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. VM520: addi r5, r6
JVM: iadd
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

\[ \text{Field names \& types etc.} \]

\[ \text{available to program via reflection} \]
Computing \( \pi \)

Numeric integration: \( \int_0^1 \frac{4}{1+x^2} \)
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);
    }
}

\[
\pi = \sum \frac{4}{1+x^2} 
\]

\[
\text{JakeVCE \ pi.java} \quad \pi \text{.class}
\]
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
    1:   dstore 4
    2:   dconst_0
    3:   dstore_2
    4:   dload 4
    5:   ldc2_w #4; //double 0.5d
    6:   dmul
    7:   dstore 6
    8:   iconst_0
    9:   istore_1
    10:  iload_1
    11:  ldc #6; //int 400000
    12:  if_icmpge 50
    13:  dload_2
    14:  ldc2_w #7; //double 4.0d
    15:  dconst_1
    16:  dload 6
    17:  dload 6
    18:  dmul
    19:  dadd
    20:  ddiv
    21:  dadd
    22:  dstore_2
    23:  dload 6
    24:  dload 4
    25:  dadd
    26:  dstore 6
    27:  iinc 1, 1
    28:  goto 17
    29:  dload_2
    30:  dload 4
    31:  dmul
    32:  dstore_2
    33:  getstatic  #9; //Field java/lang/System.out:Ljava/io/PrintStream;
    34:  new    #10; //class java/lang/StringBuilder
    35:  dup
    36:  invokespecial #11; //Method java/lang/StringBuilder."<init>":()V
    37:  ldc     #12; //String Estimation of pi is
    38:  invokevirtual #13; //Method java/lang/StringBuilder.append:(Ljava/lang
    39:  dload_2
    40:  invokevirtual #14; //Method java/lang/StringBuilder.append:(D)Ljava/lang
    41:  invokevirtual #15; //Method java/lang/StringBuilder.toString:(Ljava/lang
    42:  invokevirtual #16; //Method java/io/PrintStream.println:(Ljava/lang/St
    43:  return
}
The Java™ Virtual Machine Specification

Java SE 7 Edition

Tim Lindholm
Frank Yellin
Gilad Bracha
Alex Buckley

2012-07-27
\textit{ldc2\_w}  

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

\begin{verbatim}
 ld\_c2\_w
 indexbyte1
 indexbyte2
\end{verbatim}

Forms: $ldc2\_w = 20$ (0x14)

Operand: ...

Stack: ..., value

Description: The unsigned \textit{indexbyte1} and \textit{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 \(^{(indexbyte1 \ll 8)} \mid indexbyte2\). 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 \texttt{long} or \texttt{double} (§5.1). The numeric \textit{value} of that runtime constant is pushed onto the operand stack as a \texttt{long} or \texttt{double}, respectively.

Notes: Only a wide-index version of the \textit{ldc2\_w} instruction exists; there is no \textit{ldc2} instruction that pushes a \texttt{long} or \texttt{double} with a single-byte index.

The \textit{ldc2\_w} instruction can only be used to push a value of type \texttt{double} taken from the double value set (§2.3.2) because a constant of type \texttt{double} in the constant pool (§4.4.5) must be taken from the double value set.
The _dstore_ opcode can be used in conjunction with the _wide_ instruction (§8.9) to access a local variable using a two-byte
and index+1 are set to value.

**Description**

The index is an unsigned byte. Both index and index+1 must be

...  
  \[ \texttt{dstore} \leftarrow \text{value} \]

(\texttt{dstore} = 57 (0x39))

**Operation**

Store double into local variable

```java
_dstore
```
Push the double constant $<p>$ (0.0 or 1.0) onto the operand stack.

**Description**

$<p>...$

Stack

$...$

Operand

**Operands**

$decost$ = 1.5 (0x1F)

$decost$ = 1.4 (0xe)

**Form**

Push double

**Operation**

$decost$
The operand of this JVM instruction is an offset from the address of the
branch target (the operand is calculated to be (branch target - 8))
branchtarget.
If the comparison succeeds, the unsigned branch target and

- if-icmpge succeeds if and only if value1 >= value2
- if-icmpgt succeeds if and only if value1 > value2
- if-icmple succeeds if and only if value1 <= value2
- if-icmplt succeeds if and only if value1 < value2
- if-icmpne succeeds if and only if value1 != value2
- if-icmpneq succeeds if and only if value1 = value2

The results of the comparison are as follows:

- if-icmpne succeeds if the operands are of type int. They are both popped

<table>
<thead>
<tr>
<th>Branch</th>
<th>if-icmpcond</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch1</td>
<td>if-icmpcond</td>
</tr>
<tr>
<td>branch2</td>
<td></td>
</tr>
</tbody>
</table>

The Java Virtual Machine Instruction Set
The method that contains this 8010 instruction target address must be that of an opcode of an instruction within offset from the address of the opcode of this 8010 instruction. The construct a signed 16-bit branchoffset, where branchoffset is the unsigned bytes 'branchhiword' and 'branchlow'.

**Description**

No change

8010 = 167 (0x47)

**Operands**

- **Branchhiword**
- **Branchlow**
- **8010**

**Format**

Branch always

**Operation**

8010
A Java virtual machine implementation can support a class file format of version 0, 1, or 2 only if a class in some contiguous range of major and minor version numbers has been defined as major and minor version numbers of this class file. Together, a major and a minor version number determine the version of the class file. The major version number is a nonnegative integer, and the minor version number is also a nonnegative integer.

The values of the magic, minor version, major version, and magic version items are the minor and major version numbers, respectively. The magic item supplies the magic number identifying the class file format:

```
magic
```

The items in the class file structure are as follows:

```
{ ...
attributes_count
attribute_count
method_count
method_count
field_count
field_count
interface_count
interface_count
super_classes
this_class
access_flags
constant_pool
constant_pool
magic
}
```

A class file consists of a single class file structure:

```
The Class File Structure
```

4.1 The Class File Structure

Community provides intuitive motivation, rationale, examples, etc.

To clarify the specification, is given an aide to understand horizontal lines:

Java virtual machine instructions and class file structures. Community, designed
Note: We use this form for encoding code and code fragments, and this form for

4.1 The Class File Structure