Bytecode for the Dalvik VM

General Design

- The machine model and calling conventions are meant to approximately imitate common real architectures and C-style calling conventions:
  - The VM is register-based, and frames are fixed in size upon creation. Each frame consists of a particular number of registers (specified by the method) as well as any adjunct data needed to execute the method, such as (but not limited to) the program counter and a reference to the `.dex` file that contains the method.
  - When used for bit values (such as integers and floating point numbers), registers are considered 32 bits wide. Adjacent register pairs are used for 64-bit values. There is no alignment requirement for register pairs.
  - When used for object references, registers are considered wide enough to hold exactly one such reference.
  - In terms of bitwise representation, `(Object) null == (int) 0`.
  - The *N* arguments to a method land in the last *N* registers of the method's invocation frame, in order. Wide arguments consume two registers. Instance methods are passed a `this` reference as their first argument.
- The storage unit in the instruction stream is a 16-bit unsigned quantity. Some bits in some instructions are ignored / must-be-zero.
- Instructions aren't gratuitously limited to a particular type. For example, instructions that move 32-bit register values without interpretation don't have to specify whether they are moving ints or floats.
- There are separately enumerated and indexed constant pools for references to strings, types, fields, and methods.
- Bitwise literal data is represented in-line in the instruction stream.
- Because, in practice, it is uncommon for a method to need more than 16 registers, and because needing more than eight registers *is* reasonably common, many instructions are limited to only addressing the first 16 registers. When reasonably possible, instructions allow references to up to the first 256 registers. In addition, some instructions have variants that allow for much larger register counts, including a pair of catch-all `move` instructions that can address registers in the range `v0` – `v65535`. In cases where an instruction variant isn't available to address a desired register, it is expected that the register contents get moved from the original register to a low register (before the operation) and/or moved from a low result register to a high register (after the operation).
- There are several "pseudo-instructions" that are used to hold variable-length data payloads, which are referred to by regular instructions (for example, `fill-array-data`). Such instructions must never be encountered during the normal flow of execution. In addition, the instructions must be located on even-numbered bytecode offsets (that is, 4-byte aligned). In order to meet this requirement, dex generation tools must emit an extra `nop` instruction as a spacer if such an instruction would otherwise be unaligned. Finally, though not required, it is expected that most tools will choose to emit these instructions at the ends of methods, since otherwise it would likely be the case that additional instructions would be needed to branch around them.
- When installed on a running system, some instructions may be altered, changing their format, as an install-time static linking optimization. This is to allow for faster execution once linkage is known. See the associated [instruction formats document] for the suggested variants. The word "suggested" is used advisedly; it is not mandatory to implement these.

- Human-syntax and mnemonics:
  - Dest-then-source ordering for arguments.
  - Some opcodes have a disambiguating name suffix to indicate the type(s) they operate on:
    - Type-general 32-bit opcodes are unmarked.
    - Type-general 64-bit opcodes are suffixed with `-wide`.
    - Type-specific opcodes are suffixed with their type (or a straightforward abbreviation), one of: `-boolean` `-byte` `-char` `-short` `-int` `-long` `-float` `-double` `-object` `-string` `-class` `-void`.
  - Some opcodes have a disambiguating suffix to distinguish otherwise-identical operations that have different instruction layouts or options. These suffixes are separated from the main names with a slash ("`/`") and mainly exist at all to make there be a one-to-one mapping with static constants in the code that generates and interprets executables (that is, to reduce ambiguity for humans).
  - In the descriptions here, the width of a value (indicating, e.g., the range of a constant or the number of registers possibly addressed) is emphasized by the use of a character per four bits of width.
  - For example, in the instruction "`move-wide/from16 vAA, vBBBB`":
    - "`move`" is the base opcode, indicating the base operation (move a register's value).
    - "`wide`" is the name suffix, indicating that it operates on wide (64 bit) data.
    - "`from16`" is the opcode suffix, indicating a variant that has a 16-bit register reference as a source.
    - "`vAA`" is the destination register (implied by the operation; again, the rule is that destination arguments always come first), which must be in the range `v0` – `v255`.
    - "`vBBBB`" is the source register, which must be in the range `v0` – `v65535`.
- See the instruction formats document for more details about the various instruction formats (listed under "Op & Format") as well as details about the opcode syntax.
- See the `.dex` file format document for more details about where the bytecode fits into the bigger picture.

Summary of Instruction Set

Op & Format	Mnemonic / Syntax	Arguments	Description
`00 10x`	`nop`		Waste cycles. **Note:** Data-bearing pseudo-instructions are tagged with this opcode, in which case the high-order byte of the opcode unit indicates the nature of the data. See "`packed-switch-payload` Format", "`sparse-switch-payload` Format", and "`fill-array-data-payload` Format" below.
`01 12x`	`move vA, vB`	`A:` destination register (4 bits) `B:` source register (4 bits)	Move the contents of one non-object register to another.
`02 22x`	`move/from16 vAA, vBBBB`	`A:` destination register (8 bits) `B:` source register (16 bits)	Move the contents of one non-object register to another.
`03 32x`	`move/16 vAAAA, vBBBB`	`A:` destination register (16 bits) `B:` source register (16 bits)	Move the contents of one non-object register to another.
`04 12x`	`move-wide vA, vB`	`A:` destination register pair (4 bits) `B:` source register pair (4 bits)	Move the contents of one register-pair to another. **Note:** It is legal to move from `v`*N* to either `v`*N-1* or `v`*N+1*, so implementations must arrange for both halves of a register pair to be read before anything is written.
`05 22x`	`move-wide/from16 vAA, vBBBB`	`A:` destination register pair (8	Move the contents of one register-pair to

		bits) `B:` source register pair (16 bits)	another. **Note:** Implementation considerations are the same as `move-wide`, above.
06 32x	`move-wide/16 vAAAA, vBBBB`	`A:` destination register pair (16 bits) `B:` source register pair (16 bits)	Move the contents of one register-pair to another. **Note:** Implementation considerations are the same as `move-wide`, above.
07 12x	`move-object vA, vB`	`A:` destination register (4 bits) `B:` source register (4 bits)	Move the contents of one object-bearing register to another.
08 22x	`move-object/from16 vAA, vBBBB`	`A:` destination register (8 bits) `B:` source register (16 bits)	Move the contents of one object-bearing register to another.
09 32x	`move-object/16 vAAAA, vBBBB`	`A:` destination register (16 bits) `B:` source register (16 bits)	Move the contents of one object-bearing register to another.
0a 11x	`move-result vAA`	`A:` destination register (8 bits)	Move the single-word non-object result of the most recent `invoke-`*`kind`* into the indicated register. This must be done as the instruction immediately after an `invoke-`*`kind`* whose (single-word, non-object) result is not to be ignored; anywhere else is invalid.
0b 11x	`move-result-wide vAA`	`A:` destination register pair (8 bits)	Move the double-word result of the most recent `invoke-`*`kind`* into the indicated register pair. This must be done as the instruction immediately after an `invoke-`*`kind`* whose (double-word) result is not to be ignored; anywhere else is invalid.
0c 11x	`move-result-object vAA`	`A:` destination register (8 bits)	Move the object result of the most recent `invoke-`*`kind`* into the indicated register. This must be done as the instruction immediately after an `invoke-`*`kind`* or `filled-new-array` whose (object) result is not to be ignored; anywhere else is invalid.
0d 11x	`move-exception vAA`	`A:` destination register (8 bits)	Save a just-caught exception into the given register. This must be the first instruction of any exception handler whose caught exception is not to be ignored, and this instruction must *only* ever occur as the first instruction of an exception handler; anywhere else is invalid.
0e 10x	`return-void`		Return from a `void` method.
0f 11x	`return vAA`	`A:` return value register (8 bits)	Return from a single-width (32-bit) non-object value-returning method.
10 11x	`return-wide vAA`	`A:` return value register-pair (8 bits)	Return from a double-width (64-bit) value-returning method.
11 11x	`return-object vAA`	`A:` return value register (8 bits)	Return from an object-returning method.
12 11n	`const/4 vA, #+B`	`A:` destination register (4 bits) `B:` signed int (4 bits)	Move the given literal value (sign-extended to 32 bits) into the specified register.
13 21s	`const/16 vAA, #+BBBB`	`A:` destination register (8 bits) `B:` signed int (16 bits)	Move the given literal value (sign-extended to 32 bits) into the specified register.
14 31i	`const vAA, #+BBBBBBBB`	`A:` destination register (8 bits) `B:` arbitrary 32-bit constant	Move the given literal value into the specified register.
15 21h	`const/high16 vAA, #+BBBB0000`	`A:` destination register (8 bits) `B:` signed int (16 bits)	Move the given literal value (right-zero-extended to 32 bits) into the specified register.
16 21s	`const-wide/16 vAA, #+BBBB`	`A:` destination register (8 bits) `B:` signed int (16 bits)	Move the given literal value (sign-extended to 64 bits) into the specified register-pair.
17 31i	`const-wide/32 vAA, #+BBBBBBBB`	`A:` destination register (8 bits) `B:` signed int (32 bits)	Move the given literal value (sign-extended to 64 bits) into the specified register-pair.
18 51l	`const-wide vAA,`	`A:` destination register (8 bits)	Move the given literal value into the

	`#+BBBBBBBBBBBBBBBB`	`B:` arbitrary double-width (64-bit) constant	specified register-pair.
`19 21h`	`const-wide/high16 vAA, #+BBBB000000000000`	`A:` destination register (8 bits) `B:` signed int (16 bits)	Move the given literal value (right-zero-extended to 64 bits) into the specified register-pair.
`1a 21c`	`const-string vAA, string@BBBB`	`A:` destination register (8 bits) `B:` string index	Move a reference to the string specified by the given index into the specified register.
`1b 31c`	`const-string/jumbo vAA, string@BBBBBBBB`	`A:` destination register (8 bits) `B:` string index	Move a reference to the string specified by the given index into the specified register.
`1c 21c`	`const-class vAA, type@BBBB`	`A:` destination register (8 bits) `B:` type index	Move a reference to the class specified by the given index into the specified register. In the case where the indicated type is primitive, this will store a reference to the primitive type's degenerate class.
`1d 11x`	`monitor-enter vAA`	`A:` reference-bearing register (8 bits)	Acquire the monitor for the indicated object.
`1e 11x`	`monitor-exit vAA`	`A:` reference-bearing register (8 bits)	Release the monitor for the indicated object. **Note:** If this instruction needs to throw an exception, it must do so as if the pc has already advanced past the instruction. It may be useful to think of this as the instruction successfully executing (in a sense), and the exception getting thrown *after* the instruction but *before* the next one gets a chance to run. This definition makes it possible for a method to use a monitor cleanup catch-all (e.g., `finally`) block as the monitor cleanup for that block itself, as a way to handle the arbitrary exceptions that might get thrown due to the historical implementation of `Thread.stop()`, while still managing to have proper monitor hygiene.
`1f 21c`	`check-cast vAA, type@BBBB`	`A:` reference-bearing register (8 bits) `B:` type index (16 bits)	Throw a `ClassCastException` if the reference in the given register cannot be cast to the indicated type. **Note:** Since `A` must always be a reference (and not a primitive value), this will necessarily fail at runtime (that is, it will throw an exception) if `B` refers to a primitive type.
`20 22c`	`instance-of vA, vB, type@CCCC`	`A:` destination register (4 bits) `B:` reference-bearing register (4 bits) `C:` type index (16 bits)	Store in the given destination register `1` if the indicated reference is an instance of the given type, or `0` if not. **Note:** Since `B` must always be a reference (and not a primitive value), this will always result in `0` being stored if `C` refers to a primitive type.
`21 12x`	`array-length vA, vB`	`A:` destination register (4 bits) `B:` array reference-bearing register (4 bits)	Store in the given destination register the length of the indicated array, in entries
`22 21c`	`new-instance vAA, type@BBBB`	`A:` destination register (8 bits) `B:` type index	Construct a new instance of the indicated type, storing a reference to it in the destination. The type must refer to a non-array class.
`23 22c`	`new-array vA, vB, type@CCCC`	`A:` destination register (8 bits) `B:` size register `C:` type index	Construct a new array of the indicated type and size. The type must be an array type.
`24 35c`	`filled-new-array {vC, vD, vE, vF, vG}, type@BBBB`	`A:` array size and argument word count (4 bits) `B:` type index (16 bits) `C..G:` argument registers (4 bits each)	Construct an array of the given type and size, filling it with the supplied contents. The type must be an array type. The array's contents must be single-word (that is, no arrays of `long` or `double`, but reference types are acceptable). The

			constructed instance is stored as a "result" in the same way that the method invocation instructions store their results, so the constructed instance must be moved to a register with an immediately subsequent `move-result-object` instruction (if it is to be used).
`25 3rc`	`filled-new-array/range {vCCCC .. vNNNN}, type@BBBB`	`A:` array size and argument word count (8 bits) `B:` type index (16 bits) `C:` first argument register (16 bits) `N = A + C - 1`	Construct an array of the given type and size, filling it with the supplied contents. Clarifications and restrictions are the same as `filled-new-array`, described above.
`26 31t`	`fill-array-data vAA, +BBBBBBBB` *(with supplemental data as specified below in "`fill-array-data-payload` Format")*	`A:` array reference (8 bits) `B:` signed "branch" offset to table data pseudo-instruction (32 bits)	Fill the given array with the indicated data. The reference must be to an array of primitives, and the data table must match it in type and must contain no more elements than will fit in the array. That is, the array may be larger than the table, and if so, only the initial elements of the array are set, leaving the remainder alone.
`27 11x`	`throw vAA`	`A:` exception-bearing register (8 bits)	Throw the indicated exception.
`28 10t`	`goto +AA`	`A:` signed branch offset (8 bits)	Unconditionally jump to the indicated instruction. **Note:** The branch offset must not be `0`. (A spin loop may be legally constructed either with `goto/32` or by including a `nop` as a target before the branch.)
`29 20t`	`goto/16 +AAAA`	`A:` signed branch offset (16 bits)	Unconditionally jump to the indicated instruction. **Note:** The branch offset must not be `0`. (A spin loop may be legally constructed either with `goto/32` or by including a `nop` as a target before the branch.)
`2a 30t`	`goto/32 +AAAAAAAA`	`A:` signed branch offset (32 bits)	Unconditionally jump to the indicated instruction.
`2b 31t`	`packed-switch vAA, +BBBBBBBB` *(with supplemental data as specified below in "`packed-switch-payload` Format")*	`A:` register to test `B:` signed "branch" offset to table data pseudo-instruction (32 bits)	Jump to a new instruction based on the value in the given register, using a table of offsets corresponding to each value in a particular integral range, or fall through to the next instruction if there is no match.
`2c 31t`	`sparse-switch vAA, +BBBBBBBB` *(with supplemental data as specified below in "`sparse-switch-payload` Format")*	`A:` register to test `B:` signed "branch" offset to table data pseudo-instruction (32 bits)	Jump to a new instruction based on the value in the given register, using an ordered table of value-offset pairs, or fall through to the next instruction if there is no match.

`2d..31 23x`	`cmp`*kind* `vAA, vBB, vCC`
	`2d: cmpl-float` *(lt bias)*
	`2e: cmpg-float` *(gt bias)*
	`2f: cmpl-double` *(lt bias)*
	`30: cmpg-double` *(gt bias)*
	`31: cmp-long`
		`A:` destination register (8 bits)
		`B:` first source register or pair
		`C:` second source register or pair
			Perform the indicated floating point or `long` comparison, setting `a` to `0` if `b == c`, `1` if `b > c`, or `-1` if `b < c`. The "bias" listed for the floating point operations indicates how `NaN` comparisons are treated: "gt bias" instructions return `1` for `NaN` comparisons, and "lt bias" instructions return `-1`.
			For example, to check to see if floating point `x < y` it is advisable to use `cmpg-float`; a result of `-1` indicates that the test was true, and the other values indicate it was false either due to a valid comparison or because one of the values was `NaN`.

`32..37 22t`	`if-`*test* `vA, vB, +CCCC`
	`32: if-eq`
	`33: if-ne`
	`34: if-lt`
	`35: if-ge`
	`36: if-gt`
	`37: if-le`
		`A:` first register to test (4 bits)
		`B:` second register to test (4 bits)
		`C:` signed branch offset (16 bits)
			Branch to the given destination if the given two registers' values compare as specified.
			Note: The branch offset must not be `0`. (A spin loop may be legally constructed either by branching around a backward `goto` or by including a `nop` as a target before the

				branch.)
38..3d	21t	if-*testz* vAA, +BBBB 38: if-eqz 39: if-nez 3a: if-ltz 3b: if-gez 3c: if-gtz 3d: if-lez	A: register to test (8 bits) B: signed branch offset (16 bits)	Branch to the given destination if the given register's value compares with 0 as specified. **Note:** The branch offset must not be 0. (A spin loop may be legally constructed either by branching around a backward goto or by including a nop as a target before the branch.)
3e..43	10x	*(unused)*		*(unused)*
44..51	23x	*arrayop* vAA, vBB, vCC 44: aget 45: aget-wide 46: aget-object 47: aget-boolean 48: aget-byte 49: aget-char 4a: aget-short 4b: aput 4c: aput-wide 4d: aput-object 4e: aput-boolean 4f: aput-byte 50: aput-char 51: aput-short	A: value register or pair; may be source or dest (8 bits) B: array register (8 bits) C: index register (8 bits)	Perform the identified array operation at the identified index of the given array, loading or storing into the value register.
52..5f	22c	i*instanceop* vA, vB, field@CCCC 52: iget 53: iget-wide 54: iget-object 55: iget-boolean 56: iget-byte 57: iget-char 58: iget-short 59: iput 5a: iput-wide 5b: iput-object 5c: iput-boolean 5d: iput-byte 5e: iput-char 5f: iput-short	A: value register or pair; may be source or dest (4 bits) B: object register (4 bits) C: instance field reference index (16 bits)	Perform the identified object instance field operation with the identified field, loading or storing into the value register. **Note:** These opcodes are reasonable candidates for static linking, altering the field argument to be a more direct offset.
60..6d	21c	s*staticop* vAA, field@BBBB 60: sget 61: sget-wide 62: sget-object 63: sget-boolean 64: sget-byte 65: sget-char 66: sget-short 67: sput 68: sput-wide 69: sput-object 6a: sput-boolean 6b: sput-byte 6c: sput-char 6d: sput-short	A: value register or pair; may be source or dest (8 bits) B: static field reference index (16 bits)	Perform the identified object static field operation with the identified static field, loading or storing into the value register. **Note:** These opcodes are reasonable candidates for static linking, altering the field argument to be a more direct offset.
6e..72	35c	invoke-*kind* {vC, vD, vE, vF, vG}, meth@BBBB 6e: invoke-virtual 6f: invoke-super 70: invoke-direct 71: invoke-static 72: invoke-interface	A: argument word count (4 bits) B: method reference index (16 bits) C..G: argument registers (4 bits each)	Call the indicated method. The result (if any) may be stored with an appropriate move-result* variant as the immediately subsequent instruction. invoke-virtual is used to invoke a normal virtual method (a method that is not private, static, or final, and is also not a constructor). invoke-super is used to invoke the closest superclass's virtual method (as opposed to the one with the same method_id in the calling class). The same method restrictions hold as for invoke-virtual. invoke-direct is used to invoke a non-static direct method (that is, an

			instance method that is by its nature non-overridable, namely either a `private` instance method or a constructor). `invoke-static` is used to invoke a `static` method (which is always considered a direct method). `invoke-interface` is used to invoke an `interface` method, that is, on an object whose concrete class isn't known, using a `method_id` that refers to an `interface`. **Note:** These opcodes are reasonable candidates for static linking, altering the method argument to be a more direct offset (or pair thereof).
`73 10x`	*(unused)*		*(unused)*

`74..78 3rc`	`invoke-`*kind*`/range {vCCCC .. vNNNN}, meth@BBBB`
	`74: invoke-virtual/range`
	`75: invoke-super/range`
	`76: invoke-direct/range`
	`77: invoke-static/range`
	`78: invoke-interface/range`
		`A:` argument word count (8 bits)
		`B:` method reference index (16 bits)
		`C:` first argument register (16 bits)
		`N = A + C - 1`
			Call the indicated method. See first `invoke-`*`kind`* description above for details, caveats, and suggestions.

`79..7a 10x`	*(unused)*		*(unused)*

`7b..8f 12x`	*unop* `vA, vB`
	`7b: neg-int`
	`7c: not-int`
	`7d: neg-long`
	`7e: not-long`
	`7f: neg-float`
	`80: neg-double`
	`81: int-to-long`
	`82: int-to-float`
	`83: int-to-double`
	`84: long-to-int`
	`85: long-to-float`
	`86: long-to-double`
	`87: float-to-int`
	`88: float-to-long`
	`89: float-to-double`
	`8a: double-to-int`
	`8b: double-to-long`
	`8c: double-to-float`
	`8d: int-to-byte`
	`8e: int-to-char`
	`8f: int-to-short`
		`A:` destination register or pair (4 bits)
		`B:` source register or pair (4 bits)
			Perform the identified unary operation on the source register, storing the result in the destination register.

`90..af 23x`	*binop* `vAA, vBB, vCC`
	`90: add-int`
	`91: sub-int`
	`92: mul-int`
	`93: div-int`
	`94: rem-int`
	`95: and-int`
	`96: or-int`
	`97: xor-int`
	`98: shl-int`
	`99: shr-int`
	`9a: ushr-int`
	`9b: add-long`
	`9c: sub-long`
	`9d: mul-long`
	`9e: div-long`
	`9f: rem-long`
	`a0: and-long`
	`a1: or-long`
	`a2: xor-long`
	`a3: shl-long`
	`a4: shr-long`
	`a5: ushr-long`
	`a6: add-float`
		`A:` destination register or pair (8 bits)
		`B:` first source register or pair (8 bits)
		`C:` second source register or pair (8 bits)
			Perform the identified binary operation on the two source registers, storing the result in the first source register.

	a7: sub-float a8: mul-float a9: div-float aa: rem-float ab: add-double ac: sub-double ad: mul-double ae: div-double af: rem-double		
`b0..cf 12x`	*binop*/2addr vA, vB b0: add-int/2addr b1: sub-int/2addr b2: mul-int/2addr b3: div-int/2addr b4: rem-int/2addr b5: and-int/2addr b6: or-int/2addr b7: xor-int/2addr b8: shl-int/2addr b9: shr-int/2addr ba: ushr-int/2addr bb: add-long/2addr bc: sub-long/2addr bd: mul-long/2addr be: div-long/2addr bf: rem-long/2addr c0: and-long/2addr c1: or-long/2addr c2: xor-long/2addr c3: shl-long/2addr c4: shr-long/2addr c5: ushr-long/2addr c6: add-float/2addr c7: sub-float/2addr c8: mul-float/2addr c9: div-float/2addr ca: rem-float/2addr cb: add-double/2addr cc: sub-double/2addr cd: mul-double/2addr ce: div-double/2addr cf: rem-double/2addr	`A:` destination and first source register or pair (4 bits) `B:` second source register or pair (4 bits)	Perform the identified binary operation on the two source registers, storing the result in the first source register.
`d0..d7 22s`	*binop*/lit16 vA, vB, #+CCCC d0: add-int/lit16 d1: rsub-int (reverse subtract) d2: mul-int/lit16 d3: div-int/lit16 d4: rem-int/lit16 d5: and-int/lit16 d6: or-int/lit16 d7: xor-int/lit16	`A:` destination register (4 bits) `B:` source register (4 bits) `C:` signed int constant (16 bits)	Perform the indicated binary op on the indicated register (first argument) and literal value (second argument), storing the result in the destination register. **Note:** `rsub-int` does not have a suffix since this version is the main opcode of its family. Also, see below for details on its semantics.
`d8..e2 22b`	*binop*/lit8 vAA, vBB, #+CC d8: add-int/lit8 d9: rsub-int/lit8 da: mul-int/lit8 db: div-int/lit8 dc: rem-int/lit8 dd: and-int/lit8 de: or-int/lit8 df: xor-int/lit8 e0: shl-int/lit8 e1: shr-int/lit8 e2: ushr-int/lit8	`A:` destination register (8 bits) `B:` source register (8 bits) `C:` signed int constant (8 bits)	Perform the indicated binary op on the indicated register (first argument) and literal value (second argument), storing the result in the destination register. **Note:** See below for details on the semantics of `rsub-int`.
`e3..ff 10x`	*(unused)*		*(unused)*

`packed-switch-payload` Format

Name	Format	Description

ident	ushort = 0x0100	identifying pseudo-opcode
size	ushort	number of entries in the table
first_key	int	first (and lowest) switch case value
targets	int[]	list of `size` relative branch targets. The targets are relative to the address of the switch opcode, not of this table.

Note: The total number of code units for an instance of this table is `(size * 2) + 4`.

`sparse-switch-payload` Format

Name	Format	Description
ident	ushort = 0x0200	identifying pseudo-opcode
size	ushort	number of entries in the table
keys	int[]	list of `size` key values, sorted low-to-high
targets	int[]	list of `size` relative branch targets, each corresponding to the key value at the same index. The targets are relative to the address of the switch opcode, not of this table.

Note: The total number of code units for an instance of this table is `(size * 4) + 2`.

`fill-array-data-payload` Format

Name	Format	Description
ident	ushort = 0x0300	identifying pseudo-opcode
element_width	ushort	number of bytes in each element
size	uint	number of elements in the table
data	ubyte[]	data values

Note: The total number of code units for an instance of this table is `(size * element_width + 1) / 2 + 4`.

Mathematical Operation Details

Note: Floating point operations must follow IEEE 754 rules, using round-to-nearest and gradual underflow, except where stated otherwise.

Opcode	C Semantics	Notes
neg-int	`int32 a;` `int32 result = -a;`	Unary twos-complement.
not-int	`int32 a;` `int32 result = ~a;`	Unary ones-complement.
neg-	`int64 a;`	Unary twos-complement.

`long`	`int64 result = -a;`	
`not-long`	`int64 a;` `int64 result = ~a;`	Unary ones-complement.
`neg-float`	`float a;` `float result = -a;`	Floating point negation.
`neg-double`	`double a;` `double result = -a;`	Floating point negation.
`int-to-long`	`int32 a;` `int64 result = (int64) a;`	Sign extension of `int32` into `int64`.
`int-to-float`	`int32 a;` `float result = (float) a;`	Conversion of `int32` to `float`, using round-to-nearest. This loses precision for some values.
`int-to-double`	`int32 a;` `double result = (double) a;`	Conversion of `int32` to `double`.
`long-to-int`	`int64 a;` `int32 result = (int32) a;`	Truncation of `int64` into `int32`.
`long-to-float`	`int64 a;` `float result = (float) a;`	Conversion of `int64` to `float`, using round-to-nearest. This loses precision for some values.
`long-to-double`	`int64 a;` `double result = (double) a;`	Conversion of `int64` to `double`, using round-to-nearest. This loses precision for some values.
`float-to-int`	`float a;` `int32 result = (int32) a;`	Conversion of `float` to `int32`, using round-toward-zero. `NaN` and `-0.0` (negative zero) convert to the integer `0`. Infinities and values with too large a magnitude to be represented get converted to either `0x7fffffff` or `-0x80000000` depending on sign.
`float-to-long`	`float a;` `int64 result = (int64) a;`	Conversion of `float` to `int64`, using round-toward-zero. The same special case rules as for `float-to-int` apply here, except that out-of-range values get converted to either `0x7fffffffffffffff` or `-0x8000000000000000` depending on sign.
`float-to-double`	`float a;` `double result = (double) a;`	Conversion of `float` to `double`, preserving the value exactly.
`double-to-int`	`double a;` `int32 result = (int32) a;`	Conversion of `double` to `int32`, using round-toward-zero. The same special case rules as for `float-to-int` apply here.
`double-to-long`	`double a;` `int64 result = (int64) a;`	Conversion of `double` to `int64`, using round-toward-zero. The same special case rules as for `float-to-long` apply here.
`double-to-float`	`double a;` `float result = (float) a;`	Conversion of `double` to `float`, using round-to-nearest. This loses precision for some values.
`int-to-byte`	`int32 a;` `int32 result = (a << 24) >> 24;`	Truncation of `int32` to `int8`, sign extending the result.
`int-to-char`	`int32 a;` `int32 result = a & 0xffff;`	Truncation of `int32` to `uint16`, without sign extension.
`int-to-short`	`int32 a;` `int32 result = (a << 16) >> 16;`	Truncation of `int32` to `int16`, sign extending the result.
`add-int`	`int32 a, b;` `int32 result = a + b;`	Twos-complement addition.
`sub-int`	`int32 a, b;` `int32 result = a - b;`	Twos-complement subtraction.
`rsub-int`	`int32 a, b;` `int32 result = b - a;`	Twos-complement reverse subtraction.
`mul-int`	`int32 a, b;` `int32 result = a * b;`	Twos-complement multiplication.
`div-int`	`int32 a, b;` `int32 result = a / b;`	Twos-complement division, rounded towards zero (that is, truncated to integer). This throws `ArithmeticException` if `b == 0`.

`rem-int`	`int32 a, b;` `int32 result = a % b;`	Twos-complement remainder after division. The sign of the result is the same as that of `a`, and it is more precisely defined as `result == a - (a / b) * b`. This throws `ArithmeticException` if `b == 0`.
`and-int`	`int32 a, b;` `int32 result = a & b;`	Bitwise AND.
`or-int`	`int32 a, b;` `int32 result = a | b;`	Bitwise OR.
`xor-int`	`int32 a, b;` `int32 result = a ^ b;`	Bitwise XOR.
`shl-int`	`int32 a, b;` `int32 result = a << (b & 0x1f);`	Bitwise shift left (with masked argument).
`shr-int`	`int32 a, b;` `int32 result = a >> (b & 0x1f);`	Bitwise signed shift right (with masked argument).
`ushr-int`	`uint32 a, b;` `int32 result = a >> (b & 0x1f);`	Bitwise unsigned shift right (with masked argument).
`add-long`	`int64 a, b;` `int64 result = a + b;`	Twos-complement addition.
`sub-long`	`int64 a, b;` `int64 result = a - b;`	Twos-complement subtraction.
`mul-long`	`int64 a, b;` `int64 result = a * b;`	Twos-complement multiplication.
`div-long`	`int64 a, b;` `int64 result = a / b;`	Twos-complement division, rounded towards zero (that is, truncated to integer). This throws `ArithmeticException` if `b == 0`.
`rem-long`	`int64 a, b;` `int64 result = a % b;`	Twos-complement remainder after division. The sign of the result is the same as that of `a`, and it is more precisely defined as `result == a - (a / b) * b`. This throws `ArithmeticException` if `b == 0`.
`and-long`	`int64 a, b;` `int64 result = a & b;`	Bitwise AND.
`or-long`	`int64 a, b;` `int64 result = a | b;`	Bitwise OR.
`xor-long`	`int64 a, b;` `int64 result = a ^ b;`	Bitwise XOR.
`shl-long`	`int64 a, b;` `int64 result = a << (b & 0x3f);`	Bitwise shift left (with masked argument).
`shr-long`	`int64 a, b;` `int64 result = a >> (b & 0x3f);`	Bitwise signed shift right (with masked argument).
`ushr-long`	`uint64 a, b;` `int64 result = a >> (b & 0x3f);`	Bitwise unsigned shift right (with masked argument).
`add-float`	`float a, b;` `float result = a + b;`	Floating point addition.
`sub-float`	`float a, b;` `float result = a - b;`	Floating point subtraction.
`mul-float`	`float a, b;` `float result = a * b;`	Floating point multiplication.
`div-float`	`float a, b;` `float result = a / b;`	Floating point division.
`rem-float`	`float a, b;` `float result = a % b;`	Floating point remainder after division. This function is different than IEEE 754 remainder and is defined as `result == a - roundTowardZero(a / b) * b`.
`add-`	`double a, b;`	Floating point addition.

double	double result = a + b;	
sub-double	double a, b; double result = a - b;	Floating point subtraction.
mul-double	double a, b; double result = a * b;	Floating point multiplication.
div-double	double a, b; double result = a / b;	Floating point division.
rem-double	double a, b; double result = a % b;	Floating point remainder after division. This function is different than IEEE 754 remainder and is defined as `result == a - roundTowardZero(a / b) * b`.