1. Introduction

1.1 Intended audience

This manual is intended for developers implementing Java® virtual machines or Java debuggers using the AVR©32 Java Extension Module, or others needing information on the hardware aspects of such implementations.

Developers who wish to develop and run Java programs on an existing Java virtual machine should read the documentation for the virtual machine and the Java documentation from Sun Microsystems™, but do not need to be familiar with the contents of this manual.

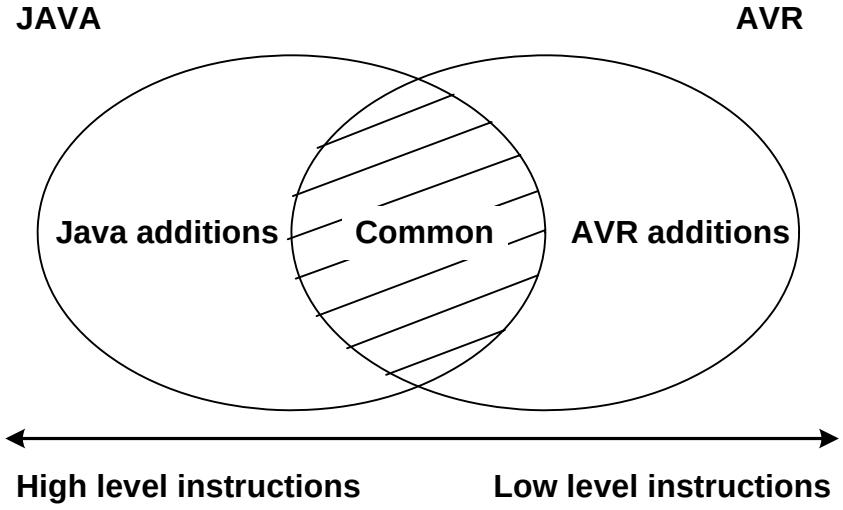
1.2 The AVR32 Java Extension Module

By including a Java Extension Module (JEM), the AVR32 optionally supports execution of Java bytecodes. This support is included with minimal hardware overhead.

Comparing Java instructions with AVR32 instructions, we see that a large part of the instructions overlap as illustrated in Figure 1-1 on page 1. The idea is thus to reuse the hardware resources by adding a separate Java instruction decoder and control module that takes over execution in a special Java state. The processor keeps track of its execution state through the status register and changes execution mode seamlessly.

A large part of the instruction set is shared between the AVR RISC and the Java Virtual Machine. The Java instruction set includes instructions with high semantically contents while the AVR RISC instruction set complements Java's set with traditional hardware near RISC instructions, see Figure 1-1 on page 1.

Figure 1-1. Instruction set shared between the AVR and the Java Virtual Machine.



In a larger runtime system, there will typically be an operating system that keeps track of, and dispatches different processes. A Java program will typically be one, or several, of these processes. There is little or no extra overhead connected with a context switch to a Java process compared to a context switch to a native process.

AVR® 32-bit Microcontroller

Java® Technical Reference

32049A–AVR32–09/06

The Java binary instructions are called bytecodes. They include some instructions with a high semantic content. In order to reduce the hardware overhead, these instructions are trapped and executed as small RISC programs. These programs are stored in the program memory and can be changed by the programmer. This gives full flexibility with regards to future extensions of the Java instruction set. Runtime analyses show that the instructions trapped are less than 10% of the executed instructions. Performance is ensured through an efficient trapping mechanism and “Java tailored” RISC instructions.

The Java bytecodes are run in the application context, while trap handlers run in the supervisor context. This change of context is handled by the java entry and exit mechanisms.

1.2.1 Nomenclature

The word "Stack" in this chapter refers to the stack pointed to by the system stack pointer SP (R13). "Java Operand Stack", or short "Operand Stack" refers to the Java operand stack.

Java state can be implemented both as a traditional Java Virtual Machine, but also as a Java Card Virtual Machine. In this document the terms Java state and Java Virtual Machine will be used.

A “bytecode” is a Java instruction one or more bytes long. A bytecode consists of an opcode and optional arguments.

1.3 The AVR32 Java Virtual Machine

The AVR32 Java Virtual machine consists of two parts, the Java Extension Module in hardware and the AVR32 specific Java Virtual Machine software. Together, the two modules comply with the Java Virtual Machine specification.

The AVR32 Java Virtual Machine software loads and controls the execution of the Java classes. The bytecodes are executed in hardware, except for some instructions, for example the instructions that create or manipulate objects. These are trapped and executed in software within the Java Virtual Machine. See Figure 1-2 on page 3.

Figure 1-2. Overview of the AVR32 Java Virtual Machine and the Java Extension Module. The grey area represent the software parts of the virtual machine, while the white box to the right represents the hardware module.

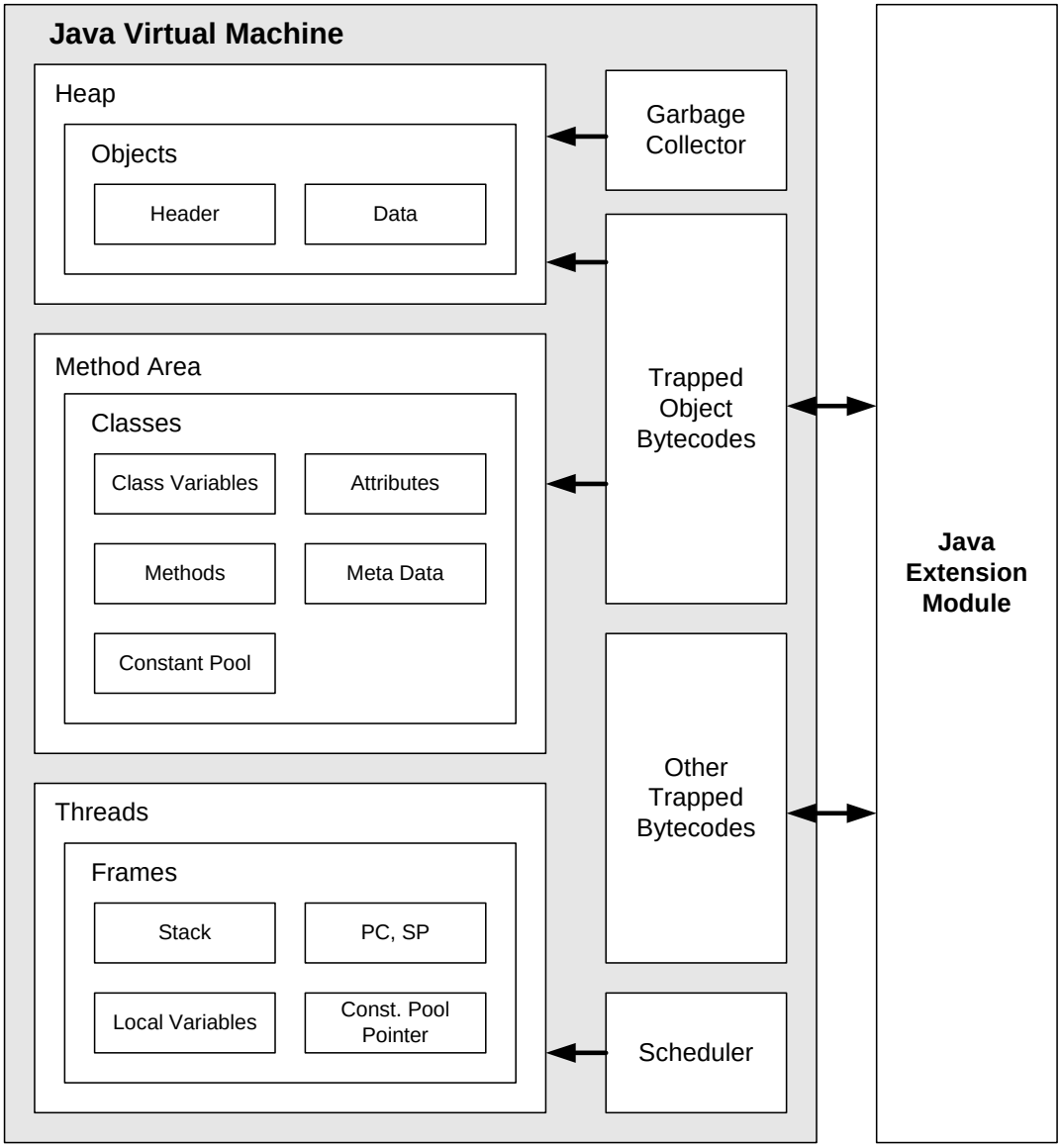
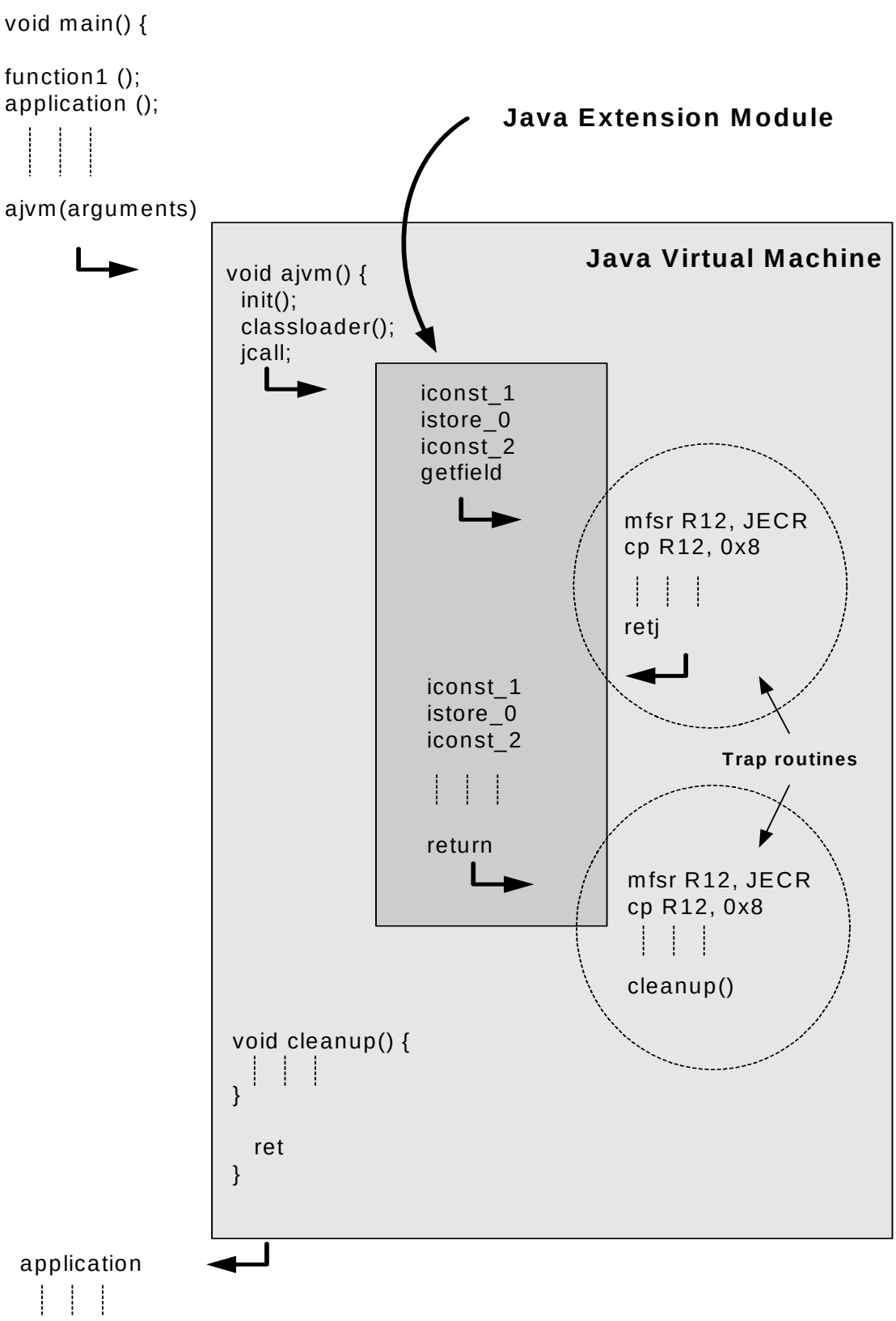


Figure 1-3 on page 5 shows one example on how a Java program is executed. The processor boots in AVR32 (RISC) state and it executes applications as a normal RISC processor. To invoke a Java program, the Java Virtual Machine is called like any other application. The Java Virtual Machine will execute an init routine followed by a class loader that parses the class and initialize all registers necessary to start executing the Java program. The last instruction in the class loader is the “retj” instruction that sets the processor in the Java state. This means that the instruction decoder now decodes Java opcodes instead of the normal AVR32 opcodes.

During execution of the Java program, the Java Extension Module will encounter some bytecodes that are not supported in hardware. The instruction decoder will automatically recognize these bytecodes and switch the processor back into RISC state and at the same time jump to a predefined location where it will execute a software routine that performs the semantic of the trapped bytecode. When finished, the routine ends with a “retj” instruction. This instruction will make the AVR32 return to Java state and the Java program will continue at the correct location.

The Java instruction "return" is also trapped. It normally returns to the previous method. If the method returning is the main method, the trap routine will return control to the AJVM which will run a cleanup procedure before exiting.

Figure 1-3. Example of running a Java program



2. Programming model

2.1 The AVR32 Java Mode

The java mode of the AVR32 processor is controlled by the J bit in the status register. When the J bit is set, instructions are interpreted as Java bytecodes, otherwise they are interpreted as native RISC instructions.

Some java instructions need to access a Constant Pool. A pointer to the beginning of the constant pool is required in register R8. The format of the constant pool is as specified in the Java Virtual Machine Specification.

Some java instructions need to access a Current Method Frame. A pointer to this frame is required in register R9. See chapter 2.2 for details.

The system registers JECR, JOSP, JTBA, JBCR, and JAVA_LV0 through JAVA_LV7 are also used in Java mode, and their use are descibed in subsequent chapters.

2.1.1 Entering Java mode

Before Java state is entered, the JVM must have initialized the appropriate registers.

Java state is entered by executing the RETJ instruction. The link register (LR) contains the absolute address of the first Java byte code. The contents of register R0 through R7, R11, and R12 will be destroyed upon calling Java state and should be stored on stack before RETJ is executed.

The instruction RETJ performs the following operations:

- Moves the value in the link register to the PC register.
- Sets the J flag in the status register.
- Clears the R flag in the status register.
- If executed in supervisor mode, the GM flag in the status register is set.

It is also possible to enter Java mode by using RETE or RETS, but in this case the value in RSR must be modified by software.

Entering java mode by setting the J bit in the status register through MTSR or SSRF instructions are not supported, and will lead to undefined behaviour.

2.1.2 Leaving Java mode

The only way to exit Java state is by executing an instruction that is not supported in hardware. This instruction is trapped, and executed as a RISC routine. The Java return instructions are all trapped, and the trap routine will detect when the main method executes a return. This signals that the Java program has ended, and the JVM will execute a cleanup procedure before exiting.

See chapter 2.5 for details on trapping instructions.

2.2 The Current Method Frame

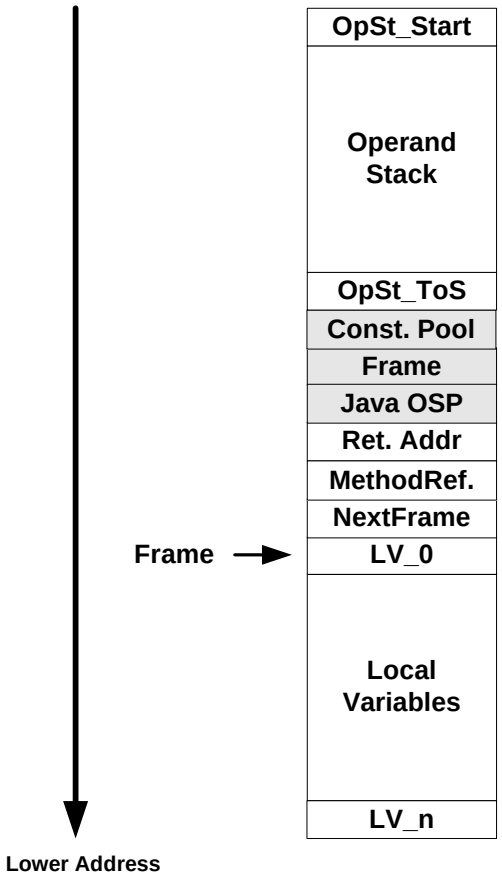
2.2.1 The Java Frame

The frame is a runtime data structure that holds all variables and references to support the execution of a method. The frame may contain incoming arguments to the method, as well as outgoing arguments to an invoked method. A frame is allocated when the method is invoked and destroyed when the program returns from the method.

An example of an AVR32 frame is shown in Figure 2-1. Different virtual machine implementations are free to implement the frame, as long as the Frame pointer in register 9 in the register file points to the first entity of the local variables.

The frame contains the operand stack, the local variables, a pointer to the next free memory space where the next frame can be placed, and the return address for the method. The frame for the current method will in addition contain the Frame pointer for the previous method together with the constant pool pointer for the previous method and the Java operand stack pointer for the previous method.

Figure 2-1. The method frame in the AVR32 Java Virtual Machine.



2.2.2 Allocating a New Frame

All methods use the same process of creating a method frame. When the AJVM 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
- Operand stack
- Return value from a method invocation

2.2.3 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 accessible as local variables by the invoked method.

A Java compiler statically produces a method structure containing the number of arguments:

- For nonstatic method invocation, the object reference and the first argument are accessible as local variable 0 and local variable 1, respectively.
- For static method invocation, the first argument becomes local variable 0.

2.2.3.1 Local Variables

When the core invokes a method, it allocates an area on the stack for storage for local variables.

A Java compiler statically determines the number of local variable words that are required; the AJVM allocates them accordingly.

2.2.3.2 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, FRAME, and Constant Pool pointer registers.

2.2.3.3 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.

2.2.4 Invoking a Method

The following is the procedure for invoking a method

- Resolve a method reference
- Access the method structure
- Allocate a new method frame
- Save the invoker's method context
- Pass control to the invoked method by branching to the method's entry point.

2.2.4.1 *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 trap routines should use this symbolic information to determine one of the following:

- An index to the method, which the core then uses to look up a method structure pointer
- A direct pointer to a method structure.

Resolving a method reference may involve class loading and resolution with subsequent method searches based on the referenced method name and signature.

2.2.4.2 *Accessing a Method Structure*

TBD

2.2.4.3 *Allocating a New Method Frame*

When the core invokes a method, it allocates a new frame and initializes the following registers, Frame, JOSP and Const. Pool pointer.

In addition it writes the address to the first available memory address in the field designated for NextFrame.

2.2.4.4 *Saving the Invokers Method Context*

When the core invokes a method, it will need to store the invokers method context. This is done by writing the local variables, the operand stack, the frame register and the Const. Pool register onto the correct frames.

2.2.4.5 *Passing Control to the Invoked Method*

When the frame is set up by the trap routine, the instruction retj is used to return to Java mode. The method invocation routine must have placed the correct program counter in the link register before executing retj.

2.2.5 Invoking a Synchronized Method

TBD.

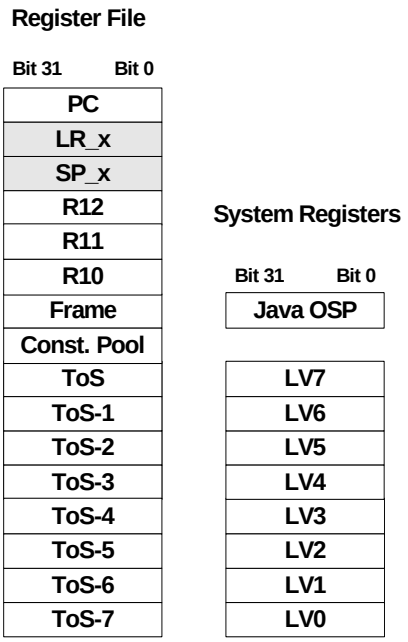
2.2.6 Returning from a Method

The java return instructions are trapped and executed as trap routines. These routines must restore the previous frame and return control to the previous method.

2.3 The RISC Register File and the Java Operand Stack

The function of the register file is altered in Java State. The Java Virtual Machine executes as a zero operand stack machine. The register file functionality is therefore changed to facilitate a push-pop stack and Java system registers as shown in Figure 2-2.

Figure 2-2. The register file functionality is altered in Java state to facilitate a push/pop stack.



ToS: Top of operand Stack

Java OSP: Java operand stack pointer. A 16 bit pointer that points to the top of the operand stack.

Frame: This register has to contain the address to local variable 0 in the current frame.

Const. Pool: This register should contain the address to the Constant Pool of the current class. Some implementations of the Java virtual machine may choose to use this register for other purposes.

LVx: Local variable stack.

2.3.1 Java Operand Stack

The operand stack holds the values to be operated on and the result of the operation. The operand stack is part of the current method frame, and a part of the operand stack is held in the register file in order to speed up operation. The register file holds up to eight elements of the operand stack. This is sufficient to execute a large majority of the Java methods.

Bits [2:0] of the Java Operans Stack Pointer (JOSP) points to the register that holds the ToS element. The rest of the register is available for use by the JVM.

The operand stack pointer is initialized to the value of 0, meaning that the operand stack is empty.

2.3.2 Operand Stack overflow / underflow

The operand stack overflows when element number nine is pushed on the stack. An overflow is automatically detected and a trap is triggered. The trap routine will have to copy necessary operand stack elements onto the designated reserved memory location in the frame.

Similarly, an underflow occurs when accessing an element if the operand stack is empty. This is also detected automatically and a trap is triggered.

2.4 Data structures

2.4.1 Byte values

A byte is a signed 8-bit integer. When on the operand stack or in a local variable, a byte value is sign extended and treated as an int. When stored in an array no sign extension is done, and the bytes are stored as 8-bit integers (RISC-mode bytes).

2.4.2 Short values

A short is a signed 16-bit integer. When on the operand stack or in a local variable, a short value is sign extended and treated as an int. When stored in an array no sign extension is done, and the shorts are stored as 16-bit integers (RISC-mode halfwords).

2.4.3 Char values

A char is an unsigned 16-bit integer. When on the operand stack or in a local variable, a char value is zero extended and treated as an int. When stored in an array no sign extension is done, and the shorts are stored as 16-bit integers (RISC-mode halfwords).

2.4.4 Int values

A int value is a signed 32-bit integer. An int is always treated as a RISC-mode word.

2.4.5 Float values

A float is a IEEE® 754 single precision floating point number. For the purpose of storage, a float is treated the same as an int. No floating point arithmetic is currently implemented in hardware.

2.4.6 Long values

A long value is a 64-bit signed integer, but is handled by the JEM as two 32-bit integers MSW (Most Significant Word) and LSW (Least Significant Word). These two words are located as follows:

- Operand stack; when a long is on top of the stack: MSW is in TOS and LSW is in TOS-1.
- Local variables; when a long is in local variable *n* and *n*+1: MSW is in LV_<*n*+1> and LSW is in LV_<*n*>.
- Memory; when a long is in memory at address *addr*: MSW is at address *addr*, and LSW is at address *addr*+4.

2.4.7 Double values

A float is a IEEE 754 double precision floating point number. For the purpose of storage, a double is treated the same as a long. No floating point arithmetic is currently implemented in hardware.

2.4.8 Object references

An object reference is a 32-bit memory address. For the purpose of storage, an object reference is treated as a RISC-mode word.

In order to support several heap structures, the JEM contains a H or handle bit in the status register. If the handle bit is cleared, the object reference points directly to instance variable zero of the object or array as shown in Figure 2-3 on page 13. If the handle bit is set, the object reference points to the handle for the object or array as shown in Figure 2-4 on page 13.

Figure 2-3. Object reference with handle bit cleared.

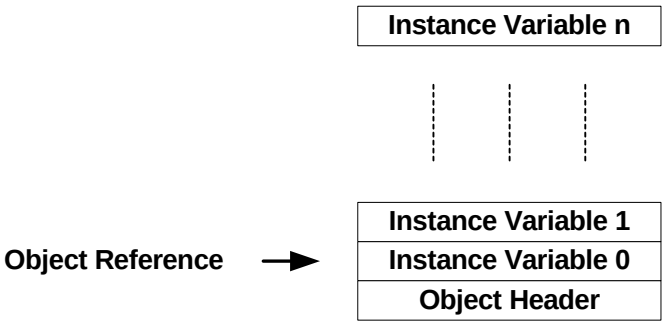
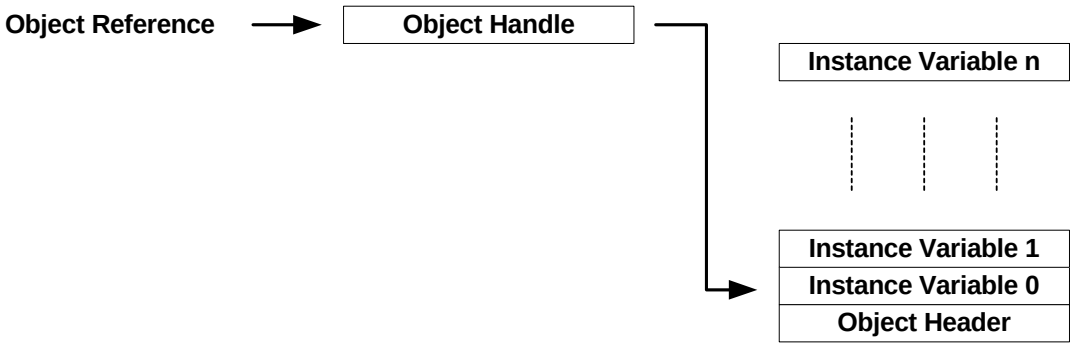


Figure 2-4. Object reference with handle bit set



2.4.8.1 *Format of the object handle*

In order to support garbage collection in software, some of the bits in the object handle are masked away when the hardware uses them. The masked bits are reserved for use by a garbage collector.

The format of the array and object references are shown in Figure 2-5 on page 13. Masked bits are shown as an X.

Figure 2-5. Format of object and array handles

2.4.8.2 *Definition of NULL*

The NULL object reference is defined as 0x00000000.

2.4.9 Arrays

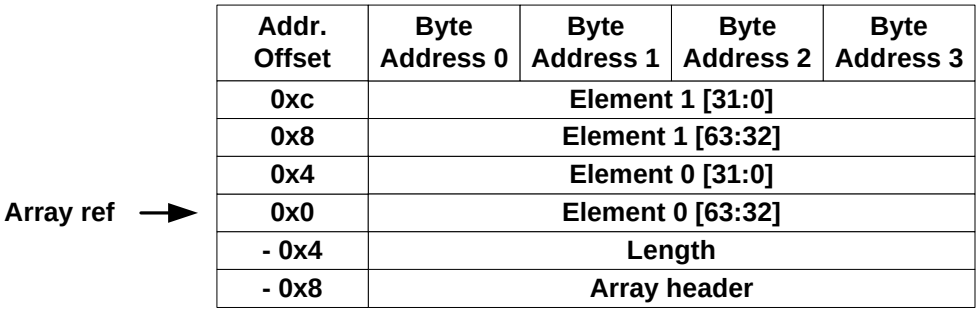
The array data structure is similar to the object data structure, shown in Figure 2-3 on page 13 and Figure 2-4 on page 13, except that array size information is stored into the first word of the data storage.

If the handle bit is set, the references must go through the handle to access the array, similar to what is shown in Figure 2-5 on page 13.

The array data structures are as follows:

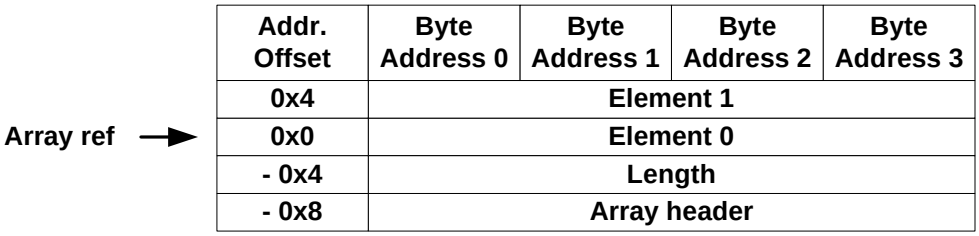
2.4.9.1 *Array of 64 bit data types*

Figure 2-6. Array of longs structure



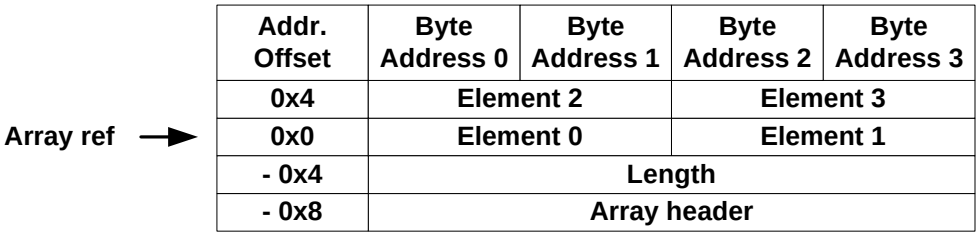
2.4.9.2 *Array of 32 bit data types*

Figure 2-7. Array of 32 bit data types



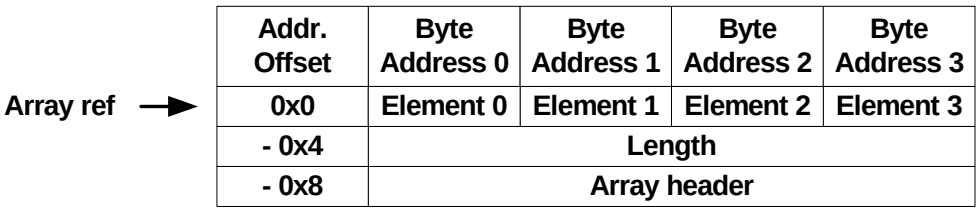
2.4.9.3 *Array of 16 bit data types*

Figure 2-8. **Array of shorts**



2.4.9.4 *Array of 8 bit data types*

Figure 2-9. Array of bytes



2.5 Trapping instruction

2.5.1 Entering traps

Unimplemented instructions are automatically trapped. A trap is executed very much like the call instruction. Each unimplemented instruction has an entry point associated with it that is the address that the JEM automatically jump to when the unimplemented instruction is encountered.

The entry address is generated by OR'ing together a entry point dependent offset and the value in the JTBA (Java Trap Base Address) system register. The trap offsets are shown in Table 2-1. Because the offset is OR'ed in instead of added, JTBA should be aligned to a power-of-two boundary that is larger than the largest possible trap offset.

When a Java instruction is trapped, the instruction is automatically written to the JECR register. The opcode will always be resident in Byte 2, while the arguments will be placed in byte 1 and 0. If the instruction is longer than three bytes, the opcode will be placed in byte 2, while the contents of byte 1 and 0 will be undefined. The trap routine will in this case have to fetch the arguments from the program memory. See the instruction set reference in chapter 3. for more information.

When entering the trap, the J bit in the system register is cleared, returning the CPU to RISC mode. The R bit in the system register is set, simplifying accessing the operand stack from RISC mode. If the JVM runs in supervisor mode, the GM bit is set to help ensure thread and interrupt safety.

2.5.2 Exiting traps

Traps are exited by executing the "RETJ" instruction. This will return to Java mode, and continue executing bytecodes from the address in LR.

The stack overflow/underflow INCJOSP traps must be exited in another way. See chapter 2.5.5.4 for detals.

When the trapped bytecode is a return from the main function, the java program is finished. In this case the JVM can clear the R bit and run native RISC code, e.g. the JVM cleanup and exit routines.

2.5.3 Accessing the operand stack from a trap routine

Trap routines need to access the operand stack. To avoid the cumbersome process of having to read out the Java operand stack pointer, and translate it to the correct register, the Java extension module contains a unique register remapping unit. This ensures that top of stack (ToS) is always in register R7, ToS-1 is always in R6 etc.

When pushing new values to the operand stack, the special instruction “incjosp” is used. Incjosp increment the operand stack pointer and thereby pushes all of the operand stack elements N elements down. This frees N element on the top of the stack. The value going onto the stack can thereafter be moved to R7. The incjosp instruction is also used to pop elements off the stack.

The incjosp instruction will automatically handle stack overflow/underflow. See chapter 2.5.5.4 for special considerations when using incjosp.

2.5.4 Accessing local variables from a trap routine

The first eight local variables are accessible as system registers. Higher numbered local variables are found in the current frame as described in chapter 2.2.

2.5.5 Trap Entry Points

The Figure 2-1 on page 16 shows the possible trap entry points. Some of them are described in further detail below.

Table 2-1. Java trap entry points

EP number	JTBA Offset		JECR	LR	Instruction Groups
0	0x000	0	Exception number	PC	Java Exceptions
1	0x080	128	Undefined	PC	Stack overflow
2	0x100	256	Undefined	PC	Stack underflow
3	0x180	384	Undefined	PC	Stack overflow INCJOSP
4	0x200	512	Undefined	PC	Stack underflow INCJOSP
5	0x280	640	Bytecode	PC + size	ireturn/areturn/freturn
6	0x300	768	Bytecode	PC + size	return
7	0x380	896	Bytecode	PC + size	lreturn/dreturn
8	0x400	1024	Bytecode	PC + size	iinc
9	0x480	1152	Bytecode	PC + size	checkcast_quick
10	0x500	1280	Bytecode	PC + size	instanceof_quick
11	0x580	1408	Bytecode	PC + size	getstatic
12	0x600	1536	Bytecode	PC + size	putstatic
13	0x680	1664	Bytecode	PC + size	new
14	0x700	1792	Bytecode	PC + size	invokestatic
15	0x780	1920	Bytecode	PC + size	invokeinterface_quick
16	0x800	2048	Bytecode	PC + size	invokevirtual_quick
17	0x880	2176	Bytecode	PC + size	invokestatic_quick
18	0x900	2304	Bytecode	PC + size	invokenonvirtual_quick
19	0x980	2432	Bytecode	PC + size	newarray/anewarray/multianewarray
20	0xA00	2560	Bytecode	PC + size	Long instructions
21	0xA80	2688	Bytecode	PC + size	Float instructions
22	0xB00	2816	Bytecode	PC + size	Double instructions
23	0xB80	2944	Bytecode	PC + size	All other trapped instructions

2.5.5.1 *Java Exception trap*

This trap is used when the Java Extension Module discovers a case where a Java RuntimeException should be thrown. JECR is set to a number that identifies the exception, and the trap routine is responsible for creating an instance of the correct Exception class, and throwing this.

Table 2-2. JECR values when a Java System Exception generated by the HW occur.

JECR	Exception Type
0x0	ArithmeticException
0x1	NullPointerException
0x2	ArrayIndexOutOfBoundsException
0x53 xx xx	Aastore write barrier
0xe7 xx xx	Aputfield_quick write barrier

2.5.5.2 *Stack overflow trap*

This trap is used when a bytecode tries to push the ninth element onto the part of the operand stack resident in the register file. The trap routine should copy some operand stack entries onto the current frame, decrement JOSP, and move remaining registers to the new top-of-stack. This makes room for new elements on the operand stack, and the bytecode can be resumed.

2.5.5.3 *Stack underflow trap*

This trap is used when a bytecode tries to pop from the operand stack when no elements are resident in the the register file. The trap routine should increment JOSP, move the entries currently in the register file, and copy some operand stack entries from the current frame. This puts more elements on the operand stack, and the bytecode can be resumed.

2.5.5.4 *Stack overflow INCJOSP trap*

This trap is used when a *incjosp* instruction tries to increase JOSP so more than eight elements of the operand stack are resident in the register file. It should behave just like the normal stack overflow trap, but is given a separete entry point because it requires a different return mechanism.

Note the following:

- Incjosp puts the return address in *R12* when it traps. Executing incjosp may thus destroy R12.
- The best way to return from this trap is "mov PC, R12".

2.5.5.5 *Stack underflow INCJOSP trap*

This trap is used when a *incjosp* instruction tries to remove more elements from the operand than are currently resident in the the register file. It should behave just like the normal stack underflow trap, but is given a separete entry point because it requires a different return mechanism.

All the considerations described above for "Stack overflow INCJOSP" also apply to this trap.

2.5.5.6 *Other traps*

The behaviour of other traps depends on the length of the trapped instruction:

Table 2-3. JECR and LR in other traps

Bytecode length	JECR [23:16]	JECR[15:8]	JECR[7:0]	LR
1 byte	Opcode	Undefined	Undefined	PC + 1
2 bytes	Opcode	Parameter 1	Undefined	PC + 2
3 bytes or more	Opcode	Parameter 1	Parameter 2	PC + 3

All unknown bytecodes trap to entry point 23, and are treated as 1 byte long.

2.6 Garbage collection support

2.6.1 Object handles

When the Handle bit in the status register is set, the JEM refers to all objects through object handles, as described in chapter 2.4.8. This extra level of indirection allows the JVM to keep all object references in a separate memory area, and allows the four masked bits in each object reference to be used for implementing e.g. a mark-sweep algorithm.

2.6.2 Write barriers

In order to support more efficient garbage collection algorithms, the JEM includes support for a write barrier. The write barrier may be used to cause a trap when an object reference is stored to either an array or a field in an object. This can be used to keep e.g. a MC2 garbage collector updated.

The Java Barrier Control Register (JBCR) holds the lower bound of a memory area that causes a trap when storing a reference to an object there. Comparisons against JBCR are only enabled when the Handle bit in the status register is set. JBCR is not reset by hardware, so it must be manually set before entering Java mode. The write barrier can be effectively disabled by setting JBCR to a value above the memory accessible in application mode, e.g. 0x80000000.

The *aastore* and *aputfield_quick* bytecodes test the reference against JBCR when the Handle bit is set. See the Instruction Set Description in chapter 4. for details. The *putfield_quick* and *putstatic_quick* bytecodes does not test against JBCR, so the class loader must make sure to use either *aputfield_quick* or *putstatic* for storing object references.

3. Java Extension Module Instruction Set

3.1 Operator Symbols

&	Bitwise AND operation.
|	Bitwise OR operation.
^	Bitwise EOR operation.
~	Bitwise NOT operation.
&&	Logical AND operation
||	Logical OR operation
!	Logical NOT operation
SE	Sign Extend
ZE	Zero Extend
a[b:c]	Extract the bits from bit index *b* down to bit index *c* from the value *a.*

3.2 The Operand Stack

ToS	"Top of Stack". Refers to the position that is the top of the operand stack.
ToS-N	Refers to the N'th position under the "Top of Stack"

All operations refers to the operand stack positions before the instruction is executed.

3.3 Special Registers

Const. Pool	Register R8
Frame	Register R9

3.4 Java Bytecode Summary

Any bytecode not described in the following table is assumed to have length 1 and trap to Trap Entry Point 23.

Mnemonic	Opcode	Length	Trap Entry Point	Description
aaload	0x32	1		Load reference from array
aastore	0x53	1		Store into reference array
aconst_null	0x01	1		Push *null* object reference on stack.
aload	0x19	2		Load reference from local variable
aload_0	0x2A	1		Load reference from local variable 0
aload_1	0x2B	1		Load reference from local variable 1
aload_2	0x2C	1		Load reference from local variable 2
aload_3	0x2D	1		Load reference from local variable 3
anewarray	0xBD	3	23	Create a new array of reference
anewarray_quick	0xDE	3	19	Create a new array of reference
aputfield_quick	0xE7	3		Set field in object to reference
areturn	0xB0	1	5	Return reference from method
arraylength	0xBE	1		Get length of array
astore	0x3A	2		Store reference in local variable
astore_0	0x4B	1		Store reference in local variable 0
astore_1	0x4C	1		Store reference in local variable 1
astore_2	0x4D	1		Store reference in local variable 2
astore_3	0x4E	1		Store reference in local variable 3
athrow	0xBF	1	23	Throw exception or Error
baload	0x33	1		Load byte from array
bastore	0x54	1		Store into byte array
bipush	0x10	2		Push the byte argument on the stack
caload	0x34	1		Load char from array
castore	0x55	1		Store into char array
checkcast	0xC0	3	23	Check whether object is of given type
checkcast_quick	0xE0	3	9	Check whether object is of given type
d2f	0x90	1	22	Convert double to float
d2i	0x8E	1	22	Convert double to int
d2l	0x8F	1	22	Convert double to long
dadd	0x63	1	22	Add double
daload	0x31	1		Load double from array
dastore	0x52	1		Store into double array
dcmpg	0x98	1	22	Compare double

dcmpl	0x97	1	22	Compare double
dconst_0	0x0E	1		Push the double value *0.0* on stack
dconst_1	0x0F	1		Push the double value *1.0* on stack
ddiv	0x6F	1	22	Divide Double
dload	0x18	2		Load double from local variable
dload_0	0x26	1		Load double from local variable 0
dload_1	0x27	1		Load double from local variable 1
dload_2	0x28	1		Load double from local variable 2
dload_3	0x29	1		Load double from local variable 3
dmul	0x6B	1	22	Multiply double
dneg	0x77	1	22	Negate double
drem	0x73	1	22	Remainder double
dreturn	0xAF	1	7	Return double from method
dstore	0x39	2		Store double in local variable
dstore_0	0x47	1		Store double in local variable 0
dstore_1	0x48	1		Store double in local variable 1
dstore_2	0x49	1		Store double in local variable 2
dstore_3	0x4A	1		Store double in local variable 3
dsub	0x67	1	22	Subtract double
dup	0x59	1		Duplicate top operand stack word
dup_x1	0x5A	1		Duplicate top operand stack word and put two down
dup_x2	0x5B	1	23	Duplicate top operand stack word and put three down
dup2	0x5C	1		Duplicate top two operand stack words
dup2_x1	0x5D	1	23	Duplicate top two operand stack words and put three down
dup2_x2	0x5E	1	23	Duplicate top two operand stack words and put four down
f2d	0x8D	1	21	Convert float to double
f2i	0x8B	1	21	Convert float to int
f2l	0x8C	1	21	Convert float to long
fadd	0x62	1	21	Add float
faload	0x30	1		Load float from array
fastore	0x51	1		Store into float array
fcmpg	0x96	1	21	Compare float
fcmpl	0x95	1	21	Compare float
fconst_0	0x0B	1		Push the float value *0.0* on stack
fconst_1	0x0C	1		Push the float value *1.0* on stack
fconst_2	0x0D	1		Push the float value *2.0* on stack
fdiv	0x6E	1	21	Divide float
fload	0x17	2		Load float from local variable

fload_0	0x22	1		Load float from local variable 0
fload_1	0x23	1		Load float from local variable 1
fload_2	0x24	1		Load float from local variable 2
fload_3	0x25	1		Load float from local variable 3
fmul	0x6A	1	21	Multiply float
fneg	0x76	1	21	Negate float
frem	0x72	1	21	Remainder float
freturn	0xAE	1	5	Return float from method
fstore	0x38	2		Store float in local variable
fstore_0	0x43	1		Store float in local variable 0
fstore_1	0x44	1		Store float in local variable 1
fstore_2	0x45	1		Store float in local variable 2
fstore_3	0x46	1		Store float in local variable 3
fsub	0x66	1	21	Subtract float
getfield	0xB4	3	23	Get field from object
getfield_quick	0xCE	3		Get field from obhect
getfield2_quick	0xD0	3		Get field from object
getstatic	0xB2	3	11	Get static field from class
getstatic_quick	0xD2	3		Get static field from class
getstatic2_quick	0xD4	3		Get static field from class
goto	0xA7	3		Branch always
goto_w	0xC8	5	23	Branch always (wide index)
i2b	0x91	1		Convert int to byte
i2c	0x92	1		Convert int to char
i2d	0x87	1	22	Convert int to double
i2f	0x86	1	21	Convert int to float
i2l	0x85	1		Convert int to long
i2s	0x93	1		Convert int to short
iadd	0x60	1		Add int
iaload	0x2E	1		Load int from array
iand	0x7E	1		Boolean AND int
iastore	0x4F	1		Store into int array
iconst_0	0x03	1		Push the integer value *0* on stack
iconst_1	0x04	1		Push the integer value *1* on stack
iconst_2	0x05	1		Push the integer value *2* on stack
iconst_3	0x06	1		Push the integer value *3* on stack
iconst_4	0x07	1		Push the integer value *4* on stack
iconst_5	0x08	1		Push the integer value *5* on stack

iconst_m1	0x02	1		Push the integer value *-1* on stack
idiv	0x6C	1		Divide int
if ge	0x9C	3		Branch if integer comparison TOS >= 0
if_acmpeq	0xA5	3		Branch if reference comparison TOS == TOS-1
if_acmpne	0xA6	3		Branch if reference comparison TOS != TOS-1
if_icmpeq	0x9F	3		Branch if integer comparison TOS == TOS-1
if_icmpge	0xA2	3		Branch if integer comparison TOS >= TOS-1
if_icmpgt	0xA3	3		Branch if integer comparison TOS > TOS-1
if_icmple	0xA4	3		Branch if integer comparison TOS <= TOS-1
if_icmplt	0xA1	3		Branch if integer comparison TOS < TOS-1
if_icmpne	0xA0	3		Branch if integer comparison TOS != TOS-1
ifeq	0x99	3		Branch if integer comparison TOS == 0
ifgt	0x9D	3		Branch if integer comparison TOS > 0
ifle	0x9E	3		Branch if integer comparison TOS <= 0
iflt	0x9B	3		Branch if integer comparison TOS < 0
ifne	0x9A	3		Branch if integer comparison TOS != 0
ifnonnull	0xC7	3		Branch if reference not null
ifnull	0xC6	3		Branch if reference is null
iinc	0x84	3	8	Increment local variable by constant
iload	0x15	2		Load int from local variable
iload_0	0x1A	1		Load int from local variable 0
iload_1	0x1B	1		Load int from local variable 1
iload_2	0x1C	1		Load int from local variable 2
iload_3	0x1D	1		Load int from local variable 3
imul	0x68	1		Multiply int
ineg	0x74	1		Negate int
instanceof	0xC1	3	23	Determine if object is of given type
instanceof_quick	0xE1	3	10	Determine if object is of given type
invokeinterface	0xB9	5	23	Invoke interface method
invokeinterface_quick	0xDA	5	15	Invoke a interface method
invokenonvirtual_quick	0xD7	3	18	Invoke instance initialization or private method
invokespecial	0xB7	3	23	Invoke instance method
invokestatic	0xB8	3	14	Invoke a class method
invokestatic_quick	0xD9	3	17	Invoke a class method
invokesuper_quick	0xD8	3	23	Invoke a class method
invokevirtual	0xB6	3	23	Invoke instance method
invokevirtual_quick	0xD6	3	16	Invoke instance method
invokevirtual_quick_w	0xE2	3	16	Invoke instance method

invokevirtualobject_quick	0xDB	3	23	Invoke instance method of class java.lang.Object
ior	0x80	1		Boolean OR int
irem	0x70	1		Remainder int
ireturn	0xAC	1	5	Return int from method
ishl	0x78	1		Shift left int
ishr	0x7A	1		Arithmetic shift right int
istore	0x36	2		Store int in local variable
istore_0	0x3B	1		Store int in local variable 0
istore_1	0x3C	1		Store int in local variable 1
istore_2	0x3D	1		Store int in local variable 2
istore_3	0x3E	1		Store int in local variable 3
isub	0x64	1		Subtract int
iushr	0x7C	1		Logical shift right int
ixor	0x82	1		Boolean XOR int
jsr	0xA8	3		Jump to subroutine
jsr_w	0xC9	5	23	Jump to subroutine (wide index)
l2d	0x8A	1	22	Convert long to double
l2f	0x89	1	21	Convert long to float
l2i	0x88	1		Convert long to int
ladd	0x61	1		Add long
laload	0x2F	1		Load long from array
land	0x7F	1		Boolean AND long
lastore	0x50	1		Store into long array
lcmp	0x94	1	20	Compare long
lconst_0	0x09	1		Push the long value *0* on stack
lconst_1	0x0A	1		Push the long value *1* on stack
ldc	0x12	2	23	Push item from constant pool
ldc_quick	0xCB	2		Push item from constant pool
ldc_w	0x13	3	23	Push item from constant pool
ldc_w_quick	0xCC	3		Push item from constant pool (wide index)
ldc2_w	0x14	3	23	Push long or double from constant pool
ldc2_w_quick	0xCD	3		Push long or double from constant pool (wide index)
ldiv	0x6D	1	20	Divide long
lload	0x16	2		Load long from local variable
lload_0	0x1E	1		Load long from local variable 0
lload_1	0x1F	1		Load long from local variable 1
lload_2	0x20	1		Load long from local variable 2
lload_3	0x21	1		Load long from local variable 3

lmul	0x69	1	20	Multiply long
lneg	0x75	1	20	Negate long
lookupswitch	0xAB	9+	23	Access jump table by key match and jump
lor	0x81	1		Boolean OR long
lrem	0x71	1	20	Remainder long
lreturn	0xAD	1	7	Return long from method
lshl	0x79	1	20	Shift left long
lshr	0x7B	1	20	Arithmetic shift right long
lstore	0x37	2		Store long in local variable
lstore_0	0x3F	1		Store long in local variable 0
lstore_1	0x40	1		Store long in local variable 1
lstore_2	0x41	1		Store long in local variable 2
lstore_3	0x42	1		Store long in local variable 3
lsub	0x65	1		Subtract long
lushr	0x7D	1	20	Logical shift right long
lxor	0x83	1		Boolean XOR long
monitorenter	0xC2	1	23	Enter monitor for object
monitorexit	0xC3	1	23	Exit monitor for object
multianewarray	0xC5	4	19	Create new multidimensional array
multianewarray_quick	0xDF	4	19	Create new multidimensional array
new	0xBB	3	23	Create a new object
new_quick	0xDD	3	13	Create a new object
newarray	0xBC	2	19	Create new array
nop	0x00	1		Do nothing
pop	0x57	1		Pop top operand stack word
pop2	0x58	1		Pop top two operand stack words
putfield	0xB5	3	23	Set field in object
putfield_quick	0xCF	3		Set field in object
putfield2_quick	0xD1	3		Set field in object
putstatic	0xB3	3	12	Set static field in class
putstatic_quick	0xD3	3		Set static field in class
putstatic2_quick	0xD5	3		Set static field in class
ret	0xA9	2		Return from subroutine
return	0xB1	1	6	Return void from method
saload	0x35	1		Load short from array
sastore	0x56	1		Store into short array
sipush	0x11	3		Push the short argument on the stack

swap	0x5F	1		Swap top two operand stack words
tableswitch	0xAA	13+	23	Access jump table by index and jump
wide	0xC4	4/6	23	Extend local variable index by additional bytes

4. Java Instruction Set Description

The following chapter describes the instructions in the java instruction set.

4.1 AALOAD – Load reference from array

4.1.1 Description

Index and *arrayref* is popped of the top of the stack, and the reference value at *index* is retrieved and pushed onto top of stack

4.1.2 Stack

..., arrayref, index →

..., value

4.1.3 Operation

```
if ( arrayref == NULL )
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
value ← *(addr+(index << 2));

JOSP --;
```

4.1.4 Syntax

I. aaload

4.1.5 Operands

none

4.1.6 Format

aaload (0x32)

4.1.7 Note

If *arrayref* is null, *aaload* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, *aaload* throws an ArrayIndexOutOfBoundsException.

4.2 AASTORE – Store into reference array

4.2.1 Description

Index, *arrayref* and *value* are popped from the operand stack. If the Handle bit is set *and* the address of the array is bigger or equal to the value of JBCR, the bytecode traps. If the bytecode doesn't trap, the reference value is stored as the component of array at *index.*

4.2.2 Stack

..., arrayref, index, value →

...

4.2.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11← *arrayref;
  if ( (R11 & 0x3FFFFFFC) >= JBCR )
  TRAP 0;
  addr ← (R11 & 0x3ffffffc);
  else
  addr ← arrayref;

  R12 ← *(addr-4);

if ( (index < 0)‖ (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 2)) ← value;
JOSP ← JOSP-3;
```

4.2.4 Syntax

I. aastore

4.2.5 Operands

none

4.2.6 Format

aastore (0x53)

4.2.7 Note

If *arrayref* is null, *aastore* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, the *aastore* instruction throws an ArrayIndexOutOfBoundsException.

4.3 ACONST_NULL – Push null

Description

Push the *null* object reference onto the operand stack

Stack

... →

..., null

Operation

```
JOSP ++;
ToS ←  0x0;
```

Syntax

I. aconst_null

Operands

none

Format

aconst_null (0x01)

Note

The Java Virtual Machine does not mandate a concrete value for null

4.4 ALOAD – Load reference from local variable

4.4.1 Description

The *objectref* in the local variable at *index* is pushed onto the operand stack

4.4.2 Stack

... →

..., objectref

4.4.3 Operation

```
JOSP++;
if ( index < 8 )
  objectref ← Sysreg(LV_<index>);
else
  objectref ← *(FRAME - (index << 2));
```

4.4.4 Syntax

I. aload

4.4.5 Operands

index

4.4.6 Format

aload (0x19)

4.4.7 Note

The *aload* instruction cannot be used to load a value of type returnAdress from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.5 ALOAD_<N> – Load reference from local variable

4.5.1 Description

The *objectref* in the local variable at *index* is pushed onto the operand stack.

4.5.2 Stack

... →

..., objectref

4.5.3 Operation

```
JOSP++;
```

I. objectref ← Sysreg(LV_0);

II. objectref ← Sysreg(LV_1);

III. objectref ← Sysreg(LV_2);

IV. objectref ← Sysreg(LV_3);

4.5.4 Syntax

I. aload_0

II. aload_1

III. aload_2

IV. aload_3

4.5.5 Operands

none

4.5.6 Format

aload_0 (0x2a)

aload_1 (0x2b)

aload_2 (0x2c)

aload_3 (0x2d)

4.5.7 Note

An *aload_<n>* instruction cannot be used to load a value of type returnAdress from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction is intentional. Each of the *aload_<n>* instructions is the same as aload with an *index* of *<n>*, except that the operand *<n>* is implicit.

4.6 ANEWARRAY – Create new array of reference

4.6.1 Description

The *count* is popped off the operand stack. The *count* represents the number of components of the array to be created. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class where the value of the index is (indexbyte1<<8) | *indexbyte2.* The item at that index in the constant pool must be tagged CONSTANT_Class, a symbolic reference to a class, array or interface type. The symbolic reference is resolved. A new array with components of that type, of length *count*, is allocated from the garbage-collected heap, and a reference *arrayref* to this new array object is pushed onto the operand stack. All components of the new array are initialized to null, the default value for reference types.

4.6.2 Stack

..., count →

..., arrayref

4.6.3 Operation

```
TRAP 23
```

4.6.4 Syntax

I. anewarray

4.6.5 Operands

indexbyte1, indexbyte2

4.6.6 Format

anewarray (0xbd)

4.6.7 Note

The *anewarray* instruction is used to create a single dimension of an array of object references. It can also be used to create part of a multidimensional array.

4.7 ANEWARRAY_QUICK – Create new array of reference

4.7.1 Description

The count must be of type int. It is popped off the operand stack. The count represents the number of components of the array to be created. The unsigned indexbyte1 and undexbyte 2 are used to construct an index into the constant pool of the current class. The item at that index must already be resolved and must be a class or interface type. A new array of that type, of length count, is allocated from the garbage-collected heap, and a reference arrayref to this new array object is pushed onto the operand stack. All components of the new array are initialized to null.

4.7.2 Stack

..., count →

..., arrayref

4.7.3 Operation

```
TRAP 19
```

4.7.4 Syntax

I. anewarray_quick

4.7.5 Operands

indexbyte1, indexbyte2

4.7.6 Format

anewarray_quick (0xde)

4.7.7 Note

The opcode of this instruction was originally anewarray.

4.8 APUTFIELD_QUICK – Set field in object to reference

4.8.1 Description

The *value* and *objectref* are popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an offset into the class instance referenced by *objectref*, where the offset is *(indexbyte1* << 8) | *indexbyte2*. If the Handle bit is set *and* the address of the field is bigger or equal to the value of JBCR, the bytecode traps. If the bytecode doesn't trap, the one-word field at the offset from the start of the object referenced by *objectref* is set to the *value.*

4.8.2 Stack

...,objectref, value →

...

4.8.3 Operation

```
if (objectref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *objectref;
  if ( (R11 & 0x3FFFFFFC) >= JBCR )
    TRAP 0;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← objectref;

*(addr+((indexbyte1 << 8 | indexbyte2) << 2)) ← value;

JOSP ← JOSP - 2;
```

4.8.4 Syntax

I. aputfield_quick

4.8.5 Operands

indexbyte1, indexbyte2

4.8.6 Format

putfield_quick (0xe7)

4.8.7 Note

If objectref is null, aputfield_quick throws a NullPointerException.

4.9 ARETURN – Return reference from method

4.9.1 Description

The *objectref* is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

4.9.2 Stack

..., objectref →

[empty]

4.9.3 Operation

```
TRAP 5
```

4.9.4 Syntax

I. areturn

4.9.5 Operand

none

4.9.6 Format

areturn (0xb0)

4.10 ARRAYLENGTH – Get length of array

4.10.1 Description

The *arrayref* is popped from the operand stack. The *length* of the array it references is determined. That *length* is pushed onto the operand stack as an int.

4.10.2 Stack

..., arrayref →

..., length

4.10.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

length ← *(addr-4);
```

4.10.4 Syntax

I. arraylength

4.10.5 Operands

none

4.10.6 Format

arraylength (0xbe)

4.10.7 Note

If the *arrayref* is null, the *arraylength* instruction throws a NullPointerException.

4.11 ASTORE – Store reference into local variable

4.11.1 Description

The *objectref* on the top of the operand stack must be of type returnAdress or of type reference. It is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

4.11.2 Stack

..., objectref→

...

4.11.3 Operation

```
if ( index < 8 )
  Sysreg(LV_<index>) ← objectref;
else
  *(Frame - (index << 2) ) ← objectref;
JOSP--;
```

4.11.4 Syntax

I. astore

4.11.5 Operands

index

4.11.6 Format

astore (0x3a)

4.11.7 Note

The astore instruction is used with an objectref of type returnAddress when implementing the finally clauses of the Java programming language. The *aload* instruction cannot be used to load a value of type returnAdress from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *astore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.12 ASTORE_<N> – Store reference into local variable

4.12.1 Description

The *objectref* on the top of the operand stack is popped, and the value of the local variable at <n> is set to *objectref*

4.12.2 Stack

..., objectref →

...

4.12.3 Operation

I. Sysreg(LV_0) ← objectref;

II. Sysreg(LV_1) ← objectref;

III. Sysreg(LV_2) ← objectref;

IV. Sysreg(LV_3) ← objectref;

```
JOSP--;
```

4.12.4 Syntax

I. astore_0

II. astore_1

III. astore_2

IV. astore_3

4.12.5 Operands

none

4.12.6 Format

astore_0 (0x4b)

astore_0 (0x4c)

astore_0 (0x4d)

astore_0 (0x4e)

4.12.7 Note

An *astore_<n>* instruction is used with an objectref of type return Address when implementing the finally clause of the Java programming language. An *aload_<n>* instruction cannot be used to load a value of type returnAdress from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction is intentional.

Each of the *astore_<n>* instructions is the same as *astore* with an *index* of <n>, except that the operand <n> is implicit.

4.13 ATHROW – Throw exception or error

4.13.1 Description

The *obectref* is popped from the operand stack. It is then thrown by searching the current frame for the most recent catch clause that catches the class of *objectref* or one of its superclasses.

If a catch is clause is found, it contains the location of the code intended to handle this exception. The pc register is reset to that location, the operand stack of the current frame is cleared, *objectref* is pushed back onto the operand stack, and execution continues. If no appropriate clause is found in the current frame, that frame is popped, the frame of its invoker is reinstated, and the *objectref* is rethrown.

If no catch clause is found that handles this exception, the current thread exits.

4.13.2 Stack

..., objectref →

objectref

4.13.3 Operation

```
TRAP 23
```

4.13.4 Syntax

I. athrow

4.13.5 Operands

none

4.13.6 Format

athrow (0xbf)

4.13.7 Note

If *objectref* is null, *athrows* throws a NullPointerException instead of *objectref.*

4.14 BALOAD – Load byte or boolean from array

4.14.1 Description

Both *arrayref* and *index* are popped from the operand stack. The byte *value* in the component of the array at *index* is retrieved, sign-extended to an int *value,* and pushed onto the top of the operand stack.

4.14.2 Stack

..., arrayref, index →

..., value

4.14.3 Operation

```
if ( arrayref == NULL )
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ←  *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
value ← SE( *(addr+index));

JOSP--;
```

4.14.4 Syntax

I. baload

4.14.5 Operands

none

4.14.6 Format

baload (0x33)

4.14.7 Note

If *arrayref* is null, *baload* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref,* the *baload* instruction throws an ArrayIndexOutOfBoundsException.

4.15 BASTORE – Store into byte or boolean array

4.15.1 Description

The *arrayref, index,* and *value* are popped from the operand stack. The int *value* is truncated to a byte and stored as the component of the array indexed by *index*.

4.15.2 Stack

..., arrayref, index, value →

...

4.15.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);
if ( (index < 0)  || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+index) ← value[7:0];
JOSP = JOSP-3;
```

4.15.4 Syntax

I. bastore

4.15.5 Operands

none

4.15.6 Format

bastore (0x54)

4.15.7 Note

If *arrayref* is null, bastore throws a NullPointerException.

If index is not within the bounds of the array referenced by *arrayref,* the *bastore* instruction throws an ArrayIndexOutOfBoundsException.

4.16 BIPUSH – Push byte

4.16.1 Description

The immediate *byte* is sign-extended to an int, and the resulting *value* is pushed onto the operand stack.

4.16.2 Stack

... →

..., value

4.16.3 Operation

```
JOSP++;
value ← SE(byte);
```

4.16.4 Syntax

I. bipush

4.16.5 Operands

byte

4.16.6 Format

bipush (0x10)

4.17 CALOAD – Load char from array

4.17.1 Description

Both *arrayref* and *index* are popped from the operand stack. The char *value* in the component of the array at *index* is retrieved, zero-extended to an int *value*, and pushed onto the top of the operand stack.

4.17.2 Stack

..., arrayref, index →

..., value

4.17.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;
  R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception
value ← ZE( *(addr+(index << 1)) );

JOSP--;
```

4.17.4 Syntax

I. caload

4.17.5 Operands

none

4.17.6 Format

caload (0x34)

4.17.7 Note

If *arrayref* is null, *caload* throws a NullPointerException.

If *index* is not within the bounds of array referenced by *arrayref,* the *caload* instruction throws an ArrayIndexOutOfBoundsException.

4.18 CASTORE – Store into char array

4.18.1 Description

The *arrayref, index,* and *value* are popped from the operand stack. The int *value* is truncated to a char and stored as the component of array indexed by *index.*

4.18.2 Stack

..., arrayref, index, value →

...

4.18.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ←(R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12))
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 1)) ← value[15:0];

JOSP = JOSP-3;
```

4.18.4 Syntax

I. castore

4.18.5 Operands

none

4.18.6 Format

castore (0x55)

4.18.7 Note

If *arrayref* is null, *castore* throws a NullPointerException.

If *index* is not within the bounds of array referenced by *arrayref,* the *castore* instruction throws an ArrayIndexOutOfBoundsException.

4.19 CHECKCAST – Check whether object is of given type

4.19.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the value of index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item at the index must be CONSTANT_Class, a symbolic reference to a class, array, or interface type. The symbolic reference is resolved.

4.19.2 Stack

..., objectref →

..., objectref

4.19.3 Operation

```
TRAP 23
```

4.19.4 Syntax

I. checkcast

4.19.5 Operands

indexbyte1, indexbyte2

4.19.6 Format

checkcast (0xc0)

4.19.7 Note

The *checkcast* instruction is very similar to the *instanceof* instruction. It differs in its treatment of null, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

4.20 CHECKCAST_QUICK – Check whether object is of given type

4.20.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the value of index is (*indexbyte1*<<8) | *indexbyte2*. The object at this index must already have been successfully resolved.

4.20.2 Stack

..., objectref →

..., objectref

4.20.3 Operation

```
TRAP 9
```

4.20.4 Syntax

I. checkcast_quick

4.20.5 Operands

indexbyte1, indexbyte2

4.20.6 Format

checkcast_quick (0xe0)

4.20.7 Note

The *checkcast_quick* instruction is very similar to the *instanceof* instruction. It differs in its treatment of null, its behavior when its test fails (*checkcast_quick* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

4.21 D2F – Convert double to float

4.21.1 Description

The *value* on the top of the operand stack is popped and converted to a float *result* using IEEE 754 round-to-nearest mode. The *result* is pushed onto the operand stack.

4.21.2 Stack

..., value.word1, value.word2 →

..., result

4.21.3 Operation

```
TRAP 22
```

4.21.4 Syntax

I. d2f

4.21.5 Operands

none

4.21.6 Format

d2f (0x90)

4.21.7 Note

The d2f instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value* and may also lose precision.

4.22 D2I – Convert double to int

4.22.1 Description

The *value* on the top of the operand stack is popped and converted to an int. The *result* is pushed onto the operand stack.

4.22.2 Stack

..., value.word1, value.word2 →

..., result

4.22.3 Operation

```
TRAP 22
```

4.22.4 Syntax

I. d2i

4.22.5 Operands

none

4.22.6 Format

d2i (0x8e)

4.22.7 Note

The *d2i* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value* and may also lose precision.

4.23 D2L – Convert double to long

4.23.1 Description

The *value* on the top of the operand stack is popped and converted to a long. The *result* is pushed onto the operand stack.

4.23.2 Stack

..., value.word1, value.word2 →

..., result.word1, result.word2

4.23.3 Operation

```
TRAP 22
```

4.23.4 Syntax

I. d2l

4.23.5 Operands

none

4.23.6 Format

d2l (0x8f)

4.23.7 Note

The *d2l* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value* and may also lose precision.

4.24 DADD – Add double

4.24.1 Description

The values are popped from the operand stack. The double *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

4.24.2 Stack

..., value1.word1, value1.word1, value2.word1, value2.word2 →

..., result.word1, result.word2

4.24.3 Operation

```
TRAP 22
```

4.24.4 Syntax

I. dadd

4.24.5 Operands

none

4.24.6 Format

dadd (0x63)

4.25 DALOAD – Load double from array

4.25.1 Description

Both *arrayref* and *index* are popped from the operand stack. The double *value* in the component of array at *index* is retrieved and pushed onto the top of the operand stack.

4.25.2 Stack

..., arrayref, index →

..., value.word1, value.word2

4.25.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ←  *(addr-4);

if ( (index < 0) || (index <= R12) )
  throw ArrayIndexOutOfBounds exception
value.word2:value.word1 ← *(addr+(index << 3));
```

4.25.4 Syntax

I. daload

4.25.5 Operands

none

4.25.6 Format

daload (0x31)

4.25.7 Note

If *arrayref* is null, *daload* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref,* the *daload* instruction throws an ArrayIndexOutOfBoundsException.

4.26 DASTORE – Store into double array

4.26.1 Description

The *arrayref, index* and *value* are popped from the operand stack. The double *value* is stored as the component of the array indexed by *index*.

4.26.2 Stack

..., arrayref, index, value.word1, value.word2 →

...

4.26.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 3)) ← value.word2:value.word1;
```

4.26.4 Syntax

I. dastore

4.26.5 Operands

none

4.26.6 Format

dastore (0x52)

4.26.7 Note

If *arrayref* is null, *dastore* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, the *dastore* instruction throws an ArrayIndexOutOfBoundsException.

4.27 DCMP<OP> – Compare double

4.27.1 Description

Both *value1* and *value2* is popped from the operand stack, and a float -point comparison is performed. If *value1* is greater than *value2*, the int value *1* is pushed onto the operand stack. If *value1* is equal to *value2*, the int value *0* is pushed onto the operand stack. If *value1* is less than *value2*, then int value *-1* is pushed onto the operand stack. If either *value1* or *value2* is Nan, the *dcmpg* instruction pushes the int value *1* onto the operand stack and the *dcmpl* instruction pushes the int value *-1*.

4.27.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result

4.27.3 Operation

```
TRAP 22
```

4.27.4 Syntax

I. dcmpg

II. dcmpl

4.27.5 Operands

none

4.27.6 Format

dcmpg (0x98)

dcmpl (0x97)

4.27.7 Note

The *dcmpg* and *dcmpl* instructions differ only in their treatment of a comparison involving NaN.

4.28 DCONST_<D> – Push double

4.28.1 Description

Push the double constant <d> onto the operand stack

4.28.2 Stack

... →

..., result.word1, result.word2

4.28.3 Operation

```
JOSP ← JOSP + 2;
```

I. result.word2:result.word1← 0x0000000000000000;

II. result.word2:result.word1← 0x3ff0000000000000;

4.28.4 Syntax

I. dconst_0

II. dconst_1

4.28.5 Operands

none

4.28.6 Format

dconst_0 (0x0e)

dconst_1 (0x0f)

4.29 DDIV – Divide double

4.29.1 Description

Both *value1* and *value2* are popped from the operand stack. The double *result* is *value1*/*value2*. The *result* is pushed onto the operand stack.

4.29.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.29.3 Operation

```
TRAP 22
```

4.29.4 Syntax

I. ddiv

4.29.5 Operands

none

4.29.6 Format

ddiv (0x6f)

4.30 DLOAD – Load double from local variable

4.30.1 Description

The local variables at *index* and *index* + 1 together must contain a double. The value of the local variables at *index* and *index* + 1 is pushed onto the operand stack.

4.30.2 Stack

... →

..., value.word1, value.word2

4.30.3 Operation

```
JOSP ← JOSP +2;
if (index <= 6 )
  value.word1 ← Sysreg(LV_<index>);
  value.word2 ← Sysreg(LV_(index+1));
else if (index == 7)
  value.word1 ← Sysreg(LV_7);
  value.word2 ← *(Frame - ((index+1) << 2));
else
  value.word2:value.word1 ← *(Frame - (index << 2));
```

4.30.4 Syntax

I. dload

4.30.5 Operands

index

4.30.6 Format

dload (0x18)

4.30.7 Note

The *dload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.31 DLOAD_<N> – Load double from local variable

4.31.1 Description

The local variables at *<n>* and *<n>* + 1 together must contain a double. The value of the local variables at *<n>* and *<n>* + 1is pushed onto the operand stack.

4.31.2 Stack

... →

..., value.word1, value.word2

4.31.3 Operation

```
JOSP ← JOSP +2;
```

I. value.word1 ← Sysreg(LV_0);
   value.word2 ← Sysreg(LV_1);

II. value.word1 ← Sysreg(LV_1);
   value.word2 ← Sysreg(LV_2);

III. value.word1 ← Sysreg(LV_2);
   value.word2 ← Sysreg(LV_3);

IV. value.word1 ← Sysreg(LV_3);
   value.word2 ← Sysreg(LV_4);

4.31.4 Syntax

I. dload_0

II. dload_1

III. dload_2

IV. dload_3

4.31.5 Operands

index

4.31.6 Format

dload_0 (0x26)

dload_1 (0x27)

dload_2 (0x28)

dload_3 (0x29)

4.31.7 Note

Each of the *dload_<n>* instructions is the same as *dload* with an *index* of <n>, except that the operand <n> is implicit.

4.32 DMUL – Multiply double

4.32.1 Description

Both *value1* and *value2* are popped from the operand stack. The double *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

4.32.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.32.3 Operation

```
TRAP 22
```

4.32.4 Syntax

I. dmul

4.32.5 Operands

none

4.32.6 Format

dmul (0x6b)

4.33 DNEG – Negate double

4.33.1 Description

The *value* is popped from the operand stack. The *result* is the arithmetic negation of *value*, namely -*value*. The *result* is pushed onto the operand stack.

4.33.2 Stack

..., value.word1, value.word2 →

..., result.word1, result.word2

4.33.3 Operation

```
TRAP 22
```

4.33.4 Syntax

I. dneg

4.33.5 Operands

none

4.33.6 Format

dneg (0x77)

4.34 DREM – Remainder double

4.34.1 Description

Both *value1* and *value2* are popped from the operating stack. The *result* is calculated and pushed onto the operand stack as a double.

4.34.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.34.3 Operation

```
TRAP 22
```

4.34.4 Syntax

I. drem

4.34.5 Operands

none

4.34.6 Format

drem (0x73)

4.34.7 Note

The result of a *drem* instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a trucating division, and so its behavior is not analogous to that of the usual integer remainder operator.

The IEEE 754 remainder operation may be computed by Java library routine Math.IEEEremainder

4.35 DRETURN – Return double from method

4.35.1 Description

The *value* is popped from the operand stack of the current frame and pushed onto the operand stack of the invoker. Any other values on the operand stack of the current method are discarded.

4.35.2 Stack

..., value.word1, value.word2 →

[empty]

4.35.3 Operation

```
TRAP 7
```

4.35.4 Syntax

I. dreturn

4.35.5 Operands

none

4.35.6 Format

dreturn (0xaf)

4.36 DSTORE – Store double into local variable

4.36.1 Description

The *value* is popped from the operating stack. The local variables at *index* and *index* + 1 are set to *value*.

4.36.2 Stack

..., value.word1, value.word2 →

...

4.36.3 Operation

```
if (index <= 6 )
  Sysreg(LV_<index>) ← value.word1;
  Sysreg(LV_(index+1)) ← value.word2;
else if (index == 7)
  Sysreg(LV_7) ← value.word1;
  *(Frame - ((index+1) << 2)) ← value.word2;
else
  *(Frame - (index << 2)) ← value.word2:value.word1;
JOSP ← JOSP - 2;
```

4.36.4 Syntax

I. dstore

4.36.5 Operands

index

4.36.6 Format

dstore (0x39)

4.36.7 Note

The *dstore* opcode can be used in conjunction with *wide* instruction to access a local using a two-byte unsigned index.

4.37 DSTORE_<n> – Store double into local variable

4.37.1 Description

The value on the top of the operand stack is popped and the local variables at <n> and <n> + 1 are set to value.

4.37.2 Stack

..., value.word1, value.word2 →

...

4.37.3 Operation

I. Sysreg(LV_0) ← value.word1;

Sysreg(LV_1) ← value.word2;

II. Sysreg(LV_1) ← value.word1;

Sysreg(LV_2) ← value.word2;

III. Sysreg(LV_2) ← value.word1;

Sysreg(LV_3) ← value.word2;

IV. Sysreg(LV_3) ← value.word1;

Sysreg(LV_4) ← value.word2;

```
JOSP ← JOSP - 2;
```

4.37.4 Syntax

I. dstore_0

II. dstore_1

III. dstore_2

IV. dstore_3

4.37.5 Operands

none

4.37.6 Format

dstore_0 (0x47)

dstore_1 (0x48)

dstore_2 (0x49)

dstore_3 (0x4a)

4.37.7 Note

Each of the *dstore_<n>* instructions is the same as *dstore* with an index <n>, except that the operand <n> is implicit.

4.38 DSUB – Subtract double

4.38.1 Description

The values are popped from the operand stack. The double *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

4.38.2 Stack

..., value1.word1, value1.word1, value2.word1, value2.word2 →

..., result.word1, result.word2

4.38.3 Operation

```
TRAP 22
```

4.38.4 Syntax

I. dsub

4.38.5 Operands

none

4.38.6 Format

dsub (0x67)

4.39 DUP – Duplicate top operand stack word

4.39.1 Description

The top word on the operand stack is duplicated and pushed onto the operand stack.

4.39.2 Stack

..., word1 →

..., word1, word1

4.39.3 Operation

```
JOSP++;
TOS ← word1;
```

4.39.4 Syntax

I. dup

4.39.5 Operands

none

4.39.6 Format

dup (0x59)

4.39.7 Note

The *dup* instruction must not be used unless word contains a 32-bit data type.

Except for restrictions preserving the integrity of 64-bit data types, the *dup* instruction operates on an untyped word, ignoring the type of datum it contains.

4.40 DUP_X1 – Duplicate top operand stack word and put two down

4.40.1 Description

The top word on the operand stack is duplicated and the copy inserted two words down in the operand stack.

4.40.2 Stack

..., word2, word1 →

..., word1, word2, word1

4.40.3 Operation

```
JOSP++;
TOS ← word1;
TOS-1 ← word2;
TOS-2 ← word1;
```

4.40.4 Syntax

I. dup_x1

4.40.5 Operands

none

4.40.6 Format

dup_x1 (0x5a)

4.40.7 Note

The *dup_x1* instruction must not be used unless each of *word1* and *word2* is a word that contains a 32-bit data type.

Except for restrictions preserving the integrity of 64-bit data types, the *dup_x1* instruction operates on untyped words, ignoring the types of datum they contain.

4.41 DUP_X2 – Duplicate top operand stack word and put three down

4.41.1 Description

The top word on the operand stack is duplicated and the copy inserted two words down in the operand stack.

4.41.2 Stack

..., word3, word2, word1 →

..., word1, word3, word2, word1

4.41.3 Operation

```
TRAP 23
```

4.41.4 Syntax

I. dup_x2

4.41.5 Operands

none

4.41.6 Format

dup_x2 (0x5b)

4.41.7 Note

The *dup_x2* instruction must not be used unless each of *word2* and *word3* is a word that contains a 32-bit data type or together are the two words of a single 64-bit datum, and unless *word1* contains a 32-bit data type.

Except for restrictions preserving the integrity of 64-bit data types, the *dup_x2* instruction operates on untyped words, ignoring the types of datum they contain.

4.42 DUP2 – Duplicate top two operand stack words

4.42.1 Description

The top two words on the operand stack are duplicated and pushed onto the operand stack, in the original order.

4.42.2 Stack

..., word2, word1 →

..., word2, word1, word2, word1

4.42.3 Operation

```
JOSP++;
TOS ← word1;
TOS-1 ← word2;
```

4.42.4 Syntax

I. dup2

4.42.5 Operands

none

4.42.6 Format

dup2 (0x5c)

4.42.7 Note

The *dup2* instruction must not be used unless each of *word1* and *word2* is a word that contains a 32-bit data type or both together are the two words of a single 64-bit datum.

Except for restrictions preserving the integrity of 64-bit data types, the *dup2* instruction operates on untyped words, ignoring the types of datum they contain.

4.43 DUP2_X1 – Duplicate top two operand stack words and put three down

4.43.1 Description

The top two words on the operand stack are duplicated and the copies inserted, in the original order, three words down in the operand stack.

4.43.2 Stack

..., word3, word2, word1 →

..., word2, word1, word3, word2, word1

4.43.3 Operation

```
TRAP 23
```

4.43.4 Syntax

I. dup2_x1

4.43.5 Operands

none

4.43.6 Format

dup2_x1 (0x5d)

4.43.7 Note

The *dup2_x1* instruction must not be used unless each of *word1* and *word2* is a word that contains a 32-bit data type or both together are the two words that contain a single 64-bit datum, and unless *word3* contains a 32-bit data type.

Except for restrictions preserving the integrity of 64-bit data types, the *dup2_x1* instruction operates on untyped words, ignoring the types of datum they contain.

4.44 DUP2_X2 – Duplicate top two operand stack words and put four down

4.44.1 Description

The top two words on the operand stack are duplicated and the copies inserted, in the original order, four words down in the operand stack.

4.44.2 Stack

..., word4, word3, word2, word1 →

..., word2, word1, word4, word3, word2, word1

4.44.3 Operation

```
TRAP 23
```

4.44.4 Syntax

I. dup2_x2

4.44.5 Operands

none

4.44.6 Format

dup2_x2 (0x5e)

4.44.7 Note

The *dup2_x2* instruction must not be used unless each of *word1* and *word2* is a 32-bit data type or both together are the two words of a single 64-bit datum, and unless *word3* and *word4* are each a word that contains a 32-bit data type or both together the two words of single 64-bit datum.

Except for restrictions preserving the integrity of 64-bit data types, the *dup2_x2* instruction operates on untyped words, ignoring the types of datum they contain.

4.45 F2D – Convert float to double

4.45.1 Description

The *value* on the top of the operand is popped and converted to a double. The *result* is pushed onto the operand stack.

4.45.2 Stack

..., value →

..., result.word1, result.word2

4.45.3 Operation

```
TRAP 21
```

4.45.4 Syntax

I. f2d

4.45.5 Operands

none

4.45.6 Format

f2d (0x8d)

4.45.7 Note

The f2d instruction performs a widening primitive conversion. Because all values of type float are exactly representable by type double, the conversion is exact.

4.46 F2I – Convert float to int

4.46.1 Description

The value on the top of the operand stack is popped an converted to an int. The result is pushed onto the operand stack.

4.46.2 Stack

..., value →

..., result

4.46.3 Operation

```
TRAP 21
```

4.46.4 Syntax

I. f2i

4.46.5 Operands

none

4.46.6 Format

f2i (0x8b)

4.46.7 Note

The *f2i* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*, and may also lose precision.

4.47 F2L – Convert float to long

4.47.1 Description

The *value* is popped from the operand stack and converted to a long. The *result* is pushed onto the operand stack.

4.47.2 Stack

..., value →

..., result.word1, result.word2

4.47.3 Operation

```
TRAP 21
```

4.47.4 Syntax

I. f2l

4.47.5 Operands

none

4.47.6 Format

f2l (0x8c)

4.47.7 Note

The *f2l* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*, and may also lose precision.

4.48 FADD – Add float

4.48.1 Description

Value1 and *value2* are popped from the operand stack. The float *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

4.48.2 Stack

..., value1, value2 →

..., result

4.48.3 Operation

```
TRAP 21
```

4.48.4 Syntax

I. fadd

4.48.5 Operands

none

4.48.6 Format

fadd (0x62)

4.49 FALOAD – Load float from array

4.49.1 Description

Both *arrayref* and *index* are popped from the operand stack. The float *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

4.49.2 Stack

..., arrayref, index →

..., value

4.49.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ←  *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
value ← *(addr+(index << 2));

JOSP--;
```

4.49.4 Syntax

I. faload

4.49.5 Operands

none

4.49.6 Format

faload (0x30)

4.49.7 Note

If *arrayref* is null, *faload* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref,* the *faload* instruction throws an ArrayIndexOutOfBoundsException.

4.50 FASTORE – Store into float array

4.50.1 Description

The *arrayref*, *index* and *value* are popped from the operand stack. The float *value* is stored as the component of the array indexed by *index*.

4.50.2 Stack

..., arrayref, index, value →

...

4.50.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 2)) ← value;

JOSP = JOSP-3;
```

4.50.4 Syntax

I. fastore

4.50.5 Operands

none

4.50.6 Format

fastore (0x51)

4.50.7 Note

If *arrayref* is null, *fastore* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref,* the *fastore* instruction throws an ArrayIndexOutOfBoundsException.

4.51 FCMP<OP> – Compare float

4.51.1 Description

Value1 and *value2* are popped from the operand stack, and a floating-point comparison is performed. If *value1* is greater than *value2*, the int value *1* is pushed onto the operand stack. If *value1* is equal to *value2*, the int value *0* is pushed onto the operand stack. If *value1* is less than *value2*, the int value *-1* is pushed onto the operand stack. If either *value1* or *value2* is NaN, the *fcmpg* instruction pushes the int value *1* onto the operand stack and the *fcmpl* instruction pushes the int value *-1*.

4.51.2 Stack

..., value1, value2 →

..., result

4.51.3 Operation

```
TRAP 21
```

4.51.4 Syntax

I. fcmpg

II. fcmpl

4.51.5 Operands

none

4.51.6 Format

fcmpg (0x96)

fcmpl (0x95)

4.51.7 Note

The *fcmpg* and *fcmpl* instruction differ only in their treatment of comparison involving NaN.

4.52 FCONST_<F> – Push float

4.52.1 Description

Push the float constant <f> onto the operand stack

4.52.2 Stack

... →

..., <f>

4.52.3 Operation

```
JOSP++;
```

I. <f> ← 0x00000000;

II. <f> ← 0x3f800000;

III. <f> ← 0x40000000;

4.52.4 Syntax

I. fconst_0

II. fconst_1

III. fconst_2

4.52.5 Operands

none

4.52.6 Format

fconst_0 (0x0b)

fconst_1 (0x0c)

fconst_2 (0x0d)

4.53 FDIV – Divide float

4.53.1 Description

Both *value1* and *value2* are popped from the operand stack. The float *result* is *value1* / *value2*. The *result* is pushed onto the operand stack.

4.53.2 Stack

..., value1, value2 →

..., result

4.53.3 Operation

```
TRAP 21
```

4.53.4 Syntax

I. fdiv

4.53.5 Operands

none

4.53.6 Format

fdiv (0x6e)

4.54 FLOAD – Load float from local variable

4.54.1 Description

The *value* of the local variable at *index* is pushed onto the operand stack.

4.54.2 Stack

... →

..., value

4.54.3 Operation

```
JOSP++;
if (index < 8)
  value ← Sysreg(LV_<index>);
else
  value ← *(Frame - (index << 2));
```

4.54.4 Syntax

I. fload

4.54.5 Operands

index

4.54.6 Format

fload (0x17)

4.54.7 Note

The *float* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.55 FLOAD_<N> – Load float from local variable

4.55.1 Description

The *value* of the local variable at <n> is pushed onto the operand stack.

4.55.2 Stack

... →

..., value

4.55.3 Operation

```
JOSP++;
```

I. value ← Sysreg(LV_0);

II. value ← Sysreg(LV_1);

III. value ← Sysreg(LV_2);

IV. value ← Sysreg(LV_3);

4.55.4 Syntax

I. fload_0

I. fload_1

I. fload_2

I. fload_3

4.55.5 Operands

none

4.55.6 Format

fload_0 (0x22)

fload_0 (0x23)

fload_0 (0x24)

fload_0 (0x25)

4.55.7 Note

Each of the *fload_<n>* instructions is the same as *fload* with an index <n>, except that the operand <n> is implicit.

4.56 FMUL – Multiply float

4.56.1 Description

Both *value1* and *value2* are popped from the operand stack. The float *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

4.56.2 Stack

..., value1, value2 →

..., result

4.56.3 Operation

```
TRAP 21
```

4.56.4 Syntax

I. fmul

4.56.5 Operands

none

4.56.6 Format

fmul (0x6a)

4.57 FNEG – Negate float

4.57.1 Description

The *value* is popped from the operand stack. The float *result* is the arithmetic negation of *value*, -*value*. The *result* is pushed onto the operand stack.

4.57.2 Stack

..., value →

..., result

4.57.3 Operation

```
TRAP 21
```

4.57.4 Syntax

I. fneg

4.57.5 Operands

none

4.57.6 Format

fneg (0x76)

4.58 FREM – Remainder float

4.58.1 Description

Both *value1* and *value2* are popped from the operand stack. The *result* is calculated and pushed onto the operand stack as float.

4.58.2 Stack

..., value1, value2 →

..., result

4.58.3 Operation

```
TRAP 21
```

4.58.4 Syntax

I. frem

4.58.5 Operands

none

4.58.6 Format

frem (0x72)

4.58.7 Note

The result of a *frem* instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a trucating division, and so its behavior is not analogous to that of the usual integer remainder operator.

The IEEE 754 remainder operation may be computed by Java library routine Math.IEEEremainder

4.59 FRETURN – Return float from method

4.59.1 Description

The value is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

4.59.2 Stack

..., value →

[empty]

4.59.3 Operation

```
TRAP 5
```

4.59.4 Syntax

I. freturn

4.59.5 Operands

none

4.59.6 Format

freturn (0xae)

4.60 FSTORE – Store float into local variable

4.60.1 Description

The *value* on the top of the operand stack is popped, and the value of the local variable at *index* is set to *value*.

4.60.2 Stack

..., value →

...

4.60.3 Operation

```
if (index < 8)
  Sysreg(LV_<index>) ← value;
else
  *(Frame - (index <<2)) ← value;
JOSP--;
```

4.60.4 Syntax

I. fstore

4.60.5 Operands

index

4.60.6 Format

fstore (0x38)

4.60.7 Note

The *fstore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.61 FSTORE_<N> – Store float into local variable

4.61.1 Description

The *value* on top of the operand stack is popped and the value of the local variable at <n> is set to *value*.

4.61.2 Stack

..., value →

...

4.61.3 Operation

I. Sysreg(LV_0) ← value;

II. Sysreg(LV_1) ← value;

III. Sysreg(LV_2) ← value;

IV. Sysreg(LV_3) ← value;

```
JOSP--;
```

4.61.4 Syntax

I. fstore_0

II. fstore_1

III. fstore_2

IV. fstore_3

4.61.5 Operands

none

4.61.6 Format

fstore_0 (0x43)

fstore_1 (0x44)

fstore_2 (0x45)

fstore_3 (0x46)

4.61.7 Note

Each of *fstore_<n>* is the same as *fstore* with an index of <n>, except that the operand <n> is implicit.

4.62 FSUB – Subtract float

4.62.1 Description

Both *value1* and *value2* are popped from the operand stack. The float *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

4.62.2 Stack

..., value1, value2 →

..., result

4.62.3 Operation

```
TRAP 21
```

4.62.4 Syntax

I. fsub

4.62.5 Operands

none

4.62.6 Format

fsub (0x66)

4.63 GETFIELD – Get field from object

4.63.1 Description

The *objectref* is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item at the index must be a CONSTANT_Fieldref, a reference to a class name and a field name. The item is resolved, determining both the field width and the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched and pushed onto the operand stack.

4.63.2 Stack

..., objectref →

..., value

OR

..., objectref →

..., value.word1, value.word2

4.63.3 Operation

```
TRAP 23
```

4.63.4 Syntax

I. getfield

4.63.5 Operands

indexbyte1, indexbyte2

4.63.6 Format

getfield (0xb4)

4.63.7 Note

If objectref is null, getfield throws a NullPointerException.

4.64 GETFIELD_QUICK – Get field from object

4.64.1 Description

The *objectref* is popped from the operand stack. The unsigned *offsetbyte1* and *offsetbyte2* are used to construct an offset into the class instance referenced by *objectref*, where the offset is *(offsetbyte1* << 8) | *offsetbyte2*. The one-word value at that offset is fetched and pushed onto the operand stack.

4.64.2 Stack

..., objectref →

..., value

4.64.3 Operation

```
if (objectref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *objectref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← objectref;

value ← *(addr+( ( (offsetbyte1<<8) | offsetbyte2) << 2) );
```

4.64.4 Syntax

I. getfield_quick

4.64.5 Operands

offsetbyte1, offsetbyte2

4.64.6 Format

getfield_quick (0xce)

4.64.7 Note

If objectref is null, getfield_quick throws a NullPointerException.

4.65 GETFIELD2_QUICK – Get field from object

4.65.1 Description

The *objectref* is popped from the operand stack. The unsigned *offsetbyte1* and *offsetbyte2* are used to construct an offset into the class instance referenced by *objectref*, where the offset is *(offsetbyte1* << 8) | *offsetbyte2*. The two-word value at that offset is fetched and pushed onto the operand stack.

4.65.2 Stack

..., objectref →

..., value.word1, value.word2

4.65.3 Operation

```
JOSP++;

if (objectref == NULL)
  throw NullPointer exception;
if (SR(H))
  R11 ← *objectref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← objectref;

value.word2:value.word1 ← *(addr+( ( (offsetbyte1<<8) | offsetbyte2) << 2)
);
```

4.65.4 Syntax

I. getfield2_quick

4.65.5 Operands

offsetbyte1, offsetbyte2

4.65.6 Format

getfield2_quick (0xd0)

4.65.7 Note

If objectref is null, getfield2_quick throws a NullPointerException.

4.66 GETSTATIC – Get static field from class

4.66.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item at the index must be a CONSTANT_Fieldref, a reference to a class name and a field name. The item is resolved, determining both the field width and the field offset. The *value* of the class field is fetched and pushed onto the operand stack.

4.66.2 Stack

...,→

..., value

OR

4.66.3 Stack

...,→

..., value.word1, value.word2

4.66.4 Operation

```
TRAP 11
```

4.66.5 Syntax

I. getstatic

4.66.6 Operands

indexbyte1, indexbyte2

4.66.7 Format

getstatic (0xb2)

4.67 GETSTATIC_QUICK – Get static field from class

4.67.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The value at that index is fetched and used as a pointer to the the one-word field, which is fetched and pushed onto the operand stack.

4.67.2 Stack

...,→

..., value

4.67.3 Operation

```
JOSP++;

R11 ← *(Const.Pool + ((indexbyte1 << 8 | indexbyte2) << 2));
value ← *R11;
```

4.67.4 Syntax

I. getstatic _quick

4.67.5 Operands

indexbyte1, indexbyte2

4.67.6 Format

getstatic_quick (0xd2)

4.68 GETSTATIC2_QUICK – Get static field from class

4.68.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The value at that index is fetched and used as a pointer to the the two-word field, which is fetched and pushed onto the operand stack.

4.68.2 Stack

...,→

..., value.word1, value.word2

4.68.3 Operation

```
JOSP ← JOSP + 2;
R11 ← *(Const.Pool + ((indexbyte1 << 8 | indexbyte2) << 2));
value.word2:value.word1 ← *(R11);
```

4.68.4 Syntax

I. getstatic2_quick

4.68.5 Operands

indexbyte1, indexbyte2

4.68.6 Format

getstatic2_quick (0xd4)

4.69 GOTO – Branch always

4.69.1 Description

The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1*<<8) | *branchbyte2*. Execution proceeds at that offset from address of the opcode of this goto instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

4.69.2 Stack

No change

4.69.3 Operation

```
PC ← PC + SE((branchbyte1 << 8) | branchbyte2)
```

4.69.4 yntax

I. goto

4.69.5 Operands

branchbyte1, branchbyte2

4.69.6 Format

goto (0xa7)

4.70 GOTO_W – Branch always (wide index)

4.70.1 Description

The unsigned bytes *branchbyte1*, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit *branchoffset*, where *branchoffset* is (*branchbyte1*<<24) | (*branchbyte2* << 16) | (*branchbyte3* <<8) | *branchbyte4*. Execution proceeds at that offset from the address of the opcode of this *goto_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto_w* instruction.

4.70.2 Stack

No change

4.70.3 Operation

```
TRAP 23
```

4.70.4 Syntax

I. goto_w

4.70.5 Operands

branchbyte1, branchbyte2, branchbyte3, branchbyte4

4.70.6 Format

goto_w (0xc8)

4.71 I2B – Convert int to byte

4.71.1 Description

The *value* from the top of the operand stack is popped, truncated to a byte, then sign extended to an int *result*. The *result* is pushed onto the operand stack.

4.71.2 Stack

..., value →

..., result

4.71.3 Operation

```
result ← SE(value[7:0]);
```

4.71.4 Syntax

I. i2b

4.71.5 Operands

none

4.71.6 Format

i2b (0x91)

4.71.7 Note

The *i2b* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign *value*.

4.72 I2C – Convert int to char

4.72.1 Description

The *value* from the top of the operand stack is popped, truncated to a char, then zero extended to an int *result*. The *result* is pushed onto the operand stack.

4.72.2 Stack

..., value →

..., result

4.72.3 Operation

```
result ← ZE(value[15:0]);
```

4.72.4 Syntax

I. i2c

4.72.5 Operands

none

4.72.6 Format

i2c (0x92)

4.72.7 Note

The *i2c* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign *value*.

4.73 I2D – Convert int to double

4.73.1 Description

The *value* from the top of the operand stack is popped and converted to a double *result*. The *result* is pushed onto the operand stack.

4.73.2 Stack

..., value →

..., result.word1, result.word2

4.73.3 Operation

```
TRAP 22
```

4.73.4 Syntax

I. i2d

4.73.5 Operands

none

4.73.6 Format

i2d (0x87)

4.73.7 Note

The *i2d* instruction performs a widening primitive conversion. Because all values of type int are exactly representable by type double, the conversion is exact.

4.74 I2F – Convert int to float

4.74.1 Description

The *value* from the top of the operand stack is popped and converted to the float *result*. The *result* is pushed onto the operand stack.

4.74.2 Stack

..., value →

..., result

4.74.3 Operation

```
TRAP 21
```

4.74.4 Syntax

I. i2f

4.74.5 Operands

none

4.74.6 Format

i2f (0x86)

4.74.7 Note

The *i2f* instruction performs a widening primitive conversion, but may result in a loss of precision because type float has only 24 mantissa bits.

4.75 I2L – Convert int to long

4.75.1 Description

The *value* from the top of the operand stack is popped and sign-extended to a long *result*. The *result* is pushed onto the operand stack.

4.75.2 Stack

..., value →

..., result.word1, result.word2

4.75.3 Operation

```
JOSP++;
result.word1 ← value;
result.word2 ← SE(value[31]);
```

4.75.4 Syntax

I. i2l

4.75.5 Operands

none

4.75.6 Format

i2l (0x85)

4.75.7 Note

The *i2l* instruction performs a widening primitive conversion. Because all values of type int are exactly representable by type long, the conversion is exact.

4.76 I2S – Convert int to short

4.76.1 Description

The *value* from the top of the operand stack is popped, truncated to a short, then signextended to an int *result*. The *result* is pushed onto the operand stack.

4.76.2 Stack

..., value →

..., result

4.76.3 Operation

```
result ← SE(value[15:0]);
```

4.76.4 Syntax

I. i2s

4.76.5 Operands

none

4.76.6 Format

i2s (0x93)

4.76.7 Note

The *i2s* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

4.77 IADD – Add int

4.77.1 Description

Both *value1* and *value2* are popped from the operand stack. The int *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

4.77.2 Stack

..., value1, value2 →

..., result

4.77.3 Operation

```
result ← value1 + value2;
JOSP--;
```

4.77.4 Syntax

I. iadd

4.77.5 Operands

none

4.77.6 Format

iadd (0x60)

4.78 IALOAD – Load integer from array

4.78.1 Description

Index and *arrayref* is popped of the top of the stack, and the integer value at index is retrieved and pushed onto top of stack

4.78.2 Stack

..., arrayref, index →

..., value

4.78.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ←  *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception
value ← *(addr+(index << 2));

JOSP--;
```

4.78.4 Syntax

I. iaload

4.78.5 Operands

none

4.78.6 Format

iaload (0x2e)

4.78.7 Note

If *arrayref* is null, *iaload* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, *iaload* throws an ArrayIndexOutOfBoundsException.

4.79 IAND – Boolean AND int

4.79.1 Description

Both *value1* and *value2* are popped from the operand stack. An int *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

4.79.2 Stack

..., value1, value2 →

..., result

4.79.3 Operation

```
result ← value1 & value2;
JOSP--;
```

4.79.4 Syntax

I. iand

4.79.5 Operands

none

4.79.6 Format

iand (0x7e)

4.80 IASTORE – Store into int array

4.80.1 Description

The *arrayref, index* and *value* are popped from the operand stack. The int *value* is stored as the component of the array indexed by *index.*

4.80.2 Stack

..., arrayref, index, value →

...

4.80.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11← *arrayref;
  addr ← (R11 & 0x3ffffffc);

else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 2)) ← value;

JOSP = JOSP-3;
```

4.80.4 Syntax

I. iastore

4.80.5 Operands

none

4.80.6 Format

iastore (0x4f)

4.80.7 Note

If *arrayref* is null, *iastore* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, the *iastore* instruction throws an ArrayIndexOutOfBoundsException.

4.81 ICONST_<i> – Push int constant

4.81.1 Description

Push the int constant <i> (-1, 0, 1, 2, 3, 4, or 5) onto the operand stack

4.81.2 Stack

... →

..., <i>

4.81.3 Operation

```
JOSP++;
```

I. <i> ← -1;

II. <i> ← 0;

III. <i> ← 1;

IV. <i> ← 2;

V. <i> ← 3;

VI. <i> ← 4;

VII. <i> ← 5;

4.81.4 Syntax

I. iconst_m1

II. iconst_0

III. iconst_1

IV. iconst_2

V. iconst_3

VI. iconst_4

VII. iconst_5

4.81.5 Operands

none

4.81.6 Format

iconst_m1 (0x02)

iconst_0 (0x03)

iconst_1 (0x04)

iconst_2 (0x05)

iconst_3 (0x06)

iconst_4 (0x07)

iconst_5 (0x08)

4.81.7 Note

Each of this family of instructions is equivalent to *bipush* <i> for the respective value of <i>, except that the operand <i> is implicit.

4.82 IDIV – Divide int

4.82.1 Description

Both *value1* and *value2* are popped from the operand stack. The int *result* is the value of the Java expression *value1* / *value2*. The *result* is pushed onto the operand stack.

4.82.2 Stack

..., value1, value2 →

..., result

4.82.3 Operation

```
if(value2==0)
  throw ArithmeticException;
result ← value1 / value2;
JOSP--;
```

4.82.4 Syntax

I. idiv

4.82.5 Operands

none

4.82.6 Format

idiv (0x6c)

4.82.7 Note

If the value of the divisor is 0, *idiv* throws an Arithmetic Exception.

4.83 IF_ACMP<COND> - Branch if reference comparison is not equal

4.83.1 Description

Both *value1* and *value2* are popped from the operand stack and compared. The results of the comparison are as follows:

eq - branch if and only if *value1 == value2*

ne - branch if and only if *value1* != *value2*

4.83.2 Stack

..., value1, value2 →

...

4.83.3 Operation - eq

```
if ( value1 == value2 )
  PC ← PC + SE ((branchbyte1 << 8) | branchbyte2);
else
  PC ← PC + 3;
JOSP = JOSP - 2;
```

4.83.4 Operation - ne

```
if ( value1 != value2 )
  PC ← PC + SE ((branchbyte1 << 8) | branchbyte2);
else
  PC ← PC + 3;
JOSP = JOSP - 2;
```

4.83.5 Syntax

I. if_acmpeq

II. if_acmpne

4.83.6 Operands

branchbyte1, branchbyte2

4.83.7 Format

if_acmpeq (0xa5)

if_acmpne (0xa6)

4.84 IF_ICMP<COND> – Branch if int comparison succeeds

4.84.1 Description

Both *value1* and *value2* are popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

eq - succeeds if and only if *value1* is equal to *value2*

ne - succeeds if and only if *value1* is not equal to *value2*

lt - succeeds if and only if *value1* is less than *value2*

le - succeeds if and only if *value1* is less or equal to *value2*

gt - succeeds if and only if *value1* is greater than *value2*

ge - succeeds if and only if *value1* is greater or equal to *value2*

4.84.2 Stack

..., value1, value2 →

...

4.84.3 Operation

```
if ( value1 <cond> value2)
  PC ← PC + SE ((branchbyte1 << 8) | branchbyte2);
else
  PC ← PC + 3;
JOSP = JOSP - 2;
```

4.84.4 Syntax

I. if_icmpeq

II. if_icmpne

III. if_icmplt

IV. if_icmple

V. if_icmpgt

VI. if_icmpge

4.84.5 Operands

branchbyte1, branchbyte2

4.84.6 Format

if_icmpeq (0x9f)

if_icmpne (0xa0)

if_icmplt (0xa1)

if_icmple (0xa4)

if_icmpgt (0xa3)

```
if_icmpge ( 0xa2 )
```

4.85 IF<COND> – Branch if int comparison with zero succeeds

4.85.1 Description

The *value* is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

eq - succeeds if and only if *value* is equal to *0*

ne - succeeds if and only if *value* is not equal to *0*

lt - succeeds if and only if *value* is less than *0*

le - succeeds if and only if *value* is less or equal to *0*

gt - succeeds if and only if *value* is greater than *0*

ge - succeeds if and only if *value* is greater or equal to *0*

4.85.2 Stack

..., value →

...

4.85.3 Operation

```
if (value <cond> 0)
  PC ← PC + SE ((branchbyte1 << 8) | branchbyte2);
else
  PC ← PC + 3;
JOSP--;
```

4.85.4 Syntax

I. ifeq

II. ifne

III. iflt

IV. ifle

V. ifgt

VI. ifge

4.85.5 Operands

branchbyte1, branchbyte2

4.85.6 Format

ifeq (0x99)

ifne (0x9a)

iflt (0x9b)

ifle (0x9e)

ifgt (0x9d)

```
ifge ( 0x9c )
```

4.86 IFNONNULL – Branch if reference not null

4.86.1 Description

The *value* is popped from the operand stack. If *value* is not null, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1<<8)* | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *ifnonnull* instruction. The target address must be of an instruction within the method that contains this *ifnonnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnonnull* instruction.

4.86.2 Stack

..., value →

...

4.86.3 Operation

```
if (value != 0x0)
  PC ← PC + SE ((branchbyte1 << 8) | branchbyte2);
else
  PC ← PC + 3;
JOSP--;
```

4.86.4 Syntax

I. ifnonnull

4.86.5 Operands

branchbyte1, branchbyte2

4.86.6 Format

ifnonnull (0xc7)

4.87 IFNULL – Branch if reference is null

4.87.1 Description

The *value* is popped from the operand stack. If *value* is null, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1<<8)* | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *ifnull* instruction. The target address must be of an instruction within the method that contains this *ifnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnull* instruction.

4.87.2 Stack

..., value →

...

4.87.3 Operation

```
if (value == 0x0)
  PC ← PC + SE ((branchbyte1 << 8) | branchbyte2);
else
  PC ← PC + 3;
JOSP--;
```

4.87.4 Syntax

I. ifnull

4.87.5 Operands

branchbyte1, branchbyte2

4.87.6 Format

ifnull (0xc6)

4.88 IINC – Increment local variable by constant

4.88.1 Description

The value *const* is first sign-extended to an int, then the local variable at *index* is incremented by that amount.

4.88.2 Stack

No change

4.88.3 Operation

```
if ( index < 8 )
  LV_<index> ← LV_<index> + SE(const);
else
  TRAP 8
```

4.88.4 Syntax

I. iinc

4.88.5 Operands

index, const

4.88.6 Format

iinc (0x84)

4.88.7 Note

The iinc opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte immediate value.

4.89 ILOAD – Load int from local variable

4.89.1 Description

The *value* of the local variable at *index* is pushed onto the operand stack

4.89.2 Stack

... →

..., value

4.89.3 Operation

JOSP++;

if (index < 8)

    value ← Sysreg(LV_<index>);

else

    value ← *(Frame - (index << 2));

4.89.4 Syntax

I. iload

4.89.5 Operands

index

4.89.6 Format

iload (0x15)

4.89.7 Note

The *iload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index

4.90 ILOAD_<N> – Load int from local variable

4.90.1 Description

The *value* of the local variable at <n> is pushed onto the operand stack

4.90.2 Stack

... →

..., value

4.90.3 Operation

```
JOSP++;
```

I. value ← Sysreg(LV_0);

II. value ← Sysreg(LV_1);

III. value ← Sysreg(LV_2);

IV. value ← Sysreg(LV_3);

4.90.4 Syntax

I. iload_0

II. iload_1

III. iload_2

IV. iload_3

4.90.5 Operands

none

4.90.6 Format

iload_0 (0x1a)

iload_1 (0x1b)

iload_2 (0x1c)

iload_3 (0x1d)

4.90.7 Note

Each of the *iload_<n>* instructions is the same as *iload* with an *index* of <n>, except that the operand <n> is implicit

4.91 IMUL – Multiply int

4.91.1 Description

Both *value1* and *value2* are popped from the operand stack. The int *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

4.91.2 Stack

..., value1, value2 →

..., result

4.91.3 Operation

```
result ← value1 * value2 ;
JOSP--;
```

4.91.4 Syntax

I. imul

4.91.5 Operands

none

4.91.6 Format

imul (0x68)

4.92 INEG – Negate int

4.92.1 Description

The *value* is popped from the operand stack. The int *result* is the arithmetic negation of *value*, -*value*. The *result* is pushed onto the operand stack.

4.92.2 Stack

..., value →

..., result

4.92.3 Operation

```
result ← 0 - value;
```

4.92.4 Syntax

I. ineg

4.92.5 Operands

none

4.92.6 Format

ineg (0x74)

4.93 INSTANCEOF – Determine if object is of given type

4.93.1 Description

The *objectref* is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of current class, where the value of the index is (*indexbyte1* << 8) | CONSTANT_Class, a symbolic reference to a class, array, or interface. The symbolic reference is resolved.

4.93.2 Stack

..., objectref →

..., result

4.93.3 Operation

```
TRAP 23
```

4.93.4 Syntax

I. instanceof

4.93.5 Operands

indexbyte1, indexbyte2

4.93.6 Format

instanceof (0xc1)¨

4.94 INSTANCEOF_QUICK – Determine if object is of given type

4.94.1 Description

The *objectref* is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of current class, where the value of the index is (*indexbyte1* << 8) | CONSTANT_Class, a symbolic reference to a class, array, or interface.

4.94.2 Stack

..., objectref →

..., result

4.94.3 Operation

```
TRAP 10
```

4.94.4 Syntax

I. instanceof_quick

4.94.5 Operands

indexbyte1, indexbyte2

4.94.6 Format

instanceof_quick (0xe1)

4.95 INVOKEINTERFACE – Invoke interface method

4.95.1 Description

The method table of the class of the type of *objectref* is determined. If *objectref* is an array type, then the method table of class Object is used. The method table is searched for a method whose name and descriptor are identical to the name and descriptor of the resolved constant pool entry.

The result of the search is a method table entry, which includes a direct reference to the code for the interface method and the method's modifier information. The method table entry be that of a public method.

4.95.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.95.3 Operation

```
TRAP 23
```

4.95.4 Syntax

I. invokeinterface

4.95.5 Operands

indexbyte1, indexbyte2, nargs, 0

4.95.6 Format

invokeinterface (0xb9)

4.96 INVOKEINTERFACE_QUICK – Invoke interface method

4.96.1 Description

4.96.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.96.3 Operation

```
TRAP 15
```

4.96.4 Syntax

I. invokeinterface_quick

4.96.5 Operands

indexbyte1, indexbyte2, nargs, 0

4.96.6 Format

invokeinterface_quick (0xda)

4.97 INVOKENONVIRTUAL_QUICK – invoke instance initialization or private method

4.97.1 Description

4.97.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.97.3 Operation

```
TRAP 18
```

4.97.4 Syntax

I. invokenonvirtual_quick

4.97.5 Operands

indexbyte1, indexbyte2

4.97.6 Format

invokenonvirtual_quick (0xd7)

4.98 INVOKESPECIAL – Invoke instance method

4.98.1 Description

Invoke instance method; special handling for superclass, private, and instance initialization method invocations

4.98.2 Stack

..., objectref, [arg1,[arg2 ...]] →

...

4.98.3 Operation

```
TRAP 23
```

4.98.4 Syntax

I. invokespecial

4.98.5 Operands

indexbyte1, indexbyte2

4.98.6 Format

invokespecial (0xb7)

4.98.7 Note

If the selected method exists but is a class (static) method, the *invokespecial* instruction throws an IncompatibleClassChangeError.

If the selected method is abstract, *invokespecial* throws an AbstractMethodError.

If the selected method is native and the code that implements the method cannot be loaded or linked, *invokespecial* throws an UnsatisfiedLinkError.

If *objectref* is null, the *invokespecial* instruction throws a NullPointerException.

4.99 INVOKESTATIC – Invoke a class method

4.99.1 Description

Invoke a class (static) method.

4.99.2 Stack

..., [arg1, [arg2 ...]] →

...

4.99.3 Operation

```
TRAP 14
```

4.99.4 Syntax

I. invokestatic

4.99.5 Operands

indexbyte1, indexbyte2

4.99.6 Format

invokestatic (0xb8)

4.99.7 Note

If the specified method exists but is an instancemethod, the *invokestatic* instruction throws an IncompatibleClassChangeError.

If the selected method is native and the code that implements the method cannot be loaded or linked, *invokestatic* throws an UnsatisfiedLinkError.

4.100 INVOKESTATIC_QUICK – Invoke a class method

4.100.1 Description

Invoke a class (static) method.

4.100.2 Stack

..., [arg1, [arg2 ...]] →

...

4.100.3 Operation

```
TRAP 17
```

4.100.4 Syntax

I. invokestatic_quick

4.100.5 Operands

indexbyte1, indexbyte2

4.100.6 Format

invokestatic_quick (0xd9)

4.101 INVOKESUPER_QUICK – Invoke a class method

4.101.1 Description

Invoke a class method.

4.101.2 Stack

..., [arg1, [arg2 ...]] →

...

4.101.3 Operation

```
TRAP 23
```

4.101.4 Syntax

I. invokesuper_quick

4.101.5 Operands

indexbyte1, indexbyte2

4.101.6 Format

invokesuper_quick (0xd8)

4.102 INVOKEVIRTUAL – Invoke instance method

4.102.1 Description

Invoke instance method; dispatch based on class

4.102.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.102.3 Operation

```
TRAP 23
```

4.102.4 Syntax

I. invokevirtual

4.102.5 Operands

indexbyte1, indexbyte2

4.102.6 Format

invokevirtual (0xb6)

4.102.7 Note

If the selected method exists but is a class (static) method, the *invokevirtual* instruction throws an IncompatibleClassChangeError.

If the specified method is abstract, *invokevirtual* throws an AbstractMethodError.

If the selected method is native and the code that implements the method cannot be loaded or linked, *invokevirtual* throws an UnsatisfiedLinkError.

If *objectref* is null, the *invokevirtual* instruction throws a NullPointerException.

4.103 INVOKEVIRTUAL_QUICK – Invoke instance method

4.103.1 Description

Invoke instance method; dispatch based on class

4.103.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.103.3 Operation

```
TRAP 16
```

4.103.4 Syntax

I. invokevirtual_quick

4.103.5 Operands

indexbyte1, indexbyte2

4.103.6 Format

invokevirtual_quick (0xd6)

4.103.7 Note

If the selected method exists but is a class (static) method, the *invokevirtual_quick* instruction throws an IncompatibleClassChangeError.

If the specified method is abstract, *invokevirtual_quick* throws an AbstractMethodError.

If the selected method is native and the code that implements the method cannot be loaded or linked, *invokevirtual_quick* throws an UnsatisfiedLinkError.

If *objectref* is null, the *invokevirtual_quick* instruction throws a NullPointerException.

4.104 INVOKEVIRTUAL_QUICK_W – Invoke instance method

4.104.1 Description

Invoke instance method; dispatch based on class

4.104.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.104.3 Operation

```
TRAP 23
```

4.104.4 Syntax

I. invokevirtual_quick_w

4.104.5 Operands

indexbyte1, indexbyte2

4.104.6 Format

invokevirtual_quick_w (0xe2)

4.104.7 Note

If the selected method exists but is a class (static) method, the *invokevirtual_quick_w* instruction throws an IncompatibleClassChangeError.

If the specified method is abstract, *invokevirtual_quick_w* throws an AbstractMethodError.

If the selected method is native and the code that implements the method cannot be loaded or linked, *invokevirtual_quick_w* throws an UnsatisfiedLinkError.

If *objectref* is null, the *invokevirtual_quick_w* instruction throws a NullPointerException.

4.105 INVOKEVIRTUALOBJECT_QUICK – Invoke method in java.lang.Object

4.105.1 Description

Invoke instance method in java.lang.Object instance.

4.105.2 Stack

..., objectref, [arg1, [arg2 ...]] →

...

4.105.3 Operation

```
TRAP 23
```

4.105.4 Syntax

I. invokevirtualobject_quick

4.105.5 Operands

indexbyte1, indexbyte2

4.105.6 Format

invokevirtualobject_quick (0xdb)

4.105.7 Note

If the selected method exists but is a class (static) method, the *invokevirtualobject_quick* instruction throws an IncompatibleClassChangeError.

If the specified method is abstract, *invokevirtualobject_quick* throws an AbstractMethodError.

If the selected method is native and the code that implements the method cannot be loaded or linked, *invokevirtualobject_quick* throws an UnsatisfiedLinkError.

If *objectref* is null, the *invokevirtualobject_quick* instruction throws a NullPointerException.

4.106 IOR – Boolean OR int

4.106.1 Description

Both *value1* and *value2* are popped from the operand stack. An int *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

4.106.2 Stack

..., value1, value2 →

..., result

4.106.3 Operation

```
result ← value1 | value2;
JOSP--;
```

4.106.4 Syntax

I. ior

4.106.5 Operands

none

4.106.6 Format

ior (0x80)

4.107 IREM – Remainder int

4.107.1 Description

Both *value1* and *value2* are popped from the operand stack. The int *result* is *value1* - (*value1/value2)*value2*. The *result* is pushed onto the operand stack.

4.107.2 Stack

..., value1, value2 →

..., result

4.107.3 Operation

```
if ( value2 == 0)
  throw ArithmeticException;
result ← value1 % value2;
JOSP--;
```

4.107.4 Syntax

I. irem

4.107.5 Operands

none

4.107.6 Format

irem (0x70)

4.107.7 Note

If the value of the divisor for an int remainder operator is 0, *irem* throws an ArithmeticException

4.108 IRETURN – Return int from method

4.108.1 Description

The *value* is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

4.108.2 Stack

..., value →

[empty]

4.108.3 Operation

```
TRAP 5
```

4.108.4 Syntax

I. ireturn

4.108.5 Operands

none

4.108.6 Format

ireturn (0xac)

4.109 ISHL – Shift left int

4.109.1 Description

Both *value1* and *value2* are popped from the operand stack. An int *result* is calculated by shifting *value1* left by s bit positions, where s is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

4.109.2 Stack

..., value1, value2 →

..., result

4.109.3 Operation

```
result ← (value1 << value2[4:0]);
JOSP--;
```

4.109.4 Syntax

I. ishl

4.109.5 Operands

none

4.109.6 Format

ishl (0x78)

4.110 ISHR – Arithmetic shift right int

4.110.1 Description

Both *value1* and *value2* are popped from the operand stack. An int *result* is calculated by shifting *value1* right by s bit positions, with sign extension, where s is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

4.110.2 Stack

..., value1, value2 →

..., result

4.110.3 Operation

```
result = SE( value1[31:value2[4:0]] );
JOSP--;
```

4.110.4 Syntax

I. ishr

4.110.5 Operands

none

4.110.6 Format

ishr (0x7a)

4.111 ISTORE – Store int into local variable

4.111.1 Description

The *value* on top of the operand stack is popped, and the value of the local variable at *index* is set to *value*.

4.111.2 Stack

..., value →

...

4.111.3 Operation

```
if (index < 8)
  Sysreg(LV_<index>) ← value;
else
  *(Frame - (index <<2)) ← value;
JOSP--;
```

4.111.4 Syntax

I. istore

4.111.5 Operands

index

4.111.6 Format

istore (0x36)

4.111.7 Note

The istore opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.112 ISTORE_<N> – Store int into local variable

4.112.1 Description

The *value* on top of the operand stack is popped, and the value of the local variable at <*n*> is set to *value*.

4.112.2 Stack

..., value →

...

4.112.3 Operation

I. Sysreg(LV_0) ← value;

II. Sysreg(LV_1) ← value;

III. Sysreg(LV_2) ← value;

IV. Sysreg(LV_3) ← value;

```
JOSP--;
```

4.112.4 Syntax

I. istore_0

II. istore_1

III. istore_2

IV. istore_3

4.112.5 Operands

none

4.112.6 Format

istore_0 (0x3b)

istore_1 (0x3c)

istore_2 (0x3d)

istore_3 (0x3e)

4.112.7 Note

Each of the *istore_<n>* instructions is the same as *istore* with an *index* of <n>, except that the operand <n> is implicit.

4.113 ISUB – Subtract int

4.113.1 Description

Both *value1* and *value2* are popped from the operand stack. The int *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

4.113.2 Stack

..., value1, value2 →

..., result

4.113.3 Operation

```
result ← value1 - value2;
JOSP--;
```

4.113.4 Syntax

I. isub

4.113.5 Operands

none

4.113.6 Format

isub (0x64)

4.114 IUSHR – Logical shift right int

4.114.1 Description

Both *value1* and *value2* are popped from the operand stack. An int *result* is calculated by shifting *value1* right by s bit positions, with zero extension, where s is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

4.114.2 Stack

..., value1, value2 →

..., result

4.114.3 Operation

```
result = ZE( value1[31:value2[4:0]] );
JOSP--;
```

4.114.4 Syntax

I. iushr

4.114.5 Operands

none

4.114.6 Format

iushr (0x7c)

4.115 IXOR – Boolean XOR int

4.115.1 Description

Both *value1* and *value2* are popped from the operand stack. An int *result* is calculated by taking the bitwise exclusive OR of *value1*and *value2*. The *result* is pushed onto the operand stack.

4.115.2 Stack

..., value1, value2 →

..., result

4.115.3 Operation

```
result ← value1 ^ value2;
JOSP--;
```

4.115.4 Syntax

I. ixor

4.115.5 Operands

none

4.115.6 Format

ixor (0x82)

4.116 JSR – Jump subroutine

4.116.1 Description

The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type returnAddress. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1<<8)* | *branchbyte2.* Execution then proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

4.116.2 Stack

... →

..., address

4.116.3 Operation

```
JOSP++;
address ← PC+3;
PC ← PC + SE((branchbyte1 <<8) | branchbyte2);
```

4.116.4 Syntax

I. jsr

4.116.5 Operands

branchbyte1, branchbyte2

4.116.6 Format

jsr (0xa8)

4.117 JSR_W – Jump subroutine (wide index)

4.117.1 Description

The *address* of the opcode of the instruction immediately following this *jsr_w* instruction is pushed onto the operand stack as a value of type returnAddress. The unsigned *branchbyte1, branchbyte2, branchbyte3 and branchbyte4* are used to construct a signed 32-bit offset, where the offset is calculated to be (*branchbyte1<<24)* | (*branchbyte2<<16)* | (*branchbyte3<<8)* | *branchbyte4.* Execution then proceeds at that offset from the address of this *jsr_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr_w* instruction.

4.117.2 Stack

..., →

..., address

4.117.3 Operation

```
TRAP 23
```

4.117.4 Syntax

I. jsr_w

4.117.5 Operands

branchbyte1, branchbyte2, branchbyte3, branchbyte4

4.117.6 Format

jsr_w (0xc9)

4.118 L2D – Convert long to double

4.118.1 Description

The *value* on top on top of the operand stack is popped and converted to a double *result* using IEEE 754 round-to-nearest mode. The *result* is pushed onto the operand stack.

4.118.2 Stack

..., value.word1, value.word2 →

..., result.word1, result.word2

4.118.3 Operation

```
TRAP 22
```

4.118.4 Syntax

I. l2d

4.118.5 Operands

none

4.118.6 Format

l2d (0x8a)

4.118.7 Note

The l2d instruction performs a widening primitive conversion that may lose precision because type double has only 53 mantissa bits.

4.119 L2F – Convert long to float

4.119.1 Description

The *value* on top on top of the operand stack is popped and converted to a float *result* using IEEE 754 round-to-nearest mode. The *result* is pushed onto the operand stack.

4.119.2 Stack

..., value.word1, value.word2 →

..., result.word

4.119.3 Operation

```
TRAP 21
```

4.119.4 Syntax

I. l2f

4.119.5 Operands

none

4.119.6 Format

l2f (0x89)

4.119.7 Note

The l2f instruction performs a widening primitive conversion that may lose precision because type float has only 24 mantissa bits.

4.120 L2I – Convert long to int

4.120.1 Description

The *value* on the top of the operand stack is popped and converted to an int *result* by taking the lower-order 32 bits of the long value and discarding the high-order 32 bits. The *result* is pushed onto the operand stack.

4.120.2 Stack

..., value.word1, value.word2 →

..., result

4.120.3 Operation

```
result = value.word1;
JOSP--;
```

4.120.4 Syntax

l. l2i

4.120.5 Operands

none

4.120.6 Format

l2i (0x88)

4.120.7 Note

The l2i instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value.* The *result* may also not have the same sign as *value.*

4.121 LADD – Add long

4.121.1 Description

Both *value1* and *value2* are popped from the operand stack. The long *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

4.121.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.121.3 Operation

```
result.word2:result.word2 ← value1.word2:value1.word2 +
value2.word2:value2.word1;
JOSP ← JOSP-2;
```

4.121.4 Syntax

I. ladd

4.121.5 Operands

none

4.121.6 Format

ladd (0x61)

4.122 LALOAD – Load long from array

4.122.1 Description

Both *arrayref* and *index* are popped from the operand stack. The long *value* in the component of the array at index is retrieved and pushed onto the top of the operand stack.

4.122.2 Stack

..., arrayref, index →

..., value.word1, value.word2

4.122.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ←  *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
value.word2:value.word2 ← *(addr+(index << 3));
```

4.122.4 Syntax

I. laload

4.122.5 Operands

none

4.122.6 Format

laload (0x2f)

4.122.7 Note

If *arrayref* is null, *laload* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, *laload* throws an ArrayIndexOutOfBoundsException.

4.123 LAND – Boolean AND long

4.123.1 Description

Both *value1* and *value2* are popped from the operand stack. A long *result* is calculated by taking the bitwise AND of *value1* and *value2*. The *result* is pushed onto the operand stack.

4.123.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.123.3 Operation

```
result.word2 ←(value1.word2 & value2.word2);
result.word1 ← (value1.word1 & value2.word1);
JOSP ← JOSP-2;
```

4.123.4 Syntax

I. land

4.123.5 Operands

none

4.123.6 Format

land (0x7f)

4.124 LASTORE – Store into long array

4.124.1 Description

The *arrayref*, *index* and *value* are popped from the operand stack. The long i is stored as the component of the array indexed by *index*.

4.124.2 Stack

..., arrayref, index, value.word1, value.word2 →

...

4.124.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 3)) ← value.word2:value.word1;

JOSP = JOSP - 3;
```

4.124.4 Syntax

I. lastore

4.124.5 Operands

none

4.124.6 Format

lastore (0x50)

4.124.7 Note

If *arrayref* is null, *lastore* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, *lastore* throws an ArrayIndexOutOfBoundsException.

4.125 LCMP – Compare long

4.125.1 Description

Both *value1* and *value2* is popped from the operand stack, and a signed integer comparison is performed. If *value1* is greater than *value2*, the int value *1* is pushed onto the operand stack. If *value1* is equal to *value2*, the int value *0* is pushed onto the operand stack. If *value1* is less than *value2*, then int value *-1* is pushed onto the operand stack.

4.125.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result

4.125.3 Operation

```
TRAP 20
```

4.125.4 Syntax

I. lcmp

4.125.5 Operands

none

4.125.6 Format

lcmp (0x94)

4.126 LCONST_<L> – Push long constant

4.126.1 Description

Push the long constant <l> (0 or 1) onto the operand stack

4.126.2 Stack

... →

..., <l>.word1, <l>.word2

4.126.3 Operation

```
JOSP ← JOSP + 2;
```

I. <l>.word2:<l>.word1 ← 0;

II. <l>.word2:<l>.word1 ← 1;

4.126.4 Syntax

I. lconst_0

II. lconst_1

4.126.5 Operands

none

4.126.6 Format

lconst_0 (0x09)

lconst_1 (0x0a)

4.127 LDC – Push item from constant pool

4.127.1 Description

The *item*, a reference to the instance, is pushed onto the operand stack.

4.127.2 Stack

... →

..., item

4.127.3 Operation

```
TRAP 23
```

4.127.4 Syntax

I. ldc

4.127.5 Operands

index

4.127.6 Format

ldc (0x12)

4.128 LDC_QUICK – Push item from constant pool

4.128.1 Description

The *item*, a reference to the instance, is pushed onto the operand stack.

4.128.2 Stack

... →

..., item

4.128.3 Operation

```
JOSP++;
item ← *((Const. Pool)+(index << 2));
```

4.128.4 Syntax

I. ldc_quick

4.128.5 Operands

index

4.128.6 Format

ldc (0xcb)

4.129 LDC_W – Push item from constant pool (wide index)

4.129.1 Description

The *item*, a reference to the instance, is pushed onto the operand stack.

4.129.2 Stack

... →

..., item

4.129.3 Operation

```
TRAP 23
```

4.129.4 Syntax

I. ldc_w

4.129.5 Operands

indexbyte1, indexbyte2

4.129.6 Format

ldc_w (0x13)

4.129.7 Note

The *ldc_w* instruction is identical to the *ldc* instruction except for its wider constant pool index.

4.130 LDC_W_QUICK – Push item from constant pool (wide index)

4.130.1 Description

The *item*, a reference to the instance, is pushed onto the operand stack.

4.130.2 Stack

... →

..., item

4.130.3 Operation

```
JOSP++;
item ← *((Const. Pool)+(indexbyte1 << 8 | indexbyte2)<<2);
```

4.130.4 Syntax

I. ldc_w_quick

4.130.5 Operands

indexbyte1, indexbyte2

4.130.6 Format

ldc_w_quick (0xcc)

4.130.7 Note

The *ldc_w* instruction is identical to the *ldc* instruction except for its wider constant pool index.

4.131 LDC2_W – Push long or double from constant pool (wide index)

4.131.1 Description

The *item*, a reference to the instance, is pushed onto the operand stack.

4.131.2 Stack

... →

..., item.word1, item.word2

4.131.3 Operation

```
TRAP 23
```

4.131.4 Syntax

I. ldc2_w

4.131.5 Operands

indexbyte1, indexbyte2

4.131.6 Format

ldc2_w (0x14)

4.132 LDC2_W_QUICK – Push long or double from constant pool (wide index)

4.132.1 Description

The *item*, a reference to the instance, is pushed onto the operand stack.

4.132.2 Stack

... →

..., item.word1, item.word2

4.132.3 Operation

```
JOSP ← JOSP+2;
item.word2:item.word1 ← *((Const. Pool)+(indexbyte1 << 8 | indexbyte2)<<2);
```

4.132.4 Syntax

I. ldc2_w_quick

4.132.5 Operands

indexbyte1, indexbyte2

4.132.6 Format

ldc2_w_quick (0xcd)

4.133 LDIV – Divide Long

4.133.1 Description

Both *value1* and *value2* is popped from the operand stack. The long *result* is the value of the Java expression *value1 / value2.* The *result* is pushed onto the operand stack.

4.133.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.133.3 Operation

```
TRAP 20
```

4.133.4 Syntax

I. ldiv

4.133.5 Operands

none

4.133.6 Format

ldiv (0x6d)

4.133.7 Note

If the value of the divisor in a long division is 0, *ldiv* throws an ArithmeticException.

4.134 LLOAD – Load long from local variable

4.134.1 Description

The *value* of the local variables at *index* and *index* + 1 is pushed onto the operand stack.

4.134.2 Stack

... →

..., value.word1, value.word2

4.134.3 Operation

```
JOSP ← JOSP +2;

if (index <= 6 )
  value.word1 ← Sysreg(LV_<index>);
  value.word2 ← Sysreg(LV_(index+1));
if (index == 7 )
  value.word1 ← Sysreg(LV_7);
  value.word2 ← *(Frame - ((index+1) << 2));
else
  value.word2:value.word1 ← *(Frame - (index << 2));
```

4.134.4 Syntax

I. lload

4.134.5 Operands

index

4.134.6 Format

lload (0x16)

4.134.7 Note

The lload opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.135 LLOAD_<N> – Load long from local variable

4.135.1 Description

The *value* of the local variables at <n> and <n> + 1 is pushed onto the operand stack.

4.135.2 Stack

... →

..., value.word1, value.word2

4.135.3 Operation

```
JOSP ← JOSP +2;
```

I. value.word1 ← Sysreg(LV_0);
   value.word2 ← Sysreg(LV_1);

II. value.word1 ← Sysreg(LV_1);
   value.word2 ← Sysreg(LV_2);

III. value.word1 ← Sysreg(LV_2);
   value.word2 ← Sysreg(LV_3);

IV. value.word1 ← Sysreg(LV_3);
   value.word2 ← Sysreg(LV_4);

4.135.4 Syntax

I. lload_0

II. lload_1

III. lload_2

IV. lload_3

4.135.5 Operands

none

4.135.6 Format

lload_0 (0x1e)

lload_0 (0x1f)

lload_0 (0x20)

lload_0 (0x21)

4.135.7 Note

Each of the *lload_<n>* instructions is the same as *lload* with an index of <n>, except that the operand <n> is implicit.

4.136 LMUL – Multiply long

4.136.1 Description

Both *value1* and *value2* are popped from the operand stack. The long *result* is *value1* value2.* The *result* is pushed onto the operand stack.

4.136.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.136.3 Operation

```
TRAP 20
```

4.136.4 Syntax

I. lmul

4.136.5 Operands

none

4.136.6 Format

lmul (0x69)

4.137 LNEG – Negate long

4.137.1 Description

The *value* is popped from the operand stack. The long *result* is the arithmetic negation of *value, -value.* The *result* is pushed onto the operand stack.

4.137.2 Stack

..., value.word1, value.word2 →

..., result.word1, result.word2

4.137.3 Operation

```
TRAP 20
```

4.137.4 Syntax

I. lneg

4.137.5 Operands

none

4.137.6 Format

lneg (0x75)

4.138 LOOKUPSWITCH – Access jump table by key match and jump

4.138.1 Description

The *key* is compared against the *match* values. If it is equal to one of them, then a target address is calculated by adding the corresponding *offset* to the address of the opcode of this *lookupswitch* instruction. If the *key* does not *match,* the target address is calculated by adding *default* to the address of the opcode of this *lookupswitch* instruction. Execution then continues at the target address.

4.138.2 Stack

..., key →

...

4.138.3 Operation

```
TRAP 23
```

4.138.4 Syntax

I. lookupswitch

4.138.5 Operands

<0-3 byte pad>,

defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4,

npairs1, npairs2, npairs3, npairs4,

match-offset pairs...

4.138.6 Format

lookupswitch (0xab)

4.139 LOR – Boolean OR long

4.139.1 Description

Both *value1* and *value2* are popped from the operand stack. A long *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2.* The *result* is pushed onto the operand stack.

4.139.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.139.3 Operation

```
result.word2 ← (value1.word2 | value2.word2);
result.word1 ← (value1.word1 | value2.word1);
JOSP ← JOSP-2;
```

4.139.4 Syntax

I. lor

4.139.5 Operands

none

4.139.6 Format

lor (0x81)

4.140 LREM – Remainder long

4.140.1 Description

Both *value1* and *value2* are popped from the operand stack. The long *result* is *value1 - (value1 / value2) * value2.* The *result* is pushed onto the operand stack.

4.140.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.140.3 Operation

```
TRAP 20
```

4.140.4 Syntax

I. lrem

4.140.5 Operands

none

4.140.6 Format

lrem (0x71)

4.140.7 Note

If the value of the divisor for a long remainder operator is 0, *lrem* throws an ArithmeticException.

4.141 LRETURN – Return long from method

4.141.1 Description

The *value* is popped from the operand stack of the current frame and pushed onto the operand stack of the invoker. Any other values on the operand stack of the current method are discarded.

4.141.2 Stack

..., value.word1, value.word2 →

[empty]

4.141.3 Operation

```
TRAP 7
```

4.141.4 Syntax

I. lreturn

4.141.5 Operands

none

4.141.6 Format

lreturn (0xad)

4.142 LSHL – Shift left long

4.142.1 Description

Both *value1* and *value2* are popped from the operand stack. A long *result* is calculated by shifting *value1* left by s bit positions, where s is the low six bits of *value2.* The *result* is pushed onto the operand stack.

4.142.2 Stack

..., value1.word1, value1.word2, value2 →

..., result.word1, result.word2

4.142.3 Operation

```
TRAP 20
```

4.142.4 Syntax

I. lshl

4.142.5 Operands

none

4.142.6 Format

lshl (0x79)

4.143 LSHR – Arithmetic shift right long

4.143.1 Description

Both *value1* and *value2* are popped from the operand stack. A long *result* calculated by shifting *value1* right by s bit positions, with sign extension, where s is the value of the low six bits of *value2.* The *result* is pushed onto the operand stack.

4.143.2 Stack

..., value1.word1, value1.word2, value2 →

..., result.word1, result.word2

4.143.3 Operation

```
TRAP 20
```

4.143.4 Syntax

I. lshr

4.143.5 Operands

none

4.143.6 Format

lshr (0x7b)

4.144 LSTORE – Store long into local variable

4.144.1 Description

The *value* is popped from the operand stack, and the local variable at *index* and *index + 1* are set to *value*.

4.144.2 Stack

..., value.word1, value.word2 →

...

4.144.3 Operation

```
if (index <= 6 )
  Sysreg(LV_<index>) ← value.word1;
  Sysreg(LV_(index+1)) ← value.word2;
else if (index == 7)
  Sysreg(LV_7) ← value.word1;
  *(Frame - ((index+1) << 2)) ← value.word2;
else
  *(Frame - (index << 2)) ← value.word2:value.word1;

JOSP ← JOSP -2;
```

4.144.4 Syntax

I. lstore

4.144.5 Operands

index

4.144.6 Format

lstore (0x37)

4.144.7 Note

The *lstore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.145 LSTORE_<N> – Store long into local variable

4.145.1 Description

The *value* is popped from the operand stack, and the local variables at <n> and <n> +1 are set to *value.*

4.145.2 Stack

..., value.word1, value.word2 →

...

4.145.3 Operation

I. Sysreg(LV_0) ← value.word1;
   Sysreg(LV_1) ← value.word2;

II. Sysreg(LV_1) ← value.word1;
   Sysreg(LV_2) ← value.word2;

III. Sysreg(LV_2) ← value.word1;
   Sysreg(LV_3) ← value.word2;

IV. Sysreg(LV_3) ← value.word1;
   Sysreg(LV_4) ← value.word2;

```
JOSP ← JOSP - 2;
```

4.145.4 Syntax

I. lstore_0

II. lstore_1

III. lstore_2

IV. lstore_3

4.145.5 Operands

none

4.145.6 Format

lstore_0 (0x3f)

lstore_1 (0x40)

lstore_2 (0x41)

lstore_3 (0x42)

4.145.7 Note

Each of the *lstore_<n>* instructions is the same as *lstore* with an *index* of <n>, except that the operand <n> is implicit.

4.146 LSUB – Subtract long

4.146.1 Description

Both *value1* and *value2* are popped from the operand stack. The long *result* is *value1 - value2.* The *result* is pushed onto the operand stack.

4.146.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.146.3 Operation

```
result.word2:result.word1 ← value1.word2:value1.word2 –
value2.word2:value2.word1;
JOSP ← JOSP-2;
```

4.146.4 Syntax

I. lsub

4.146.5 Operands

none

4.146.6 Format

lsub (0x65)

4.147 LUSHR – Logical shift right long

4.147.1 Description

Both *value1* and *value2* are popped from the operand stack. A long *result* calculated by shifting *value1* right logically (with zero extension) by the amount indicated by the low six bits of *value2.* The *result* is pushed onto the operand stack.

4.147.2 Stack

..., value1.word1, value1.word2, value2 →

..., result.word1, result.word2

4.147.3 Operation

```
TRAP 20
```

4.147.4 Syntax

I. lushr

4.147.5 Operands

none

4.147.6 Format

lushr (0x7d)

4.148 LXOR – Boolean XOR long

4.148.1 Description

Both *value1* and *value2* are popped from the operand stack. A long *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2.* The *result* is pushed onto the operand stack.

4.148.2 Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 →

..., result.word1, result.word2

4.148.3 Operation

```
result.word2 ← (value1.word2 ^ value2.word2);
result.word1 ← (value1.word1 ^ value2.word1);
JOSP ← JOSP-2;
```

4.148.4 Syntax

I. lxor

4.148.5 Operands

none

4.148.6 Format

lxor (0x83)

4.149 MONITORENTER – Enter monitor for object

4.149.1 Description

Each object has a monitor associated with it. The thread that executes *monitorenter* gains ownership of the monitor associated with *objectref*.

4.149.2 Stack

..., objectref →

...

4.149.3 Operation

```
TRAP 23
```

4.149.4 Syntax

I. monitorenter

4.149.5 Operands

none

4.149.6 Format

monitorenter (0xc2)

4.149.7 Note

If *objectref* is null, *monitorenter* throws a NullPointerException.

4.150 MONITOREXIT – Exit monitor for object

4.150.1 Description

The thread decrements the counter indicating the number of times it has entered this monitor. If as a result the value of the counter becomes zero, the current thread releases the monitor. If the monitor associated with *objectref* becomes free, other threads that are waiting to acquire that monitor are allowed to attempt to do so.

4.150.2 Stack

..., objectref →

...

4.150.3 Operation

```
TRAP 23
```

4.150.4 Syntax

I. monitorexit

4.150.5 Operands

none

4.150.6 Format

monitorexit (0xc3)

4.150.7 Note

If *objectref* is null, *monitorexit* throws a NullPointerException.

If the current thread is not the owner of the monitor, *monitorexit* throws an IllegalMonitorStateException.

4.151 MULTIANEWARRAY – Create new multidimensional array

4.151.1 Description

A new multidimensional array of the array type is allocated from the garbage-collected heap. The components of the first dimension of the array are initialized to subarrays of the type of the second dimension, and so on. The components of the first dimension of the array are initialized to the default initial value for the type of the components. A reference *arrayref* to the new array is pushed onto the operand stack.

4.151.2 Stack

..., count1, [count2, ...] →

..., arrayref

4.151.3 Operation

TRAP 19

4.151.4 Syntax

I. multianewarray

4.151.5 Operands

indexbyte1, indexbyte2, dimensions

4.151.6 Format

multianewarray (0xc5)

4.151.7 Note

If the current class does not have permission to access the base class of the resolved array class, *multianewarray* throws an IllegalAccessError.

If any of the *dimensions* values on the operand stack is less than zero, the *multianewarray* instruction throws a NegativeArraySizeException.

4.152 MULTIANEWARRAY_QUICK – Create new multidimensional array

4.152.1 Description

A new multidimensional array of the array type is allocated from the garbage-collected heap. The components of the first dimension of the array are initialized to subarrays of the type of the second dimension, and so on. The components of the first dimension of the array are initialized to the default initial value for the type of the components. A reference *arrayref* to the new array is pushed onto the operand stack.

4.152.2 Stack

..., count1, [count2, ...] →

..., arrayref

4.152.3 Operation

TRAP 19

4.152.4 Syntax

I. multianewarray_quick

4.152.5 Operands

indexbyte1, indexbyte2, dimensions

4.152.6 Format

multianewarray_quick (0xdf)

4.152.7 Note

If the current class does not have permission to access the base class of the resolved array class, *multianewarray_quick* throws an IllegalAccessError.

If any of the *dimensions* values on the operand stack is less than zero, the *multianewarray_quick* instruction throws a NegativeArraySizeException.

4.153 NEW – Create new object

4.153.1 Description

The *indexbyte1* and *indexbyte2* are used to construct an index. The item at that index in the constant pool must be a CONSTANT_Class. The symbolic reference is resolved and must result in a class type. Memory for the new object is allocated. Instance variables of the new object are initialized to their default initial values. The *objectref,* a reference to the instance, is pushed onto the operand stack.

4.153.2 Stack

... →

..., objectref

4.153.3 Operation

```
TRAP 23
```

4.153.4 Syntax

I. new

4.153.5 Operands

indexbyte1, indexbyte2

4.153.6 Format

new (0xbb)

4.153.7 Note

If the CONSTANT_Class constant pool item resolves to an interface or is an abstract class, *new* throws an InstantiationError.

If the current class does not have permission to access the resolved class, *new* throws an IllegalAccessError.

4.154 NEW_QUICK – Create new object

4.154.1 Description

The *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index must have already been resolved to a class type. A new instance of that class is created.

4.154.2 Stack

... →

..., objectref

4.154.3 Operation

```
TRAP 13
```

4.154.4 Syntax

I. new_quick

4.154.5 Operands

indexbyte1, indexbyte2

4.154.6 Format

new_quick (0xdd)

4.155 NEWARRAY – Create new array

4.155.1 Description

The value *count* is popped from the operand stack. The *count* represents the number of elements in the array to be created. The *atype* is a code that indicates the type of array to create. A new array whose components are of type *atype*, of length *count,* is allocated from the garbage-collected heap. A reference *arrayref* to this new array object is pushed into the operand stack. All of the elements of the new array are initialized to the default initial values for its type.

4.155.2 Stack

..., count →

..., arrayref

4.155.3 Operation

```
TRAP 19
```

4.155.4 Syntax

I. newarray

4.155.5 Operands

atype

4.155.6 Format

newarray (0xbc)

4.155.7 Note

If count is less than zero, *newarray* throws a NegativeArraySizeException.

4.156 NOP – Do nothing

4.156.1 Description

Do nothing

4.156.2 Stack

No change

4.156.3 Operation

```
none
```

4.156.4 Syntax

I. nop

4.156.5 Operands

none

4.156.6 Format

nop (0x00)

4.157 POP – Pop top operand stack word

4.157.1 Description

The top word is popped from the operand stack.

4.157.2 Stack

..., word →

...

4.157.3 Operation

```
JOSP--;
```

4.157.4 Syntax

l. pop

4.157.5 Operands

none

4.157.6 Format

pop (0x57)

4.158 POP2 – Pop top two operand stack words

4.158.1 Description

The top words are popped from the operand stack.

4.158.2 Stack

..., word2, word1 →

...

4.158.3 Operation

```
JOSP = JOSP - 2;
```

4.158.4 Syntax

I. pop2

4.158.5 Operands

none

4.158.6 Format

pop2 (0x58)

4.159 PUTFIELD – Set field in object

4.159.1 Description

The *objectref* is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item at the index must be a CONSTANT_Fieldref, a reference to a class name and a field name. The item is resolved, determining both the field width and the field offset. The *value* and *objectref* are popped from the operand stack, and the field at the offset from the start of the object referenced by *objectref* is set to *value*.

4.159.2 Stack

...,objectref, value →

...

OR

...,objectref, value.word1, value.word2 →

...

4.159.3 Operation

```
TRAP 23
```

4.159.4 Syntax

I. putfield

4.159.5 Operands

indexbyte1, indexbyte2

4.159.6 Format

putfield (0xb5)

4.159.7 Note

If objectref is null, putfield throws a NullPointerException.

4.160 PUTFIELD_QUICK – Set field in object

4.160.1 Description

The *value* and *objectref* are popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an offset into the class instance referenced by *objectref*, where the offset is *(indexbyte1* << 8) | *indexbyte2*. The one-word field at the offset from the start of the object referenced by *objectref* is set to the *value.*

4.160.2 Stack

...,objectref, value →

...

4.160.3 Operation

```
if (objectref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *objectref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← objectref;

*(addr+((indexbyte1 << 8 | indexbyte2) << 2)) ← value;

JOSP ← JOSP - 2;
```

4.160.4 Syntax

I. putfield_quick

4.160.5 Operands

indexbyte1, indexbyte2

4.160.6 Format

putfield_quick (0xcf)

4.160.7 Note

If objectref is null, putfield_quick throws a NullPointerException.

4.161 PUTFIELD2_QUICK – Set field in object

4.161.1 Description

The *value* and *objectref* are popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an offset into the class instance referenced by *objectref*, where the offset is *(indexbyte1* << 8) | *indexbyte2*. The two-word field at the offset from the start of the object referenced by *objectref* is set to the *value.*

4.161.2 Stack

...,objectref, value.word1, value.word2 →

...

4.161.3 Operation

```
if (objectref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *objectref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← objectref;

*(addr+((indexbyte1 << 8 | indexbyte2) << 2)) ← value.word2value.word1;

JOSP ← JOSP - 3;
```

4.161.4 Syntax

I. putfield2_quick

4.161.5 Operands

indexbyte1, indexbyte2

4.161.6 Format

putfield2_quick (0xd1)

4.161.7 Note

If objectref is null, putfield2_quick throws a NullPointerException.

4.162 PUTSTATIC – Set static field in class

4.162.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The constant pool item at the index must be a CONSTANT_Fieldref, a reference to a class name and a field name. The item is resolved, determining both the field width and the field offset. The *value* is popped from the operand stack, and the class field is set to *value*.

4.162.2 Stack

..., value →

...

4.162.3 Operation

TRAP 12

4.162.4 Syntax

I. putstatic

4.162.5 Operands

indexbyte1, indexbyte2

4.162.6 Format

putstatic (0xb3)

4.162.7 Note

If the specified field exists but is not a static field (class variable), *putstatic* throws an IncompatibleClassChangeError.

4.163 PUTSTATIC_QUICK – Set static field in class

4.163.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The data that index is fetched and used as a pointer to the the one-word field. The *value* is popped from the operand stack, written to the field.

4.163.2 Stack

..., value →

...

4.163.3 Operation

```
R11 ← *(Const.Pool + ((indexbyte1 << 8 | indexbyte2) << 2));
*(R11) ← value;
JOSP--;
```

4.163.4 Syntax

I. putstatic_quick

4.163.5 Operands

indexbyte1, indexbyte2

4.163.6 Format

putstatic_quick (0xd3)

4.164 PUTSTATIC2_QUICK – Set static field in class

4.164.1 Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class, where the index is (*indexbyte1*<<8) | *indexbyte2*. The data that index is fetched and used as a pointer to the the two-word field. The *value* is popped from the operand stack, written to the field.

4.164.2 Stack

..., value.word1, value.word2 →

...

4.164.3 Operation

```
R11 ← *(Const.Pool + ((indexbyte1 << 8 | indexbyte2) << 2));
*(R11) ← (value.word2):value.word1;

JOSP ← JOSP - 2;
```

4.164.4 Syntax

I. putstatic

4.164.5 Operands

indexbyte1, indexbyte2

4.164.6 Format

putstatic2_quick (0xd5)

4.165 RET – Return from subroutine

4.165.1 Description

The contents of the local variable at *index* in the current frame are written into the pc register, and execution continues there.

4.165.2 Stack

No change

4.165.3 Operation

```
if (index < 8)
  PC ← Sysreg(LV_<index>);
else
  PC ← *(Frame + (index << 2));
```

4.165.4 Syntax

I. ret

4.165.5 Operands

index

4.165.6 Format

ret (0xa9)

4.165.7 Note

The *ret* instruction should not be confused with the *return* instruction. A *return* instruction returns control from a Java method to its invoker, without passing any value back to the invoker.

The *ret* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

4.166 RETURN – Return void from method

4.166.1 Description

Any values on the operand stack of the current frame are discarded. The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

4.166.2 Stack

... →

[empty]

4.166.3 Operation

```
TRAP 6
```

4.166.4 Syntax

I. return

4.166.5 Operands

none

4.166.6 Format

return (0xb1)

4.167 SALOAD – Load short from array

4.167.1 Description

Both *arrayref* and *index* are popped from the operand stack. The short *value* in the component of the array at *index* is retrieved, sign-extended to an int *value,* and pushed onto the top of the operand stack.

4.167.2 Stack

..., arrayref, index →

..., value

4.167.3 Operation

```
if ( arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
value ← SE(*(addr+(index << 1)));

JOSP--;
```

4.167.4 Syntax

I. saload

4.167.5 Operands

none

4.167.6 Format

saload (0x35)

4.167.7 Note

If *arrayref* is null, *saload* throws a NullPointerException.

If index is not within bounds of the array referenced by *arrayref,* the *saload* instruction throws an ArrayIndexOutOfBoundsException.

4.168 SASTORE – Store into short array

4.168.1 Description

The *arrayref, index,* and *value* are popped from the operand stack. The int *value* is truncated to a short and stored as the component of the array indexed by *index.*

4.168.2 Stack

..., array, index, value →

...

4.168.3 Operation

```
if (arrayref == NULL)
  throw NullPointer exception;
if ( SR(H) )
  R11 ← *arrayref;
  addr ← (R11 & 0x3ffffffc);
else
  addr ← arrayref;

R12 ← *(addr-4);

if ( (index < 0) || (index >= R12) )
  throw ArrayIndexOutOfBounds exception;
*(addr+(index << 1)) ← value[15:0];

JOSP = JOSP-3;
```

4.168.4 Syntax

I. sastore

4.168.5 Operands

none

4.168.6 Format

sastore (0x56)

4.168.7 Note

If *arrayref* is null, *sastore* throws a NullPointerException.

If *index* is not within the bounds of the array referenced by *arrayref*, the *sastore* instruction throws an ArrayIndexOutOfBoundsException.

4.169 SIPUSH – Push short

4.169.1 Description

The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate short where the value of the short is (*byte1*<<8) | *byte2*. The intermediate value is then sign-extended to an int, and the resulting *value* is pushed onto the operand stack.

4.169.2 Stack

... →

..., value

4.169.3 Operation

```
JOSP++;
value ← SE((byte1<<8) | byte2)
```

4.169.4 Syntax

I. sipush

4.169.5 Operands

byte1, byte2

4.169.6 Format

sipush (0x35)

4.170 SWAP – Swap top two operand stack words

4.170.1 Description

The top two words on the operand stack are swapped.

4.170.2 Stack

..., word2, word1 →

..., word1, word2

4.170.3 Operation

R11 ← word1;

TOS ← word2;

TOS-1 ← R11;

4.170.4 Syntax

I. swap

4.170.5 Operands

none

4.170.6 Format

swap (0x5f)

4.171 TABLESWITCH – Access jump table by index and jump

4.171.1 Description

The *index* is popped from the operand stack. If *index* is less than *low* or greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *tableswitch* instruction. Otherwise, the offset at position *index* - *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *tableswitch* instruction. Execution then continues at the target address.

4.171.2 Stack

..., index →

...

4.171.3 Operation

TRAP 23

4.171.4 Syntax

I. tableswitch

4.171.5 Operands

<0-3 byte pad>,

defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4,

lowbyte1, lowbyte2, lowbyte3, lowbyte4,

highbyte1, highbyte2, highbyte3, highbyte4,

jump offsets...

4.171.6 Format

tableswitch (0xaa)

4.172 WIDE – Extend local variable index by additional bytes

4.172.1 Description

The *wide* instruction modifies the behavior of another instruction. It takes one of two formats, depending on the instruction being modified. The first form of the *wide* instruction modifies one of the instructions *iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore,* or *ret*. The second form applies only to the *iinc* instruction.

4.172.2 Stack

Same as modified instruction

4.172.3 Operation

```
TRAP 23
```

4.172.4 Syntax

I. wide

4.172.5 Operands 1

<opcode>, indexbyte1, indexbyte2

- where <opcode> is one of *iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore,* or *ret*.

4.172.6 Operands 2

iinc, indexbyte1, indexbyte2, constbyte1, constbyte2

4.172.7 Format

wide (0xc4)

Table of Contents