in big-endian addressing mode.

7.1.2 Local Variables

When the core invokes a method, it allocates an area on the stack for storage of local variables.

A Java compiler statically determines the number of local variable words that are required; the core allocates them accordingly.

7.1.3 Invoker's Method Context

When a new frame is built for the current method, the core pushes this information onto the newly allocated frame and later uses it to restore the invoker's method context before returning.

The method context consists of return `PC`, `VARS`, `FRAME`, and `CONST_POOL` registers. The method pointer word in the method context area represents the current method context, not the invoker's context. The method pointer refers to the method structure of the current method. See *Method Structure on page 72*, for additional information.

7.1.4 Operand Stack

The core uses the operand stack area:

- To provide the source and target operands for various instructions
- To hold the arguments and return values of other Java methods invoked by this method

7.2 Invoking a Method

The following is the picoJava-II procedure for invoking a method:

1. **Resolve a method reference.**
2. **Access the method structure.**
3. **Allocate a new method frame.**
4. **Save the invoker's method context.**
5. **Pass control to the invoked method by branching to the method's entry point.**

7.2.1 Resolving a Method Reference

Typically, the first time the core encounters a method call site, the invoke instruction refers to a constant pool entry that provides symbolic information on the method to be invoked, such as its name and argument types, as described in *The Java Virtual Machine Specification*. Depending on the invoke type, software in the emulation trap routines should use this symbolic information to determine one of the following:

- An index to the method vector (see *Method Vector and Runtime Class Info Structure* on page 71), which the core then uses to look up a method structure pointer
- A direct pointer to a method structure, described on *Method Structure on page 72*

Resolving a method reference may involve class loading and resolution with subsequent method searches based on the referenced method name and signature.

7.2.2 Accessing a Method Structure

The core obtains the method information—the number of arguments, the size of local variable storage, and the method entry point—from the method structure that was built for the invoked method during class loading.

The core saves this method structure pointer in the current method frame.

7.2.3 Allocating a New Method Frame

When the core invokes a method, it allocates a new frame and initializes the following registers:

- VARS — Location of the callee's first argument, either the implicit object reference or the caller's first actual argument.
- FRAME — Location of the first word of the invoker's method context, which is where the return PC is saved. This location is offset from VARS by the total number of bytes of both the incoming arguments and local variables.
- OPTOP — Location of the first empty stack word after allocating a number of locals and saving the invoker's context: FRAME – 20.

See FIGURE 7-2 for details.

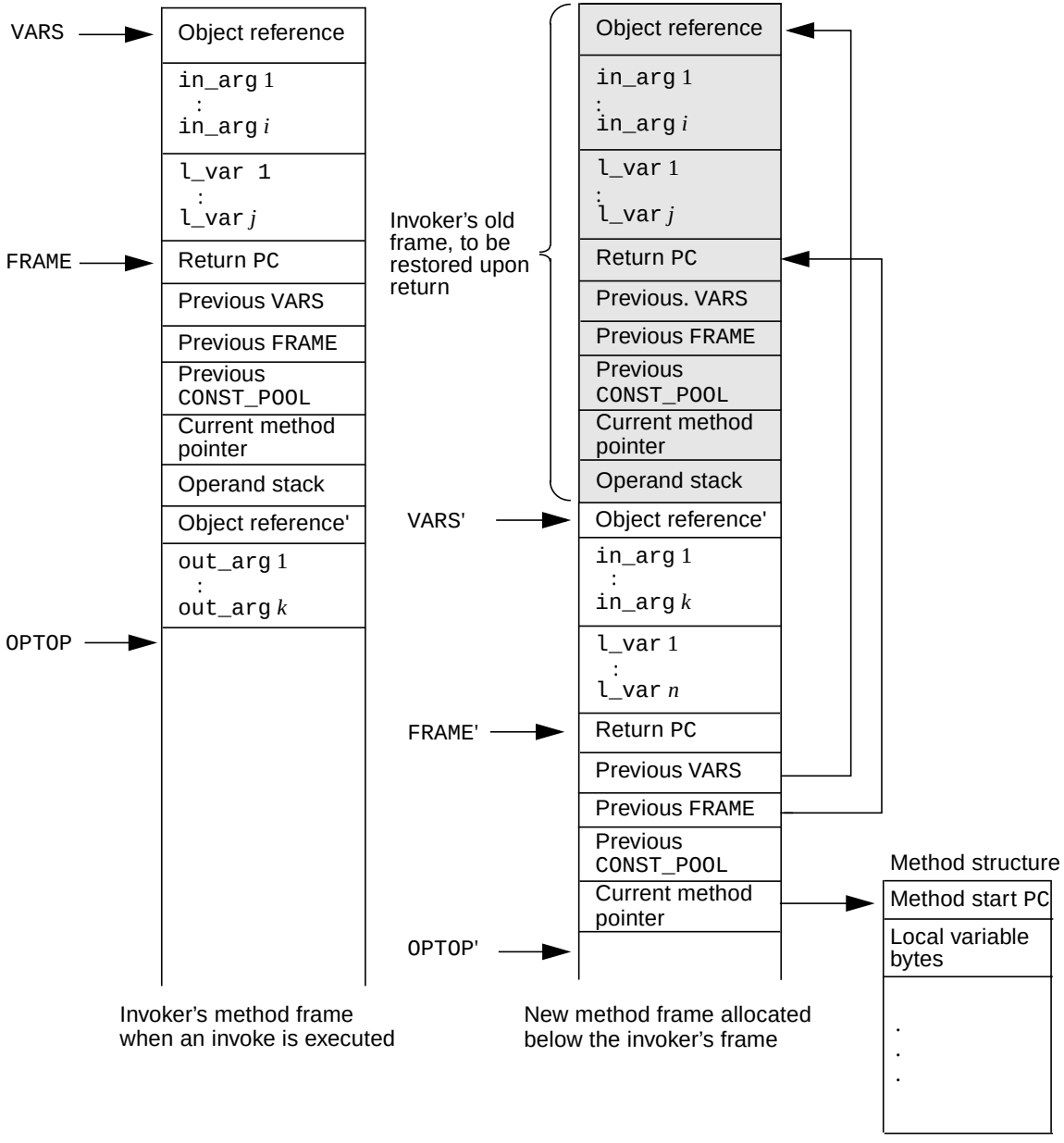


FIGURE 7-2 Allocation of a New Frame

7.2.4 Saving the Invoker's Method Context

The core saves the invoker's context in the newly allocated frame so that it can restore the invoker's frame when the current method returns. Following is the definition for five words of method context:

Return PC	Location of the Java instruction next to the invoke instruction
Return VARS	Location of the calling method's local variable starts
Return FRAME	Location of the calling method's frame
Return CONST_POOL	Pointer to the calling method's constant pool table
Current method pointer	Pointer to the method structure of the current method

7.2.5 Passing Control to the Invoked Method

When the above steps are complete, execution branches to the program counter (PC) value, which is specified as the method entry point in the method structure.

7.3 Invoking a Synchronized Method

A synchronized, nonstatic method must enter the monitor associated with the object reference on the stack before execution of that method. Similarly, a synchronized static method must enter the monitor associated with the class of the current method. Upon a return resulting from a `return` instruction or exception handling from this method, the monitor must be exited.

The core requires explicit `monitorenter` and `monitorexit` instructions to perform the necessary lock acquisitions and releases, thus keeping the invokes and returns simple. The class loader modifies synchronized methods to meet this requirement.

The class loader must take the following steps:

1. **Replace all `return`, `areturn`, `ireturn`, `freturn`, `lreturn`, and `dreturn` instructions with the `exit_sync_method` instruction.**

   `exit_sync_method` is a 1-byte instruction (which avoids the need to update branch offsets in the original code). It simply branches to the PC that is stored at FRAME – 20.

2. **Insert code to the beginning of the method to execute `monitorenter` and `monitorexit` explicitly.**

TABLE 7-1 and TABLE 7-2 list the code that the class loader must prepend to the code for nonstatic and static methods, respectively.

TABLE 7-1 Code Prepended to Synchronized Nonstatic Methods

Address	Instruction	Action
0:	`aload_0`	Get the object reference.
1:	`monitorenter`	Synchronize on the object reference.
2:	`jsr` + 6	Push `PC` on top of the stack, which is also `FRAME` – 20, and jump to the next instruction.
5:	`aload_0`	Get the object reference.
6:	`monitorexit`	Exit the monitor.
7:	*x*`return`	Return to the caller of the correct type.
8:	Original code for the method with `exit_sync_method`; replace all *x*`return`s in the original.	

TABLE 7-2 Code Prepended to Synchronized Static Methods

Address	Instruction	Action
0:	`get_current_class`	Get the current class pointer.
2:	`monitorenter`	Synchronize on the class pointer.
3:	`jsr` + 9	Push `PC` on top of the stack, which is also `FRAME` – 20, and jump to the original code.
6:	`get_current_class`	Get the current class pointer.
8:	`monitorexit`	Exit the monitor.
9:	*x*`return`	Return to the caller of the correct type.
10:	`nop`	
11:	`nop`	Ensure that the code is a multiple of 4 bytes to prevent changes in padding for `lookupswitch` and `tableswitch`.
12:	Original code for the method with `exit_sync_method`; replace all *x*`return`s in the original.	

You must ensure that the second `aload_0` in the prologue for nonstatic methods returns the same object reference as the first one—that is, the local variable 0 cannot be modified within an instance method. If you develop synchronized methods in a low-level language, you must follow this rule. Similarly, no automatic optimizer should overload the contents of local variable zero in nonstatic methods.

3. **Change the exception table for the method so that** `start_pc`**,** `end_pc`**, and** `handler_pc` **of the entries are each incremented by 8 or 12 to map onto the now-relocated code.**

   When an exception is thrown to a synchronized method with no corresponding exception table entry, the exception handling agent (for example, the `athrow` trap) releases the acquired monitor before discarding the method frame.

   The above steps simplify the invoke and return hardware and cause an implicit operation for monitor release upon normal or abnormal completion of a synchronized method.

7.4 Returning from a Method

The following is the procedure for returning from a method:

1. **Pop off the input arguments.**
2. **Restore the invoker's method frame.**
3. **Forward a return value into the invoker's frame, if any.**
4. **Branch to the return** `PC` **to resume execution of the invoking method.**

   The core pops off the input arguments by restoring `OPTOP` to the value of the current (invoked) method's `VARS`. If a return value exists, the core pushes it onto the invoker's operand stack, adjusting `OPTOP` to provide stack space (either one or two words) on which to push the return value. The core restores the remainder of the old frame from the invoker's frame saving area.

   See FIGURE 7-3 for details.

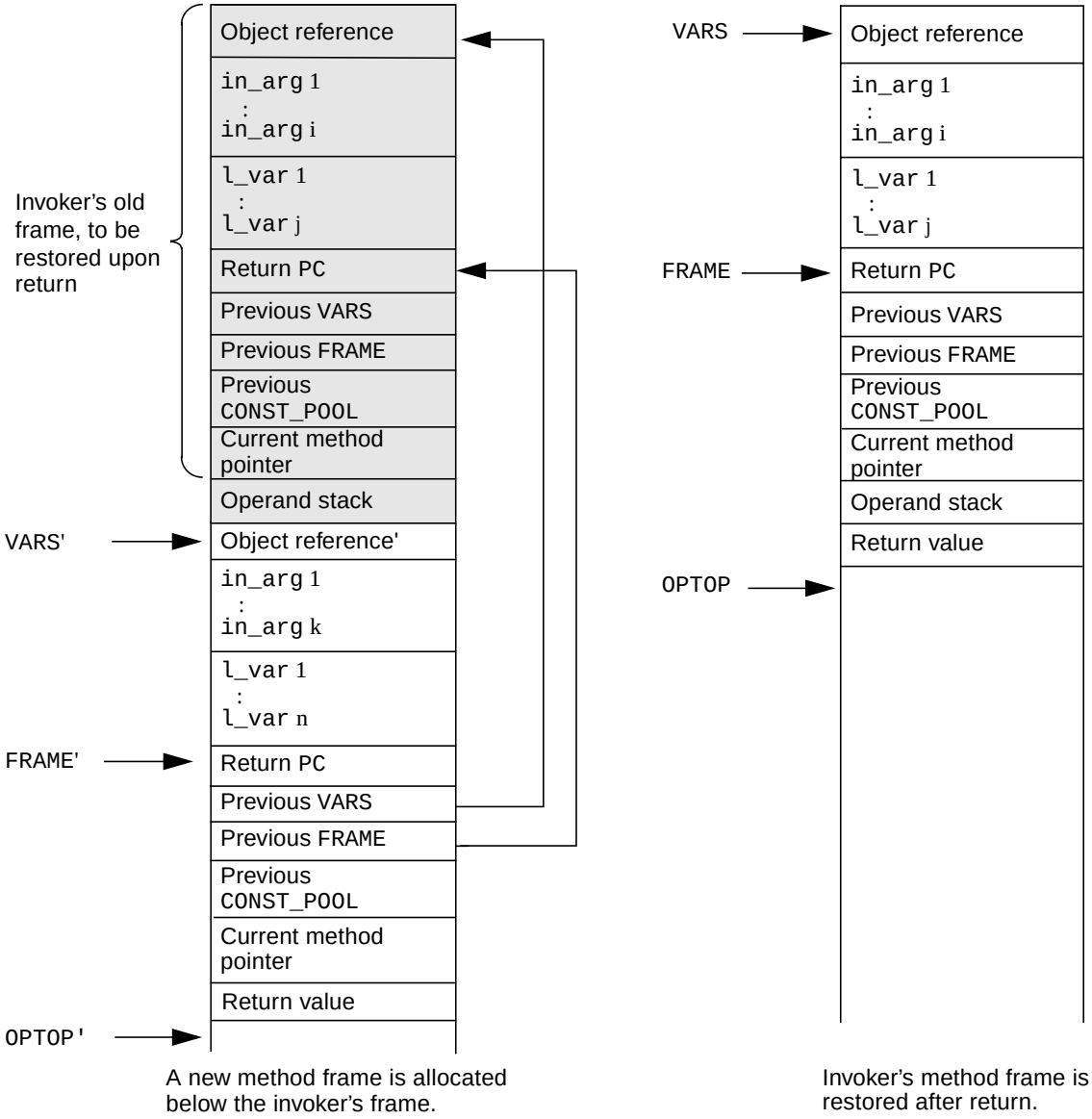


FIGURE 7-3 Return from a Method

CHAPTER 8

Monitors

To synchronize operations between concurrent threads of execution, the Java language uses *monitors*. A monitor is a high-level mechanism for allowing only one thread at a time to execute a region of code associated with an object that the monitor protects. To enter and exit a monitor for an object, the Java virtual machine uses the `monitorenter` and `monitorexit` instructions.

One frequent situation with monitors is that a thread of execution acquires a monitor (lock) on an unlocked object (perhaps reacquiring the lock many times), holds the lock for some time, and releases the lock as many times as it was acquired. Often, during the time a thread holds the lock, no other threads of execution attempt to acquire the lock on the same object. A thread can perform this lock-unlock sequence a number of times during its time slice before a context switch.

To speed up common situations in which monitors are used, execution of `monitorenter` and `monitorexit` is by means of hardware in the picoJava-II core. Because the core supports only uniprocessor Java execution, caching of the “locked” state of an object can occur in the core. Updates to the object in memory can then be deferred until context-switch time.

picoJava-II hardware is optimized for the typical case in which a thread has entered, at most, two monitors at one time and no other threads contend for these monitors.

This chapter discusses the following subjects that pertain to monitors:

- *Structures* on page 396
- *Hardware Synchronization* on page 396
- *Software Support* on page 397

8.1 Structures

Two LOCKADDR registers contain references to the objects or arrays for which a lock is acquired or released. A LOCKADDR register that contains a value of 0 is considered to be empty.

Note – If both LOCKADDR registers contain the same reference, the behavior of `monitorenter` and `monitorexit` is undefined.

Two LOCKCOUNT registers each contain an 8-bit counter to indicate the number of times a lock has been acquired for the object referenced by the corresponding LOCKADDR register. Bit 14 in the LOCKCOUNT registers, the LOCKWANT bit, indicates whether another thread is waiting for this lock. Bit 15, the CO bit, indicates whether the only record of the lock is the state of this LOCKADDR and LOCKCOUNT register pair.

Furthermore, each object header in memory has one bit reserved as a LOCK bit, which specifies whether the object is locked. Optionally, each object can maintain a pointer to a monitor data structure that tracks threads that may be waiting to acquire this monitor. If a per-object pointer field is not maintained, then software must provide a hashing scheme or other mechanism to find the monitor data structure for an object.

Rather than the “locked” status of a current object being strictly kept in memory, an object is locked if its address is contained in a LOCKADDR register and its corresponding LOCKCOUNT is nonzero, or if the object address is not in any LOCKADDR register and the LOCK bit is set in the object header.

Note – When `monitorenter` and `monitorexit` compare an object reference with the contents of a LOCKADDR register, they consider only bits <29:2>.

8.2 Hardware Synchronization

The Java virtual machine defines two instructions that support synchronization: `monitorenter` (see page 297) and `monitorexit` (see page 299).

See also *Invoking a Synchronized Method* on page 391.

The core does not support general-purpose, atomic, mutual-exclusion primitives, such as test-and-set or compare-and-swap.

8.3 Software Support

Software support for monitor handling comprises four handlers and context switch code, described in the following sections.

8.3.1 LockCountOverflow Handler

The only requirement in *The Java Virtual Machine Specification* is that software throw the `IllegalMonitorStateException` if `monitorexit` causes a decrement of a `LOCKCOUNT` field that is already 0.

The core also generates `LockCountOverflow` when the `LOCKCOUNT.COUNT` field is incremented too far by `monitorenter`. The trap handler software can either maintain a higher-precision version of that field or raise various exceptions. If the trap handler maintains a higher precision value of `LOCKCOUNT`, then it must clear the `LOCKCOUNT.CO` bit.

8.3.2 LockEnterMiss Handler

We recommend that the trap handler for `LockEnterMiss` perform the following steps:

1. **Ensure that an empty `LOCKADDR-LOCKCOUNT` register pair is available.**

   The `LockEnterMiss` trap handler first checks each of the two `LOCKADDR` registers to determine whether they contain 0. If not, then the trap handler must determine which pair of `LOCKADDR-LOCKCOUNT` registers, such as the pair that was least recently replace, will cache the lock to be entered.

   If the `LOCKCOUNT.COUNT` field in the pair to be replaced is 0 *and* any higher precision version of the `LOCKCOUNT.COUNT` field is also 0, then the trap handler clears the `LOCK` bit in the corresponding object header in memory. Otherwise, the trap handler sets the `LOCK` bit in the corresponding object header and saves the current thread identifier and the value of the `COUNT` field in the corresponding monitor data structure.

2. **Examine the `LOCK` bit in the object header of the object to be locked.**

a. If the LOCK bit is 0, then install the object in the empty LOCKADDR register.

The trap handler sets the LOCK bit to 1 in the object header and initializes the monitor data structure for the object. It then writes the object reference to the empty LOCKADDR data structure and initializes the corresponding LOCKCOUNT to 0.

If you need not maintain compatibility with previous versions of the picoJava architecture, the trap handler can simply return and re-execute `monitorenter`.

b. If the LOCK bit is 1, then examine the monitor data structure for the object.

i. If another thread owns the lock, wait for it to be released.

If another thread currently owns the lock, then the trap handler places the current thread in the queue associated with the monitor and blocks the current thread until the other thread exits the monitor.

ii. If this thread owns the lock, install the object in the empty LOCKADDR register.

The trap handler sets the empty LOCKADDR register to the object reference and sets the LOCKCOUNT.COUNT field to the lock count from the monitor data structure. It must also set the LOCKWANT bit if any other threads are waiting for the release of this lock. The CO bit must be 0.

3. Return from the trap handler such that `monitorenter` is re-executed.

8.3.3 `LockRelease` Handler

The core generates `LockRelease` when `monitorexit` is executed, the LOCKCOUNT.COUNT field has been decremented to 0, and the LOCKWANT bit is set. If any higher-precision version of the LOCKCOUNT.COUNT field is not also 0, the trap handler should immediately return to the instruction following `monitorexit`.

Otherwise, a monitor that has another thread waiting to acquire it has been exited and the trap handler should notify the waiting threads that the current thread has released the monitor.

Depending on whether other threads are waiting for the lock and the thread scheduling policy, it may be necessary to clear the LOCK bit in the corresponding object header. Also, the trap handler should set the corresponding LOCKADDR and LOCKCOUNT registers to 0 because the current thread can no longer reacquire the lock by incrementing the LOCKCOUNT field.

8.3.4 LockExitMiss Handler

The core generates LockExitMiss when a monitorexit attempts to unlock an object that is *not* present in either LOCKADDR register. We recommend that the trap handler perform the following steps:

1. **Examine the LOCK bit in the object header of the object to be unlocked.**

   a. **If the LOCK bit is 0, then throw an IllegalMonitorStateException object.**

   b. **If the LOCK bit is 1, then examine the monitor data structure for the object.**

      i. **If another thread currently owns the lock, then throw an IllegalMonitorStateException object.**

      ii. **If this thread owns the lock, then install the lock in a LOCKADDR-LOCKCOUNT register pair.**

      The LockExitMiss trap handler checks each LOCKADDR register to determine whether it contains 0. If at least one of the LOCKADDR registers is 0, then the object will be installed in that pair.

      Otherwise, if both LOCKADDR registers are not 0, then the trap handler must determine which pair of LOCKADDR-LOCKCOUNT registers, such as the pair that was least recently replaced, will cache the lock to be exited. If the LOCKCOUNT.COUNT field in the pair to be replaced is 0 and any higher-precision version of that field is also 0, then the trap handler clears the LOCK bit in the corresponding object header in memory. Otherwise, the trap handler sets the LOCK bit in the corresponding object header and saves the current thread identifier and the value of the COUNT field in the corresponding monitor data structure.

      The trap handler sets the LOCKADDR register in which the lock to be exited will be installed to the object reference to be unlocked and sets the COUNT field in the LOCKCOUNT register to the lock count from the monitor data structure. It must also set the LOCKWANT bit if any other threads are waiting for the release of this lock. The CO bit must be 0.

2. **If no exceptions are thrown, return from the trap handler such that monitorexit is re-executed.**

8.3.5 Context Switch Support

At the time a thread is switched out, the context switch routine must update the status of the monitors it holds in memory for each nonzero LOCKADDR register, as follows:

- If the LOCKCOUNT.COUNT field is 0 and any higher-precision version of that field is also 0, then the context switch routine clears the LOCK bit in the corresponding object header in memory.
- If the COUNT field is *not* 0, then the context switch routine sets the LOCK bit in the corresponding object header and saves the current thread identifier and the value of the LOCKCOUNT register in the corresponding monitor data structure.

For more details, see *Context Switch* on page 58.

CHAPTER 9

Support of the C Programming Language

In the C programming language, the *function* is the fundamental element of execution. The Application Binary Interface (ABI) specifies the conventions for interfunction interfaces in terms of the *caller* and the *callee*.

In this chapter, we first describe the generation of C code for the picoJava-II core, specifically:

- *Register Conventions* on page 402
- *Runtime Stack Architecture* on page 402
- *Calling Conventions for Java-to-C Calls* on page 420
- *Optimizations* on page 421
- *Function Tables* on page 422
- *Handling of Argument Mismatches* on page 425

Finally, we discuss the object file formats defined by the System V ABI specification, which apply for various C language object and executable files. See:

- *Object File Formats* on page 426

Note – Aside from C, the contents of this chapter also apply to other programming languages that cannot be compiled to the Java virtual machine, such as C++ and FORTRAN.

9.1 Register Conventions

The picoJava-II core is a stack-based machine and does not provide any general-purpose registers for use in expression evaluation. Because most operations are stack- based, there are no caller-save or callee-save registers for arguments and locals. However, the architecture does provide some registers with assigned uses, as listed in TABLE 9-1.

TABLE 9-1 Register Uses by C Calling Convention

Name	Function
VARS	Points to the first entry of a frame.
FRAME	Points to the caller's context in a Java frame. This register is *not* used in a C frame.
OPTOP	Points to the top of the operand stack.
OPLIM	Points to the maximum limit of the operand stack. If OPTOP is less than or equal to OPLIM, the core generates a trap.
GLOBAL0	Points to the space allocated for a frame on the aggregate stack. This register is *not* used in a Java frame. This register obeys the caller save convention in C code.
GLOBAL1, GLOBAL2	Returns values from functions. These are volatile caller-save registers and are *not* used in Java functions.

9.2 Runtime Stack Architecture

The runtime stack provides space for local variable storage, temporaries, and arguments. The picoJava-II architecture has a stack cache of 64 entries; elements in this cache can be accessed using `iload` and `istore`. The stack cache and data cache are not coherent; therefore, access to data on the stack by means of `load_word` or `store_word` may produce unexpected results; for an explanation, see *Cache Coherency* on page 30.

To make efficient and correct use of the stack cache, a separate stack (aggregate stack) is maintained for aggregate locals or locals and parameters that can be referenced through a pointer:

- The operand stack is for parameter passing, allocation of scalar locals that can reside in registers, and regular machine operations.

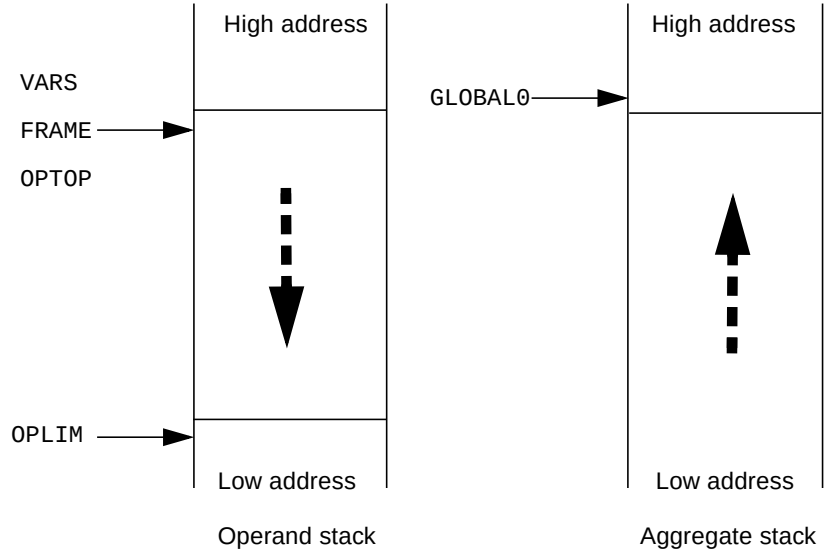
- The aggregate stack is for space allocation for aggregate locals and scalar locals that can be accessed through pointers and the unnamed parameters of a function with a variable number of arguments.

  Sixty-four-bit values in the aggregate stack are stored in big-endian order: The most significant word is stored in the lower-numbered address.

  An implementation can allocate an aggregate local on the operand stack if it can ensure that there is no pointer to the aggregate local and that it is accessed only via local variable load and store instructions.

For each thread that can execute C code, you must allocate space for the aggregate stack. This stack grows from low addresses to high addresses. You should also set aside an area of memory from which C programs can dynamically allocate memory. If a contiguous area is reserved for both of these regions, you can set the USERRANGE register to point to the low and high addresses of this space so that a trap occurs if the C code refers to memory outside this region.

FIGURE 9-1 illustrates the runtime stack allocation when a new thread is created for C code.



Note - The two stacks grow in opposite directions.

FIGURE 9-1 Runtime Stack Allocation for a New Thread

9.2.1 Calling Convention for C-to-C Calls

C function calls and returns use `call` and `return0`. The frame layout and prologue code for a function call depend on the type of parameters and locals in the function.

The following sections outline the general rules for the frame layout and provide some examples.

9.2.2 Rules for Passing Arguments

In passing arguments, the following conventions apply:

- Scalar parameters are pushed on the operand stack in the same order as declared.
- Aggregate parameters are passed by reference. The caller allocates temporary space for the aggregate parameter and passes the address of this space as the reference.
- Before executing a `call` instruction, the caller pushes the following items onto the stack, in this sequence:

  a. The parameters being passed to the callee, in the order in which they were declared in the caller

  b. The target address of the call

  c. The number of words pushed for the call (including the target address and this word)

CODE EXAMPLE 9-1 lists a typical function call.

CODE EXAMPLE 9-1 Sample Code for a Function Call

```
func(1,2,3);

iconst_1
 iconst_2
  iconst_3
   sipush lo(_func)
   sethi hi(_func)
    bipush 5
call
```

9.2.3 Function Return Values

Functions that do not return a value (type `void`) use the `return0` instruction.

Functions that return a size of one to four bytes use the `GLOBAL1` register for the return value and also use the `return0` instruction.

Functions that return a size of five to eight bytes use the `GLOBAL1` and `GLOBAL2` registers for the return value and use a `return0` instruction to transfer control back to the caller. A return value that requires more than four bytes must be represented by the concatenation of `GLOBAL1` and `GLOBAL2`. The most significant word is returned in `GLOBAL1`.

A function that returns an aggregate value copies the return value into the space provided by the caller. The caller passes a reference to the space where the return value will be stored as the first implicit (hidden) parameter. The called function uses this address to copy the return value and then executes `return0`.

9.2.4 Function Prologue and Epilogue

A function prologue sets up the execution environment for a function. A function epilogue unwinds the execution environment and reestablishes the old environment so that execution can continue after a return from a call.

- The function prologue code allocates space for scalar locals on the operand stack by decreasing the address in the `OPTOP` register. If necessary, it also allocates space for any aggregate locals by increasing the address in the `GLOBAL0` register, which points to the aggregate stack. If there exists any simple parameters whose addresses are taken in the code, space is allocated on the aggregate stack for the locals and their initial values are copied into their respective locations on the aggregate stack.

  In addition, for functions with variable number of arguments, the prologue code moves all the unnamed arguments onto the aggregate stack and moves the return `VARS` and return `PC` entries up so that they are in the locations assigned for them during compilation.

- The function epilogue code does the following prior to executing the appropriate `return` instruction, which restores the environment to that of the caller:
  - Unwinds the `OPTOP` register and frees up the local space allocated on the operand stack
  - Unwinds the `GLOBAL0` register and frees up the space allocated on the aggregate stack for the function

See *Optimizations* on page 421 for more details on optimization.

9.2.5 Functions with Simple Parameters and Locals

Consider CODE EXAMPLE 9-2, which shows the frame structure and call sequence for a function with simple parameters and locals.

CODE EXAMPLE 9-2 Function with Simple Parameters and Locals

```
int zoo1(int param1, int param2, int param3)
{
    int local1;
    return 0;
}
zoo()
{
    int zlocal1, zlocal2, zlocal3;
    int i;
    .......
    i = zoo1(zlocal1, zlocal2, zocal3);
   .......
}
```

FIGURE 9-2 is the operand stack frame layout for functions `zoo` and `zoo1`.

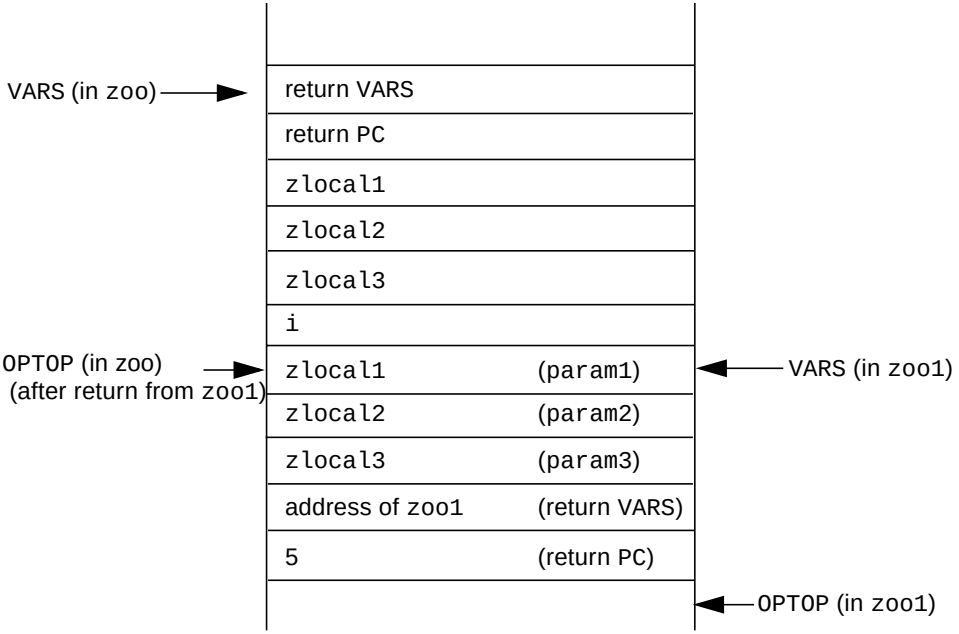


FIGURE 9-2 Operand Stack Frame Layout for `zoo` and `zoo1`

The zoo function first pushes the parameters zlocal1, zlocal2, and zlocal3 on the stack, then the address of the function zoo1 to call and the number of entries pushed (number of parameters + 2). zoo then issues the call instruction.

On entry into the zoo1 function, the prologue code allocates space for the local variable of zoo1. On return, the function zoo1 stores the return value in the GLOBAL1 register, unwinds the allocation for local variables from the stack, and issues the return0 instruction. FIGURE 9-3 shows the stack at this point.

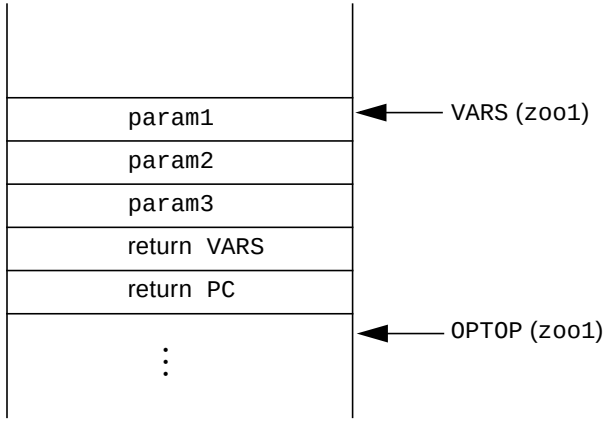


FIGURE 9-3 Operand Stack Frame Layout Before zoo1 Returns to zoo

Because this function does not include any aggregate variables or variables for which an address is taken, the function does not allocate any space on its aggregate stack.

CODE EXAMPLE 9-3 contains the compiled code for a function with simple parameters and locals.

CODE EXAMPLE 9-3 Compiled Code for a Function with Simple Parameters and Locals

```
_zoo1:
       // PROLOGUE
        iconst_0      // Allocate space for locals
       // END PROLOGUE
         iconst_0
        write_global1
L_1:
       // EPILOGUE
       pop            // Deallocate space for locals
       return0
       // END EPILOGUE
_zoo:
```

CODE EXAMPLE 9-3 Compiled Code for a Function with Simple Parameters and Locals

```
        // PROLOGUE
          lconst_0     // Allocate space for locals
            lconst_0
        // END PROLOGUE
             iload_2
              iload_3
               iload 4
                sipush lo(_zoo1)
                sethi hi(_zoo1)
                 bipush 5
            call
             read_global1
            istore 5
L_2:
        // EPILOGUE
          pop2
        pop2
        return0
        // END EPILOGUE
```

9.2.6 Functions with Complex Parameters and Locals

Consider CODE EXAMPLE 9-4, which contains the frame structure and call sequence for functions with complex parameters and locals.

CODE EXAMPLE 9-4 Functions with Aggregate Parameters and Locals

```
struct s {
    int i,j,l;
    char c;
};
int zoo1(struct s s1, int i, int j)
{
    int li,*lip = &li;
    int *ip = &i;
    struct s s2;

    return 0;
}
void zoo()
{
    struct s s2,i;
    zoo1(s2, 1, 2);
}
```

Functions in this code have aggregate variables and variables for which an address is taken. The prologue code for functions `zoo` and `zoo1` allocates space on the aggregate stack for these functions.

FIGURE 9-4 shows the stack frame layout for functions `zoo1` and `zoo`.

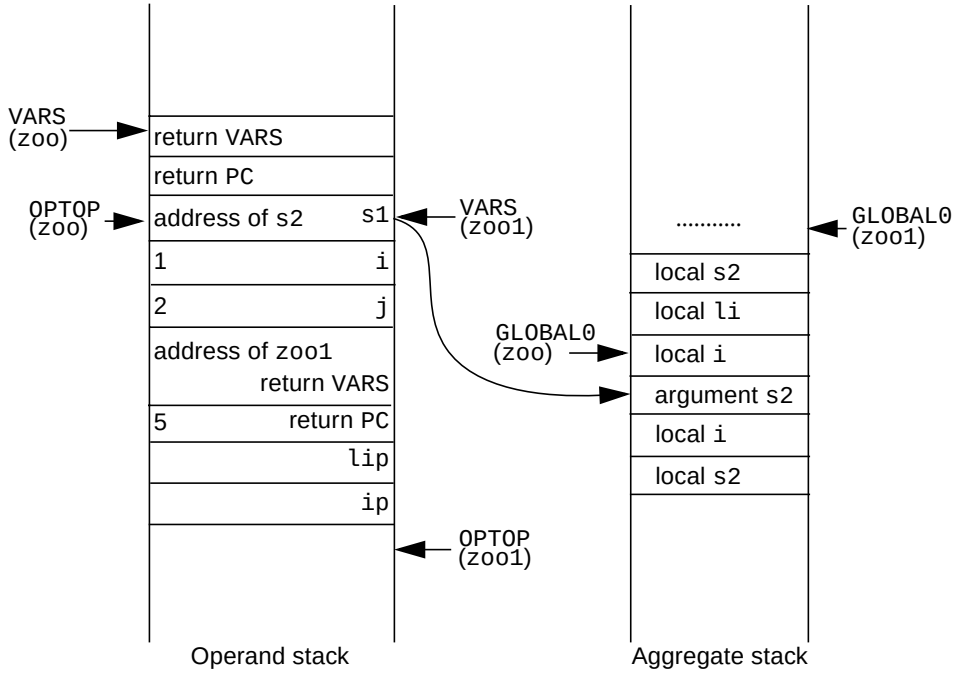


FIGURE 9-4 Stack Frame Layout for `zoo` Calling `zoo1`

Because aggregate parameters are pushed by reference, `zoo` does the following prior to issuing the `call` instruction:

- Allocates temporary space for parameter `s2` on its aggregate stack and passes a reference to this space as a parameter on the operand stack
- Pushes parameters 1 and 2 onto the operand stack, then the `zoo1` address and the number of entries pushed on the operand stack (number of simple parameters + 2)

On entry into `zoo1`, the function `prologue` does the following:

- Increments `GLOBAL0` to allocate space for the local aggregate objects and variables for which the C code obtains the address
- Allocates space for the simple locals of `zoo1` on the operand stack by decrementing `OPTOP`

On return, `zoo1` does the following prior to issuing the `return0` instruction:

- Stores the return value in the `GLOBAL1` register
- Unwinds the local space allocated on the operand stack.
- Unwinds the local and param space allocated on the aggregate stack by subtracting the local space allocated from the `GLOBAL0` register.

FIGURE 9-5 shows the stacks at this point.

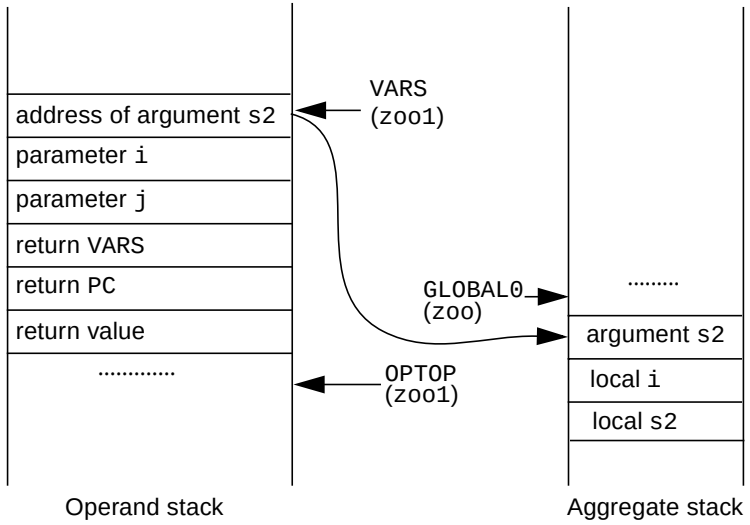


FIGURE 9-5 Stack Frame Layout for `zoo1` Before Returning

CODE EXAMPLE 9-5 lists the compiled code for CODE EXAMPLE 9-4.

CODE EXAMPLE 9-5 Compiled Code for Function with Aggregate Parameters

```
_zoo1:
       // PROLOGUE
        read_global0      // Allocate space on aggr stack
         bipush 24
        iadd
       write_global0
         lconst_0         // Allocate space for lip, ip
          iload_1         // Copy i to aggr stack
           read_global0
            bipush 36
           isub
         store_word
       // END PROLOGUE
          read_global0
```

CODE EXAMPLE 9-5 Compiled Code for Function with Aggregate Parameters *(Continued)*

```
            bipush 32
           isub
          istore 5          // Store address of li in local lip
           read_global0
            bipush 36
           iadd
          istore 6          // Store address of i in local ip
           iconst_0
          write_global1
L_1:
        // EPILOGUE
        pop2                // Free space of locals
         read_global0       // Free aggr stack space
          bipush 36
         isub
        write_global0
        return0
        // END EPILOGUE
_zoo:
        // PROLOGUE
         read_global0
          bipush 32
         iadd
        write_global0
        // END PROLOGUE
        ......
        ......
         read_global0       // Pass reference to arg s2
          dup
           bipush 16
          iadd              // Allocate space for arg s2
         write_global0
          iconst_1
           iconst_2
            sipush lo(_zoo1)
            sethi hi(_zoo1)
             bipush 5
        call
L_2:
        // EPILOGUE
         read_global0
          bipush 48
         isub
        write_global0
        return0
        // END EPILOGUE
```

9.2.7 Functions That Return Aggregate Values

Consider CODE EXAMPLE 9-6, which lists the frame structure and call sequence of functions that return aggregate values.

CODE EXAMPLE 9-6 Function Returning Aggregate Values

```
struct s {
    int i,j,l;
    char c;
};
struct s zoo1(int j)
{
    struct s ls;
    return ls;
}
void zoo()
{
    int i;
    zoo1(2);
}
```

The function `zoo` does the following:

- Allocates space on its frame for the return value of function `zoo1` on the aggregate stack
- Passes the address of this space as the first implicit parameter to `zoo1`

On return, `zoo1` does the following:

- Copies the return aggregate value to this area, using the address in the first hidden parameter
- Issues `return0`

FIGURE 9-6 illustrates the stack frame layout at this point.

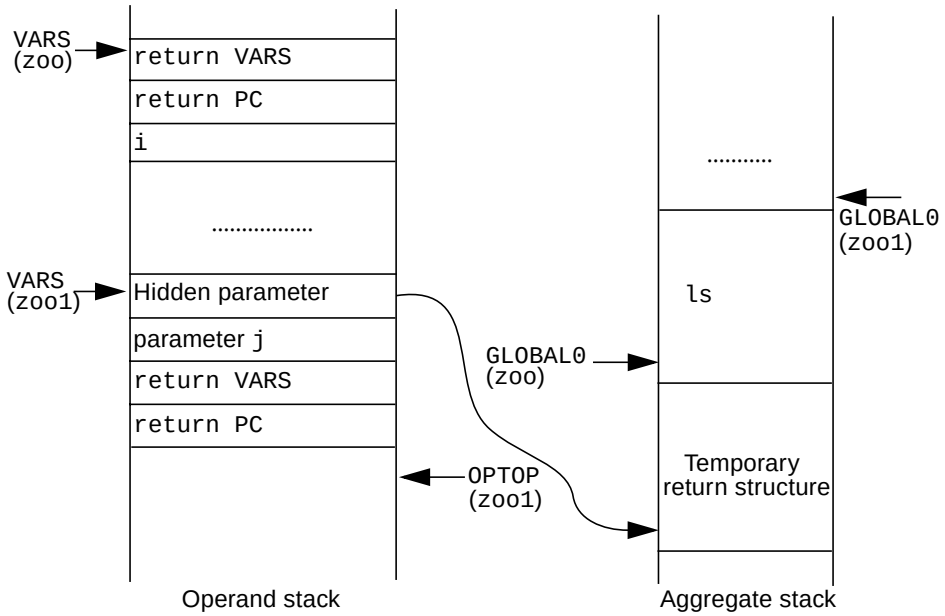


FIGURE 9-6 Stack Frame for Function `zoo1` Returning Aggregate Values

CODE EXAMPLE 9-7 contains the compiled code for a function that returns an aggregate value.

CODE EXAMPLE 9-7 Compiled Code for a Function That Returns An Aggregate Value

```
_zoo1:
       // PROLOGUE
        read_global0
         bipush 16
        iadd
       write_global0
       // END PROLOGUE
        read_global0          // Copy return value
         bipush 16
        isub
        load_word
         iload_0
       store_word
        read_global0
         bipush 12
        isub
        load_word
         iload_0
          iconst_4
```

CODE EXAMPLE 9-7 Compiled Code for a Function That Returns An Aggregate Value

```
          iadd
        store_word
         read_global0
          bipush 8
         isub
         load_word
          iload_0
           bipush 8
          iadd
        store_word
         read_global0
          iconst_4
         isub
         load_word
          iload_0
           bipush 12
          iadd
        store_word
L_1:
        // EPILOGUE
         read_global0
          bipush 16
         isub
        write_global0
        return0
        // END EPILOGUE
_zoo
        // PROLOGUE
         iconst_0          // Allocate space for local variable i
        // END PROLOGUE
          read_global0
           bipush 16
          iadd             // Allocate space for return structure
         write_global0
          read_global0     // Pass addr of return area
           bipush 16
          isub
           iconst_2
            sipush lo(_zoo1)
            sethi hi(_zoo1)
             iconst_4
         call
L_2:
        // EPILOGUE
          read_global0
           bipush 16
          isub
```

CODE EXAMPLE 9-7 Compiled Code for a Function That Returns An Aggregate Value

```
         write_global0
        pop                     ; Deallocate space for i
        return0
        // END EPILOGUE
```

CODE EXAMPLE 9-8 shows code from CODE EXAMPLE 9-7 that was optimized for the core. Highlighted are the uses of `load_word_index` and `store_word_index`.

CODE EXAMPLE 9-8 Optimized Code for Function Returning Aggregate Values

```
_zoo1:  read_global0          // PROLOGUE
         bipush 16
         iadd
        write_global0         // END PROLOGUE
         read_global0         // Puts global0 in local var 4
          load_word_index 4 -16// Copy return value
         store_word_index 0 0
          load_word_index 4 -12
         store_word_index 0 4
          load_word_index 4 -8
         store_word_index 0 8
          load_word_index 4 -4
         store_word_index 0 12
        pop
L_1:     read_global0         // EPILOGUE
         bipush 16
         isub
        write_global0
        return0               // END EPILOGUE
        . . .
```

9.2.8 Functions with Variable Number of Arguments

Consider CODE EXAMPLE 9-9, which exemplifies a function with a variable number of arguments.

CODE EXAMPLE 9-9 Function with Variable Number of Arguments

```
#include “stdarg.h”
zoo1(int i, int j, ...)
{
  int          li = 0;
  int          lj = 1;
  va_list       arglist;
```

CODE EXAMPLE 9-9 Function with Variable Number of Arguments *(Continued)*

```
  va_start (arglist,j);
  li = va_arg(arglist, int);
}

zoo()
{
  foo(1,2,3,4,5);
}
```

An example `stdarg.h` is listed in CODE EXAMPLE 9-10.

CODE EXAMPLE 9-10 Sample Code In `stdarg.h`

```
enum __va_type_classes {
  __no_type_class = -1,
  __void_type_class,
  __integer_type_class,
  __char_type_class,
  __enumeral_type_class,
  __boolean_type_class,
  __pointer_type_class,
  __reference_type_class,
  __offset_type_class,
  __real_type_class,
  __complex_type_class,
  __function_type_class,
  __method_type_class,
  __record_type_class,
  __union_type_class,
  __array_type_class,
  __string_type_class,
  __set_type_class,
  __file_type_class,
  __lang_type_class
};

typedef int *va_list;
#define va_start(pvar, ARG) (pvar = _builtin_saveregs())
#define va_end(pvar)
#define va_arg(pvar, TYPE)                                    \
    ((__builtin_classify_type (*((TYPE
       *)0))<__record_type_class)?                          \
    (((char *)pvar)-=((sizeof(TYPE)+3) & ~3),                 \
       ((sizeof(TYPE) < 4) ?                                  \
```

CODE EXAMPLE 9-10 Sample Code In `stdarg.h` *(Continued)*

```
        *((TYPE *)pvar + ((4-sizeof(TYPE))/sizeof(TYPE))):     \
        *(TYPE *)pvar)):                                       \
    (((char *)pvar) -= 4,  *(TYPE *)(*((void **)pvar))))
```

`zoo1` has two named parameters, `i` and `j`. All other parameters are unnamed; access to them is through the `va_arg` macros. The `PROLOGUE` code of `zoo1` moves these unnamed parameters to the aggregate stack and adjusts up return `VARS` and return `PC` so that the frame corresponds to the compile time image of two named parameters. The `va_arg` macro then accesses the unnamed parameters from the aggregate stack. The `PROLOGUE` code also saves the return `GLOBAL0` value so that the `EPILOGUE` code can restore it to the caller's value.

FIGURE 9-7 and FIGURE 9-8 illustrate the stack frames for functions that use variable arguments.

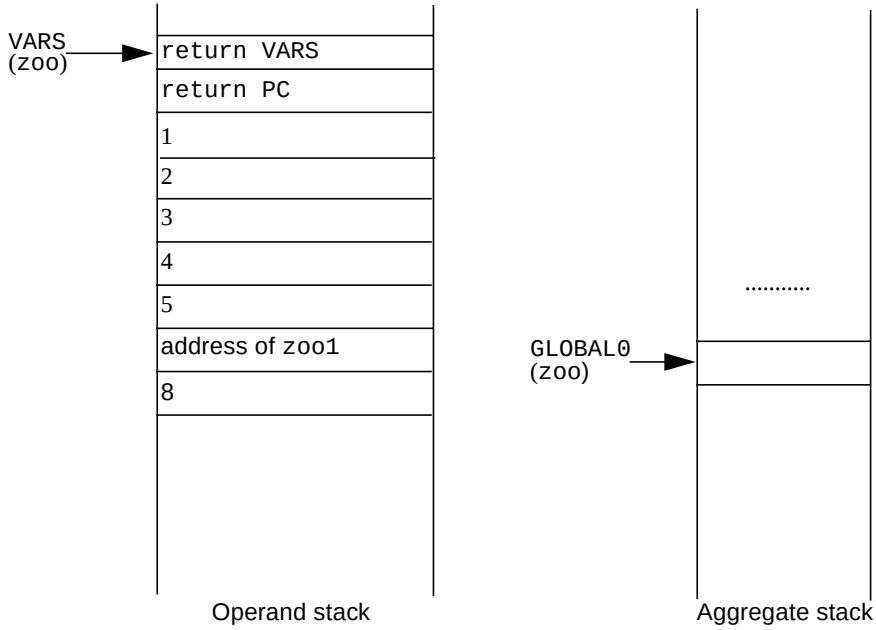


FIGURE 9-7 Stack Frame for Function `zoo` Calling `zoo1` with Variable Number of Arguments

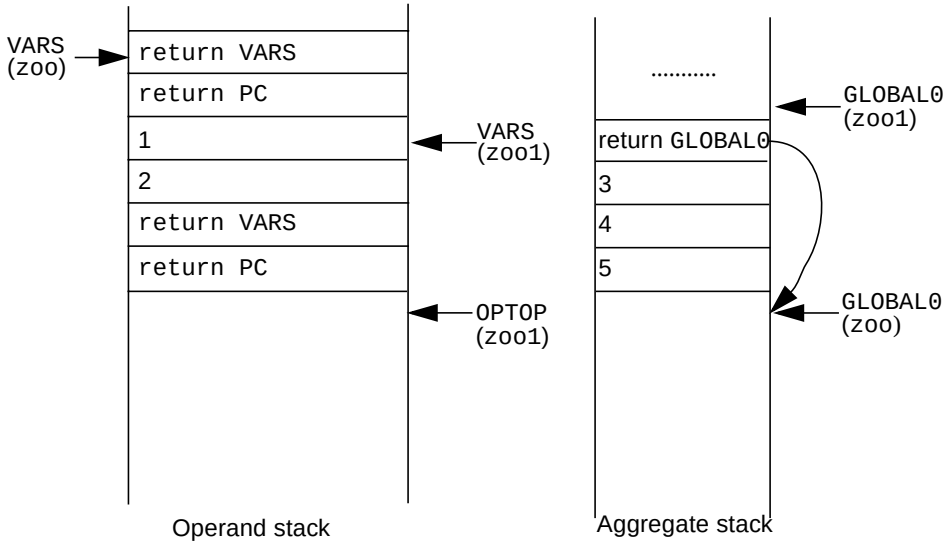


FIGURE 9-8 Stack Frame of `zoo1` After Execution of `PROLOGUE` Code

CODE EXAMPLE 9-11 contains the compiled code for CODE EXAMPLE 9-9.

CODE EXAMPLE 9-11 Compiled Code for Function with Variable Number of Arguments

```
_zoo1:
        // PROLOGUE
        // Code to move unnamed arguments onto aggregate stack
        // and ensure exactly the named parameters are left
     write_global1     // GLOBAL1 = return PC
    write_global2      // GLOBAL2 = return VARS
     read_vars
      read_optop
     isub
      dup
       bipush 12
     if icmpgt L_1
    pop
     read_global0
      read_global0     // Could use dup, but this helps
    store_word         // Folding as dup is BG1
     read_global0
      iconst_4
     iadd
    write_global0;
     read_vars         // Ensure named parameters are on stack
      bipush 16
     iadd
```

CODE EXAMPLE 9-11 Compiled Code for Function with Variable Number of Arguments

```
    write_optop        // OPTOP should be 4 entries lower
     read_global2
    istore_2           // return VARS should be local 2
     read_global1
    istore_3           // return PC should be local 3
    goto _zoo1_Known
L_1:  iconst_4
     isub              // TOS = #passed - #named + 4
      dup
       read_global0
      iadd             // TOS = GLOBAL0+passed-named+4
       read_global1    // Make room for loop variable in GLOBAL1
       swap
     store_word        // TOS = #passed - #named + 4
      iconst_4
     isub
      read_global0
     iadd
    write_global1      // End address for copy onto aggregate stack
     read_global0
      read_global1
    store_word         // Save aggregate stack pointer
L_2:  read_global0
    store_word         // store a stack element to agg. stack
     read_global0
      iconst_4
     iadd
    write_global0
     read_global1
      read_global0
    if_icmplt L_2
     read_global0
      iconst_4
     iadd
    write_global0      // Update GLOBAL0
     read_global2      // Restore return VARS
      read_global0
      load_word        // Restore return PC

    // END PROLOGUE

_zoo1_Known:
    // Note - Once GLOBAL1 is written into the aggregate stack, the
    // stack is shorter than at _zoo1 by two entries and does
    // not become greater than it was at that point
    // so no OPLIM trap occurs after the writing of global1
    // and before the label _zoo1_Known.
```

CODE EXAMPLE 9-11 Compiled Code for Function with Variable Number of Arguments

```
     ...
     ...
        // EPILOGUE
      read_global0
       iconst_4
      isub
      load_word
     write_global0
      iload_2       // Instead of using pop2's a multicycle
       iload_3      // instruction, use iloads that are folded
     return0
         // END EPILOGUE
_zoo:
         // PROLOGUE
         // END PROLOGUE
      iconst_1
       iconst_2
        iconst_3
         iconst_4
          iconst_5
           sipush lo(_zoo1)
           sethi hi(_zoo1)
            bipush 6
     call
     return0
         // EPILOGUE
         // END EPILOGUE
```

9.3 Calling Conventions for Java-to-C Calls

There are three ways for Java code to call a C function:

- The trap handlers for `invoke_virtual`, `invoke_interface`, and `invoke_static` set up the parameters for calling the native C method and issue the `call` instruction. On completion, the native C method returns to the trap handler code, which arranges for the Java thread to continue execution.
- The Java compiler produces code to invoke a Java method. A tool similar to `javah` provides "trampoline" code, which understands a native function frame structure and produces a call to the function `ClassName_FunctionName`. You then provide the code for this function, create the stub, and include it in the code. A `javah`-like tool is dependent on the Java virtual machine structure and is provided with each Java virtual machine. All C call optimizations for C calls are applicable to the call to `ClassName_FunctionName`.

A C function calls a Java method by calling special functions provided with the virtual machine that, based on the parameters passed to them, set up the appropriate Java frame, then invoke the Java method.

Alternatively, if the C compiler understands the structure of Java frames and the constant pool, it can generate invokes (such as `invokevirtual`) directly.

9.4 Optimizations

A function that takes no arguments, has no local variables, and that does not return an aggregate type can be executed in the caller's environment. A `jsr` instruction can call the *leaf* function, thereby bypassing building a frame. Depending on the return type, the leaf uses `ret_from_sub` to return to the caller and leaves no temporaries on the stack. Such functions are usually for data abstraction.

There can be any number of frameless calls between two framed calls.

The return instructions take two entries off the stack (return PC and return VARS) and thus are defined as a BG2 type of instruction with respect to folding. Therefore, if there are temporaries between `retPC` and `OPTOP` at the time of returning, `iload`s (which are folded with the `return0` and thus are faster than `pop`s in this case) can replace the `pop`s in the epilogue code.

Optimize prologue code by initializing local variables in the prologue instead of just pushing zeroes on the stack to create space for them. If initialization occurs in the prologue, it need not occur in the body, thus saving one push and one store per local variable.

An implementation can optimize by passing aggregate structures on the operand stack if it can ensure the following:

- The aggregate structure is only accessed directly. This situation occurs only if there are no indirect references to the structure and a pointer to the structure is not passed to another function.
- All objects are consistently compiled with the same options.

9.5 Function Tables

The operating system keeps a set of tables (two-dimensional arrays).

The Code Segment Table, the address of which is in the operating system global variable, contains one entry per code segment. Each entry in this table points to a Function Table, which contains one entry for each C function in the code segment. These tables support stack chunking and are used by the `oplim_trap` routine.

These tables are used by debuggers, the `oplim_trap` handler, and other functions to unwind a C call stack.

9.5.1 Structure

FIGURE 9-9 illustrates the structure of the tables.

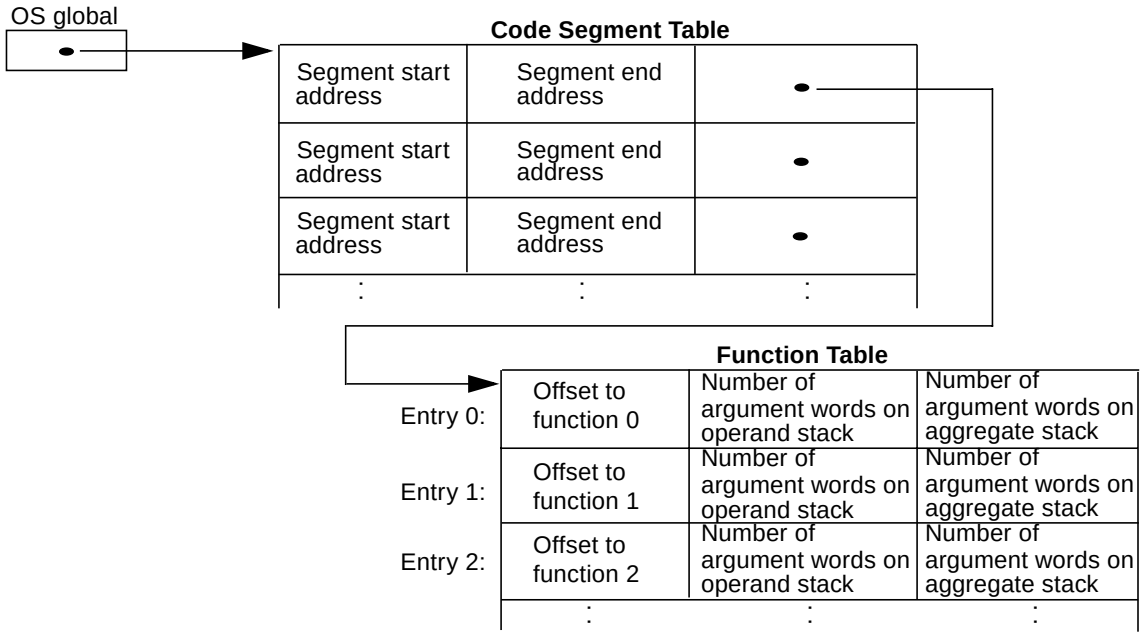


FIGURE 9-9 Table Structures

Each entry in the Code Segment Table contains two integers and a pointer to a Text Table. The first integer or entry contains the start address of the text segment. The second entry contains the end address of the text segment.

Each entry in the Function Table contains three integers:

- The first entry stores the offset (from the text segment start address) of the start PC for the function.
- The second entry contains the number of nonaggregate argument words that the function takes—the number of argument words that are pushed on the operand stack.
- The last entry contains the number of aggregate argument words that the function takes.

If the function takes a variable number of arguments, then the last entry contains a value of –1. A value of –2 indicates a special case in the argument number mismatch handler.

9.5.2 Properties

The Code Segment Table entries are sorted in ascending order by segment start address. The Function Table entries are sorted in ascending order by offset from segment start. In creating an executable, the C compiler and linker must ensure that Code Segment and Function Tables, as described above, exist in the file and that their entries are sorted properly.

This information is passed to the operating system in the executable and linking format (ELF) file as follows.

- A symbol `_text_table` residing in the static data segment points to the Function Table structure.
- The symbols `_text_start` and `_text_end` point to the start and end of the code segment area.

9.5.3 Provisions in the Operating System

The operating system must maintain the Code Segment and Function Tables in memory. These tables reside in a memory space accessible to the kernel and can therefore be shared by all threads.

Three functions are available:

```
void OSAddTextTable(unsigned int TextStart,
                    unsigned int TextEnd,
                    struct TextTable *tt);

void OSDeleteTextTable(unsigned int TextStart);

void OSGetNargs(unsigned int PC, int *operand_nargs,
                int *aggregate_nargs);
```

Here is the structure of `TextTable`:

```
struct TextTable {
     int num_entries;
     int table_entry[][3];
};
```

Where:

- `table_entry` is a two-dimensional array such that `table_entry[`*i*`][0]` is the offset of the start of function *i* within the text segment.
- `table_entry[`*i*`][1]` and `table_entry[`*i*`][2]` are the number of operand stack arguments and aggregate stack arguments, respectively.

9.5.4 `_init` and `_fini`

In the `_init` startup function, the Text Table is registered by a call to the `OSAddTextTable` function with the symbols `_text_start`, `_text_end`, and `_text_table` as parameters.

Similarly, in the `_fini` wrapup functions, if the text segment is to be unloaded on completion, the table is unregistered by a call to the `OSDeleteTextTable` function with the `_text_start` symbol as a parameter.

9.5.5 `OSGetNArgs` Algorithm

The `OSGetNArgs` routine performs a binary search on the segment table (the address of which is in a global variable) to get a pointer to a Text Table. The search is based on the `PC` passed in as a parameter; the desired text segment is the one into which the `PC` points (that is, where `text_start` ≤ `PC` ≤ `text_end`). The `PC`'s offset from the beginning of the text segment (`PC` – `text_start`) is then the basis for a binary

search of the appropriate Text Table to find the entry corresponding to the function that contains the PC. The number of expected arguments for that function is found in that Text Table entry.

The return values are set to the values from the Text Table. The operand_nargs value may indicate further action, as follows:

- A return of -1 indicates that the function takes a variable number of arguments. Local variable 0 of the function contains the present number of arguments.
- A return of -2 indicates that the function is handling a possible argument mismatch. See *Handling of Argument Mismatches*, below.
- A return of -3 indicates an unsuccessful search; therefore, Java code must have been executing at the time of the OPLIM trap.
- A return value that is greater than or equal to 0 indicates the number of arguments that the function expects after a successful search.

9.5.6 Extensions to Support .so (.dll) Files

You can use the function tables to support .so (.dll) files, which can be C code with a Java wrapper or code with C (native) methods. When you create a .class file that requires a companion .so (.dll) file, the latter must contain a table similar to the one described in the previous section.

9.6 Handling of Argument Mismatches

The compiler creates two entry points per function. CODE EXAMPLE 9-12 is an example of a function, foo, that takes five arguments.

CODE EXAMPLE 9-12 Rearranging the Function Frame for an Argument Mismatch

```
_foo:       write_global1 // global1 = return PC
           write_global2 // global2 = return VARS
            read_vars // OPTOP and VARS should be
             bipush 28 // 28 bytes apart for a
            isub      // five-argument function frame
           write_optop
            read_global2
           istore 5   // return VARS should be local var 5
            read_global1
           istore 6   // return PC should be local var 6
_foo_Known:           // Normal prologue of the function
```

All functions within the same module (.o file) call foo by pushing the address _foo_Known only if the number of arguments the callee expects matches the number of arguments the caller provides.

In a different module where the function foo is called, the linker must resolve external functions to produce an executable. The linker must already generate some of the code associated with a call, such as pushing the target PC. Now it must also infer the number of arguments that are being passed. If this number matches the number of arguments foo expects, then the entry point used for the call to foo is at _foo_Known; otherwise, it is _foo.

The linker can infer the number of arguments passed by disassembling an iconst, bipush, or sipush instruction immediately before the call instruction or immediately after the instructions that push the call address. If the linker does not detect the instruction pattern:

<push *target*> (generally sipush, sethi)
<push *nargs*> (a single instruction)
call

then the linker generates a call to the function at the first entry point—the point where all calls are checked for argument count agreement (*_name*).

If the compiler supports this flag or mode, the text table has two entries per C function, one for *_name* and the other for *_name*_Known. A special return value (-2) from the table notifies the trap routine that the argument check and adjustment is in progress. The return PC and return VARS values to unwind the stack can then be reconstructed as needed.

Also, indirect calls (those made through a pointer) to a function foo are made to _foo, that is, to the point where all calls are "safe" (checked for an argument mismatch).

9.7 Object File Formats

In general, the ELF object file format defined by the System V ABI specification applies for various C language object and executable files. In that case, the 16-bit e_machine field of the ELF file header contains the value EM_PICOJAVA.

```
EM_PICOJAVA = 0x63
```

The 32-bit e_flags field contains values that are associated with the following fields:

```
EF_PICOJAVA_ARCH = 0x0000000F
EF_PICOJAVA_NEWCALLS= 0x00000010
```

The 4-bit field that corresponds to EF_PICOJAVA_ARCH has a value of 0 if the object file uses only the instructions implemented in hardware by the picoJava-I core. This value is 1 if any of the additional instructions implemented by picoJava-II are used. In the case of a combination of multiple object files, the resulting value in the EF_PICOJAVA_ARCH field is the highest value in these fields.

Emulation trap routines on a picoJava-I system can allow code from object files with nonzero EF_PICOJAVA_ARCH fields to execute.

The 1-bit flag that corresponds to EF_PICOJAVA_NEWCALLS has the value of 1 if the calling convention in this chapter applies. If this flag has a value of 0, then unsupported, older calling conventions apply. A loader can reject object files that use the old calling convention.

CHAPTER 10

Stack Chunking

This chapter explains how stack chunking can be implemented for the picoJava-II core. It contains the following sections:

- *Overview* on page 429
- *`oplim_trap` Handler* on page 430
- *Manual Updates of the VARS Register* on page 431
- *Returns to Previously Saved Program States* on page 432
- *Possible Write-After-Write Hazards* on page 432

10.1 Overview

Normally, at thread creation time, the core allocates a stack chunk to provide space for the new thread's stack. At times, however, a thread may use up all of the allocated space. To support the thread's continued execution, the core provides a mechanism for it to allocate additional space, as follows.

Stacks are allocated by software in chunks of some reasonable size. If there is not enough room on a stack (`OPTOP` < `OPLIM`) to execute an instruction, an `oplim_trap` occurs, the `oplim_trap` handler can allocate additional space, and execution can continue. Allocation and deallocation occur in software.

There are three caveats:

- The stack cache assumes that the underlying stack, which is used by the spill and fill mechanisms, is contiguous. If the new stack space allocated by the trap handler is not contiguous, you must take certain actions, as described in *oplim_trap Handler* on page 430.
- There is no analogous underflow trap when `OPTOP` becomes less than the top of the chunk.

- The stack cache may cause reads and writes to any of the 64 words of memory at addresses greater than OPTOP. Therefore, the stack cache should be positioned at least 64 words below a memory area that is used for anything except read-only data or another stack chunk. See *Stack Cache* on page 39 for more details.

The next sections describe the oplim_trap handler and underflow conditions and highlight some aspects of the trap architectures.

10.2 oplim_trap Handler

FIGURE 10-1 shows possible states of the stack before the oplim_trap handler is entered.

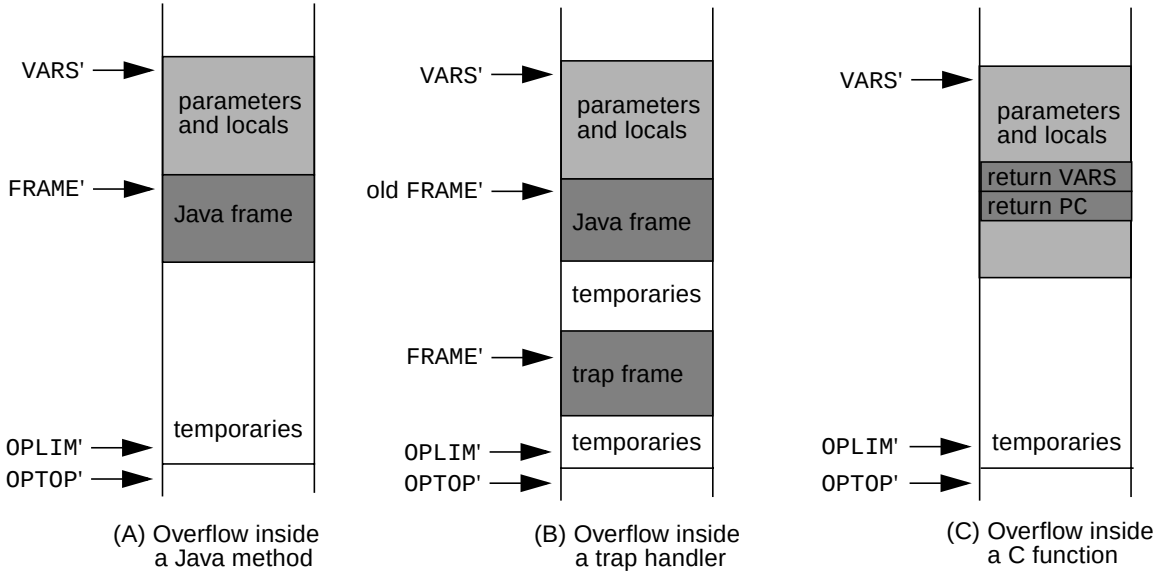


FIGURE 10-1 Possible Stack States Before Entering oplim_trap Handler

In each of these scenarios, the trap handler can enable execution to continue by providing additional stack area. To grow the stack, you can use the following strategies:

- If contiguous free memory is available, adjust OPLIM to allow the stack to continue to grow from its current location. If necessary, use the garbage collector to move objects out of the way.

- Allocate a larger stack region for the thread and copy the entire current stack to the new larger region. Be sure to update all references to VARS, FRAME, and OPTOP in the stack you are copying to refer to the new locations in memory.
- Allocate a noncontiguous region of memory and copy the currently reachable portion of the stack to the new chunk. Be sure to update all references to VARS, FRAME, and OPTOP in the stack you are copying to refer to the new locations in memory. You must also insert an underflow handler into the call stack to return execution to the original stack chunk.

All of these techniques require that you can relocate a stack, which entails redirecting all VARS, FRAME, and OPTOP values on the stack to refer to the new location. If you use the first technique, you may need to move stacks to free up contiguous memory into which another stack can grow.

Within Java code, all VARS, FRAME, and OPTOP values reside within method frames only. To update stacks that contain only Java method calls, trace the call stack and update the values within the frames.

Tracing C and trap frames is more complex but generally straightforward. Be aware that VARS, FRAME, and OPTOP values can lie outside of the call frames and be directly pushed on the stack by trap or C code. You must ensure that you can identify all the values on the stack before you can move them.

10.3 Manual Updates of the VARS Register

Although VARS can be written with any value, such practice can result in erroneous execution of the thread because the technique used by the `oplim_trap` handler may only ensure that the area from VARS to OPTOP is contiguous. Moving VARS upward may effectively result in an underflow of the allocated stack area.

One way around the problem is to duplicate all the information needed from the locals and parameters area to the temps area by integer loads and to move VARS in one direction only (for example, downward).

10.4 Returns to Previously Saved Program States

If you move the stack of a thread that is not currently running, be sure to update the VARS, FRAME, and OPTOP values in any saved context that is to be restored when the thread resumes execution.

Similarly, when implementing the C language functions `setjmp` and `longjmp`, you must consider the problem of returning to the previously saved program state. For example, if you save the state of a program using `setjmp`, then you have moved the stack. A subsequent `longjmp` restores VARS, FRAME, and OPTOP values that are no longer valid. Therefore, you must update the `setjmp` state when you move the stack.

10.5 Possible Write-After-Write Hazards

The `oplim_trap` handler may change the value of return PC (and return VARS) in a call frame to return to an underflow handler. This technique introduces a restriction on the implementation of trap handlers.

For example, a trap being used to implement an instruction must change the return PC to continue execution at the next instruction. It does so by modifying the return PC of the trap frame, which may actually be the return address to enter into the underflow trap handler if an `oplim_trap` occurred during the original trap (that is, the emulation trap handler). Hence, you must revise the code that updates the return PC of the emulation handler such that the trap ensures that its return PC is not the same as the entry point of the underflow handler. The address of the overflow handler can be provided by the kernel.

CHAPTER 11

Support for Garbage Collection

Rather than having the programmer indicate when an object is no longer used in a program (and thus reclaim the memory the object occupies), the Java language specifies that once an object is no longer referenced in any thread of execution, a garbage collection (GC) scheme collects that object's memory and returns it to the pool of available memory.

A Java virtual machine implementation must provide automatic garbage collection. An abundance of literature is available on various GC techniques, most of which are surveyed in [Wilson 1992].

This chapter defines support by the picoJava-II core for various GC schemes in three sections and provides several references, as follows:

- *Hardware Support* on page 433
- *Write Barriers* on page 434
- *Examples* on page 439
- *References* on page 441

Note – Throughout this chapter, abbreviated reference titles are enclosed in square brackets, for example, [Wilson 1992].

11.1 Hardware Support

The core provides the following low-level mechanisms to assist you in implementing a garbage collector:

- Support for handles
- Reserved bits in object references and headers

11.1.1 Support for Handles

As described in *Reference Types and Values* on page 62, object and array references can be direct or indirect, through a handle. Handles provide a simple and convenient mechanism for a garbage collector to relocate objects in memory. Typically, the garbage collector relocates the objects that are still referenced to a contiguous range of memory, thus leaving the remaining free memory in a contiguous chunk. Using the level of indirection provided by handles significantly simplifies moving an object in memory because you need update only the object storage pointer in a handle to relocate an object. If you use direct object references, you must update every reference to an object if you move the object.

Using handles requires additional memory and execution time. Each object access instruction must traverse the extra level of indirection at a cost of at least two additional clock cycles. Also, each handled reference requires one additional word of memory to hold the object storage pointer. Be aware of these tradeoffs when designing a garbage collector.

11.1.2 Reserved Bits in References and Headers

Three bits in each reference, <31:30> and <1>, and four bits in each object header, <31:30> and <2:1>, are reserved for use by software. (For more details regarding reference and header formats, see *References and Headers* on page 63.) The garbage collector can use these reserved bits for a variety of purposes, depending on the algorithm chosen. For example, a *mark-sweep* garbage collector can use one of the bits in the object header to mark objects that have been traversed.

Two of the reserved bits in each reference, <31:30>, influence the behavior of the write-barrier hardware in the core. Use these bits only if you are aware of the configuration of the write-barrier hardware.

11.2 Write Barriers

To enable a variety of garbage collection algorithms, the core provides a flexible write-barrier mechanism to notify the garbage collector when certain reference fields are stored into memory. The garbage collector uses this information to maintain data structures that allow it to reclaim unused memory. The core notifies the garbage collector via a `gc_notify` (type = 0x27) trap. The trap handler should take action depending on the garbage collection algorithm used.

The core triggers the write-barrier trap under certain conditions when a reference field of some object (or an element of an array) is written with some new reference, as illustrated by FIGURE 11-1. For simplicity, this figure assumes that no handle is used and, therefore, the reference denotes the object directly.

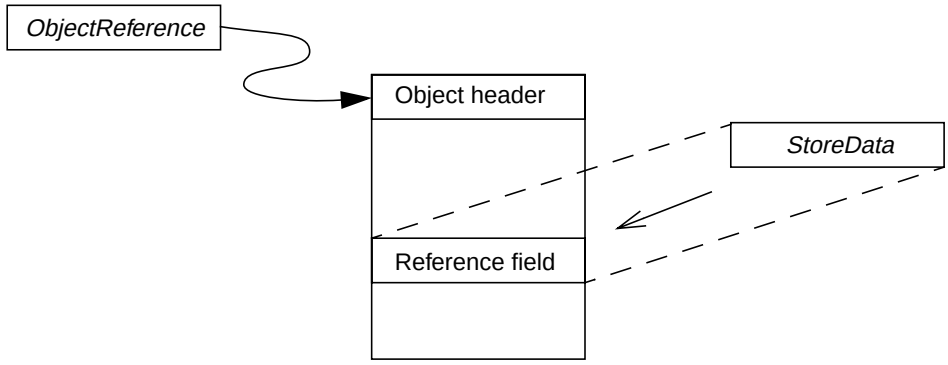


FIGURE 11-1 Storing a Reference into a Field of an Object

Note – In the remainder of this chapter, *ObjectReference* refers to the reference that denotes the object or array in which the field being written; *StoreData* refers to the new value that is to be written into that field.

The conditions under which a write-barrier trap is generated are governed by the values in the PSR.GCE field and the GC_CONFIG register. These configuration registers govern two types of write-barrier mechanisms:

- A page-based write barrier that uses the relative memory locations of *ObjectReference* and *StoreData* to determine when to signal a gc_notify trap
- A reference-based write barrier that uses the two-bit GC_TAG reserved fields of both *ObjectReference* and *StoreData* to determine when to signal a gc_notify trap

You can use both mechanisms simultaneously as required by your garbage collection scheme. Also, you can disable either or both of the mechanisms if you do not want to use them.

11.2.1 Instructions Subject to Write-Barrier Checks

The instructions that store references into other objects or arrays, thus being subject to write-barrier checks, are listed in TABLE 11-1.

TABLE 11-1 Instructions That Store References

`aastore`	`aastore_quick`
`putfield`	`aputfield_quick`
`putstatic`	`aputstatic_quick`

The core performs the write-barrier checks in hardware for `aputfield_quick`, `aputstatic_quick` and `aastore_quick`. The emulation trap routines must perform the appropriate checks for the other instructions. In the case of `putfield` and `putstatic`, after the corresponding field has been resolved and when the field being written is a reference, the emulation trap handler typically changes the `putfield` or `putstatic` into `aputfield_quick` or `aputstatic_quick`, which performs the check in hardware. However, `aastore` must emulate the write-barrier checks in software, or perform the store within the emulation trap handler using `aastore_quick`.

Caution – You must also emulate garbage collection checks in any routine that stores an object reference (whether within an object or an array) using an instruction not listed in TABLE 11-1. Optimized implementations of a library routine, such as `java.lang.System.arraycopy`, may be one such case.

In the case of `aputfield_quick`, `aputstatic_quick` and `aastore_quick`, the core takes the write-barrier trap before the store that triggered it takes place. Therefore, the `gc_notify` trap handler must ensure one of the following:

- Returning from the trap handler and reexecuting the store does not generate another `gc_notify` trap.
- The store is explicitly emulated, which entails setting the return `PC` in the trap frame to point to the instruction that follows the store, then popping the arguments from the operand stack appropriately.

11.2.2 Page-Based Write Barrier

The page-based write barrier detects situations where, within a larger region of memory divided into a number of fixed-sized pages, the *StoreData* reference is located in a different page than the *ObjectReference*. The `PSR.GCE` bit enables this check when it is set to 1. When the check is enabled, the `GC_CONFIG.REGION_MASK`

and GC_CONFIG.CAR_MASK fields govern the operation of the check. (The term *car* is synonymous with page and is the terminology introduced by train algorithm garbage collectors.)

Typically, you should do the following:

- Initialize the REGION_MASK field of GC_CONFIG with a value that, when represented in binary form, consists of some number of zeros, followed by some number of ones.
- Initialize the CAR_MASK field with a value that consists of some number of ones followed by some number of zeros. TABLE 11-2 and TABLE 11-3 list those values.

Region Size	REGION_MASK
256 Kbytes	0x000
512 Kbytes	0x001
1 Mbytes	0x003
2 Mbytes	0x007
4 Mbytes	0x00f
8 Mbytes	0x01f
16 Mbytes	0x03f
32 Mbytes	0x07f
64 Mbytes	0x0ff
128 Mbytes	0x1ff
256 Mbytes	0x3ff
512 Mbytes	0x7ff

TABLE 11-2 GC_CONFIG.REGION_MASK Values

Page Size	CAR_MASK
Reserved	0x00
128 Kbytes	0x10
64 Kbytes	0x18
32 Kbytes	0x1c
16 Kbytes	0x1e
8 Kbytes	0x1f

TABLE 11-3 GC_CONFIG.CAR_MASK Values

FIGURE 11-2 illustrates the operation of the page-based write-barrier check.

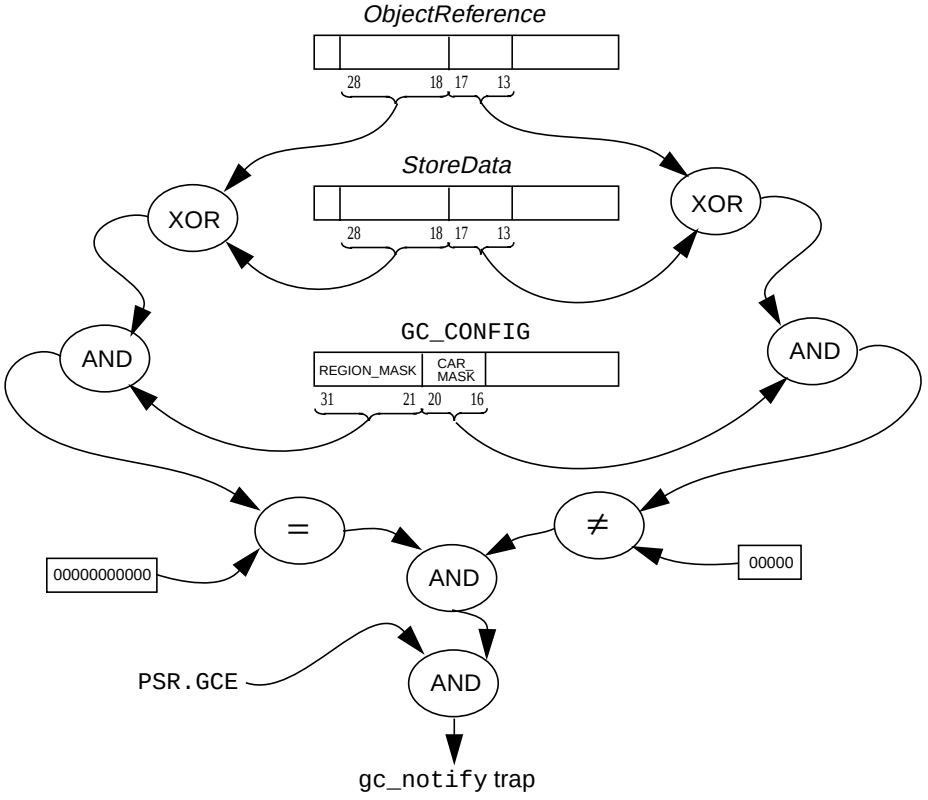


FIGURE 11-2 Operation of Page-Based Write Barrier

CODE EXAMPLE 11-1 details the same check.

CODE EXAMPLE 11-1 Pseudocode for a Page-Based Write Barrier

```
if  (
        ( PSR.GCE = 1 )
        AND (( ObjectReference<28:18> & GC_CONFIG<31:21> ) =
             ( StoreData<28:18> & GC_CONFIG<31:21> ))
        AND (( ObjectReference<17:13> & GC_CONFIG<20:16> )    ≠
             ( StoreData<17:13> & GC_CONFIG<20:16> ))
    ) then
        gc_notify trap
```

11.2.3 Reference-Based Write Barrier

The reference-based write barrier detects situations when GC_TAG fields of the *StoreData* reference and the *ObjectReference* are combined and indicate a gc_notify trap should be taken. The two GC_TAG fields form an index into the GC_CONFIG.WB_VECTOR field. If the corresponding bit is set to 1, then the core generates a trap. You can disable this check by setting GC_CONFIG.WB_VECTOR to 0.

FIGURE 11-3 illustrates the operation of the reference-based write-barrier check.

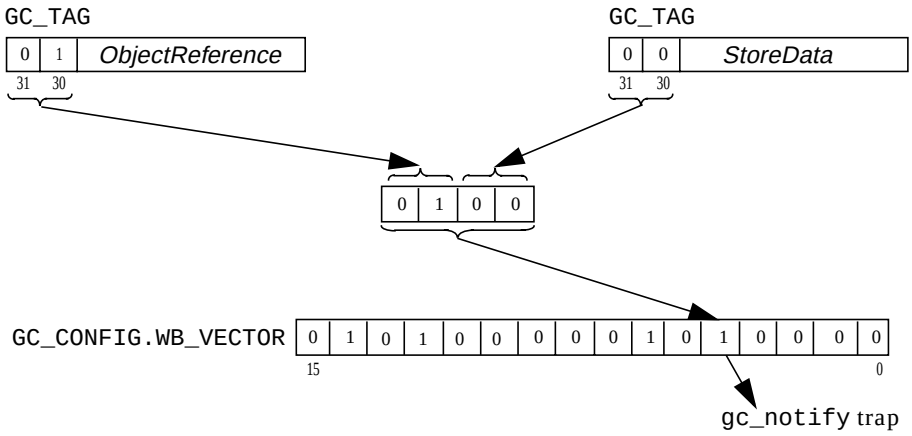


FIGURE 11-3 Operation of Reference-Based Write Barrier

CODE EXAMPLE 11-2 details the same check.

CODE EXAMPLE 11-2 Pseudocode for a Reference-Based Write Barrier

```
gc_index ⇐ ( ObjectReference<31:30> << 2 ) | ( StoreData<31:30> )
write_barrier_bit ⇐ ( GC_CONFIG >> gc_index ) | 0x00000001
if  (write_barrier_bit = 1) then
        gc_notify trap
```

11.3 Examples

The examples in this section are illustrations only to demonstrate how you can apply the write-barrier mechanisms to a range of GC algorithms. They do not describe the GC scheme and assume that you are familiar with the concepts.

You can combine these ideas in your garbage collector implementation. For instance, you can combine both page-based and reference-based write barriers in an incremental generational garbage collector, where the former mechanism tracks references that cross generation boundaries; the latter is used to synchronize application (mutator) threads with a three-color incremental graph-tracing algorithm used within each generation.

11.3.1 Train Algorithm-Based Collectors

The train algorithm allows nondisruptive collection of the oldest generation of a generational system. It uses a write barrier to track references between different memory regions (known as “cars” in [Hudson 1992] and [Grarup 1993]) within the oldest generation. Generally, these cars are defined as fixed, power-of-two-sized regions that are aligned on power-of-two boundaries (similar to pages in a virtual memory system).

The page-based write-barrier check was designed specifically to assist this class of GC algorithm. The page (car) size is configurable, based on the `CAR_MASK` field of the `GC_CONFIG` register. See TABLE 11-3 on page 437.

For more details on the train algorithm and its implementation, see [Hudson ‘92] or [Grarup ‘93].

11.3.2 Remembered Set-Based Generational Collector

FIGURE 11-4 illustrates a heap partitioned into two generations—an old and a young generation. A remembered set keeps track of those “old” objects that hold references that denote “young” objects.

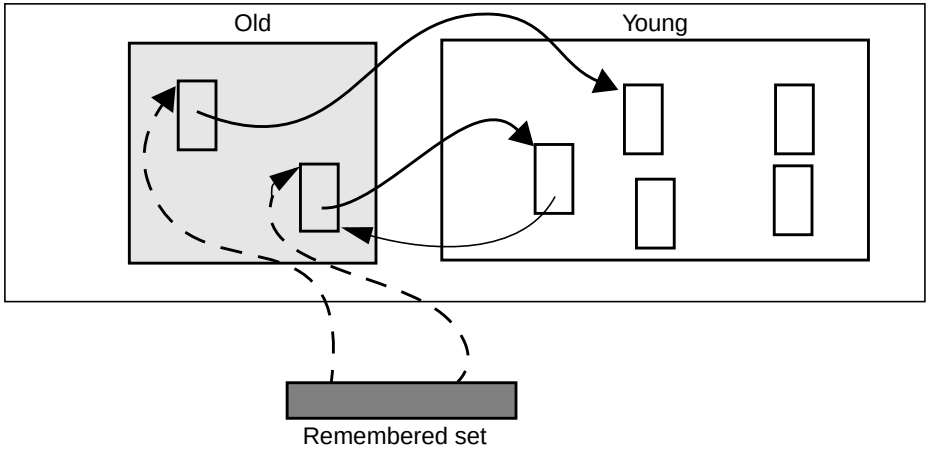


FIGURE 11-4 Generational Garbage Collection Using Remembered Sets

You can use the reference-based write-barrier mechanism to update the remembered set, as follows:

- The `WB_VECTOR` field of the `GC_CONFIG` register contains the value 0x5050.
- The `GC_TAG` bits of object references that denote an old object are set to 00; the `GC_TAG` bits of object references that denote a young object are set to 01.

This configuration ensures that the write-barrier trap is triggered whenever a reference denoting a young object is written into an old object, as illustrated by the example values in FIGURE 11-3 on page 439. Whenever that situation occurs, the `gc_notify` trap handler adds *ObjectReference* to the remembered set for the young generation accordingly.

Note – You can also implement remembered-set generation algorithms with the page-based write-barrier. If you do so, however, you lose some flexibility in the relative sizing of the generations.

11.4 References

[Grarup 1993] Grarup, S., and J. Seligmann: *Incremental Mature Garbage Collection*, M. Sc. Thesis, Aarhus University, Computer Science Department, August 1993.

[Hudson 1992] Hudson, R., and J. E. B. Moss: *Proceedings of International Workshop on Memory Management*: *Incremental Garbage Collection For Mature Objects*, St. Malo, France, September 16–18, 1992.

[Wilson 1992] Wilson, P., *Uniprocessor Garbage Collection Techniques*. In International Workshop on Memory Management, St. Malo, France, September 1992. (Also Springer-Verlag Lecture Notes in Computer Science No. 637).

CHAPTER 12

System Management and Debugging

This chapter provides information regarding power management, reset management, and debugging support.

This chapter describes following features of the picoJava-II core:

- *Power Management* on page 443
- *Reset Management* on page 444
- *Breakpoints* on page 450
- *Other Debug and Trace Features* on page 454

12.1 Power Management

The picoJava-II core can operate in two modes, normal mode and a power-saving standby mode. The standby mode can occur for as little as one cycle.

In the standby mode, the core powers down all functional units and stops the internal clock. The core enters standby mode by executing the powerdown instruction, `priv_powerdown`. The number of cycles it takes the core to enter standby mode is unspecified.

During standby, the core asserts a signal to external devices, allowing them to enter a power-conserving mode if available. Refer to your chip or system manual to find out if this signal is used.

All state within the processor is retained in standby mode as long as the main clock into the core is active.

The core wakes up from standby mode if there is an external interrupt. Resumption of normal operation within the core may take up to four clock cycles.

After re-entering normal mode, the core begins execution at the first instruction of the trap handler of the interrupt that caused the core to exit powerdown mode. Returning from this interrupt handler will continue execution with the instruction following `priv_powerdown`.

A reset also returns the core from standby mode to normal mode.

12.2 Reset Management

Upon power-on or reset, only a portion of the core enters a known state. The code that executes at this point must carefully initialize the rest of the core state before normal execution can begin.

The core can be reset in one of two ways:

- By assertion of the external reset signal to the core
- By executing the `priv_reset` instruction

12.2.1 Machine State After Reset

When the core is powered on or reset, the registers inside the core are set to the values listed in TABLE 12-1. The values marked Hardwired are initialized to values dependent on specific chip and system implementation. Consult your chip or system manual to find out the values of these fields at power-on or after a reset. The values marked Unknown must be initialized to known values before enabling features of the chip that make use of them. For example, the TRAPBASE register must be initialized before enabling interrupts or executing any instructions that trap to emulation routines.

The RAM blocks in the stack cache, the instruction cache, and the data cache are uninitialized at power-on and remain unchanged when the core is reset. After a power-on or reset, the core starts executing instructions from address 0x00000000.

TABLE 12-1 Machine State Changes on POR/SIR and Powerdown

Register	Field	POR/SIR
PC		0
VERSION_ID		Hardwired

TABLE 12-1 Machine State Changes on POR/SIR and Powerdown *(Continued)*

Register	Field	POR/SIR
PSR	DBH	Unknown
	DBL	Unknown
	DRT	0
	BM8	Hardwired
	ACE	0
	GCE	0
	FPE	0
	DCE	0
	ICE	0
	AEM	0
	DRE	0
	FLE	0
	SU	1
	IE	0
	PIL	Unknown
HCR	DCL	Hardwired
	ICL	Hardwired
	DCS	Hardwired
	ICS	Hardwired
	FPP	Hardwired
	SRN	Hardwired
TRAPBASE	TBA	Unknown
	tt	Unknown
SC_BOTTOM		0x003ffffc
OPTOP		0x003ffffc
OPLIM		0x00000000
VARS		0x003ffffc
BRK12C		0x00000000
GC_CONFIG		0x00000000
Other registers (see Chapter 2, *Registers*)		Unknown

12.2.2 Enabling the Stack Cache

As part of the reset sequence, you must be careful when enabling the stack cache. Since the picoJava-II core is a stack-based machine, all instructions depend on the correct operation of the stack cache and the dribbler. When the dribbler is off, the core does not behave like a true stack-based machine; therefore, even simple operations that cause stack movement may produce unexpected results. Specifically, if more than 64 words are pushed onto the top of the program stack before the dribbler is enabled, there can be corruption of data elements on the stack.

To avoid unpredictable behavior, it is *strongly* recommended that the first few instructions in the reset handler first set up `OPTOP` and `SC_BOTTOM` to their valid locations and enable the dribbler. The dribbler high and low watermarks must be initialized at the time the dribbler is enabled. All other reset functionality should be performed after the stack has been set up in a valid area and the dribbler has been enabled.

Systems for which the power-on/reset values of `OPTOP` and `SC_BOTTOM` (which place the stack at address 0x003ffffc) are unsuitable must set `OPTOP` and `SC_BOTTOM` to their new values *before* enabling the dribbler. This requirement is because switching to a different stack location after enabling the dribbler causes a flush of the old stack, causing unwanted writes to the memory area around 0x00400000. When the dribbler is disabled, stack switching does not cause flushing of the old stack to memory.

CODE EXAMPLE 12-1 illustrates a sample reset code for a system that does not have memory at 0x003ffffc (and must avoid writes to this region) but has its startup stack located at address 0x01fffffc and below.

CODE EXAMPLE 12-1 Enabling the Stack Cache

```
Address_0x0:
     bipush 0xfc
     sethi 0x01ff
      dup
     write_sc_bottom
    write_optop
     priv_read_psr
      sipush 0x0080
      sethi 0x0032
     ior
    priv_write_psr
```

12.2.3 Enabling the Instruction and Data Caches

As part of the reset sequence, code must use diagnostic cache writes to explicitly initialize the instruction and data cache tags to the invalid state, before enabling them. The contents of the tag RAMs are uninitialized at power-on.

The contents of the stack cache do not need to be explicitly initialized at power-on because there are no valid entries in the stack cache.

CODE EXAMPLE 12-2 and CODE EXAMPLE 12-3 illustrate how reset code can compute the size of the caches from the HCR register and set all lines in both caches to the invalid state.

CODE EXAMPLE 12-2 Enabling the Instruction Cache

```
EnableICache:
     priv_read_hcr
      iconst_0
      sethi 0x001c
     iand                  // Mask off HCR.ICS
      bipush 18
     ishr
      dup
     ifne ICachePresent
    pop
    goto EnableDCache      // No I-Cache, do D-Cache next

ICachePresent:
      priv_read_hcr
       iconst_0
       sethi 0x0700
      iand                 // Mask off HCR.ICL
       bipush 24
      ishr
       iconst_4
       swap
      ishl
     write_global1         // Put line size in bytes in global1
      sipush 0x0200
      swap                 // Turn HCR.ICS field from stack into
     ishl                  // instruction cache size in bytes
      priv_read_hcr
       iconst_0
       sethi 0x0001
      iand                 // Mask off HCR.ICA
     iushr                 // Set cache size to size of a “way”
      read_global1
     isub
```

CODE EXAMPLE 12-2 Enabling the Instruction Cache *(Continued)*

```
     write_global2          // global2 is maximum tag address

  ICacaheInvalidateLoop:
      iconst_0
       read_global2
     priv_write_icache_tag  // Invalidate line by writing 0 tag
      iconst_0
       read_global2
        priv_read_hcr
         bipush 16
        iushr
         bipush 31
        ishl                // Set bit 31 to value of HCR.ICA
       ior                  // Tag address is for way 1, if present
     priv_write_icache_tag  // invalidate line by writing 0 tag
      read_global2
       read_global1
      isub
     write_global2          // Get next tag address by subtracting
      read_global2          // line size. Loop through zero.
     ifge ICacheInvalidateLoop

      priv_read_psr
       sipush 0x0200
      ior
     priv_write_psr         // Enable I-Cache - set PSR.ICE to 1

  ICacheDone:
```

The code to enable the data cache is very similar to that required to enable the instruction cache. The only significant difference is based on the fact that data cache can be up to four-way set-associative, according to the HCR.DCA bits.

CODE EXAMPLE 12-3 Enabling the Data Cache

```
  EnableDCache:
      priv_read_hcr
       iconst_0
       sethi 0x00e0
      iand                  // Mask off HCR.DCS
       bipush 21
      ishr
       dup
      ifne DCachePresent
     pop
     goto DCacheDone        // No D-Cache, we are done
```

CODE EXAMPLE 12-3 Enabling the Data Cache *(Continued)*

```
DcachePresent:
      priv_read_hcr
       iconst_0
       sethi 0x3800
      iand                 // Mask off HCR.DCL
       bipush 27
      ishr
       iconst_4
       swap
      ishl
     write_global1         // Put line size in bytes in global1
      sipush 0x0200        // Turn HCR.DCS field from stack into
      swap                 // size of data cache in bytes
     ishl
      priv_read_hcr
       bipush 30
      iushr                // Mask off HCR.DCA
       dup
        iconst_2
      if_icmpne DCSetAssoc // Set associative, need to adjust size
     pop
     goto DCSizeDone
DCSetAssoc:
       iconst_1
      iadd
     iushr                 // Size of data cache “way” in bytes
DCSizeDone:
      read_global1
     isub
    write_global2          // global2 is maximum way 0 tag address

DCacaheInvalidateLoop:
     iconst_0
      read_global2         // Tag address is for way 0
    priv_write_dcache_tag  // Invalidate line by writing 0 tag
     priv_read_hcr
      bipush 31
     iushr                 // Bit 31 of HCR set if direct mapped
    ifne DCacheNextSet
     iconst_0
      read_global2
      sethi 0x8000         // Tag address is way 1 (or 2 if 4-way)
    priv_write_dcache_tag  // Invalidate line by writing 0 tag
     priv_read_hcr
      bipush 30
     iushr                 // HCR is zero if 2-way set associative
    ifne DCacheNextSet
```

CODE EXAMPLE 12-3 Enabling the Data Cache *(Continued)*

```
      iconst_0
       read_global2
       sethi 0x4000          // Tag address is way 1 if 4-way
     priv_write_dcache_tag   // Invalidate line by writing 0 tag
      iconst_0
       read_global2
       sethi 0xc000          // Tag address is way 3 if 4-way
     priv_write_dcache_tag   // Invalidate line by writing 0 tag
  DCacheNextSet:
      read_global2
       read_global1
      isub
     write_global2           // Get next tag address by subtracting
      read_global2           // line size. Loop through zero.
     ifge DCacheInvalidateLoop

      priv_read_psr
       sipush 0x0400
      ior
     priv_write_psr          // Enable D-Cache - set PSR.DCE to 1

  DCacheDone:
```

12.3 Breakpoints

The core supports data and instruction breakpoints. Data and instruction breakpoint traps are triggered when there is a match between an instruction or data address with the address stored in one of the two breakpoint address registers, BRK1A and BRK2A. The BRK12C register controls exactly how the address comparison is performed.

To enable a breakpoint:

1. **Set up the address breakpoint registers.**
2. **Write to the breakpoint control register.**

Note – Do not set a breakpoint to trigger the instruction that sets it. Unexpected results may occur.

12.3.1 Data Breakpoints

There are two types of data breakpoints:

- A data store breakpoint traps on a store to the specified address instruction.
- A data load breakpoint, also known as a watchpoint, traps on a load from the specified address.

Both data load and store breakpoints appear to happen before the load or store causing the breakpoint has completed. The PC stored in the trap frame is that of the instruction which caused the breakpoint. In the special case of a store breakpoint set on an address that matches the second half of a double-word store (such as `lastore`), the first half of the double-word is written but the second word is not written when the breakpoint trap handler is entered.

The core performs data breakpoint checks on all accesses that go to the data cache. Hence, data breakpoints are checked on all direct loads and stores, object accesses, and local variable accesses that miss the stack cache. Since there is no way in general to generate breakpoint traps on accesses to stack data (because local variables may or may not reside in the stack cache), debuggers may prefer to disallow the setting of data breakpoints on data residing on the stack.

Instruction bytes accessed through the instruction cache are not subject to data breakpoint checks. However, since all accesses to the data cache are checked for data breakpoints, unexpected actions can trigger data breakpoints. Some examples are: the action of reading the address of a trap vector from the trap table, or the tableswitch instruction reading its operands from the instruction stream as if they were data.

Data breakpoints can be used whether instruction folding is enabled or disabled.

12.3.2 Instruction Breakpoints

Instruction breakpoint traps take place before an instruction is executed. The PC stored in the trap frame is the PC of the instruction causing the trap. Hardware instruction breakpoints are triggered only when instruction folding is disabled (PSR.FLE is 0).

12.3.3 Breakpoint Registers

The core contains three registers: two breakpoint registers (BRK1A and BRK2A) and a breakpoint control register (BRK12C). You can set a maximum of two breakpoints at one time.

The configuration of the BRK1A register is listed below and illustrated in FIGURE 12-1.

Field	Type	Description
31:00	RW	This is the breakpoint1 address against which to compare. This register is used along with BRK12C to set a breakpoint.

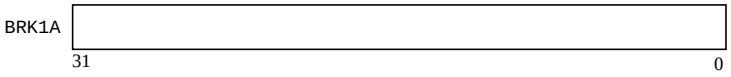


FIGURE 12-1 Breakpoint Register (BRK1A)

The configuration of the BRK2A register is listed below and illustrated in FIGURE 12-2.

Field	Type	Description
31:00	RW	This is the breakpoint2 address against which to compare. This register is used along with BRK12C to set a breakpoint.

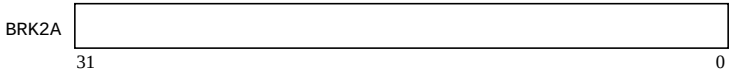


FIGURE 12-2 Breakpoint Register (BRK2A)

The configuration of the BRK12C register is listed below and illustrated in FIGURE 12-3.

Field	Type	Description
31	RW	HALT – Determines if a breakpoint halts or traps. At setting 0, the breakpoint traps (default); at setting 1, the core halts all transactions.
30:24	RW	BRKM2 – Mask bits for breakpoint2 <30> – Enable compare of BRK2A<31:13> <29> – Enable compare of BRK2A<12> <28> – Enable compare of BRK2A<11:4> <27> – Enable compare of BRK2A<3> <26> – Enable compare of BRK2A<2> <25> – Enable compare of BRK2A<1> <24> – Enable compare of BRK2A<0>
23	Reserved	

Field	Type	Description
22:16	RW	BRKM1 – Mask bits for breakpoint1 <22> – Enable compare of BRK1A<31:13> <21> – Enable compare of BRK1A<12> <20> – Enable compare of BRK1A<11:4> <19> – Enable compare of BRK1A<3> <18> – Enable compare of BRK1A<2> <17> – Enable compare of BRK1A<1> <16> – Enable compare of BRK1A<0>
15:12	Reserved	
11	RW	SUBRK2 – Supervisor (privileged) or user access for breakpoint2
10:9	RW	SRCBRK2 – Source for breakpoint2 0x0 – Data cache read 0x1 – Data cache write 0x2 – Reserved 0x3 – Instruction cache fetch (folding disabled)
8	RW	BRKEN2 – Breakpoint2 trap enable bit 1 – The breakpoint is enabled 0 – The breakpoint is disabled
7:4	Reserved	
3	RW	SUBRK1 – Supervisor (privileged) or user access for breakpoint1. If set to 1, then the core ignores the breakpoint in privileged mode.
2:1	RW	SRCBRK1 – Source for breakpoint1 0x0 – Data cache read 0x1 – Data cache write 0x2 – Reserved 0x3 – Instruction cache fetch (folding disabled)
0	RW	BRKEN1 – Breakpoint1 trap enable bit 1 – The breakpoint is enabled 0 – The breakpoint is disabled

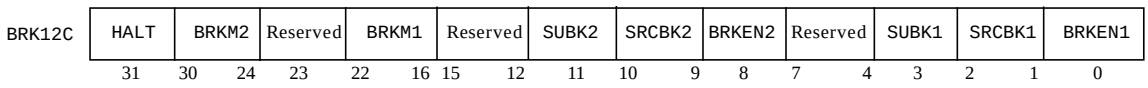


FIGURE 12-3 Breakpoint Control Register (BRK12C)

12.3.4 Breakpoint Address Matching

The BRK12C.BRKM1 and BRK12C.BRKM2 bits control which address bits in the corresponding breakpoint address register should be compared with the instruction or data address generated by the chip. If a mask bit is 1, then the corresponding range of bits in the BRK1A or BRK2A register must match the generated address. This flexibility allows setting of breakpoints to cover more than just one word of memory—a single breakpoint address register can cover a region of up to 8 Kbytes.

Breakpoints are triggered only on exact matches to the breakpoint addresses and are not triggered on accesses "covering" the breakpoint address. For example, if the breakpoint address is set to address 0x103 with all bits enabled for address comparison, a word write to address 0x100, which covers addresses 0x100–0x103, will not cause a breakpoint trap.

In addition, each breakpoint can be either enabled or disabled using the BRK12C.BRK1EN and BRK12C.BRK2EN bits. Additionally, each breakpoint can optionally be enabled only in user mode. When the BRK12C.SUBK1 and BRK12C.SUBK2 bits for each breakpoint are set to 1, breakpoint addresses are compared only when in user mode (PSR.SU is 0). When the bit is set to 0, breakpoint addresses are compared both in user and superuser modes.

12.3.5 Breakpoints and Halt Mode

The same hardware breakpoints mechanism can also be used to halt the entire processor instead of causing a trap. Hardware signals can then be used to probe the internal state of the chip. However, this requires special hardware support in the system for continuing from a breakpoint and probing the state of the chip. To find if this feature is supported in your system, see your system documentation.

To use breakpoints to force the core into halt mode instead of causing a trap, set the HALT bit in the BRK12C register.

12.4 Other Debug and Trace Features

The core provides additional hardware support for debugging through signals for halting and single-stepping the processor and a scan chain accessible through an IEEE 1149.1 JTAG interface. These methods need specialized hardware support. See your chip and system documentation for details about these and other debugging options.

PART III Appendixes

APPENDIX **A**

Opcodes

This appendix lists and describes the Java virtual machine opcodes and the picoJava-II-specific additions to the instruction set. It contains these tables:

- *TABLE A-1 picoJava-II 1-Byte Opcodes* on page 458
- *TABLE A-2 picoJava-II 2-byte Opcodes* on page 469

In the tables, the Group column lists the folding category for the instructions.

The Cycles column shows a typical number of cycles the instructions take, assuming cache hits and no pipeline stalls or exceptions. If an instruction is marked *Trap*, it is not executed by the hardware but traps to a software emulation trap handler.

Here is a key to the acronyms in the tables:

LV	Local variable load or load from global register or push constant
OP	An operation that uses the top two entries of stack
BG2	An operation that uses the top two entries of the stack and breaks the group
BG1	An operation that uses only the topmost entry of stack and breaks the group
MEM	Local variable stores, global register stores, and memory loads
NF	Nonfoldable instruction
LDUSE	Addition of an extra cycle if subsequent instructions use the load results

Here is a key to the footnotes in the tables:

1 Assumes a nonhandle reference.

2 Assumes a handle reference.

3 Optionally traps, depending on the PSR.DRT bit. See *Instruction Emulation* on page 54.

4 Assumes conditional branch is not taken.

5 Assumes conditional branch is taken.

6 Depends on the index and `tableswitch` bounds: If the index is less than the lower bound, then `tableswitch` take 10 cycles; if the index is greater than the upper bound, then `tableswitch` takes 11 cycles.

7 May trap if the object reference is not held in LOCKADDR registers. See Chapter 8, *Monitors*.

8 May trap if the hardware would have required an examination of the superclass type of the checked object.

TABLE A-1 lists and describes the Java virtual machine opcodes.

TABLE A-1 picoJava-II 1-Byte Opcodes

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
0 (0x0)	`nop`	1	Do not operate.	NF	1	
1 (0x1)	`aconst_null`	1	Push null object.	LV	1	
2 (0x2)	`iconst_m1`	1	Push integer constant –1.	LV	1	
3 (0x3)	`iconst_0`	1	Push integer constant 0.	LV	1	
4 (0x4)	`iconst_1`	1	Push integer constant 1.	LV	1	
5 (0x5)	`iconst_2`	1	Push integer constant 2.	LV	1	
6 (0x6)	`iconst_3`	1	Push integer constant 3.	LV	1	
7 (0x7)	`iconst_4`	1	Push integer constant 4.	LV	1	
8 (0x8)	`iconst_5`	1	Push integer constant 5.	LV	1	
9 (0x9)	`lconst_0`	1	Push long integer constant 0.	NF	2	
10 (0x0a)	`lconst_1`	1	Push long integer constant 1.	NF	2	
11 (0x0b)	`fconst_0`	1	Push float constant 0.0.	LV	1	
12 (0x0c)	`fconst_1`	1	Push float constant 1.0.	LV	1	
13 (0x0d)	`fconst_2`	1	Push float constant 2.0.	LV	1	
14 (0x0e)	`dconst_0`	1	Push double float 0.0.	NF	2	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
15 (0x0f)	dconst_1	1	Push double float 1.0.	NF	2	
16 (0x10)	bipush	2	Push 1-byte integer.	LV	1	
17 (0x11)	sipush	3	Push 2-byte integer.	LV	1	
18 (0x12)	ldc	2	Load constant from the constant pool.	NF	Trap	
19 (0x13)	ldc_w	3	Load constant from constant pool using a wider offset (16-bit index).	NF	Trap	
20 (0x14)	ldc2_w	3	Load long or double from constant pool.	NF	Trap	
21 (0x15)	iload	2	Load local integer variable.	LV	1	
22 (0x16)	lload	2	Load local long variable.	NF	2	
23 (0x17)	fload	2	Load local float variable.	LV	1	
24 (0x18)	dload	2	Load local double float variable.	NF	2	
25 (0x19)	aload	2	Load local object variable.	LV	1	
26 (0x1a)	iload_0	1	Load local variable 0.	LV	1	
27 (0x1b)	iload_1	1	Load local variable 1.	LV	1	
28 (0x1c)	iload_2	1	Load local variable 2.	LV	1	
29 (0x1d)	iload_3	1	Load local variable 3.	LV	1	
30 (0x1e)	lload_0	1	Load local long variable 0.	NF	2	
31 (0x1f)	lload_1	1	Load local long variable 1.	NF	2	
32 (0x20)	lload_2	1	Load local long variable 2.	NF	2	
33 (0x21)	lload_3	1	Long local long variable 3.	NF	2	
34 (0x22)	fload_0	1	Load local float variable 0.	LV	1	
35 (0x23)	fload_1	1	Load local float variable 1.	LV	1	
36 (0x24)	fload_2	1	Load local float variable 2.	LV	1	
37 (0x25)	fload_3	1	Load local float variable 3.	LV	1	
38 (0x26)	dload_0	1	Load local double variable 0.	NF	2	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
39 (0x27)	dload_1	1	Load local double variable 1.	NF	2	
40 (0x28)	dload_2	1	Load local double variable 2.	NF	2	
41 (0x29)	dload_3	1	Load local double variable 3.	NF	2	
42 (0x2a)	aload_0	1	Load local object variable 0.	LV	1	
43 (0x2b)	aload_1	1	Load local object variable 1.	LV	1	
44 (0x2c)	aload_2	1	Load local object variable 2.	LV	1	
45 (0x2d)	aload_3	1	Load local object variable 3.	LV	1	
46 (0x2e)	iaload	1	Load integer from array.	BG2	$3^1/5^2$	Yes
47 (0x2f)	laload	1	Load long from array.	BG2	$4^1/6^2$	Yes
48 (0x30)	faload	1	Load float from array.	BG2	$3^1/5^2$	Yes
49 (0x31)	daload	1	Load double from array.	BG2	$4^1/6^2$	Yes
50 (0x32)	aaload	1	Load object ref from array.	BG2	$3^1/5^2$	Yes
51 (0x33)	baload	1	Load signed byte from array.	BG2	$3^1/5^2$	Yes
52 (0x34)	caload	1	Load character from array.	BG2	$3^1/5^2$	Yes
53 (0x35)	saload	1	Load short from arrray.	BG2	$3^1/5^2$	Yes
54 (0x36)	istore	2	Store integer into local variable.	MEM	1	
55 (0x37)	lstore	2	Store long into local variable.	NF	2	
56 (0x38)	fstore	2	Store float into local variable.	MEM	1	
57 (0x39)	dstore	2	Store double into local variable.	NF	2	
58 (0x3a)	astore	2	Store object reference into local variable.	MEM	1	
59 (0x3b)	istore_0	1	Store into local variable 0.	MEM	1	
60 (0x3c)	istore_1	1	Store into local variable 1.	MEM	1	
61 (0x3d)	istore_2	1	Store into local variable 2.	MEM	1	
62 (0x3e)	istore_3	1	Store into local variable 3.	MEM	1	
63 (0x3f)	lstore_0	1	Store into local variable 0.	NF	2	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
64 (0x40)	lstore_1	1	Store into local variable 1.	NF	2	
65 (0x41)	lstore_2	1	Store into local variable 2.	NF	2	
66 (0x42)	lstore_3	1	Store into local variable 3.	NF	2	
67 (0x43)	fstore_0	1	Store into local variable 0.	MEM	1	
68 (0x44)	fstore_1	1	Store into local variable 1.	MEM	1	
69 (0x45)	fstore_2	1	Store into local variable 2.	MEM	1	
70 (0x46)	fstore_3	1	Store into local variable 3.	MEM	1	
71 (0x47)	dstore_0	1	Store into local variable 0.	NF	2	
72 (0x48)	dstore_1	1	Store into local variable 1.	NF	2	
73 (0x49)	dstore_2	1	Store into local variable 2.	NF	2	
74 (0x4a)	dstore_3	1	Store into local variable 3.	NF	2	
75 (0x4b)	astore_0	1	Store into local variable 0.	MEM	1	
76 (0x4c)	astore_1	1	Store into local variable 1.	MEM	1	
77 (0x4d)	astore_2	1	Store into local variable 2.	MEM	1	
78 (0x4e)	astore_3	1	Store into local variable 3.	MEM	1	
79 (0x4f)	iastore	1	Store into integer array.	BG2	$5^1/7^2$	Yes
80 (0x50)	lastore	1	Store into long array.	BG2	$6^1/8^2$	Yes
81 (0x51)	fastore	1	Store into float array.	BG2	$5^1/7^2$	Yes
82 (0x52)	dastore	1	Store into double float array.	BG2	$6^1/8^2$	Yes
83 (0x53)	aastore	1	Store into object reference array.	NF	Trap	
84 (0x54)	bastore	1	Store into signed byte array.	BG2	$5^1/7^2$	Yes
85 (0x55)	castore	1	Store into character array.	BG2	$5^1/7^2$	Yes
86 (0x56)	sastore	1	Store into short array.	BG2	$5^1/7^2$	Yes
87 (0x57)	pop	1	Pop top entry in stack.	NF	1	
88 (0x58)	pop2	1	Pop top two entries in stack.	NF	1	
89 (x059)	dup	1	Duplicate top stack word.	NF	1	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
90 (0x5a)	dup_x1	1	Duplicate top word and put two down.	BG2	3	
91 (0x5b)	dup_x2	1	Duplicate top word and put three down.	BG2	4	
92 (0x5c)	dup2	1	Duplicate top two words.	NF	2	
93 (0x5d)	dup2_x1	1	Duplicate top two words and put three down.	BG2	5	
94 (0x5e)	dup2_x2	1	Duplicate top two words and put four down.	BG2	6	
95 (0x5f)	swap	1	Swap top two stack words.	BG2	2	
96 (0x60)	iadd	1	Add integer.	OP	1	
97 (0x61)	ladd	1	Add long.	NF	2	
98 (0x62)	fadd	1	Add float.	OP	3	
99 (0x63)	dadd	1	Add double.	NF	11	
100 (0x64)	isub	1	Subtract integer.	OP	1	
101 (0x65)	lsub	1	Subtract long.	NF	2	
102 (0x66)	fsub	1	Subtract float.	OP	3	
103 (0x67)	dsub	1	Subtract double.	NF	14	
104 (0x68)	imul	1	Multiply integer.	OP	2 - 18	
105 (0x69)	lmul	1	Multiply long.	NF	Trap	
106 (0x6a)	fmul	1	Multiply float.	OP	3	
107 (0x6b)	dmul	1	Multiply double.	NF	14	
108 (0x6c)	idiv	1	Divide integer.	OP	32	
109 (0x6d)	ldiv	1	Divide long.	NF	Trap	
110 (0x6e)	fdiv	1	Divide float.	OP	30	
111 (0x6f)	ddiv	1	Divide double.	NF	60	
112 (0x70)	irem	1	Compute integer remainder.	OP	32	
113 (0x71)	lrem	1	Compute long remainder.	NF	Trap	
114 (0x72)	frem	1	Compute float remainder.	OP	<200	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
115 (0x73)	drem	1	Compute double remainder.	NF	<2000[3]	
116 (0x74)	ineg	1	Negate integer.	BG1	1	
117 (0x75)	lneg	1	Negate long.	NF	2	
118 (0x76)	fneg	1	Negate float.	BG1	1	
119 (0x77)	dneg	1	Negate double.	NF	1	
120 (0x78)	ishl	1	Shift left integer.	OP	1	
121 (0x79)	lshl	1	Shift left long.	NF	2	
122 (0x7a)	ishr	1	Arithmetic shift right integer.	OP	1	
123 (0x7b)	lshr	1	Arithmetic shift right long.	NF	2	
124 (0x7c)	iushr	1	Logical shift right integer.	OP	1	
125 (0x7d)	lushr	1	Logical shift right long.	NF	2	
126 (0x7e)	iand	1	Compute bitwise AND.	OP	1	
127 (0x7f)	land	1	Compute long bitwise AND.	NF	2	
128 (0x80)	ior	1	Compute integer bitwise OR.	OP	1	
129 (0x81)	lor	1	Compute long bitwise OR.	NF	2	
130 (0x82)	ixor	1	Compute integer bitwise XOR.	OP	1	
131 (0x83)	lxor	1	Compute long bitwise XOR.	NF	2	
132 (0x84)	iinc	3	Increment local variable by constant.	NF	1	
133 (0x85)	i2l	1	Convert integer to long.	NF	1	
134 (0x86)	i2f	1	Convert integer to float.	NF	6	
135 (0x87)	i2d	1	Convert integer to double.	NF	3	
136 (0x88)	l2i	1	Convert long to integer.	NF	1	
137 (0x89)	l2f	1	Convert long to float.	NF	8	
138 (0x8a)	l2d	1	Convert long to double.	NF	6	
139 (0x8b)	f2i	1	Convert float to integer.	NF	4	
140 (0x8c)	f2l	1	Convert float to long.	NF	5	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
141 (0x8d)	f2d	1	Convert float to double.	NF	3	
142 (0x8e)	d2i	1	Convert double to integer.	NF	5	
143 (0x8f)	d2l	1	Convert double to long.	NF	5	
144 (0x90)	d2f	1	Convert double to float.	NF	7	
145 (0x91)	i2b	1	Convert integer to byte.	BG1	1	
146 (0x92)	i2c	1	Convert integer to character.	BG1	1	
147 (0x93)	i2s	1	Convert integer to short.	BG1	1	
148 (0x94)	lcmp	1	Compare long.	NF	2	
149 (0x95)	fcmpl	1	Float compare –1 on incomparable.	OP	7	
150 (0x96)	fcmpg	1	Float compare 1 on incomparable.	OP	7	
151 (0x97)	dcmpl	1	Double compare –1 on incomparable.	NF	7	
152 (0x98)	dcmpg	1	Double compare 1 on incomparable.	NF	7	
153 (0x99)	ifeq	3	Branch if equal to 0.	BG1	$1^4/4^5$	
154 (0x9a)	ifne	3	Branch if not equal to 0.	BG1	$1^4/4^5$	
155 (0x9b)	iflt	3	Branch if less than 0.	BG1	$1^4/4^5$	
156 (0x9c)	ifge	3	Branch if greater than or equal 0.	BG1	$1^4/4^5$	
157 (0x9d)	ifgt	3	Branch if greater than 0.	BG1	$1^4/4^5$	
158 (0x9e)	ifle	3	Branch if less than or equal 0.	BG1	$1^4/4^5$	
159 (0x9f)	if_icmpeq	3	Compare top two stack elements, branch on equal.	BG2	$1^4/4^5$	
160 (0xa0)	if_icmpne	3	Compare top two stack elements, branch on not equal.	BG2	$1^4/4^5$	
161 (0xa1)	if_icmplt	3	Compare top two stack elements, branch on less than.	BG2	$1^4/4^5$	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
162 (0xa2)	`if_icmpge`	3	Compare top two stack elements, branch on greater than or equal.	BG2	1[4]/4[5]	
163 (0xa3)	`if_icmpgt`	3	Compare top two stack elements, branch on greater than.	BG2	1[4]/4[5]	
164 (0xa4)	`if_icmple`	3	Compare top two stack elements, branch on less than or equal.	BG2	1[4]/4[5]	
165 (0xa5)	`if_acmpeq`	3	Compare top two stack objects, branch on equal.	BG2	1[4]/4[5]	
166 (0xa6)	`if_acmpne`	3	Compare top two stack objects, branch on not equal.	BG2	1[4]/4[5]	
167 (0xa7)	`goto`	3	Branch always.	NF	4	
168 (0xa8)	`jsr`	3	Jump to subroutine.	NF	4	
169 (0xa9)	`ret`	2	Return from subroutine.	NF	4	
170 (0xaa)	`tableswitch`	--	Access jump table by index and jump.	NF	15[6]	
171 (0xab)	`lookupswitch`	--	Access jump table by match and jump.	NF	Trap	
172 (0xac)	`ireturn`	1	Return integer from procedure.	BG1	8	
173 (0xad)	`lreturn`	1	Return long from procedure.	NF	8	
174 (0xae)	`freturn`	1	Return float from procedure.	BG1	8	
175 (0xaf)	`dreturn`	1	Return double from procedure.	NF	8	
176 (0xb0)	`areturn`	1	Return object from procedure.	BG1	8	
177 (0xb1)	`return`	1	Return void from procedure.	NF	8	
178 (0xb2)	`getstatic`	3	Get static field value.	NF	Trap	
179 (0xb3)	`putstatic`	3	Set static field in class.	NF	Trap	
180 (0xb4)	`getfield`	3	Get field value.	NF	Trap	
181 (0xb5)	`putfield`	3	Set field in class.	NF	Trap	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
182 (0xb6)	`invokevirtual`	3	Call method based on object.	NF	Trap	
183 (0xb7)	`invokespecial`	3	Call method *not* based on object.	NF	Trap	
184 (0xb8)	`invokestatic`	3	Call a static method.	NF	Trap	
185 (0xb9)	`invokeinterface`	5	Call an interface method.	NF	Trap	
186 (0xba)	Undefined					
187 (0xbb)	`new`	3	Create new object.	NF	Trap	
188 (0xbc)	`newarray`	2	Allocate new array.	NF	Trap	
189 (0xbd)	`anewarray`	3	Allocate new array of objects.	NF	Trap	
190 (0xbe)	`arraylength`	1	Get length of array.	BG1	$1^1/3^2$	Yes
191 (0xbf)	`athrow`	1	Throw an exception.	NF	Trap	
192 (0xc0)	`checkcast`	3	Check if object is of given type.	NF	Trap	
193 (0xc1)	`instanceof`	3	Determine if object is of given type.	NF	Trap	
194 (0xc2)	`monitorenter`	1	Enter a monitored region of code.	NF	3^7	
195 (0xc3)	`monitorexit`	1	Exit a monitored region of code.	NF	2^7	
196 (0xc4)	`wide`	4/6	Extend local variable index by additional bytes.	NF	Trap	
197 (0xc5)	`multianewarray`	4	Allocate new multidimensional array.	NF	Trap	
198 (0xc6)	`ifnull`	3	Test if null.	BG1	$1^4/4^5$	
199 (0xc7)	`ifnonnull`	3	Test if not null.	BG1	$1^4/4^5$	
200 (0xc8)	`goto_w`	5	Branch always (wide index).	NF	4	
201 (0xc9)	`jsr_w`	5	Jump subroutine, 4-byte offset.	NF	4	
202 (0xca)	`breakpoint`	1	Call breakpoint handler.	NF	Trap	
203 (0xcb)	`ldc_quick`	2	Push item from constant pool.	NF	1	Yes
204 (0xcc)	`ldc_w_quick`	3	Push item from constant pool.	NF	1	Yes

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
205 (0xcd)	`ldc2_w_quick`	3	Push long or double from constant pool.	NF	2	Yes
206 (0xce)	`getfield_quick`	3	Get field value from constant pool.	BG1	$1^1/4^2$	Yes
207 (0xcf)	`putfield_quick`	3	Set field in object.	BG2	$1^1/4^2$	
208 (0xd0)	`getfield2_quick`	3	Get long or double from field of object.	BG1	$2^1/5^2$	Yes
209 (0xd1)	`putfield2_quick`	3	Set field of object (long or double).	NF	$2^1/5^2$	
210 (0xd2)	`getstatic_quick`	3	Get static field from class.	NF	3	Yes
211 (0xd3)	`putstatic_quick`	3	Set static field in class.	BG1	3	
212 (0xd4)	`getstatic2_quick`	3	Get static field from class—long or double.	NF	4	Yes
213 (0xd5)	`putstatic2_quick`	3	Set long or double static field in class.	NF	4	
214 (0xd6)	`invokevirtual_quick`	3	Invoke instance method.	NF	15	
215 (0xd7)	`invokenonvirtual_quick`	3	Invoke instance method.	NF	13	
216 (0xd8)	`invokesuper_quick`	3	Invoke instance method.	NF	21	
217 (0xd9)	`invokestatic_quick`	3	Invoke static method.	NF	11	
218 (0xda)	`invokeinterface_quick`	3	Invoke interface method.	NF	Trap	
219 (0xdb)	Undefined					
220 (0xdc)	`aastore_quick`	1	Store into object reference array with no type checks.	BG2	$6^1/8^2$	
221 (0xdd)	`new_quick`	3	Create new object.	NF	Trap	
222 (0xde)	`anewarray_quick`	3	Create a new array of objects.	NF	Trap	
223 (0xdf)	`multianewarray_quick`	3	Create a new multidimensional array of objects.	NF	Trap	
224 (0xe0)	`checkcast_quick`	3	Check whether object is of given type.	NF	6^8	
225 (0xe1)	`instanceof_quick`	3	Determine if object is of given type.	NF	7^8	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
226 (0xe2)	`invokevirtual_quick_w`	3	Invoke instance method, dispatching on class.	NF	19	
227 (0xe3)	`getfield_quick_w`	3	Fetch field from object (wide index).	NF	Trap	
228 (0xe4)	`putfield_quick_w`	3	Set field in object (wide index).	NF	Trap	
229 (0xe5)	`nonnull_quick`	1	Pop object reference and trap if null.	BG1	1	
230 (0xe6)	`agetfield_quick`	3	Read reference field in object.	BG1	$1^1/4^2$	Yes
231 (0xe7)	`aputfield_quick`	3	Set reference field in object with GC checks.	BG2	$1^1/4^2$	
232 (0xe8)	`agetstatic_quick`	3	Read static reference field in class.	NF	3	Yes
233 (0xe9)	`aputstatic_quick`	3	Set static reference field in class with GC checks.	BG1	3	
234 (0xea)	`aldc_quick`	2	Push reference from constant pool.	NF	1	Yes
235 (0xeb)	`aldc_w_quick`	3	Push reference from constant pool.	NF	1	Yes
236 (0xec)	`exit_sync_method`	1	Jump to return code for synchronized method.	NF	6	
237 (0xed)	`sethi`	3	Set upper 16 bits of topmost entry of stack.	BG1	1	
238 (oxee)	`load_word_index`	3	Load word, using indexed addressing.	NF	1	Yes
239 (0xef)	`load_short_index`	3	Load short, using indexed addressing.	NF	1	Yes
240 (0xf0)	`load_char_index`	3	Load character, using indexed addressing.	NF	1	Yes
241 (0xf1)	`load_byte_index`	3	Load byte, using indexed addressing.	NF	1	Yes
242 (0xf2)	`load_ubyte_index`	3	Load unsigned byte, using indexed addressing.	NF	1	Yes
243 (0xf3)	`store_word_index`	3	Store word, using indexed addressing.	NF	1	

TABLE A-1 picoJava-II 1-Byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
244 (0xf4)	`nastore_word_index`	3	Nonallocating store, using indexed addressing.	NF	1	
245 (0xf5)	`store_short_index`	3	Store short, using indexed addressing.	NF	1	
246 (0xf6)	`store_byte_index`	3	Store byte, using indexed addressing.	NF	1	
247 (0xf7) to 254 (0xfe)	Undefined					

For a key to the footnotes, see page 458.

TABLE A-2 lists and describes 2-byte opcodes in the picoJava-II core.

TABLE A-2 picoJava-II 2-byte Opcodes

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
255 (0xff)	The first opcode byte denotes the extended opcode; the second opcode denotes the instruction.					
0 (0x00)	`load_ubyte`	2	Load unsigned byte from memory.	BG1	1	Yes
1 (0x01)	`load_byte`	2	Load signed byte from memory.	BG1	1	Yes
2 (0x02)	`load_char`	2	Load unsigned short or character from memory.	BG1	1	Yes
3 (0x03)	`load_short`	2	Load signed short from memory.	BG1	1	Yes
4 (0x04)	`load_word`	2	Load integer from memory.	BG1	1	Yes
5 (0x05)	`priv_ret_from_trap`	2	Return from trap in privileged mode.	NF	8	
6 (0x06)	`priv_read_dcache_tag`	2	Diagnostic read data cache tags in privileged mode.	NF	1	Yes
7 (0x07)	`priv_read_dcache_data`	2	Diagnostic read data cache data array in privileged mode.	NF	1	Yes
8 (0x08)	Undefined					

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
9 (0x09)	Undefined					
10 (0x0a)	`load_char_oe`	2	Load unsigned short or character from memory; endian swap.	BG1	1	Yes
11 (0x0b)	`load_short_oe`	2	Load signed short from memory; endian swap.	BG1	1	Yes
12 (0x0c)	`load_word_oe`	2	Load integer from memory; endian-swap.	BG1	1	Yes
13 (0x0d)	`return0`	2	Return with no value from routine entered via call.	BG2	6	
14 (0x0e)	`priv_read_icache_tag`	2	Diagnostic read instruction cache tags in privileged mode.	NF	2	Yes
15 (0x0f)	`priv_read_icache_data`	2	Diagnostic read instruction cache data array in privileged mode.	NF	2	Yes
16 (0x10)	`ncload_ubyte`	2	Load unsigned byte from memory; noncacheable.	BG1	1	Yes
17 (0x11)	`ncload_byte`	2	Load signed byte from memory; noncacheable.	BG1	1	Yes
18 (0x12)	`ncload_char`	2	Load unsigned short or character from memory; noncacheable.	BG1	1	Yes
19 (0x13)	`ncload_short`	2	Load signed short from memory; noncacheable.	BG1	1	Yes
20 (0x14)	`ncload_word`	2	Load integer from memory; noncacheable.	BG1	1	Yes
21(0x15)	`iucmp`	2	Compare unsigned integers.	OP	1	
22 (0x16)	`priv_powerdown`	2	Enter low-power standby state in privileged mode.	NF	1	
23 (0x17)	`cache_invalidate`	2	Invalidate cache line if present in cache.	BG1	1	
24 (0x18)	Undefined					
25 (0x19)	Undefined					

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
26 (0x1a)	`ncload_char_oe`	2	Load unsigned short or character from memory; noncacheable endian swap.	BG1	1	Yes
27 (0x1b)	`ncload_short_oe`	2	Load signed short from memory; noncacheable endian swap.	BG1	1	Yes
28 (0x1c)	`ncload_word_oe`	2	Load integer from memory; noncacheable endian swap.	BG1	1	Yes
29 (0x1d)	`return1`	2	Return with one-word value from subroutine entered via call.	BG2	7	
30 (0x1e)	`cache_flush`	2	Flush cache line and invalidate if present in cache.	BG1	1	
31 (0x1f)	`cache_index_flush`	2	Flush cache line and invalidate (no tag check).	BG1	1	
32 (0x20)	`store_byte`	2	Store byte to memory.	BG2	1	
33 (0x21)	Undefined					
34 (0x22)	`store_short`	2	Store short or character to memory.	BG2	1	
35 (x023)	Undefined					
36 (0x24)	`store_word`	2	Store integer to memory.	BG2	1	
37 (0x25)	`soft_trap`	2	Initiate a software trap.	NF	Trap	
38 (0x26)	`priv_write_dcache_tag`	2	Diagnostic write data cache tags in privileged mode.	NF	1	
39 (0x27)	`priv_write_dcache_data`	2	Diagnostic write data cache data array in privileged mode.	NF	1	
40 (0x28)	Undefined					
41 (0x29)	Undefined					
42 (0x2a)	`store_short_oe`	2	Store short to memory; endian swap.	BG2	1	
43 (0x2b)	Undefined					

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
44 (0x2c)	store_word_oe	2	Store integer to memory; endian swap.	BG2	1	
45 (0x2d)	return2	2	Return with two-word value from subroutine entered via call.	BG2	7	
46 (0x2e)	priv_write_icache_tag	2	Diagnostic write instruction cache tags in privileged mode.	NF	1	
47 (0x2f)	priv_write_icache_data	2	Diagnostic write instruction cache data array in privileged mode.	NF	1	
48 (0x30)	ncstore_byte	2	Store byte to memory; noncacheable.	BG2	1	
49 (0x31)	Undefined					
50 (0x32)	ncstore_short	2	Store short or character to memory; noncacheable.	BG2	1	
51 (0x33)	Undefined					
52 (0x34)	ncstore_word	2	Store integer to memory; noncacheable.	BG2	1	
53 (0x35)	Undefined					
54 (0x36)	priv_reset	2	Generate software-initiated reset in privileged mode.	NF	1	
55 (0x37)	get_current_class	2	Push class pointer for current method.	NF	3	
56 (0x38)	Undefined					
57 (0x39)	Undefined					
58 (0x3a)	ncstore_short_oe	2	Store short to memory; noncacheable endian swap.	BG2	1	
59 (0x3b)	Undefined					
60 (0x3c)	ncstore_word_oe	2	Store integer to memory; noncacheable endian swap.	BG2	1	
61 (0x3d)	call	2	Call subroutine with specified number of arguments.	BG2	6	

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
62 (0x3e)	zero_line	2	Set cache line to valid, dirty and zero the data.	BG1	5	
63 (0x3f)	priv_update_optop	2	Atomic update of OPTOP and OPLIM registers in privileged mode.	NF	2	
64 (0x40)	read_pc	2	Read current PC register.	NF	1	
65 (0x41)	read_vars	2	Read current VARS register.	NF	1	
66 (0x42)	read_frame	2	Read current FRAME register.	NF	1	
67 (0x43)	read_optop	2	Read current OPTOP register.	NF	1	
68 (0x44)	priv_read_oplim	2	Read current OPLIM register in privileged mode.	NF	1	
69 (0x45)	read_const_pool	2	Read current CONST_POOL register.	NF	1	
70 (0x46)	priv_read_psr	2	Read current PSR register in privileged mode.	NF	1	
71 (0x47)	priv_read_trapbase	2	Read current TRAPBASE register in privileged mode.	NF	1	
72 (0x48)	priv_read_lockcount0	2	Read current LOCKCOUNT0 register in privileged mode.	NF	1	
73 (0x49)	priv_read_lockcount1	2	Read current LOCKCOUNT1 register in privileged mode.	NF	1	
74 (0x4a)	Undefined					
75 (0x4b)	Undefined					
76 (0x4c)	priv_read_lockaddr0	2	Read current LOCKADDR0 register in privileged mode.	NF	1	
77 (0x4d)	priv_read_lockaddr1	2	Read current LOCKADDR1 register in privileged mode.	NF	1	
78 (0x4e)	Undefined					
79 (0x4f)	Undefined					
80 (0x50)	priv_read_userrange1	2	Read current USERRANGE1 register in privileged mode.	NF	1	
81 (0x51)	priv_read_gc_config	2	Read current GC_CONFIG register in privileged mode.	NF	1	

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
82 (0x52)	priv_read_brk1a	2	Read current BRK1A register in privileged mode.	NF	1	
83 (0x53)	priv_read_brk2a	2	Read current BRK2A register in privileged mode.	NF	1	
84 (0x54)	priv_read_brk12c	2	Read current BRK12C register in privileged mode.	NF	1	
85 (0x55)	priv_read_userrange2	2	Read current USERRANGE2 register in privileged mode.	NF	1	
86 (0x56)	Undefined					
87 (0x57)	priv_read_versionid	2	Read current VERSIONID register in privileged mode.	NF	1	
88 (0x58)	priv_read_hcr	2	Read current HCR register in privileged mode.	NF	1	
89 (0x59)	priv_read_sc_bottom	2	Read current SC_BOTTOM register in privileged mode.	NF	1	
90 (0x5a)	read_global0	2	Read current GLOBAL0 register.	LV	1	
91 (0x5b)	read_global1	2	Read current GLOBAL1 register.	LV	1	
92 (0x5c)	read_global2	2	Read current GLOBAL2 register.	LV	1	
93 (0x5d)	read_global3	2	Read current GLOBAL3 register.	LV	1	
94 (0x5e)	Undefined					
95 (0x5f)	Undefined					
96 (0x60)	write_PC, ret_from_sub	2	Write to PC register.	NF	4	
97 (0x61)	write_vars	2	Write to VARS register.	NF	2	
98 (0x62)	write_frame	2	Write to FRAME register.	NF	2	
99 (0x63)	write_optop	2	Write to OPTOP register.	NF	2	
100 (0x64)	priv_write_oplim	2	Write to OPLIM register in privileged mode.	NF	2	
101 (0x65)	write_const_pool	2	Write to CONST_POOL register.	NF	2	

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
102 (0x66)	priv_write_psr	2	Write to PSR register in privileged mode.	NF	2	
103 (0x67)	priv_write_trapbase	2	Write to TRAPBASE register in privileged mode.	NF	2	
104 (0x68)	priv_write_lockcount0	2	Write to LOCKCOUNT0 register in privileged mode.	NF	2	
105 (0x69)	priv_write_lockcount1	2	Write to LOCKCOUNT1 register in privileged mode.	NF	2	
106 (0x6a)	Undefined					
107 (0x6b)	Undefined					
108 (0x6c)	priv_write_lockaddr0	2	Write to LOCKADDR0 register in privileged mode.	NF	2	
109 (0x6d)	priv_write_lockaddr1	2	Write to LOCKADDR1 register in privileged mode.	NF	2	
110 (0x6e)	Undefined					
111 (0x6f)	Undefined					
112 (0x70)	priv_write_userrange1	2	Write to USERRANGE1 register in privileged mode.	NF	2	
113 (0x71)	priv_write_gc_config	2	Write to GC_CONFIG register in privileged mode.	NF	2	
114 (0x72)	priv_write_brk1a	2	Write to BRK1A register in privileged mode.	NF	2	
115 (0x73)	priv_write_brk2a	2	Write to BRK2A register in privileged mode.	NF	2	
116 (0x74)	priv_write_brk12c	2	Write to BRK12C register in privileged mode.	NF	2	
117 (0x75)	priv_write_userrange2	2	Write to USERRANGE2 register in privileged mode.	NF	2	
118 (0x76)	Undefined					
119 (0x77)	Undefined					
120 (0x78)	Undefined					
121 (0x79)	priv_write_sc_bottom	2	Write to SC_BOTTOM register in privileged mode.	NF	2	

TABLE A-2 picoJava-II 2-byte Opcodes *(Continued)*

Opcode	Mnemonic	Size	Description	Group	Cycles	LDUSE
122 (0x7a)	`write_global0`	2	Write to current `GLOBAL0` register.	MEM	1	
123 (0x7b)	`write_global1`	2	Write to current `GLOBAL1` register.	MEM	1	
124 (0x7c)	`write_global2`	2	Write to current `GLOBAL2` register.	MEM	1	
125 (0x7d)	`write_global3`	2	Write to current `GLOBAL3` register.	MEM	1	
126-255	Undefined					

Index

NUMERICS

64-bit values
  endianness, order of loads, 25
  placement, 62

A

aaload, 82, 460
aastore
  instruction, 83, 436
  opcode, 461
  trap, 52
aastore_quick, 84, 436, 467
ACE field in the PSR register, 11, 27
aconst_null, 86, 458
actions, subsequent to the core taking a trap or an interrupt, 47
address
  alignment, requirements for, 22
  larger than 30-bit, handling of, 22
  noncacheable, 23
  opposite endianness, bit 30 set, 24
  space, 21, 22
Address of Deepest Stack Cache Entry Register (SC_BOTTOM), 8
AEM field in the PSR register, 11
agetfield_quick, 87, 468
agetstatic_quick, 74, 88, 468
aldc_quick, 74, 89, 468
aldc_w_quick, 74, 90, 468
alignment address, requirements for, 22
allocation of
  frames, 385, 389
  stack chunks, 429
aload, 91, 459
aload_0, 460
aload_1, 460
aload_2, 460
aload_3, 460
aload_*n*, 92
anewarray
  instruction, 93
  opcode, 466
  trap, 51
anewarray_quick
  instruction, 94
  opcode, 467
  trap, 51
aputfield_quick, 95, 436, 468
aputstatic_quick, 74, 97, 436, 468
areturn, 43, 99, 465
ArithmeticException trap, 52, 56
array
  arrays structure, 68
  booleans structure, 70
  bytes structure, 70
  chars structure, 69
  data structures, 67
  doubles structure, 67
  floats structure, 69
  integers structure, 68
  longs structure, 67
  objects structure, 68
  reference, handle (H) bit, 66

shorts structure, 70
storage, 65
ArrayIndexOutOfBounds trap, 52, 56
arraylength, 100, 466
astore, 101, 460
astore_0, 461
astore_1, 461
astore_2, 461
astore_3, 461
astore_*n*, 102
asynchronous_error trap, 27, 50, 54
athrow
instruction, 103
opcode, 466
trap, 52, 393

B

baload, 104, 460
bastore, 105, 461
bipush, 31, 106, 459
BM8 field in the PSR register, 11
breakpoint
address matching, 454
halting the core, 454
opcode, 466
registers, 15, 451
setup, 450
trap, 52
breakpoint1 trap, 50, 55
breakpoint2 trap, 50, 55
BRK12C, 445
BRK12C register, 15
BRK12C.BRKM1 bit, 454
BRK12C.BRKM2 bit, 454
BRK1A register, 15
BRK2A register, 15
BRKEN1 field in the BRK12C register, 17
BRKEN2 field in the BRK12C register, 16
BRKM1 field in the BRK12C register, 16
BRKM2 field in the BRK12C register, 16
byte ordering, *See* endianness

C

C code generation
argument mismatches, 425
function
calls, 404, 420
prologue and epilog, 405
return values, 405
functions
with aggregate params and locals, 408
with simple params and locals, 406
object file formats, 426
optimizations, 421
passing of arguments, 404
registers, 402
runtime stacks, 402
CAC field in the PSR register, 11, 27
cache coherency, 30, 33
cache_flush, 31, 34, 35, 37, 107, 471
cache_index_flush, 31, 34, 36, 37, 109, 471
cache_invalidate, 31, 34, 37, 111, 470
call, 113, 472
caload, 114, 460
CAR_MASK field in the GC_CONFIG register, 14, 440
castore, 115, 461
checkcast
instruction, 116
opcode, 466
trap, 51
checkcast_quick
instruction, 71, 74, 117
opcode, 467
trap, 51
checks, memory protection, 27
class loader, 391
CO field in the LOCKCOUNT registers, 13, 396, 398, 399
code for nonstatic and static methods, 392
coherency
between caches and memory, 30 to 33
in accesses
instruction, 32
stack and data, 30
Constant Pool Base Pointer Register (CONST_POOL), 9, 385, 387
constant pool table, 72
context switch, 44, 58, 400
COUNT field in the LOCKCOUNT registers, 13
cp, *See* constant pool table
creation of method frames, 385

D

d2f
  instruction, 118
  opcode, 464
  trap, 51
d2i
  instruction, 119
  opcode, 464
  trap, 51
d2l
  instruction, 120
  opcode, 464
  trap, 51
dadd
  instruction, 121
  opcode, 462
  trap, 50
daload, 123, 460
dastore, 124, 461
data
  breakpoint traps, 450
  cache
    configuration, 36
    function, 22
    initialization, 37
    operations, 37
  structures, array, 67
  types
    floating point, 62
    integral, 62
    primitive, 61 to 62
data_access_error trap, 50
data_access_io_error trap, 27
data_access_mem_error trap, 27, 56
data_store_error trap, 50
DBH and DBL fields in the PSR register, 11, 40, 41
DCA field in the HCR register, 18, 36
DCE field in the PSR register, 11, 53, 54
DCL field in the HCR register, 18, 36
dcmpg
  instruction, 125
  opcode, 464
  trap, 51
dcmpl
  instruction, 126
  opcode, 464
  trap, 51
dconst_0, 127, 458
dconst_1, 128, 459
DCS field in the HCR register, 19, 36
ddiv
  instruction, 129
  opcode, 462
  trap, 50
debugging by halting and single-stepping, 454
diagnostic accesses
  in the data cache, 37
  in the instruction cache, 34
dload, 131, 459
dload_0, 459
dload_1, 460
dload_2, 460
dload_3, 460
dload_*n*, 132
dmul
  instruction, 133
  opcode, 462
  trap, 50
dneg, 134, 463
DRE field in the PSR register, 11, 39
drem
  instruction, 135
  opcode, 463
  trap, 50
dreturn, 43, 137, 465
DRT field in the PSR register, 11, 53, 54
dstore, 138, 460
dstore_0, 461
dstore_1, 461
dstore_2, 461
dstore_3, 461
dstore_*n*, 139
dsub
  instruction, 140
  opcode, 462
  trap, 50
dup, 142, 461
dup_x1, 143, 462
dup_x2, 144, 462
dup2, 145, 462
dup2_x1, 146, 462
dup2_x2, 147, 462

E

emulation of instructions, 54
ENABLE field in the OPLIM register, 55

endianness
64-bit values, order of loads, 25
big-endian, 23, 24
little-endian, 23, 24
of stack, 24
exceptions, 54
exit_sync_method, 148, 391, 392, 468

F

f2d
instruction, 149
opcode, 464
trap, 51
f2i
instruction, 150
opcode, 463
trap, 50
f2l
instruction, 151
opcode, 463
trap, 51
fadd
instruction, 152
opcode, 462
trap, 50
faload, 154, 460
fastore, 155, 461
fcmpg
instruction, 156
opcode, 464
trap, 51
fcmpl
instruction, 157
opcode, 464
trap, 51
fconst_0, 158, 458
fconst_1, 159, 458
fconst_2, 160, 458
fdiv
instruction, 161
opcode, 462
trap, 50
FLE field in the PSR register, 11
fload, 163, 459
fload_0, 459
fload_1, 459
fload_2, 459
fload_3, 459
fload_*n*, 164
flushing
in the data cache, 37
in the instruction cache, 34
in the stack cache, 44
fmul
instruction, 165
opcode, 462
trap, 50
fneg, 166, 463
FPE field in the PSR register, 11, 53, 54
FPP field in the HCR register, 19, 54
frame allocation on Java method invocations and returns, 385
Frame Pointer Register (FRAME), 7, 47, 385, 387, 389
frem
instruction, 167
opcode, 462
trap, 50
freturn, 43, 169, 465
fstore, 170, 460
fstore_0, 461
fstore_1, 461
fstore_2, 461
fstore_3, 461
fstore_*n*, 171
fsub
instruction, 172
opcode, 462
trap, 50

G

garbage collection
configuration register, 14
definition, 433
hardware support from the core
handles, 434
reserved bits in references and headers, 434
write barriers, 434
examples, 439
instructions, 436
page-based, 436
reference-based, 439
references, 441
train algorithm, 440
GC_CONFIG register, 14, 435, 440, 445

GC_CONFIG.CAR_MASK, 437
GC_CONFIG.REGION_MASK, 436
gc_notify trap, 53, 56, 434, 435, 441
GC_TAG, 63, 435, 441
GCE field in the PSR register, 11
get_current_class, 174, 392, 472
getfield
  instruction, 175
  opcode, 465
  trap, 51
getfield_quick, 176, 467
getfield_quick_w
  instruction, 177
  opcode, 468
  trap, 52
getfield2_quick, 178, 467
getstatic
  instruction, 179
  opcode, 465
  trap, 51
getstatic_quick, 74, 180, 467
getstatic2_quick, 74, 181, 467
global registers (GLOBAL), 20
goto, 182, 465
goto_w, 183, 466

H

HALT field in the BRK12C register, 16
Hardware Configuration Register (HCR), 18, 33, 36, 445
  DCA field, 36
  DCL field, 36
  DCS field, 36
  ICA field, 33
  ICL field, 33
  ICS field, 33
hardware synchronization, 396
high and low watermarks, 41

I

i2b, 184, 464
i2c, 185, 464
i2d
  instruction, 186
  opcode, 463
  trap, 50
i2f
  instruction, 187
  opcode, 463
  trap, 50
i2l, 188, 463
i2s, 189, 464
iadd, 190, 462
iaload, 191, 460
iand, 192, 463
iastore, 193, 461
ICA field in the HCR register, 19, 33
ICE field in the PSR register, 11, 34
ICL field in the HCR register, 19, 33
iconst_0, 195, 458
iconst_1, 196, 458
iconst_2, 197, 458
iconst_3, 198, 458
iconst_4, 199, 458
iconst_5, 200, 458
iconst_m1, 194, 458
ICS field in the HCR register, 19, 33
idiv, 201, 462
IE field in the PSR register, 11, 47, 57
if_acmpeq, 63, 202, 465
if_acmpne, 63, 203, 465
if_icmpeq, 204, 464
if_icmpge, 205, 465
if_icmpgt, 206, 465
if_icmple, 207, 465
if_icmplt, 208, 464
if_icmpne, 209, 464
ifeq, 210, 464
ifge, 211, 464
ifgt, 212, 464
ifle, 213, 464
iflt, 214, 464
ifne, 215, 464
ifnonnull, 216, 466
ifnull, 217, 466
iinc, 218, 463
illegal_instruction trap, 50, 56
IllegalMonitorStateException, 397, 399
iload, 31, 219, 459
iload_0, 459
iload_1, 459

iload_2, 459
iload_3, 459
iload_*n*, 220
imul, 221, 462
incoming arguments to invoked methods, 386
ineg, 222, 463
instanceof
  instruction, 223
  opcode, 466
  trap, 51
instanceof_quick
  instruction, 71, 74, 224
  opcode, 467
  trap, 51
instanceof_quick, 467
instruction
  breakpoint traps, 450
  cache
    configuration, 33
    function, 22
    initialization, 34
    operations, 34
  emulation, 54
  set, 82 to 381
  space, modification of, 35
instruction_access_error trap, 27, 50, 55
instructions for caching, 31
interrupt
  control, 57
  definitions, 56
  latency of, 57
  maskable, 56
  nonmaskable (NMI), 56
interrupt request level (IRL), 57
Interrupt_level_1, 53
Interrupt_level_10, 53
Interrupt_level_11, 53
Interrupt_level_12, 53
Interrupt_level_13, 53
Interrupt_level_14, 53
Interrupt_level_15, 53
Interrupt_level_2, 53
Interrupt_level_3, 53
Interrupt_level_4, 53
Interrupt_level_5, 53
Interrupt_level_6, 53
Interrupt_level_7, 53
Interrupt_level_8, 53
Interrupt_level_9, 53
invokeinterface
  instruction, 226
  opcode, 466
  trap, 52
invokeinterface_quick
  instruction, 227
  opcode, 467
  trap, 52
invokenonvirtual_quick, 74, 228, 467
invokespecial
  instruction, 230
  opcode, 466
  trap, 52
invokestatic
  instruction, 231
  opcode, 466
  trap, 52
invokestatic_quick, 74, 232, 467
invokesuper_quick, 233, 467
invokevirtual
  instruction, 235
  opcode, 466
  trap, 52
invokevirtual_quick, 236, 467
invokevirtual_quick_w, 74, 237, 468
invoking
  methods, 388
  synchronized methods, 391
ior, 239, 463
irem, 240, 462
ireturn, 43, 241, 465
IRL, *See* interrupt request level
ishl, 242, 463
ishr, 243, 463
istore, 31, 244, 460
istore_0, 460
istore_1, 460
istore_2, 460
istore_3, 460
istore_*n*, 245
isub, 246, 462
iucmp, 247, 470
iushr, 248, 463
ixor, 249, 463

J

Java virtual machine instructions, *See* instruction set
jsr, 250, 465
jsr_w, 251, 466

L

l2d
  instruction, 252
  opcode, 463
  trap, 50
l2f
  instruction, 253
  opcode, 463
  trap, 50
l2i, 254, 463
ladd, 255, 462
laload, 256, 460
land, 257, 463
lastore, 258, 461
latency of interrupts, 57
lcmp, 259, 464
lconst_0, 44, 260, 458
lconst_1, 261, 458
ldc
  instruction, 262
  opcode, 459
  trap, 51
ldc_quick, 74, 263, 466
ldc_w
  instruction, 264
  opcode, 459
  trap, 51
ldc_w_quick, 74, 265, 466
ldc2_w
  instruction, 266
  opcode, 459
  trap, 51
ldc2_w_quick, 74, 267, 467
ldiv
  instruction, 268
  opcode, 462
  trap, 51
lload, 269, 459
lload_0, 459
lload_1, 459
lload_2, 459
lload_3, 459
lload_*n*, 270
lmul
  instruction, 271
  opcode, 462
  trap, 51
lneg, 272, 463
load_byte, 273, 469
load_byte_index, 274, 468
load_char, 31, 275, 469
load_char_index, 31, 276, 468
load_char_oe, 31, 277, 470
load_short, 31, 278, 469
load_short_index, 31, 279, 468
load_short_oe, 31, 280, 470
load_ubyte, 281, 469
load_ubyte_index, 282, 468
load_word, 31, 39, 283, 469
load_word_index, 31, 284, 468
load_word_oe, 285, 470
Local Variable Pointer Register (VARS), 6
local variables, storage of, 387
lock caching registers, 13
LOCKADDR registers, 13, 396, 397, 399
LOCKCOUNT registers, 13, 396, 397, 399
LOCKCOUNT.COUNT, 397, 399, 400
LockCounterOverflow trap, 56
LockCountOverflow
  handler, 397
  trap, 52
LockEnterMiss
  handler, 397
  trap, 52, 56
LockExitMiss
  handler, 399
  trap, 53, 56
LockRelease
  handler, 398
  trap, 52, 56
LOCKWANT field in the LOCKCOUNT registers, 13, 396, 399
lookupswitch
  instruction, 22, 286
  opcode, 465
  trap, 52
lor, 287, 463
lrem
  instruction, 288
  opcode, 462
  trap, 51

lreturn, 43, 289, 465
lshl, 290, 463
lshr, 291, 463
lstore, 292, 460
lstore_0, 460
lstore_1, 461
lstore_2, 461
lstore_3, 461
lstore_*n*, 293
lsub, 294, 462
lushr, 295, 463
lxor, 296, 463

M

machine states, 444
management of power, 443
maskable interrupt, 56
mem_address_not_aligned trap, 22, 50, 55
mem_protection_error trap, 29, 50, 55
memory
  access types, 21
  errors, 26
  noncacheable region, 23
  protection
    checks, 27
    regions, 28
    registers, 10
Memory Protection Registers (USERRANGE1 and USERRANGE2), 10
method
  context
    information in frames, 385, 387
    saving, 391
  context, five words of, 391
  entry point in the method structure, 391
  frames, creating, 385
  invoking, 388
  passing control to, 391
  reference, resolving, 388
  return, 393
  structure accessing, 388
  synchronized, invoking, 391
methods
  in class files, indexed mapping, 73
  in superclasses, overriding, 71
Minimum Value of Top-of-Stack Register (OPLIM), 8
modifying instruction space, 35
monitor
  caching registers, 13
  definition, 395
  handling, software support for, 397
  software support, 397
  structures, 396
  updates by context switch, 400
monitorenter, 297, 391, 395, 396, 466
monitorexit, 299, 391, 395, 396, 466
multianewarray
  instruction, 301
  opcode, 466
  trap, 51
multianewarray_quick
  instruction, 302
  opcode, 467
  trap, 52

N

nastore_word_index, 31, 38, 303, 469
ncload_byte, 304, 470
ncload_char, 305, 470
ncload_char_oe, 31, 306, 471
ncload_short, 31, 307, 470
ncload_short_oe, 31, 308, 471
ncload_ubyte, 309, 470
ncload_word, 31, 310, 470
ncload_word_oe, 311, 471
ncstore_byte, 312, 472
ncstore_char, 31
ncstore_short, 31, 313, 472
ncstore_short_oe, 31, 314, 472
ncstore_word, 31, 315, 472
ncstore_word_oe, 31, 316, 472
new
  instruction, 317
  opcode, 466
  trap, 51
new_quick
  instruction, 318
  opcode, 467
  trap, 51
newarray
  instruction, 319
  opcode, 466
  trap, 51

nmi, 53
NMI, *See* nonmaskable interrupt
noncacheable
  instructions, 23
  memory region, 23
nonmaskable interrupt (NMI), 56
nonnull_quick, 320, 468
nop, 321, 458
NullPointer trap, 52, 56

O

object
  reference
    definition, 63
    handle (H) bit, 64
  storage, 64
obtaining method information, 388
opcodes
  Java virtual machine, 458 to 466
  picoJava-II core
    1-byte, 458 to 469
    2-byte, 469 to 476
operand stack, 387
OPLIM register, 8, 29, 429, 445
oplim_trap trap, 29, 50, 55, 429, 430
opposite endianness, 24
OPTOP register, 7, 29, 389, 393, 429, 445

P

picoJava-II core
  overview, 3
  relationship to the Java virtual machine, 4
PIL field in the PSR register, 11, 57
placement of 64-bit values, 62
pop, 322, 461
pop2, 323, 461
power modes, 443
power-on reset (POR), 444
prev_ret_from_trap, 44
priv_powerdown, 324, 443, 470
priv_read_brk12C, 474
priv_read_brk1A, 474
priv_read_brk2A, 474
priv_read_dcache_data, 325, 469
priv_read_dcache_tag, 327, 469
priv_read_gc_config, 473
priv_read_hcr, 474
priv_read_icache_data, 34, 329, 470
priv_read_icache_tag, 34, 331, 470
priv_read_lockaddr0, 473
priv_read_lockaddr1, 473
priv_read_lockcount0, 473
priv_read_lockcount1, 473
priv_read_oplim, 473
priv_read_psr, 473
priv_read_*reg*, 333
priv_read_sc_bottom, 474
priv_read_trapbase, 473
priv_read_userrange1, 473
priv_read_userrange2, 474
priv_read_versionid, 474
priv_reset, 335, 444, 472
priv_ret_from_trap, 43, 48, 49, 336, 469
priv_update_optop, 43, 44, 337, 473
priv_write_brk12C, 475
priv_write_brk1A, 475
priv_write_brk2A, 475
priv_write_dcache_data, 338, 471
priv_write_dcache_tag, 340, 471
priv_write_gc_config, 475
priv_write_icache_data, 34, 342, 472
priv_write_icache_tag, 34, 344, 472
priv_write_lockaddr0, 475
priv_write_lockaddr1, 475
priv_write_lockcount0, 475
priv_write_lockcount1, 475
priv_write_oplim, 474
priv_write_psr, 475
priv_write_*reg*, 346
priv_write_sc_bottom, 475
priv_write_trapbase, 475
priv_write_userrange1, 475
priv_write_userrange2, 475
privileged_instruction trap, 50, 55
Processor Status Register (PSR), 10, 47, 445
Program Counter Register (PC), 5, 47, 385, 387, 391, 444
PSR.GCE, 435, 436
putfield
  instruction, 348, 436
  opcode, 465
  trap, 51
putfield_quick, 349, 467

putfield_quick_w
instruction, 350
opcode, 468
trap, 52
putfield2_quick, 351, 467
putstatic
instruction, 352, 436
opcode, 465
trap, 51
putstatic_quick, 74, 353, 467
putstatic2_quick, 74, 354, 467

R

read_const_pool, 473
read_frame, 473
read_global0, 474
read_global1, 474
read_global2, 474
read_global3, 474
read_optop, 473
read_pc, 473
read_*reg*, 355
read_vars, 473
reference
bits, 63
format, 63
types, 62 to 70
REGION_MASK field in the GC_CONFIG register, 14
registers
breakpoint (BRK1A, BRK2A, and BRK12C), 15
constant pool base pointer (CONST_POOL), 9
deepest stack cache entry pointer (SC_BOTTOM), 8
frame pointer (FRAME), 7
garbage collection configuration (GC_CONFIG), 14
hardware configuration (HCR), 18
local variable pointer (VARS), 6
memory protection (USERRANGE1 and USERRANGE2), 10
monitor-caching, 12
processor status (PSR), 10
program counter (PC), 5
stack limit pointer (OPLIM), 8
stack management, 6
top-of-stack pointer (OPTOP), 7
trap handler address (TRAPBASE), 12
version ID (VERSIONID), 17
reset management, 40, 444
restrictions in trap handlers, 432
ret, 356, 465
ret_from_sub, 357, 474
return, 43, 358, 465
return0, 43, 359, 470
return1, 43, 360, 471
return2, 43, 361, 472
returning from a method, 393
runtime structures, 71 to 74
array header / runtime class structure, 71
class structure, 73
constant pool, 73
method structure, 72

S

saload, 362, 460
sastore, 363, 461
saving the invoker's method context, 391
SC_BOTTOM register, 8, 39, 40, 445
sethi, 364, 468
setjmp and longjmp functions in C, implementation, 432
sign extension, 25, 26
sipush, 365, 459
SIR values, 444
soft_trap
instruction, 366
opcode, 471
trap, 51
software support for monitors, 397
SRCBRK1 field in the BRK12C register, 17
SRCBRK2 field in the BRK12C register, 16
SRN field in the HCR register, 19
stack, 21
at oplim_trap, illustration, 430
cache
configuration, 39
dribbling, 22, 39, 41
entry requirements, 41
flushing contents to memory, 44
function, 22
hits, 41
spill and fill transactions, 42
cacheability, caveat, 23

chunks
allocation and deallocation, 429
returns to previously saved states, 432
how to grow, 430
how to limit, 29
overflows, 43
registers, initialization of, 40
states before entering oplim_trap, 430
thread states, caveat, 431
underflows, 43
values, 387
standby mode, 443
storage of
arrays, 65
local variables during method invocation, 387
objects, 64
store_byte, 367, 471
store_byte_index, 368, 469
store_short, 31, 369, 471
store_short_index, 31, 370, 469
store_short_oe, 31, 371, 471
store_word, 31, 372, 471
store_word_index, 31, 373, 468
store_word_oe, 31, 374, 472
SU field in the PSR register, 11, 27, 47
SUBRK1 field in the BRK12C register, 17
SUBRK2 field in the BRK12C register, 16
swap, 375, 462
switch of context, 58
synchronization of hardware, 396
synchronized methods, invoking, 391

T

tableswitch, 22, 376, 465
TBA field in the TRAPBASE register, 12, 46
Top-of-Stack Pointer Register (OPTOP), 7
trap
definitions, 45
levels, 46
priorities, 50
table, 46
types, 45, 50
Trap Handler Address Register (TRAPBASE), 12, 46, 445
TT field in the TRAPBASE register, 12, 47

U

unimplemented_instr_0xba trap, 52
unimplemented_instr_0xdb trap, 52
unimplemented_instr_0xf7 trap, 52
unimplemented_instr_0xf8 trap, 52
unimplemented_instr_0xf9 trap, 52
unimplemented_instr_0xfa trap, 52
unimplemented_instr_0xfb trap, 52
unimplemented_instr_0xfc trap, 52
unimplemented_instr_0xfd trap, 52
unimplemented_instr_0xfe trap, 52
USERHIGH and USERLOW fields in the USERRANGE registers, 10, 28
USERRANGE1 register, 10
USERRANGE2 register, 10

V

value convention on the stack, 387
VARS register, 6, 47, 385, 387, 389, 393, 431, 445
Version ID register (VERSION ID), 17, 444

W

watermarks, *See* high and low watermarks
WB_VECTOR field in the GC_CONFIG register, 14, 441
wide
instruction, 378
opcode, 466
trap, 52
write_const_pool, 474
write_frame, 474
write_global0, 476
write_global1, 476
write_global2, 476
write_global3, 476
write_optop, 43, 44, 474
write_pc, 474
write_*reg*, 379
write_vars, 474

Z

zero_line
instruction, 31, 38, 54, 380
opcode, 473
trap, 38, 52

picoJava-II Opcodes

Opcode	Mnemonic	Size	Description
0 (0x0)	nop	1	-
1 (0x1)	aconst_null	1	Push null obj
2 (0x2)	iconst_m1	1	Push int const -1
3 (0x3)	iconst_0	1	Push int const 0
4 (0x4)	iconst_1	1	Push int const 1
5 (0x5)	iconst_2	1	Push int const 2
6 (0x7)	iconst_3	1	Push int const 3
7 (0x7)	iconst_4	1	Push int const 4
8 (0x8)	iconst_5	1	Push int const 5
9 (0x9)	lconst_0	1	Push long int const 00
10 (0x0a)	lconst_1	1	Push long int const 01
11 (0x0b)	fconst_0	1	Push float const 0.0
12 (0x0c)	fconst_1	1	Push float const 1.0
13 (0x0d)	fconst_2	1	Push float const 2.0
14 (0x0e)	dconst_0	1	Push dbl float 0.0
15 (0x0f)	dconst_1	1	Push dbl float 1.0
16 (0x10)	bipush	2	Push one byte int
17 (0x11)	sipush	3	Push two byte int
18 (0x12)	ldc	2	Load const from const pool
19 (0x13)	ldc_w	3	Load const from const pool (16-bit index)
20 (0x14)	ldc2_w	3	Load long / dbl from pool
21 (0x15)	iload	2	Load local int var
22 (0x16)	lload	2	Load local long var
23 (0x17)	fload	2	Load local float var
24 (0x18)	dload	2	Load local dbl float var
25 (0x19)	aload	2	Load local obj variable
26 (0x1a)	iload_0	1	Load local var 0
27 (0x1b)	iload_1	1	Load local var 1
28 (0x1c)	iload_2	1	Load local var 2
29 (0x1d)	iload_3	1	Load local var 3
30 (0x1e)	lload_0	1	Load local long var 0
31 (0x1f)	lload_1	1	Load local long var 1
32 (0x20)	lload_2	1	Load local long var 2
33 (0x21)	lload_3	1	Long local long var 3
34 (0x22)	fload_0	1	Load local float var 0
35 (0x23)	fload_1	1	Load local float var 1
36 (0x24)	fload_2	1	Load local float var 2
37 (0x25)	fload_3	1	Load local float var 3
38 (0x26)	dload_0	1	Load local dbl var 0
39 (0x27)	dload_1	1	Load local dbl var 1
40 (0x28)	dload_2	1	Load local dbl var 2
41 (0x29)	dload_3	1	Load local dbl var 3
42 (0x2a)	aload_0	1	Load local obj var 0
43 (0x2b)	aload_1	1	Load local obj var 1
44 (0x2c)	aload_2	1	Load local obj var 2
45 (0x2d)	aload_3	1	Load local obj var 3
46 (0x2e)	iaload	1	Load int from array
47 (0x2f)	laload	1	Load long from array
48 (0x30)	faload	1	Load float from array
49 (0x31)	daload	1	Load dbl from array
50 (0x32)	aaload	1	Load obj ref from array
51 (0x33)	baload	1	Load signed byte from array
52 (0x34)	caload	1	Load character from array
53 (0x35)	saload	1	Load short from arrray
54 (0x36)	istore	2	Store int into local var
55 (0x37)	lstore	2	Store long into local var
56 (0x38)	fstore	2	Store float into local var
57 (0x39)	dstore	2	Store dbl into local var
58 (0x3a)	astore	2	Store obj ref into local var
59 (0x3b)	istore_0	1	Store into local var 0
60 (0x3c)	istore_1	1	Store into local var 1
61 (0x3d)	istore_2	1	Store into local var 2
62 (0x3e)	istore_3	1	Store into local var 3
63 (0x3f)	lstore_0	1	Store into local var 0
64 (0x40)	lstore_1	1	Store into local var 1
65 (0x41)	lstore_2	1	Store into local var 2
66 (0x42)	lstore_3	1	Store into local var 3
67 (0x43)	fstore_0	1	Store into local var 0
68 (0x44)	fstore_1	1	Store into local var 1
69 (0x45)	fstore_2	1	Store into local var 2
70 (0x46)	fstore_3	1	Store into local var 3
71 (0x47)	dstore_0	1	Store into local var 0
72 (0x48)	dstore_1	1	Store into local var 1
73 (0x49)	dstore_2	1	Store into local var 2
74 (0x4a)	dstore_3	1	Store into local var 3
75 (0x4b)	astore_0	1	Store into local var 0
76 (0x4c)	astore_1	1	Store into local var 1
77 (0x4d)	astore_2	1	Store into local var 2
78 (0x4e)	astore_3	1	Store into local var 3
79 (0x4f)	iastore	1	Store into int array
80 (0x50)	lastore	1	Store into long array
81 (0x51)	fastore	1	Store into float array
82 (0x52)	dastore	1	Store into dbl float array
83 (0x53)	aastore	1	Store into obj ref array
84 (0x54)	bastore	1	Store into signed byte array
85 (0x55)	castore	1	Store into character array
86 (0x56)	sastore	1	Store into short array
87 (0x57)	pop	1	Pop top entry in stack
88 (0x58)	pop2	1	Pop top two entries in stack
89 (0x59)	dup	1	Dup top stack word
90 (0x5a)	dup_x1	1	Dup top word and put two words down
91 (0x5b)	dup_x2	1	Dup top word and put three words down
92 (0x5c)	dup2	1	Dup top two words
93 (0x5d)	dup2_x1	1	Dup top two words and put three words down
94 (0x5e)	dup2_x2	1	Dup top two words and put four words down
95 (0x5f)	swap	1	Swap top two stack words
96 (0x60)	iadd	1	Int add
97 (0x61)	ladd	1	Long add
98 (0x62)	fadd	1	Float add
99 (0x63)	dadd	1	Dbl float add
100 (0x64)	isub	1	Int subtract
101 (0x65)	lsub	1	Long subtract
102 (0x66)	fsub	1	Float subtract
103 (0x67)	dsub	1	Dbl float subtract
104 (0x68)	imul	1	Int multiply
105 (0x69)	lmul	1	Long multiply
106 (0x6a)	fmul	1	Float multiply
107 (0x6b)	dmul	1	Dbl float multiply
108 (0x6c)	idiv	1	Int divide
109 (0x6d)	ldiv	1	Long divide
110 (0x6e)	fdiv	1	Float divide
111 (0x6f)	ddiv	1	Dbl float divide
112 (0x70)	irem	1	Int remainder
113 (0x71)	lrem	1	Long remainder
114 (0x72)	frem	1	Float remainder
115 (0x73)	drem	1	Dbl float remainder
116 (0x74)	ineg	1	Int negate
117 (0x75)	lneg	1	Long negate
118 (0x76)	fneg	1	Float negate
119 (0x77)	dneg	1	Dbl float negate
120 (0x78)	ishl	1	Int shift left
121 (0x79)	lshl	1	Long shift left
122 (0x7a)	ishr	1	Int arithmetic shift right
123 (0x7b)	lshr	1	Long arithmetic shift right
124 (0x7c)	iushr	1	Int logical shift right
125 (0x7d)	lushr	1	Long logical shift right
126 (0x7e)	iand	1	Int boolean AND
127 (0x7f)	land	1	Long boolean AND
128 (0x80)	ior	1	Int boolean OR
129 (0x81)	lor	1	Long boolean OR
130 (0x82)	ixor	1	Int boolean XOR
131 (0x83)	lxor	1	Long boolean XOR
132 (0x84)	iinc	3	Increment local var by const
133 (0x85)	i2l	1	Int to long
134 (0x86)	i2f	1	Int to float
135 (0x87)	i2d	1	Int to dbl
136 (0x88)	l2i	1	Long to int
137 (0x89)	l2f	1	Long to float
138 (0x8a)	l2d	1	Long to dbl
139 (0x8b)	f2i	1	Float to int
140 (0x8c)	f2l	1	Float to long
141 (0x8d)	f2d	1	Float to dbl
142 (0x8e)	d2i	1	Dbl to int
143 (0x8f)	d2l	1	Dbl to long
144 (0x90)	d2f	1	Dbl to float
145 (0x91)	i2b	1	Int to byte
146 (0x92)	i2c	1	Int to character
147 (0x93)	i2s	1	Int to short
148 (0x94)	lcmp	1	Long int compare
149 (0x95)	fcmpl	1	Float compare -1 on incomp
150 (0x96)	fcmpg	1	Float compare 1 on incomp
151 (0x97)	dcmpl	1	Dbl comp -1 on incomp
152 (0x98)	dcmpg	1	Dbl compare 1 on incomp
153 (0x99)	ifeq	3	Branch if =0
154 (0x9a)	ifne	3	Branch if ≠ 0
155 (0x9b)	iflt	3	Branch if <0
156 (0x9c)	ifge	3	Branch if 0
157 (0x9d)	ifgt	3	Branch if > 0
158 (0x9e)	ifle	3	Branch if ≤ 0
159 (0x9f)	if_icmpeq	3	Comp top 2 stack elmt br on =
160 (0xa0)	if_icmpne	3	Comp top 2 stack elms br on ≠
161 (0xa1)	if_icmplt	3	Comp top 2 stack elm br on <
162 (0xa2)	if_icmpge	3	Comp top 2 stack elm br on
163 (0xa3)	if_icmpgt	3	Comp top 2 stack elm br on >
164 (0xa4)	if_icmple	3	Comp top 2 stack elm, br on ≤
165 (0xa5)	if_acmpeq	3	Comp top 2 stack objs br on =

Opcode	Mnemonic	Size	Description
166 (0xa6)	if_acmpne	3	Comp top two stk objs, br on ≠
167 (0xa7)	goto	3	Unconditional jump
168 (0xa8)	jsr	3	Jump to subroutine
169 (0xa9)	ret	2	Return from subroutine
170 (0xaa)	tableswitch	--	Goto case statement
171 (0xab)	lookupswitch	--	Goto case statement
172 (0xac)	ireturn	1	Return int from procedure
173 (0xad)	lreturn	1	Return long from procedure
174 (0xae)	freturn	1	Return float from procedure
175 (0xaf)	dreturn	1	Return dbl from procedure
176 (0xb0)	areturn	1	Return obj from procedure
177 (0xb1)	return	1	Return void from procedure
178 (0xb2)	getstatic	3	Get static field value
179 (0xb3)	putstatic	3	Set static field in class
180 (0xb4)	getfield	3	Get field value
181 (0xb5)	putfield	3	Set field in class
182 (0xb6)	invokevirtual	3	Call method, based on obj
183 (0xb7)	invokespecial	3	Call method, not based on obj
184 (0xb8)	invokestatic	3	Call a static method
185 (0xb9)	invokeinterface	5	Call an interface method
186 (0xba)	Undefined		
187 (0xbb)	new	3	Create new obj
188 (0xbc)	newarray	2	Allocate new array
189 (0xbd)	anewarray	3	Allocate new array of objs
190 (0xbe)	arraylength	1	Get length of array
191 (0xbf)	athrow	1	Throw an exception
192 (0xc0)	checkcast	3	Check if obj is of given type
193 (0xc1)	instanceof	3	See if obj is of given type
194 (0xc2)	monitorenter	1	Enter a monitored region
195 (0xc3)	monitorexit	1	Exit a monitored region
196 (0xc4)	wide	1	Prefix operation
197 (0xc5)	multianewarray	4	Allocate new multiarray
198 (0xc6)	ifnull	3	Test if null
199 (0xc7)	ifnonnull	3	Test if not null
200 (0xc8)	goto_w	5	Unconditional goto. 4B offset
201 (0xc9)	jsr_w	5	Jump sub 4-byte offset
202 (0xca)	breakpoint	1	Call breakpoint handler
203 (0xcb)	ldc_quick	2	Push item from const pool
204 (0xcc)	ldc_w_quick	3	Push item from const pool
205 (0xcd)	ldc2_w_quick	3	Push long or dbl frm const pool
206 (0xce)	getfield_quick	3	Get field from object
207 (0xcf)	putfield_quick	3	Set field in object
208 (0xd0)	getfield2_quick	3	Get long or dbl frm fld of obj
209 (0xd1)	putfield2_quick	3	Set field of obj (long or dbl)
210 (0xd2)	getstatic_quick	3	Get static field from class
211 (0xd3)	putstatic_quick	3	Set static field in class
212 (0xd4)	getstatic2_quick	3	Get long or dbl static field
213 (0xd5)	putstatic2_quick	3	Set long or dbl static field
214 (0xd6)	invokevirtual_quck	3	Invoke instance method
215 (0xd7)	invokenonvirtual_quick	3	Invoke instance method
216 (0xd8)	invokesuper_quick	3	Invoke instance method
217 (0xd9)	invokestatic_quick	3	Invoke static method
218 (0xda)	invokeinterface_quick	3	Invoke interface method
219 (0xdb)	Undefined		
220 (0xdc)	aastore_quick	1	St ref to array: no type checks
221 (0xdd)	new_quick	3	Create new obj
222 (0xde)	anewarray_quick	3	Create a new array of objs
223 (0xdf)	multianewarray_quick	3	Create new obj multiarray
224 (0xe0)	checkcast_quick	3	See ifobj is of given type
225 (0xe1)	instanceof_quick	3	See if obj is of given type
226 (0xe2)	invokevirtual_quick_w	3	Invoke instance method
227 (0xe3)	getfield_quick_w	3	Get field from obj (wide index)
228 (0xe4)	putfield_quick_w	3	Set field in obj (wide index)
229 (0xe5)	nonnull_quick	1	Pop obj ref and trap if null
230 (0xe6)	agetfield_quick	3	Read ref field in obj
231 (0xe7)	aputfield_quick	3	Set ref field in obj with GC checks
232 (0xe8)	agetstatic_quick	3	Read static ref field in class
233 (0xe9)	aputstatic_quick	3	Set static ref field in class with GC checks
234 (0xea)	aldc_quick	2	Push from const pool
235 (0xeb)	aldc_w_quick	3	Push ref from const pool
236 (0xec)	exit_sync_method	1	Jump to ret code for sync meth
237 (0xed)	sethi	3	Set top 16-bits of top stk entry
238 (0xee)	load_word_index	3	Indexed load word from mem
239 (0xef)	load_short_index	3	Indexed load short from mem
240 (0xf0)	load_char_index	3	Indexed load char from mem
241 (0xf1)	load_byte_index	3	Indexed load sgn byte frm mem
242 (0xf2)	load_ubyte_index	3	Indexed load unsgn byte - mem
243 (0xf3)	store_word_index	3	Indexed store word from mem
244 (0xf4)	nastore_word_index	3	Indexed naload wrd from mem
245 (0xf5)	store_short_index	3	Indexed store short from mem
246 (0xf6)	store_byte_index	3	Indexed store byte from mem
247 (0xf7) – 254 (0xfe)	Undefined		
255 (0xff)	**2-Byte Opcodes** (all opcodes start with 0xff)		
0 (0x00)	load_ubyte	2	Load unsigned byte from mem
1(0x01)	load_byte	2	Load signed byte from mem
2 (0x02)	load_char	2	Load unsigned short or char from mem
3 (0x03)	load_short	2	Load signed shrt from mem
4 (0x04)	load_word	2	Load int from mem
5 (0x05)	priv_ret_from_trap	2	Return from trap
6 (0x06)	priv_read_dcache_tag	2	Diag read of D$ tags
7 (0x07)	priv_read_dcache_data	2	Diag read of D$ data array
8 (0x08)	Undefined		
9 (0x09)	Undefined		
10 (0x0a)	load_char_oe	2	End-swap load unsigned short or char from mem
11 (0x0b)	load_short_oe	2	End-swap ld signed shrt from mem
12 (0x0c)	load_word_oe	2	End-swap ld int from mem
13 (0x0d)	return0	2	Ret with no value from sub
14 (0x0e)	priv_read_icache_tag	2	Diag read of I$ tags
15 (0x0f)	priv_read_icache_data	2	Diag read of I$ data array
16 (0x10)	ncload_ubyte	2	NC ld unsigned byte from mem
17 (0x11)	ncload_byte	2	NC ld signed byte from mem
18 (0x12)	ncload_char	2	NC ld unsigned short or char from mem
19 (0x13)	ncload_short	2	NC ld signed short from mem
20 (0x14)	ncload_word	2	NC load int from mem
21 (0x15)	iucmp	2	Unsigned integer compare
22 (0x16)	priv_powerdown	2	Low-power standby state
23 (0x17)	cache_invalidate	2	Inval cache line
24 (0x18)	Undefined		
25 (0x19)	Undefined		
26 (0x1a)	ncload_char_oe	2	NC end-swp ld unsigd short or char from mem
27 (0x1b)	ncload_short_oe	2	NC end-swp ld sig short from mem
28 (0x1c)	ncload_word_oe	2	NC end-swp ld int from mem
29 (0x1d)	return1	2	Return with one-word val
30 (0x1e)	cache_flush	2	Flush cache line and inval
31 (0x1f)	cache_index_flush	2	Flush cache line and inval
32 (0x20)	store_byte	2	Store byte to mem
33 (0x21)	Undefined		
34 (0x22)	store_short	2	Store short or char to mem
35 (0x23)	Undefined		
36 (0x24)	store_word	2	Store int to mem
37 (0x25)	soft_trap	2	Initiate a software trap
38 (0x26)	priv_write_dcache_tag	2	Diag write of D$ tags
39 (0x27)	priv_write_dcache_data	2	Diag write of D$ data array
40 (0x28)	Undefined		
41 (0x29	Undefined		
42 (0x2a)	store_short_oe	2	End-swp store short to mem
43 (0x2b)	Undefined		
44 (0x2c)	store_word_oe	2	End-swap store int to mem
45 (0x2d)	return2	2	Ret with 2-word val from sub
46 (0x2e)	priv_write_icache_tag	2	Diag write of I$ tags
47 (0x2f)	priv_write_icache_data	2	Diag write of I$ data array
48 (0x30)	ncstore_byte	2	NC store byte to mem
49 (0x31)	Undefined		
50 (0x32)	ncstore_short	2	NC store short or char to mem
51 (0x33)	Undefined		
52 (0x34)	ncstore_word	2	NC store int to mem
53 (0x35)	Undefined		
54 (0x36)	priv_reset	2	Software-initiated reset
55 (0x37)	get_current_class	2	Push class pntr for curt meth
56 (0x38)	Undefined		
57 (0x39)	Undefined		
58 (0x3a)	ncstore_short_oe	2	NC end-swp store shrt to mem
59 (0x3b)	Undefined		
60 (0x3c)	ncstore_word_oe	2	NC end-swp store int to mem
61 (0x3d)	call	2	Call sub with spec # of args
62 (0x3e)	zero_line	2	Zero cache line data
63 (0x3f)	priv_update_optop	2	Atomic write OPTOP & OPLIM
64 (0x40)	read_pc	2	Read PC reg
65 (0x41)	read_vars	2	Read VARS reg
66 (0x42)	read_frame	2	Read FRAME reg
67 (0x43)	read_optop	2	Read OPTOP reg
68 (0x44)	priv_read_oplim	2	Priv read OPLIM reg
69 (0x45)	read_const_pool	2	Read CONST_POOL reg
70 (0x46)	priv_read_psr	2	Priv read PSR reg
71 (0x47)	priv_read_trapbase	2	Priv read TRAPBASE reg
72 (0x48)	priv_read_lockcount0	2	Priv read LOCKCOUNT0 reg
73 (0x49)	priv_read_lockcount1	2	Priv read LOCKCOUNT1 reg
74 (0x4a)	Undefined		
75 (0x4b)	Undefined		
76 (0x4c)	priv_read_lockaddr0	2	Priv read LOCKADDR0 reg
77 (0x4d)	priv_read_lockaddr1	2	Priv read LOCKADDR1 reg
78 (0x4e)	Undefined		
79 (0x4f)	Undefined		
80 (0x50)	priv_read_userrange1	2	Priv read USERRANGE1 reg
81 (0x51)	priv_read_gc_config	2	Priv read GC_CONFIG reg

Opcode	Mnemonic	Size	Description
82 (0x52)	priv_read_brk1a	2	Priv read BRK1A reg
83 (0x53)	priv_read_brk2a	2	Priv read BRK2A reg
84 (0x54)	priv_read_brk12c	2	Priv read BRK12C reg
85 (0x55)	priv_red_userrange2	2	Priv read USERRANGE2 reg
86 (0x56)	Undefined		
87 (0x57)	priv_read_versionid	2	Priv read VERSIONID reg
88 (0x58)	priv_read_hcr	2	Priv read HCR reg
89 (0x59)	priv_read_sc_bottom	2	Priv read SC_BOTTOM reg
90 (0x5a)	read_global0	2	Read GLOBAL0 reg
91 (0x5b)	read_global1	2	Read GLOBAL1 reg
92 (0x5c)	read_global2	2	Read GLOBAL2 reg
93 (0x5d)	read_global3	2	Read GLOBAL3 reg
94 (0x5e)	Undefined		
95 (0x5f)	Undefined		
96 (0x60)	write_pc	2	Write PC reg
97 (0x61)	write_vars	2	Write VARS reg
98 (0x62)	write_frame	2	Write FRAME reg
99 (0x63)	write_optop	2	Write OPTOP reg
100 (0x64)	priv_write_oplim	2	Priv write OPLIM reg
101 (0x65)	write_const_pool	2	Write CONST_POOL reg
102 (0x66)	priv_write_psr	2	Priv write PSR reg
103 (0x67)	priv_write_trapbase	2	Priv write TRAPBASE reg
104 (0x68)	priv_write_lockcount0	2	Priv write LOCKCOUNT0 reg
105 (0x69)	priv_write_lockcount1	2	Priv write LOCKCOUNT1 reg
106 (0x6a)	Undefined		
107 (0x6b)	Undefined		
108 (0x6c)	priv_write_lockaddr0	2	Priv write LOCKADDR0 reg
109 (0x6d)	priv_write_lockaddr1	2	Priv write LOCKADDR1 reg
110 (0x6e)	Undefined		
111 (0x6f)	Undefined		
112 (0x70)	priv_write_userrange1	2	Priv write USERRANGE1 reg
113 (0x71)	priv_write_gc_config	2	Priv write GC_CONFIG reg
114 (0x72)	priv_write_brk1a	2	Priv write BRK1A reg
115 (0x73)	priv_write_brk2a	2	Priv write BRK2A reg
116 (0x74)	priv_write_brk12c	2	Priv write BRK12C reg
117 (0x75)	priv_red_userrange2	2	Priv write USERRANGE2 reg
118 (0x76)	Undefined		
119 (0x77)	Undefined		
120 (0x78)	Undefined		
121 (0x79)	priv_write_sc_bottom	2	Priv write SC_BOTTOM reg
122 (0x7a)	write_global0	2	Write GLOBAL0 reg
123 (0x7b)	write_global1	2	Write GLOBAL1 reg
124 (0x7c)	write_global2	2	Write GLOBAL2 reg
125 (0x7d)	write_global3	2	Write GLOBAL3 reg