The Java Virtual Machine

CS 520
Dept. of Computer Science
Univ. of New Hampshire
The Java VM

Stack-based VM

allows for more compact programs
operands are implicit

e.g. vn520: addi r5, r6

JVM: isdd
The Java VM

Variable-length instructions

*isdd* - 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 $\pi$

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 curve. As the width of the rectangles decreases, so does the difference between the area of the rectangles and the area under the curve.

```java
public class pi {
    final static int INTERVALS = 400000;

    public static void main(String args[]) {
        int i;
        double sum; // sum of rectangle areas
        double width; // width of a rectangle
        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);
    }
}
```

\[\pi\]
public class pi extends java.lang.Object{
    static final int INTERVALS;
    public pi();
        Code: 0: aload_0
               1: invokevirtual #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
               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/la
            77: invokevirtual #16; //Method java/io/PrintStream.println:(Ljava/lang/St
**ldc2_w**

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

**Format**

<table>
<thead>
<tr>
<th>ldc2_w</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**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.

**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**

**Operation**  
Store `double` into local variable

**Format**

```
+-------+
|      |   |
|  dstore |   |
|  index  |   |
+-------+
```

**Forms**  
`dstore = 57 (0x39)`

**Operand**

`, value →`

**Stack**

`...`

**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>**

**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>

Operation
Branch if int comparison succeeds

Format

<table>
<thead>
<tr>
<th>if_icmp&lt;cond&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

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
..., value1, value2 →

Stack
...

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_icmpge 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

Operation  Branch always

Format

<table>
<thead>
<tr>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

Forms  

\[
goto = 167 \text{ (0xa7)}
\]

Operand  No change

Stack

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

```java
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.