An Introduction to Android Dalvik's bytecode and .smali file

Tony.Guo.Peng@gmail.com

2014-5-10

Types

Dalvik's bytecode has two major classes of types, primitive types and reference types. Reference types are objects and arrays, everything else is a primitive.

Primitives are represented by a single letter. I didn't come up with these abbreviations - they are what is actually stored in the dex file, in string form. They are specified in the dex-format.html document (*dalvik/docs/dex-format.html in the AOSP repository*)

- **V** - Void - can only be used for return types
- **Z** - Boolean
- **B** - Byte
- **S** - Short
- **C** - Char
- **I** - Integer
- **J** - Long (64 bits)
- **F** - Float
- **D** - Double (64 bits)

Objects take the form of **Lpackage/name/ObjectName;** - where the leading **L** indicates that it is an object type, **package/name/** is the package that the object is in, **ObjectName** is the name of the object, and **;** denotes the end of the object name. This would be equivalent to **package.name.ObjectName** in java. Or for a more concrete example, **Ljava/lang/String;** is equivalent to **java.lang.String**

Arrays take the form **[I** - this would be an array of integers with a single dimension. i.e. **int[]** in Java. For arrays with multiple dimensions, you simply add more **[** characters. **[[I** = **int[][]**, **[[[I** = **int[][][]**, etc. (Note: The maximum number of dimensions you can have is 255).

You can also have arrays of objects, **[Ljava/lang/String;** would be an array of Strings.

Methods

Methods are always specified in a very verbose form that includes the type that contains the method, the method name, the types of the parameters and the return type. All this information is required for the virtual machine to be able to find the correct method, and to be able to perform static analysis on the bytecode (for verification/optimization purposes)

They take the form of **Lpackage/name/ObjectName;->MethodName(III)Z**

In this example, you should recognize **Lpackage/name/ObjectName**; as a type. **MethodName** is obviously the name of the method. (**III**)**Z** is the method's signature. **III** are the parameters (in this case, 3 integers), and **Z** is the return type (bool).

The method parameters are listed one right after another, with no separators between them.

Here's a more complex example:
Lpackage/name/ObjectName;->MethodName(I[[IILjava/lang/String;[Ljava/lang/Object;)
Ljava/lang/String;

In java, this would be
String MethodName(int, int[][], int, String, Object[])

Fields

Fields are likewise always specified in verbose form that includes the type that contains the field, the name of the field, and the type of the field. Again, this is to allow the virtual machine to be able to find the correct field, as well as to perform static analysis on the bytecode.

They take the form of **Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;**

This should be pretty self-explanatory - it is the package and object name, the field name and the type of the field respectively.

Introduction

In dalvik's bytecode, registers are always 32 bits, and can hold any type of value. 2 registers are used to hold 64 bit types (Long and Double).

Specifying the number of registers in a method

There are two ways to specify how many registers are available in a method. the **.registers** directive specifies the **total** number of registers in the method, while the alternate **.locals** directive specifies the number of non-parameter registers in the method. The total number of registers would also include however many registers are needed to hold the method parameters.

How method parameters are passed into a method

When a method is invoked, the parameters to the method are placed into the last n registers. If a method has 2 arguments, and 5 registers (**v0-v4**), the arguments would be placed into the last 2 registers - **v3** and **v4**.

The first parameter to a non-static methods is always the object that the method is being invoked on (the **this** object) For example, let's say you are writing a non-static method **LMyObject;->callMe(II)V**. This method has 2 integer parameters, but it also has an implicit **LMyObject**; parameter before both integer parameters, so there are a total of 3 arguments to the method.

Let's say you specify that there are 5 registers in the method (**v0-v4**), with either the **.registers 5** directive or the **.locals 2** directive (i.e. 2 local registers + 3 parameter registers). When the method is invoked, the object that the method is being invoked on (i.e. the **this** reference) will be in **v2**, the first integer parameter will be in **v3**, and the second integer parameter will be in **v4**.

For static methods it's the same thing, except there isn't an implicit this argument.

Register names

There are two naming schemes for registers - the normal **v#** naming scheme and the **p#** naming scheme for parameter registers. The first register in the **p#** naming scheme is the first parameter register in the method. So let's go back to the previous example of a method with 3 arguments and 5 total registers. The following table shows the normal **v#** name for each register, followed by the **p#** name for the parameter registers

v0		The first local register
v1		The second local register
v2	p0	The first parameter register
v3	p1	The second parameter register
v4	p2	The third parameter register

You can reference parameter registers by either name - it makes no difference.

Motivation for introducing parameter registers

The **p#** naming scheme was introduced as a practical matter, to solve a common annoyance when editing smali code.

Say you have an existing method with a number of parameters and you are adding some code to the method, and you discover that you need an extra register. You think "No big deal, I'll just increase the number of registers specified in the **.registers** directive!".

Unfortunately, it isn't quite that easy. Keep in mind that the method parameters are stored in the last registers in the method. If you increase the number of registers - you change which registers the method arguments get put into. So you would have to change the **.registers** directive and renumber every parameter register.

But if the **p#** naming scheme was used to reference parameter registers throughout the method, you can easily change the number of registers in the method, without having to worry about renumbering any existing registers.

Long/Double values

As mentioned previously, long and double primitives (**J** and **D** respectively) are 64 bit values, and require 2 registers. This is important to keep in mind when you are referencing method arguments. For example, let's say you have a (non-static) method **LMyObject;->MyMethod(IJZ)V**. The parameters to the method are **LMyObject**;, **int**, **long**, **bool**. So this method would require 5 registers for all of its parameters.

- p0 **this**
- p1 **I**
- p2, p3 **J**
- p4 **Z**

Also, when you are invoking the method later on, you do have to specify both registers for any double-wide arguments in the register list for the **invoke-***kind* instruction.

1. Array

array-length vA, vB

A: Destination register (4 bits)
B: Array reference-bearing register (4 bits)
Store the length (number of entries) of the indicated array **vB** in the **vA**

fill-array-data vA+, :target

A: Register **pair** containing an array reference
B: The target label defining the array-data table
Fill the given array **vA+** with the indicated data at **target**. The reference must be to an array of primitives, and the data table must match it in type and size. The width of the array is defined in the table.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**
Example data table:

```
:target
    .array-data 0x2
        0x01 0x02
        0x03 0x04
    .end array-data
```

filled-new-array/range { vA .. vX }, Lclass;->type

vA .. vX: Range of registers containing array parameters (4 bits each)
B: Type reference (16 bits)
Construct a new array of the indicated type. The type must be an array type. Reference to the newly generated array can be obtained by a **move-result-object** instruction, immediately following the **filled-new-array/range** instruction.

filled-new-array { vA [vB, v.., vX]}, Lclass;->type

vA-vX: Argument registers (4 bits each)
B: Type reference
Construct a new array of the indicated type and size. The type must be an array type. Reference to the newly generated array can be obtained by a **move-result-object** instruction, immediately following the **filled-new-array** instruction.

new-array vA, vB, Lclass;->type

A: Destination register (8 bits)
B: Size register

C: Type reference
Construct a new array of the indicated type and size. The type must be an array type.

2. Array accessors

aget-boolean **vA**, **vB**, **vC**
A: Destination register
B: Reference to array
C: Index into array
Retrieves a **boolean** at index **vC** from array referenced by **vB** and stores in **vA**

aget-byte **vA**, **vB**, **vC**
A: Destination register
B: Reference to array
C: Index into array
Retrieves a **byte** at index **vC** from array referenced by **vB** and stores in **vA**

aget-char **vA**, **vB**, **vC**
A: Destination register
B: Reference to array
C: Index into array
Retrieves a **char** at index **vC** from array referenced by **vB** and stores in **vA**

aget-object **vA**, **vB**, **vC**
A: Destination register
B: Reference to array
C: Index into array
Retrieves an **object** at index **vC** from array referenced by **vB** and stores in **vA**

aget-short **vA**, **vB**, **vC**
A: Destination register
B: Reference to array
C: Index into array
Retrieves a **short** at index **vC** from array referenced by **vB** and stores in **vA**

aget-wide **vA+**, **vB**, **vC**
A: Destination register **pair**
B: Reference to array
C: Index into array
Retrieves a **long** or **double** at index **vC** from array referenced by **vB** and stores in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

aget **vA**, **vB**, **vC**
A: Destination register
B: Reference to array
C: Index into array
Retrieves an **integer** at index **vC** from array referenced by **vB** and stores in **vA**

aput-boolean **vA**, **vB**, **vC**
A: Source register
B: Reference to array
C: Index into array
Stores an **boolean** from **vA** in array referenced by **vB** at index **vC**

aput-byte **vA**, **vB**, **vC**
A: Source register

B: Reference to array
C: Index into array
Stores a **byte** from **vA** in array referenced by **vB** at index **vC**

aput-char **vA**, **vB**, **vC**
A: Source register
B: Reference to array
C: Index into array
Stores a **char** from **vA** in array referenced by **vB** at index **vC**

aput-object **vA**, **vB**, **vC**
A: Source register
B: Reference to array
C: Index into array
Stores an **object** reference from **vA** in array referenced by **vB** at index **vC**

aput-short **vA**, **vB**, **vC**
A: Source register
B: Reference to array
C: Index into array
Stores a **short** from **vA** in array referenced by **vB** at index **vC**

aput-wide **vA+**, **vB**, **vC**
A: Source register **pair**
B: Reference to array
C: Index into array
Stores a **double** or **long** from **vA+** in array referenced by **vB** at index **vC**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

aput **vA**, **vB**, **vC**
A: Source register
B: Reference to array
C: Index into array
Stores an **integer** from **vA** in array referenced by **vB** at index **vC**

3. Comparison

cmp-long **vA**, **vB+**, **vC+**
A: Destination register
B: First source register **pair**
C: Second source register **pair**
Compares the long values in the source registers, storing **0** if the two arguments are equal, **1** if the second argument is larger, or **-1** if the first argument is larger.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

cmpg-double **vA**, **vB+**, **vC+**
A: Destination register
B: First source register or **pair**
C: Second source register or **pair**
Compares the double values in the source registers, storing **0** if the two arguments are equal, **1** if the second argument is larger, or **-1** if the first argument is larger. If either **vB+** or **vC** are not a number, **1** is returned.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

cmpg-float **vA**, **vB**, **vC**
A: Destination register
B: First source register

C: Second source register
Compares the floating point values in the source registers, storing **0** if the two arguments are equal, **1** if the second argument is larger, or **-1** if the first argument is larger. If either **vB** or **vC** are not a number, **1** is returned.

cmpl-double vA, vB+, vC+
A: Destination register
B: First source register **pair**
C: Second source register **pair**
Compares the double values in the source registers, storing **0** if the two arguments are equal, **1** if the second argument is larger, or **-1** if the first argument is larger. If either **vB+** or **vC+** are not a number, **-1** is returned.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

cmpl-float vA, vB, vC
A: Destination register
B: First source register
C: Second source register
Perform the indicated floating point comparison, storing **0** if the two arguments are equal, **1** if the second argument is larger, or **-1** if the first argument is larger. If either **vB** or **vC** are not a number, **-1** is returned.

4. Constants

const/16 vAA, #+BBBB
A: Destination register (8 bits)
B: Signed integer (16 bits)
Puts the **integer** constant **BBBB** into the destination register **vAA**

const/4 vA, #+B
A: Destination register (4 bits)
B: Signed **integer** constant (4 bits)
Puts the given 4 bit **integer** constant into the destination register **vA**.

const/high16 vAA, #+BBBB
A: Destination register (8 bits)
B: Signed **integer** constant (16 bits)
Puts the 16 bit constant into the top-most bits of the register **vAA**. Used to initialize **float** values.

const-class vAA, Lclass
A: Destination register (8 bits)
B: Class reference
Move a reference to the **class** specified into the destination register **vAA**. In the case where the indicated type is primitive, this will store a reference to the primitive type's degenerate class.

const-string/jumbo vAA, "BBBBBBBB"
A: Destination register (8 bits)
B: String value
Move a reference to the **string** specified into the destination register **vAA**

const-string vAA, "BBBB"
A: Destination register (8 bits)
B: String value
Move a reference to the **string** specified into the destination register **vAA**

const-wide/16 vA+, #+BBBB
A: Destination register **pair** (8 bits)
B: Signed integer (16 bits)

Puts the **integer** constant into **vA+**, expanding the **integer** constant into a **long** constant.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

const-wide/32 vA+, #+BBBBBBBB
A: Destination register pair (8 bits)
B: Signed integer (32 bits)
Puts the 32 bit **integer** constant into **vA+**, expanding the **integer** constant into a **long** constant.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

const-wide/high16 vA+, #+BBBB
A: Destination register **pair** (8 bits)
B: Signed **double** constant (16 bits)
Puts the 16 bit **double** constant (**BBBB**) into the highest 16 bit of **vA+**. Used to initialize **double** values.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

const-wide vA+, #+BBBBBBBBBBBBBBBB
A: Destination register **pair** (8 bits)
B: Signed **double** constant (64 bits)
Puts the 64 bit **double** constant into **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

const vAA, #+BBBBBBBB
A: Destination register (8 bits)
B: 32-bit signed constant **integer**
Move the given constant **integer** value into the specified register **vAA**.

5. Conversion

double-to-float ***vA, vB+***
A: Destination register (4 bits)
B: Source register **pair** (8 bits)
Converts **double vB+** to **float** and stores it in **vA**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

double-to-int ***vA, vB+***
A: Destination register (4 bits)
B: Source register **pair** (8 bits)
Converts **double vB+** to an **integer** and stores it in **vA**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

double-to-long ***vA+, vB+***
A: Destination register **pair** (8 bits)
B: Source register **pair** (8 bits)
Converts **double vB+** to a **long** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

float-to-double ***vA+, vB***
A: Destination register **pair** (8 bits)
B: Source register (4 bits)
Converts **float vB** to a **double** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

float-to-int ***vA, vB***
A: Destination register (4 bits)
B: Source register (4 bits)
Converts **float vB** to an **integer** and stores it in **vA**

float-to-long ***vA+, vB***
A: Destination register **pair** (4 bits)
B: Source register (4 bits)
Converts **float vB** to a **long** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

int-to-byte ***vA, vB***
A: Destination register (4 bits)
B: Source register (4 bits)
Converts **integer vB** to a **byte** and stores it in **vA**

int-to-char ***vA, vB***
A: Destination register (4 bits)
B: Source register (4 bits)
Converts **integer vB** to a **char** and stores it in **vA**

int-to-double ***vA+, vB***
A: Destination register **pair** (8 bits)
B: Source register (4 bits)
Converts **integer vB** to **double** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

int-to-float ***vA, vB***
A: Destination register pair (4 bits)
B: Source register pair (4 bits)
Converts **integer vB** to **float** and stores it in **vA**

int-to-long ***vA+, vB***
A: Destination register **pair** (8 bits)
B: Source register pair (4 bits)
Converts **integer vB** to **long** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

int-to-short ***vA, vB***
A: Destination register (4 bits)
B: Source register (4 bits)
Converts **integer vB** to **short** and stores it in **vA**

long-to-double ***vA+, vB+***
A: Destination register **pair** (8 bits)
B: Source register **pair** (8 bits)
Converts **long vB+** to **double** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

long-to-float ***vA, vB+***
A: Destination register (4 bits)
B: Source register **pair** (8 bits)
Converts **long vB+** to **float** and stores it in **vA**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

long-to-int ***vA, vB+***
A: Destination register (4 bits)
B: Source register **pair** (8 bits)
Converts **long vB+** to **integer** and stores it in **vA**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

neg-double ***vA+, vB+***
A: Destination register **pair** (8 bits)
B: Source register **pair** (8 bits)
Calculates -**vB+** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

neg-float ***vA, vB***
A: Destination register (4 bits)
B: Source register (4 bits)
Calculates -**vB** and stores it in **vA**

neg-int ***vA, vB***
A: Destination register (4 bits)
B: Source register (4 bits)
Calculates -**vB** and stores it in **vA**

neg-long ***vB+, vB+***
A: Destination register **pair** (8 bits)
B: Source register **pair** (8 bits)
Calculates -**vB+** and stores it in **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

6. Goto

goto/16 :target
target: The label of the target instruction
Unconditional branch to **:target**.
Note: goto's literally use +/- offsets from the current instruction. APKTool translates these into labels for readability. If under the hood, the offset requires a 16-bit value, **goto/16** must be used. it is near impossible to determine if **goto/16** is required while adding a new instruction. When in doubt, a **goto/16** can replace any **goto** but not vice-versa.

goto/32 :target
target: The label of the target instruction
Unconditional branch to **:target**.
Note: goto's literally use +/- offsets from the current instruction. APKTool translates these into labels for readability. If under the hood, the offset requires a 32-bit value, **goto/32** must be used. it is near impossible to determine if **goto/32** is required while adding a new instruction. When in doubt, a **goto/32** can replace any **goto** or **goto/16,** but not vice-versa.

goto :target
target: The label of the target instruction
Unconditional branch to **:target**.

goto :target
target: The label of the target instruction
Unconditional branch to **:target**.

7. If

if-eq vA, vB, :target
A: First register to test (integer)
B: Second register to test (integer)
target: Target label
Execution jumps to **:target** if **vA == vB**

if-eqz **vA**, **:target**
A: First register to test (integer)
target: Target label
Execution jumps to **:target** if **vA == 0**

if-ge **vA**, **vB**, **:target**
A: First register to test (integer)
B: Second register to test (integer)
target: Target label
Execution jumps to **:target** if **vA >= vB**

if-gez **vA**, **:target**
A: First register to test
target: Target label
Execution jumps to **:target** if **vA >= 0**

if-gt **vA**, **vB**, **:target**
A: First register to test (integer)
B: Second register to test (integer)
target: Target label
Execution jumps to **:target** if **vA > vB**

if-gtz **vA**, **:target**
A: First register to test
target: Target label
Execution jumps to **:target** if **vA > 0**

if-le **vA**, **vB**, **:target**
A: First register to test (integer)
B: Second register to test (integer)
target: Target label
Execution jumps to **:target** if **vA <= vB**

if-lez **vA**, **:target**
A: First register to test
target: Target label
Execution jumps to **:target** if **vA <= 0**

if-lt **vA**, **vB**, **:target**
A: First register to test (integer)
B: Second register to test (integer)
target: Target label
Execution jumps to **:target** if **vA < vB**

if-ltz **vA**, **:target**
A: First register to test
target: Target label
Execution jumps to **:target** if **vA < 0**

if-ne **vA**, **vB**, **:target**
A: First register to test (integer)
B: Second register to test (integer)
target: Target label
Execution jumps to **:target** if **vA != vB**

if-nez vA, :target
A: First register to test
target: Target label
Execution jumps to **:target** if **vA != 0**

8. Instance accessor

iget-boolean **vA**, **vB**, **Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **boolean** value of **field** from the instance object referenced by **vB**, into **vA**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iget-byte **vA**, **vB**, **Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **byte** value of **field** from the instance object referenced by **vB**, into **vA**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iget-char **vA**, **vB**, **Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **char** value of **field** from the instance object referenced by **vB**, into **vA**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iget-object **vA**, **vB**, **Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **object** reference of **type** from the instance object referenced by **vB**, into **vA**
vB is typically **"p0"**, aka the '**this**' reference.

iget-short **vA**, **vB**, **Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **short** value of **field** from the instance object referenced by **vB**, into **vA**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iget-wide **vA+**, **vB**, **Lclass;->field:type**
A: Source register **pair**
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the value of **type** from the instance object referenced by **vB** in **vA+**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

iget **vA**, **vB**, **Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the value of **field** from the instance object referenced by **vB**, into **vA**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iput-boolean **vA, vB, Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **boolean** value of **vA** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iput-byte **vA, vB, Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **byte** value of **vA** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iput-char **vA, vB, Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **char** value of **vA** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iput-object **vA, vB, Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **object** reference of **vA** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iput-short **vA, vB, Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the **short** value of **vA** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

iput-wide **vA+, vB, Lclass;->field:type**
A: Source register **pair**
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the value of **vA+** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

iput **vA, vB, Lclass;->field:type**
A: Source register
B: Reference to an instance object
C: Class, field and type of the instance object
Stores the value of **vA** into **field** of the instance object referenced by **vB**
vB is typically **"p0"** (the **this** reference) or obtained by first calling **iget-object**

9. Invoke

invoke-direct/range ***{ vA .. vX }, Lclass;->method()R***
vA-vX : Range of arguments being passed to the method
class: Name of the class containing the method

method: Name of the method to invoke
R : The return type.
Invokes 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). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.
invoke-direct { v1, v2, v3 } is equal to invoke-direct/range { v1 .. v3 }

invoke-direct ***{[vA, v.., vX]}, Lclass;->method()R***
vA-vX : Optional arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes 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). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.

invoke-interface/range ***{ vA .. vX }, Lclass;->method()R***
vA-vX : Range of arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes an interface method (that is, on an object whose concrete class isn't known, using a **method** that refers to an interface). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.
invoke-interface { v1, v2, v3 } is equal to invoke-interface/range { v1 .. v3 }

invoke-interface ***{[vA, v.., vX]}, Lclass;->method()R***
vA-vX : Optional arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes an interface method (that is, on an object whose concrete class isn't known, using a **method** that refers to an interface). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.

invoke-static/range ***{ vA .. vX }, Lclass;->method()R***
vA-vX : Range of arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes a static method (which is always considered a direct method). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.
invoke-static { v1, v2, v3 } is equal to invoke-static/range { v1 .. v3 }

invoke-static ***{[vA, v.., vX]}, Lclass;->method()R***
vA-vX : Optional arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes a static method (which is always considered a direct method). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.

invoke-super/range ***{ vA .. vX }, Lclass;->method()R***
vA-vX : Range of arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke

R : The return type.
Invokes the immediate parent class's virtual method (as opposed to the one with the same **method** name in the calling class). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.
invoke-super { v1, v2, v3 } is equal to invoke-super/range { v1 .. v3 }

invoke-super ***{[vA, v.., vX]}, Lclass;->method()R***
vA-vX : Optional arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes the immediate parent class's virtual method (as opposed to the one with the same **method** name in the calling class). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.

invoke-virtual/range ***{ vA .. vX }, Lclass;->method()R***
vA-vX : Range of arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes a virtual method (a method that is not static or final, and is not a constructor) with an inclusive range of registers. If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.
invoke-virtual { v1, v2, v3 } is equal to invoke-virtual/range { v1 .. v3 }

invoke-virtual ***{[vA, v.., vX]}, Lclass;->method()R***
vA-vX : Optional arguments being passed to the method
class: Name of the class containing the method
method: Name of the method to invoke
R : The return type.
Invokes a virtual method (a method that is not static or final, and is not a constructor). If the method returns a value (**R** is not "V" for ***Void***) it must be captured in the next line with on eof the ***move-result*** statements or it will be lost.

10. Misc

check-cast vAA, Lclass
A: Reference-bearing register (8 bits)
B: Type reference (16 bits)
Checks whether the object reference in **vAA** can be cast to an instance of a type referenced by **class**.
Throws **ClassCastException** if the cast is not possible, continues execution otherwise.

instance-of vA, vB, Lclass
A: Destination register (4 bits)
B: Reference-bearing register (4 bits)
C: Class reference (16 bits)
Checks whether **vB** is ian nstance of a **class**. Sets **vA** non-zero if it is, 0 if it is not.

new-instance vAA, Lclass
A: Destination register (8 bits)
B: Type reference
Instantiates an object of type **class** and puts the reference of the newly created instance into **vAA**.
The type must refer to a non-array **class**.

nop
No Operation

throw vAA
A: Exception-bearing register (8 bits)
Throws the indicated exception. The exception **object** reference is in **vAA**.

11. Monitor

monitor-enter vAA
A: Reference-bearing register (8 bits)
Acquire the monitor for the indicated **object**.

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 device has already advanced past the instruction. It maybe 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.

12. Move

move/16 *vAAAA, vBBBB*
A: Destination register (16 bits)
B: Source register (16 bits)
Move the contents of one non-object register to another.

move/from16 *vAA, vBBBB*
A: Destination register (8 bits)
B: Source register (16 bits)
Move the contents of one non-object register to another.

move-exception *vAA*
A: Destination register (8 bits)
Save a just-caught exception into **vAA**. This should be the first instruction of any exception handler whose caught exception is not to be ignored, and this instruction may only ever occur as the first instruction of an exception handler. Use anywhere else is invalid.

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.

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.

move-object *vA, vB*
A: Destination register (4 bits)
B: Source register (4 bits)
Move the contents of one object-bearing register to another.

move-result-object *vAA*
A: Destination register (8 bits)
Move the **object** result of the most recent **invoke**-*kind* into **vAA**. This must be done as the instruction immediately

after an **invoke**-*kind* or **filled-new-array** whose (**object**) result is not to be ignored. Use anywhere else is invalid.

move-result-wide *vA+*
A: Destination register **pair** (8 bits)
Move the double-word result of the most recent **invoke**-*kind* into **vAA**. This must be done as the instruction immediately after an **invoke**-*kind* whose (double-word) result is not to be ignored. Use anywhere else is invalid.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

move-result *vAA*
A: Destination register (8 bits)
Move the single-word non-**object** result of the most recent **invoke**-*kind* into **vAA**. This must be done as the instruction immediately after an **invoke**-*kind* whose (single-word, non-**object**) result is not to be ignored. Use anywhere else is invalid.

move-wide/16 *vA+*, *vB+*
A: Destination register **pair** (16 bits)
B: Source register **pair** (16 bits)
Move the contents of one register-pair to another.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.** It is not legal to move from vN to either vN-1 or vN+1.

move-wide/from16 *vA+*, *vBBBB*
A: Destination register **pair** (8 bits)
B: Source register (16 bits)
Move the contents of one register-pair to another.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.** It is not legal to move from vN to either vN-1 or vN+1.

move-wide *vA+*, *vB+*
A: Destination register pair (4 bits)
B: Source register (16 bits)
Move the contents of one register-pair to another.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.** It is not legal to move from vN to either vN-1 or vN+1.

move *vA*, *vB*
A: Destination register (4 bits)
B: Source register (4 bits)
Move the contents of one non-object register to another.

13. Operations

add-double/2addr *vA+*, *vB+*
A: Source register **pair** 1 / Destination register (8 bits)
B: Source register **pair** 2 (8 bits)
Calculates **vA+ + vB+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

add-double *vA+*, *vB+*, *vC+*
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ + vC+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

add-float/2addr *vA*, *vB*
A: Source register 1 / Destination register (4 bits)

B: Source register 2 (4 bits)
Calculates **vA** **+** **vB** and stores the result in **vA**
*All registers are of type **float***

add-float ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** **+** **vC** and stores the result in **vA**
*All registers are of type **float***

add-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA** **+** **vB** and stores the result in **vA**
*All registers are of type **int***

add-int/lit16 ***vA, vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB** **+** **0xC** and stores the result in **vA**
*All registers are of type **int***

add-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB** **+** **0xC** and stores the result in **vA**
*All registers are of type **int***

add-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** + **vC** and stores the result in **vA**
*All registers are of type **int***

add-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+** **+** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

add-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+** **+** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

and-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)

Calculates **vA & vB** and stores the result in **vA**
*All registers are of type **int***

and-int/lit16 ***vA, vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB & 0xC** and stores the result in **vA**
*All registers are of type **int***

and-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB & 0xC** and stores the result in **vA**
*All registers are of type **int***

and-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB & vC** and stores the result in **vA**
*All registers are of type **int***

and-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+ & vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

and-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ & vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

div-double/2addr ***vA+, vB+***
A: Source register **pair** 1 / Destination register (8 bits)
B: Source register **pair** 2 (8 bits)
Calculates **vA+ / vB+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

div-double ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ / vC+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

div-float/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)

B: Source register 2 (4 bits)
Calculates **vA** **/** **vB** and stores the result in **vA**
*All registers are of type **float***

div-float ***vA**, **vB**, **vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** **/** **vC** and stores the result in **vA**
*All registers are of type **float***

div-int/2addr ***vA**, **vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA** **/** **vB** and stores the result in **vA**
*All registers are of type **int***

div-int/lit16 ***vA**, **vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB** **/** **0xC** and stores the result in **vA**
*All registers are of type **int***

div-int/lit8 ***vA**, **vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB** **/** **0xC** and stores the result in **vA**
*All registers are of type **int***

div-int ***vA**, **vB**, **vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** / **vC** and stores the result in **vA**
*All registers are of type **int***

div-long/2addr ***vA+**, **vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+** **/** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

div-long ***vA+**, **vB+**, **vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+** **/** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

mul-double/2addr ***vA+**, **vB+***
A: Source register **pair** 1 / Destination register (8 bits)
B: Source register **pair** 2 (8 bits)

Calculates **vA+** * **vB+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

mul-double ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+** * **vC+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

mul-float/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA** * **vB** and stores the result in **vA**
*All registers are of type **float***

mul-float ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)

Calculates **vB** * **vC** and stores the result in **vA**
*All registers are of type **float***

mul-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA** * **vB** and stores the result in **vA**
*All registers are of type **int***

mul-int/lit16 ***vA, vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB** * **0xC** and stores the result in **vA**
*All registers are of type **int***

mul-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB** * **0xC** and stores the result in **vA**
*All registers are of type **int***

mul-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** * **vC** and stores the result in **vA**
*All registers are of type **int***

mul-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)

Calculates **vA+** ***** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

mul-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+** ***** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

or-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA** **|** **vB** and stores the result in **vA**
*All registers are of type **int***

or-int/lit16 ***vA, vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB** **|** **0xC** and stores the result in **vA**
*All registers are of type **int***

or-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB** **|** **0xC** and stores the result in **vA**
*All registers are of type **int***

or-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** **|** **vC** and stores the result in **vA**
*All registers are of type **int***

or-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+** **|** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

or-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+** **|** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

rem-double/2addr ***vA+, vB+***
A: Source register **pair** 1 / Destination register

B: Source register **pair** 2
Calculates **vA+ % vB+** and stores the result in **vA+**
Note: % means the remainder after dividing
All registers are of type ***double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

rem-double ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vBB % vCC** and stores the result in **vAA**
Note: % means the remainder after dividing
All registers are of type ***double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

rem-float/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA % vB** and stores the result in **vA**
Note: % means the remainder after dividing
All registers are of type ***float***

rem-float ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** % **vC** and stores the result in **vA**
Note: % means the remainder after dividing
All registers are of type ***float***

rem-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA % vB** and stores the result in **vA**
Note: % means the remainder after dividing
All registers are of type ***int***

rem-int/lit16 ***vA, vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB % 0xC** and stores the result in **vA**
Note: % means the remainder after dividing
All registers are of type ***int***

rem-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB % 0xC** and stores the result in **vA**
Note: % means the remainder after dividing
All registers are of type ***int***

rem-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)

C: Source register 2 (4 bits)
Calculates **vB** % **vC** and stores the result in **vA**
Note: % means the remainder after dividing
*All registers are of type **int***

rem-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+ / vC+** and stores the result in **vA+**
Note: % means the remainder after dividing
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

rem-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ % vC+** and stores the result in **vA+**
Note: % means the remainder after dividing
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

shl-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA << vB** and stores the result in **vA**
*All registers are of type **int***

shl-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB << 0xC** and stores the result in **vA**
*All registers are of type **int***

shl-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB << vC** and stores the result in **vA**
*All registers are of type **int***

shl-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+ << vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

shl-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ << vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

shr-int/2addr ***vA**, **vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA >> vB** and stores the result in **vA**
*All registers are of type **int***

shr-int/lit8 ***vA**, **vB**, **0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB >> 0xC** and stores the result in **vA**
*All registers are of type **int***

shr-int ***vA**, **vB**, **vC***
A: Destination register (4 bits)
B: Source register1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB >> vC** and stores the result in **vA**
*All registers are of type **int***

shr-long/2addr ***vA+**, **vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+ >> vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

shr-long ***vA+**, **vB+**, **vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ >> vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

sub-double/2addr ***vA+**, **vB+***
A: Source register **pair** 1 / Destination register (8 bits)
B: Source register **pair** 2 (8 bits)
Calculates **vA+ - vB+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

sub-double ***vA+**, **vB+**, **vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ - vC+** and stores the result in **vA+**
*All registers are of type **double***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

sub-float/2addr ***vA**, **vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA - vB** and stores the result in **vA**
*All registers are of type **float***

sub-float ***vA**, **vB**, **vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB - vC** and stores the result in **vA**
*All registers are of type **float***

sub-int/2addr ***vA**, **vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA - vB** and stores the result in **vA**
*All registers are of type **int***

sub-int/lit16 ***vA**, **vB**, **0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB - 0xC** and stores the result in **vA**
*All registers are of type **int***

sub-int/lit8 ***vA**, **vB**, **0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB - 0xC** and stores the result in **vA**
*All registers are of type **int***

sub-int ***vA**, **vB**, **vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB** - **vC** and stores the result in **vA**
*All registers are of type **int***

sub-long/2addr ***vA+**, **vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+ - vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

sub-long ***vA+**, **vB+**, **vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+ - vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

ushr-int/2addr ***vA**, **vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates unsigned **vA >> vB** and stores the result in **vA**
*All registers are of type **int***

ushr-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates unsigned **vB >> 0xC** and stores the result in **vA**
*All registers are of type **int***

ushr-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates unsigned **vB >> vC** and stores the result in **vA**
*All registers are of type **int***

ushr-long/2addr ***vA+, vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates unsigned **vA+ >> vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

ushr-long ***vA+, vB+, vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates unsigned **vB+ >> vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

xor-int/2addr ***vA, vB***
A: Source register 1 / Destination register (4 bits)
B: Source register 2 (4 bits)
Calculates **vA XOR vB** and stores the result in **vA**
*All registers are of type **int***

xor-int/lit16 ***vA, vB, 0xC***
A: destination register (4 bits)
B: source register (4 bits)
C: signed int constant literal value (16 bits)
Calculates **vB XOR 0xC** and stores the result in **vA**
*All registers are of type **int***

xor-int/lit8 ***vA, vB, 0xC***
A: destination register (8 bits)
B: source register (8 bits)
C: signed int constant literal value (8 bits)
Calculates **vB XOR 0xC** and stores the result in **vA**
*All registers are of type **int***

xor-int ***vA, vB, vC***
A: Destination register (4 bits)
B: Source register 1 (4 bits)
C: Source register 2 (4 bits)
Calculates **vB XOR vC** and stores the result in **vA**
*All registers are of type **int***

xor-long/2addr ***vA+***, ***vC+***
A: Source register **pair** 1 / Destination register (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vA+** **XOR** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

xor-long ***vA+***, ***vB+***, ***vC+***
A: Destination register **pair** (8 bits)
B: Source register **pair** 1 (8 bits)
C: Source register **pair** 2 (8 bits)
Calculates **vB+** **XOR** **vC+** and stores the result in **vA+**
*All registers are of type **long***
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

14. Return

return-object vAA
A: Return value register (8 bits)
Return from an **object**-returning method with the **object**-reference in **vAA**.

return-void
Return from a **void** method without a value.

return-wide vA+
A: Return value register **pair** (8 bits)
Return a **double/long** (64-bit) value in **vA+**.
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

return vAA
A: Return value register (8 bits)
Return from a non-**object** value-returning method with value **vAA**.

15. Static accessor

sget-boolean **vA**, **Lclass;->field:type**
A: Destination register
B: Reference to a **static boolean** field
Stores the **boolean** value of **field** into **vA**

sget-byte **vA**, **Lclass;->field:type**
A: Destination register
B: Reference to a **static byte** field
Stores the **byte** value of **field** into **vA**

sget-char **vA**, **Lclass;->field:type**
A: Destination register
B: Reference to a **static char** field
Stores the **char** value of **field** into **vA**

sget-object **vA**, **Lclass;->field:type**
A: Destination register
B: Reference to a **static object** field
Stores the **object** reference of **field** into **vA**

sget-short **vA**, **Lclass;->field:type**

A: Destination register
B: Reference to a **static short** field
Stores the **short** value of **field** into **vA**

sget-wide **vA+**, **Lclass**;->**field**:**type**
A: Destination register **pair**
B: Reference to a **static** field
Stores the **long** or **double** value of **field** into **vA+**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

sget **vA**, **Lclass**;->**field**:**type**
A: Destination register
B: Reference to a **static integer** field
Stores the value of **field** into **vA**

sput-boolean **vA**, **Lclass**;->**field**:**type**
A: Source register
B: Reference to a **static boolean** field
Stores the **boolean** value of **vA** into **field**

sput-byte **vA**, **Lclass**;->**field**:**type**
A: Source register
B: Reference to a **static byte** field
Stores the **byte** value of **vA** into **field**

sput-char **vA**, **Lclass**;->**field**:**type**
A: Source register
B: Reference to a **static char** field
Stores the **char** value of **vA** into **field**
sput-object **vA**, **Lclass**;->**field**:**type**
A: Source register
B: Reference to a **static object** field
Stores the **object** reference of **vA** into **field**

sput-short **vA**, **Lclass**;->**field**:**type**
A: Source register
B: Reference to a **static short** field
Stores the **short** value of **vA** into **field**

sput-wide **vA+**, **Lclass**;->**field**:**type**
A: Source register **pair**
B: Reference to a **static** field
Stores the **long** or **double** value of **vA+** into **field**
Register pairs occupy **vX** and **vX+1**. eg **v1, v2.**

sput **vA**, **Lclass**;->**field**:**type**
A: Source register
B: Reference to a **static** field
Stores the value of **vA** into **field**

16. Switch

packed-switch **vAA**, **:target**
A: Register to test
target: Target label of packed-switch table
Implements a switch statement where the case constants are sequencial. The instruction uses an index table. **vAA**

indexes into this table to find the offset of the instruction for a particular case. If **vAA** falls out of the index table, the execution continues on the next instruction (default case). **packed-switch** is used if the possible values of **vAA** are sequencial regardless of the lowest value.

A packed-switch table is written as

```
    :target
    .packed-switch 0x1      # 0x1 = the lowest value of vAA
        :pswitch_0      # Branch to pswitch_0 if vAA == 0x1
        :pswitch_1      # Branch to pswitch_1 if vAA == 0x2
    .end packed-switch
```

sparse-switch vAA, :target

A: Register to test

target: Target label of packed-switch table

Implements a switch statement where the case constants are not sequencial. The instruction uses a lookup table with case constants and offsets for each case constant. If there is no match in the table, execution continues on the next instruction (default case).

A sparse-switch table is written as

```
    :target
    .sparse-switch
        0x3 -> :sswitch_1  # Branch to sswitch_1 if vAA == 0x3
        0x65 -> :sswitch_2      # Branch to sswitch_2 if vAA == 0x65
    .end sparse-switch
```