The Java Virtual Machine

CS520
Dept. of Computer Science
Univ. of New Hampshire

The Java VM

stack-based VM

allows for more compact programs

operands are implicit

e.g. vm520: addi r5, r6

JVM: iadd

The Java VM

Variable-length instructions

- iadd – one byte (opcode)
- dstore 4 – two bytes (opcode & local slot #)

Java VM

Key run-time data structures

pc – address of instruction currently being executed

stack – stores frames

↳ block of memory created for a method invocation

contains: local variables
partial results
return address

think about recursion

Java VM

Key run-time data structures (continued)

heap - stores objects

method area - stores instructions for methods

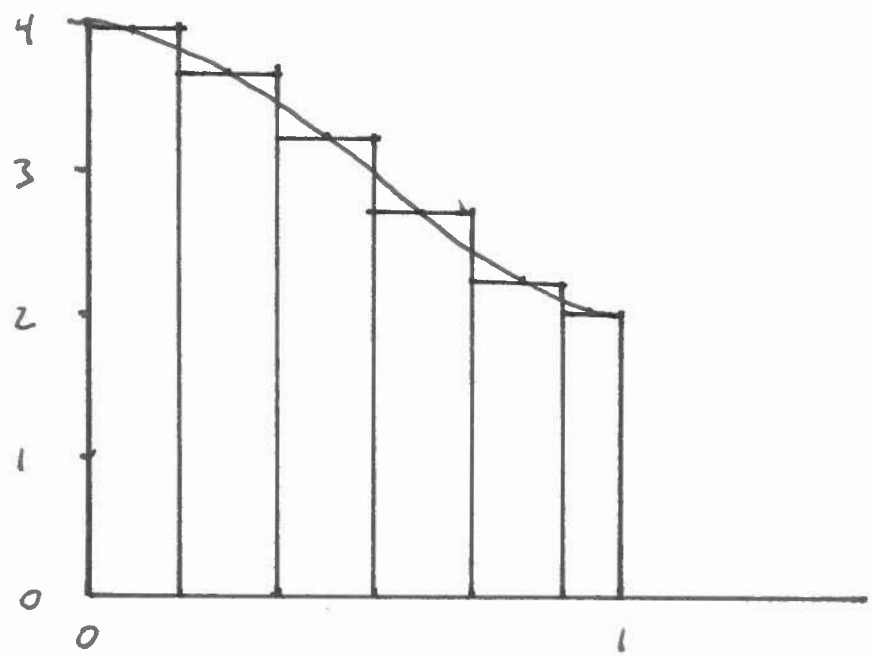
constant pool - stores constant data & meta-data

↳ field names & types etc.

↑ available to program via reflection

Computing π

numeric integration: $\int_0^1 \frac{4}{1+x^2}$



```
// Approximation of pi by calculating the area under the curve 4/(1+x^2)
// between 0 and 1 using numerical integration.
//
// The idea behind this numerical integration is to divide the area
// under the curve into rectangles. The width of every rectangle is
// the same. The height of each rectangle is chosen so that the curve
// intersects the top of the rectangle at its midpoint. The sum of
// the rectangles' areas is an approximation to the area under the.
// As the width of the rectangles decreases, so does the difference
// between the area of the rectangles and the area under the curve.

public class pi {

  final static int INTERVALS = 400000;

  public static void main(String args[])
  {                                    → slot ø
1 -  int i;
2 -  double sum;            // sum of rectangle areas
4 -  double width;          // width of a rectangle
6 -  double x;              // midpoint of rectangle on x axis

    width = (double) 1.0 / (float) INTERVALS;

    sum = (double) 0.0;
    x = width * (double) .5;
    for (i = 0; i < INTERVALS; i++) {
      sum += ((double) 4.0) / (((double) 1.0) + x * x);
      x += width;
    }
    sum *= width;

    System.out.println("Estimation of pi is " + sum);
  }
}
```

pi.java

javac pi.java

→ pi.class

```
Compiled from "pi.java"
public class pi extends java.lang.Object{
static final int INTERVALS;

public pi();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   ldc2_w  #2; //double 2.5E-6d
   3:   dstore  4
   5:   dconst_0
   6:   dstore_2
   7:   dload   4
   9:   ldc2_w  #4; //double 0.5d
   12:  dmul
   13:  dstore  6
   15:  iconst_0
   16:  istore_1
   17:  iload_1
   18:  ldc     #6; //int 400000
   20:  if_icmpge       50
   23:  dload_2
   24:  ldc2_w  #7; //double 4.0d
   27:  dconst_1
   28:  dload   6
   30:  dload   6
   32:  dmul
   33:  dadd
   34:  ddiv
   35:  dadd
   36:  dstore_2
   37:  dload   6
   39:  dload   4
   41:  dadd
   42:  dstore  6
   44:  iinc    1, 1
   47:  goto    17
   50:  dload_2
   51:  dload   4
   53:  dmul
   54:  dstore_2
   55:  getstatic       #9; //Field java/lang/System.out:Ljava/io/PrintStream;
   58:  new     #10; //class java/lang/StringBuilder
   61:  dup
   62:  invokespecial   #11; //Method java/lang/StringBuilder."<init>":()V
   65:  ldc     #12; //String Estimation of pi is
   67:  invokevirtual   #13; //Method java/lang/StringBuilder.append:(Ljava/lang
   70:  dload_2
   71:  invokevirtual   #14; //Method java/lang/StringBuilder.append:(D)Ljava/la
   74:  invokevirtual   #15; //Method java/lang/StringBuilder.toString:()Ljava/l
   77:  invokevirtual   #16; //Method java/io/PrintStream.println:(Ljava/lang/St
   80:  return

}
```

dissassembly of pi.class

javap -c pi

14 00 02

39 04

0E

A2 00 1E

new pc = cur pc + offset

17 = 47 + (-30)

A7 FF E2

1E

0001 1110

1110 0001

+1

1111 1111 1110 0010

F F E 2

The Java™ Virtual Machine Specification

Java SE 7 Edition

Tim Lindholm
Frank Yellin
Gilad Bracha
Alex Buckley

2012-07-27

ldc2_w *ldc2_w*

Operation Push `long` or `double` from runtime constant pool (wide index)

Format

ldc2_w
indexbyte1
indexbyte2

Forms *ldc2_w* = 20 (0x14)

Operand Stack

... →

..., *value*

Description The unsigned *indexbyte1* and *indexbyte2* are assembled into an unsigned 16-bit index into the runtime constant pool of the current class (§2.6), where the value of the index is calculated as (*indexbyte1* << 8) | *indexbyte2*. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index must be a runtime constant of type `long` or `double` (§5.1). The numeric *value* of that runtime constant is pushed onto the operand stack as a `long` or `double`, respectively.

Big Endian

Notes Only a wide-index version of the *ldc2_w* instruction exists; there is no *ldc2* instruction that pushes a `long` or `double` with a single-byte index.

The *ldc2_w* instruction can only be used to push a value of type `double` taken from the double value set (§2.3.2) because a constant of type `double` in the constant pool (§4.4.5) must be taken from the double value set.

dstore *dstore*

Operation Store `double` into local variable

Format

dstore
index

Forms *dstore* = 57 (0x39)

Operand Stack ..., *value* →

...

Description The *index* is an unsigned byte. Both *index* and *index*+1 must be indices into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion (§2.8.3), resulting in *value*'. The local variables at *index* and *index*+1 are set to *value*'.

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

dconst_<d> *dconst_<d>*

Operation Push `double`

Format

dconst_<d>

Forms *dconst_0* = 14 (0xe)

dconst_1 = 15 (0xf)

Operand Stack ... →

..., *<d>*

Description Push the `double` constant *<d>* (0.0 or 1.0) onto the operand stack.

if_icmp<cond> *if_icmp<cond>*

Operation Branch if `int` comparison succeeds

Format

if_icmp<cond>
branchbyte1
branchbyte2

Forms

if_icmpeq = 159 (0x9f)

if_icmpne = 160 (0xa0)

if_icmplt = 161 (0xa1)

if_icmpge = 162 (0xa2)

if_icmpgt = 163 (0xa3)

if_icmple = 164 (0xa4)

Operand Stack

..., *value1*, *value2* →

...

Description Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

- *if_icmpeq* succeeds if and only if *value1* = *value2*
- *if_icmpne* succeeds if and only if *value1* ≠ *value2*
- *if_icmplt* succeeds if and only if *value1* < *value2*
- *if_icmple* succeeds if and only if *value1* ≤ *value2*
- *if_icmpgt* succeeds if and only if *value1* > *value2*
- *if_icmpge* succeeds if and only if *value1* ≥ *value2*

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* `<<` 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_icmp<cond>* instruction. The target address must

goto *goto*

Operation Branch always

Format

goto
branchbyte1
branchbyte2

Forms *goto* = 167 (0xa7)

Operand Stack No change

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

Note: We use this font for Prolog code and code fragments, and `this font` for Java virtual machine instructions and `class` file structures. Commentary, designed to clarify the specification, is given as indented text between horizontal lines:

Commentary provides intuition, motivation, rationale, examples, etc.

4.1 The `ClassFile` Structure

A `class` file consists of a single `ClassFile` structure:

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items in the `ClassFile` structure are as follows:

`magic`

The `magic` item supplies the magic number identifying the `class` file format; it has the value `0xCAFEBABE`.

`minor_version, major_version`

The values of the `minor_version` and `major_version` items are the minor and major version numbers of this `class` file. Together, a major and a minor version number determine the version of the `class` file format. If a `class` file has major version number M and minor version number m, we denote the version of its `class` file format as M.m. Thus, `class` file format versions may be ordered lexicographically, for example, 1.5 < 2.0 < 2.1.

A Java virtual machine implementation can support a `class` file format of version v if and only if v lies in some contiguous range Mi.0 ≤ v ≤ Mj.m.

```
0000000 ca fe ba be 00 00 00 32 00 38 0a 00 12 00 20 06
0000020 3e c4 f8 b5 88 e3 68 f1 06 3f e0 00 00 00 00 00
0000040 00 03 00 06 1a 80 06 40 10 00 00 00 00 00 00 09
0000060 00 21 00 22 07 00 23 0a 00 0a 00 20 08 00 24 0a
0000100 00 0a 00 25 0a 00 0a 00 26 0a 00 0a 00 27 0a 00
0000120 28 00 29 07 00 2a 07 00 2b 01 00 09 49 4e 54 45
0000140 52 56 41 4c 53 01 00 01 49 01 00 0d 43 6f 6e 73
0000160 74 61 6e 74 56 61 6c 75 65 01 00 06 3c 69 6e 69
0000200 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01
0000220 00 0f 4c 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c
0000240 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61
0000260 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29
0000300 56 01 00 0d 53 74 61 63 6b 4d 61 70 54 61 62 6c
0000320 65 07 00 2c 01 00 0a 53 6f 75 72 63 65 46 69 6c
0000340 65 01 00 07 70 69 2e 6a 61 76 61 0c 00 16 00 17
0000360 07 00 2d 0c 00 2e 00 2f 01 00 17 6a 61 76 61 2f
0000400 6c 61 6e 67 2f 53 74 72 69 6e 67 42 75 69 6c 64
0000420 65 72 01 00 14 45 73 74 69 6d 61 74 69 6f 6e 20
0000440 6f 66 20 70 69 20 69 73 20 0c 00 30 00 31 0c 00
0000460 30 00 32 0c 00 33 00 34 07 00 35 0c 00 36 00 37
0000500 01 00 02 70 69 01 00 10 6a 61 76 61 2f 6c 61 6e
0000520 67 2f 4f 62 6a 65 63 74 01 00 13 5b 4c 6a 61 76
0000540 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 01 00
0000560 10 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65
0000600 6d 01 00 03 6f 75 74 01 00 15 4c 6a 61 76 61 2f
0000620 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 01
0000640 00 06 61 70 70 65 6e 64 01 00 2d 28 4c 6a 61 76
0000660 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 4c
0000700 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67
0000720 42 75 69 6c 64 65 72 3b 01 00 1c 28 44 29 4c 6a
0000740 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 42
0000760 75 69 6c 64 65 72 3b 01 00 08 74 6f 53 74 72 69
0001000 6e 67 01 00 14 28 29 4c 6a 61 76 61 2f 6c 61 6e
0001020 67 2f 53 74 72 69 6e 67 3b 01 00 13 6a 61 76 61
0001040 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 01
0001060 00 07 70 72 69 6e 74 6c 6e 01 00 15 28 4c 6a 61
0001100 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29
0001120 56 00 21 00 11 00 12 00 00 00 01 00 18 00 13 00
0001140 14 00 01 00 15 00 00 00 02 00 06 00 02 00 01 00
0001160 16 00 17 00 01 00 18 00 00 00 1d 00 01 00 01 00
0001200 00 00 05 2a b7 00 01 b1 00 00 00 01 00 19 00 00
0001220 00 06 00 01 00 00 00 0c 00 09 00 1a 00 1b 00 01
0001240 00 18 00 00 00 a4 00 0a 00 08 00 00 00 51 14 00
0001260 02 39 04 0e 49 18 04 14 00 04 6b 39 06 03 3c 1b
0001300 12 06 a2 00 1e 28 14 00 07 0f 18 06 18 06 6b 63
0001320 6f 63 49 18 06 18 04 63 39 06 84 01 01 a7 ff e2
0001340 28 18 04 6b 49 b2 00 09 bb 00 0a 59 b7 00 0b 12
0001360 0c b6 00 0d 28 b6 00 0e b6 00 0f b6 00 10 b1 00
0001400 00 00 02 00 19 00 00 00 2a 00 0a 00 00 00 17 00
0001420 05 00 19 00 07 00 1a 00 0f 00 1b 00 17 00 1c 00
0001440 25 00 1d 00 2c 00 1b 00 32 00 1f 00 37 00 21 00
0001460 50 00 22 00 1c 00 00 00 11 00 02 ff 00 11 00 05
0001500 07 00 1d 01 03 03 03 00 00 20 00 01 00 1e 00 00
0001520 00 02 00 1f
0001524
```

← code for main method