COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia

`http://www.cse.unsw.edu.au/~cs3131`

`http://www.cse.unsw.edu.au/~cs9102`

Week 7: JVM (Two Lectures)

1st

1. Our code generation

2nd

2. JVM:
   - Data types
   - Operand stack
   - Local variable array (indices)
   - Instructions ($\Longleftrightarrow$ Jasmin instructions)
   - Parameter-passing ($\Longleftrightarrow$ Jasmin method invocations)

Week 7 (2nd Lecture): Jasmin (or JVM) Instructions

1. Arithmetic Instructions
2. Load and store instructions
3. Control transfer instructions
4. Type conversion instructions
5. Operand stack management instructions
6. Object creation and manipulation
7. Method invocation instructions
8. Throwing instructions (not used)
9. Implementing **finally** (not used)
10. Synchronisation (not used)

Arithmetic Instructions ($3.11.3, JVM Spec)

- add: iadd, fadd,
- subtract: isub, fsub
- multiply: imul, fmul
- divide: idiv, fdiv
- negative: ineg, fneg
- comparison: fcmpg, fcmpl
- ...

Load and Store Instructions

- Loading a local variable into the operand stack:

```
iload iload_0, ..., iload_3
fload fload_0, ..., fload_3
aload aload_0, ..., aload_3 // for array and object refs
iaload                      // load from an int array
faload                      // load from a float array
```

- Storing a value from the operand stack into a local variable:

```
istore istore_0, ..., istore_3
fstore fstore_0, ..., fstore_3
astore astore_0, ..., astore_3 // for array and object refs
iastore                        // store into an int array
fastore                        // store into a float array
```

- Load a constant into the operand stack:

```
bipush, sipush, ldc, iconst_m1, iconst_0, ..., iconst_5,
fconst_0, ..., fconst_2
```

Example: Load and Stored (Slide 425 Repeated)

1. Integral expression:

```
1 + 2 * 3 + 4
```

2. Jasmin code:

```
iconst_1
iconst_2
iconst_3
imul
iadd
iconst_4
iadd
```

Example: Load and Store (Cont'd)

1. Integral expression:

```
1 + 100 * 200 + 40000
```

2. Jasmin code:

```
iconst_1
bipush 100
sipush 200
imul
iadd
ldc 40000
iadd
```

3. bipush: $-2^7 - 2^7 - 1$
4. sipush: $-2^{15} - 2^{15} - 1$
5. ldc val, where val is an int, float or string

Example: Load and Store (Cont'd)

1. Floating-point expression:

```
1.0F + 2.0F * 3.0F + 4.0F
```

2. Jasmin code:

```
fconst_1
fconst_2
ldc 3.0
fmul
fadd
ldc 4.0
fadd
```

Example: Load and Store for Arrays

1. Array operations:

```
a[0] = 100;
i = a[0]
```

2. Jasmin code – (assuming a and i have the indices 1 and 2, resp.)

```
// a[0] = 100;
aload_1
iconst_0
bipush 100
iastore
// i = a[0];
aload_1
iconst_0
iaload
istore_2
```

Control Transfer Instructions

1. Unconditional: goto
2. Instructions on int:

```
ifeq  ifne  ifle  iflt  ifne  ifge
if_icmpeq  if_icmpne  if_icmple
if_icmplt  if_icmpge  if_icmpgt
```

3. For floating-point operands, use first

   `fcmpg or fcmpl`

   and then

   `ifeq  ifne  ifle  iflt  ifne  ifge`

if_icmge label

```
      op stack (before)
+--------------------
| ... value1 value2
+--------------------
      op stack (after)
+-----
| ...
+-----
```

1. Pop off the two ints off and compare them
2. If value1 $>=$ value2, jump to label. Otherwise, execution continues at the next instruction

Other if_icmpxx instructions are similar.

Example 1: Control Transfer Instructions

1. Ctrl1.java

```
public class Ctrl1 {
  public static void main(String argv[]) {
     int i = 1, int j = 2, int k;
     if (i < j)
       k = 0;
     else
       k = 1;
  }
}
```

2. Ctrl1.j (irrelevant lines removed)

```
.method public static main([Ljava/lang/String;)V
        iconst_1
        istore_1
        iconst_2
        istore_2
        iload_1
        iload_2
        if_icmpge Label0
        iconst_0
        istore_3
        goto Label1
Label0:
        iconst_1
        istore_3
Label1:
        return
.end method
```

Example 2: Control Transfer Instructions

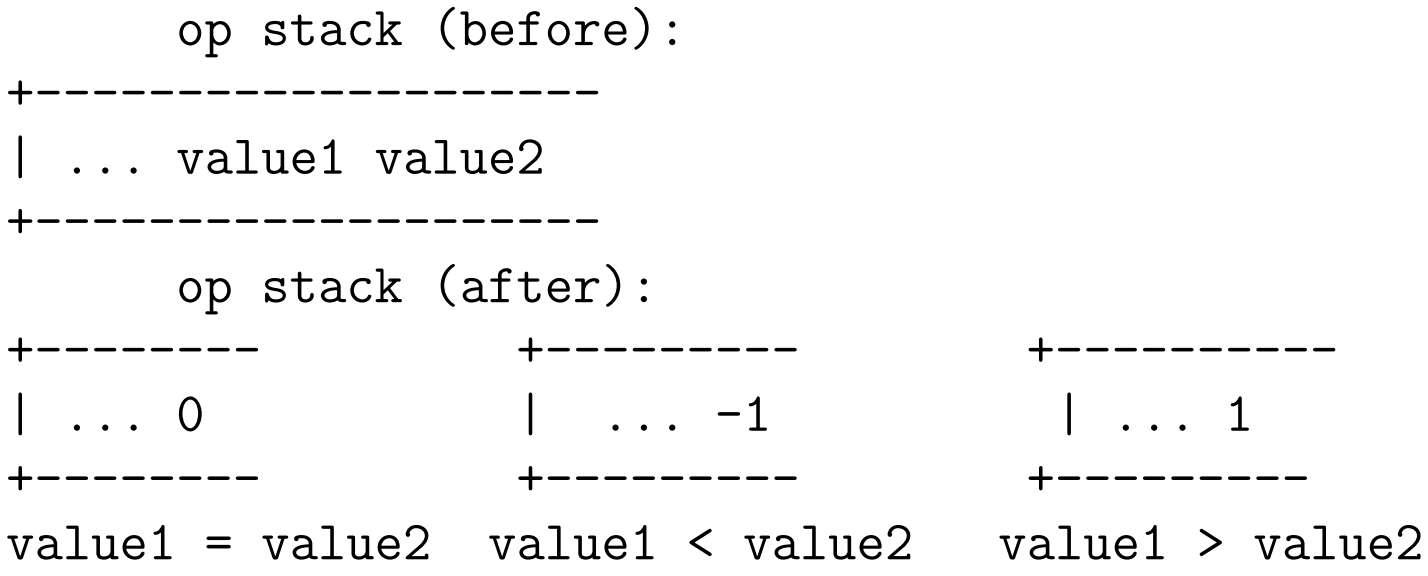
1. Ctrl2.java

```
public class Ctrl1 {
  public static void main(String argv[]) {
     float i = 1, float j = 2, float k;
     if (i < j)
       k = 0.0F;
     else
       k = 1.0F;
  }
}
```

2. Ctrl2.j (irrelevant lines removed)

```
.method public static main([Ljava/lang/String;)V
        fconst_1
        fstore_1
        fconst_2
        fstore_2
        fload_1
        fload_2
        fcmpg
        ifge Label0
        fconst_0
        fstore_3
        goto Label1
Label0:
        fconst_1
        fstore_3
Label1:
        return
.end method
```

fcmpg and fcmpl



- If either is NaN, fcmpg pushes 1 and fcmpl pushes -1.
- See JVM SPEC §7.5 for an explanation why the two instructions are provided (`https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-3.html#jvms-3.5`)

Type Conversion Instructions

1. Source:

```
int i = 1;      // index 1
float f = i;    // index 2
```

2. Jasmin code:

```
iconst_1
istore_1
iload_1
i2f
fstore 2
```

3. Only i2f is used in the VC compiler

4. i2c, i2b, f2i, etc. not used

Operand Stack Management Instructions

- Instructions:
  - dup: duplicate the stack top operand
  - pop: remove the stack top operand
  - swap: swap the top two operands
  - others: pop2, dup2, etc.
- Only the first two will be used in the VC compiler:
  - dup: for translating a = b = ...
  - pop: for translating expression statements such as 1;

Example: dup in Translating Compound Assignments

1. Assignment expressions:

```
int i; \\ index 1
int j; \\ index 2
int k; \\ index 3
i = j = k = 1;
```

2. Jasmin code:

```
iconst_1
dup
istore_3
dup
istore_2
istore_1
```

Example: pop in Translating VC Expression Statements

1. Assignment expressions:

```
int i; // index 1
1 + (i = 2);
```

2. Jasmin code:

```
iconst_1
iconst_2
dup
istore_1
iadd
pop
```

Method Invocation Instructions

- Method calls:

```
invokestatic
invokevirtual
invokespecial // also known as invokenonvirtual
  -- the instance initialisation method <init>
  -- a private method of "this"
  -- a method in a super class of "this"
invokeinterface
```

(why invokeinterface?
stackoverflow.com/questions/1504633/what-is-the-point-of-invokeinterface)

- Method returns:

```
return
ireturn
freturn
...
```

The Syntax for Method Invocation Instructions

- The syntax for invokestatic/virtual/special

  `invokexx method-spec`

  where method-spec consists of a classname, a method name and a descriptor.

- invokeinterface not used in the VC compiler
- invokespecial used only once (but already done for you in the supporting code provided).

Method Invocation Instructions (Cont'd)

- invokestatic

```
       op stack                      after
+----------------------   +--------------------
| ... arg1 arg2 ... argn   | ... result (if any)
+----------------------   +-------------------
        before
```

- invokevirtual and invokespecial

```
       op stack                       after
+-------------------------  +--------------------
| ... objref arg1 ... argn   | ... result (if any)
+-------------------------  +-------------------
        before
```

  - invokevirtual: on the dynamic type of objref
  - invokespecial: based on the static class of objref

- Note that programmer-defined operators (methods) are treated exactly the same as the built-in operators (e.g., + and −)

Example: Static Method Invocation

1. Met1.java:

```
public class Met1 {
  static int add(int i1, int i2) {
    return i1 + i2;
  }
  public static void main(String argv[]) {
    add(1, 2);
  }
}
```

2. Met1.j (irrelevant lines removed):

```
.method static add(II)I .limit stack 2 .limit locals 2
        iload_0
        iload_1
        iadd
        ireturn
.end method
.method public static main([Ljava/lang/String;)V .limit stack 2 .limit locals 1
        iconst_1
        iconst_2
        invokestatic Met1/add(II)I
        pop
        return
.end method
```

Example 9: Instance Method Invocation

1. Met2.java:

```
public class Met2 {
  int add(int i1, int i2) {
    return i1 + i2;
  }
  public static void main(String argv[]) {
    Met2 m = new Met2();
    m.add(1, 2);
  }
}
```

2. Met2.j (irrelevant lines removed):

```
.method  add(II)I  .limit stack 2 .limit locals 3
        iload_1
        iload_2
        iadd
        ireturn
.end method
.method public static main([Ljava/lang/String;)V .limit stack 3 .limit locals 2
        new Met2
        dup
        invokespecial Met2/<init>()V
        astore_1
        aload_1
        iconst_1
        iconst_2
        invokevirtual Met2/add(II)I
        return
.end method
```

Polymorphism: invokevirtual and invokespecial

```
public  class Fruit {
  public static void main(String argv[]) {
    Apple apple = new Apple();
    Fruit fruit = apple;
    fruit.whoAmI();
  }
  void whoAmI() {
    System.out.println("This is a fruit.");
    }
}
class Apple extends Fruit {
  void whoAmI() {
    System.out.println("This is an apple.");
    super.whoAmI();
    }
}
```

Fruit.j

```
;; Produced by JasminVisitor (BCEL package)
;; http://www.inf.fu-berlin.de/~dahm/JavaClass/
;; Mon Oct 09 16:09:47 GMT+10:00 2000

.source Fruit.java
.class public Fruit
.super java/lang/Object


.method public <init>()V
.limit stack 1
.limit locals 1
.var 0 is this LFruit; from Label0 to Label1

Label0:
.line 1
        aload_0
        invokespecial java/lang/Object/<init>()V
Label1:
        return

.end method

.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 3
.var 0 is arg0 [Ljava/lang/String; from Label0 to Label1

Label0:
.line 4
        new Apple
        dup
        invokespecial Apple/<init>()V
        astore_1
.line 6
        aload_1
```

```
        astore_2
.line 7
        aload_2
        invokevirtual Fruit/whoAmI()V  <===  a virtual call
Label1:
.line 3
        return

.end method

.method  whoAmI()V
.limit stack 2
.limit locals 1
.var 0 is this LFruit; from Label0 to Label1

Label0:
.line 11
        getstatic java.lang.System.out Ljava/io/PrintStream;
        ldc "This is a fruit."
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
Label1:
.line 10
        return

.end method
```

Apple.j

```
.source Fruit.java
.class  Apple
.super Fruit
.method  <init>()V
.limit stack 1
.limit locals 1
.var 0 is this LApple; from Label0 to Label1
Label0:
.line 16
        aload_0
        invokespecial Fruit/<init>()V
Label1:
        return
.end method

.method  whoAmI()V
.limit stack 2
.limit locals 1
.var 0 is this LApple; from Label0 to Label1

Label0:
.line 18
        getstatic java.lang.System.out Ljava/io/PrintStream;
        ldc "This is an apple."
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
.line 19
        aload_0
        invokespecial Fruit/whoAmI()V      <=== a static call
Label1:
.line 17
        return
.end method
```

Object Creation and Manipulation

- Create a new class instance: new
- Access fields of classes: getstatic, putstatic, getfield, putfield
- Create a new array: newarray, anewarray, multianewarray
  (only the first is used in the VC compiler)
- Load an array component onto the operand stack: iaload, faload, baload, caload, ...
- Store a value from the operand stack as an array component: iastore, fastore, bastore, castore, ...
- ...

Example: Object Creation and Manipulation

- Static and field variables:

```
public class StaticField {
  static int i = 1;
  int j = 1;
  public static void main(String argv[]) {
    StaticField o = new StaticField();
    System.out.println(i);
    System.out.println(o.j);
  }
}
```

- Jasmin code (irrelevant lines removed)

```
.field static i I
.field  j I

.method public static main([Ljava/lang/String;)V
.line 5
        new StaticField
        dup
        invokespecial StaticField/<init>()V
```

```
        astore_1
.line 6
        getstatic java.lang.System.out Ljava/io/PrintStream;
        getstatic StaticField.i I
        invokevirtual java/io/PrintStream/println(I)V
.line 7
        getstatic java.lang.System.out Ljava/io/PrintStream;
        aload_1
        getfield StaticField.j I
        invokevirtual java/io/PrintStream/println(I)V
Label0:
.line 8
        return
.end method
```

See https://luckytoilet.wordpress.com/2010/05/21/

how-system-out-println-really-works/ on how System.out is initialised.

The Syntax for Field Instructions

- Syntax:

  `putstatic/getstatic field-spec type-descriptor`

  where field-spec consists of a classname followed by a field name.

- Examples:

```
getstatic java/lang/System/out Ljava/io/PrintStream;
getstatic StaticField/i I
```

getfield and putfield not used in the VC compiler

Arrays

- Java Statements // assuming a is at slot 1 and a[2] = 5

```
int a[] = new int[10];
a[1] = a[2] + 1;
```

- Bytecode:

```
bipush 10
newarray int
astore_1

aload_1
iconst_1
aload_1
iconst_2
iaload
iconst_1
iadd
iastore
```

In Java, LHS must be evaluated before RHS.

Jasmin Directives

- Jasmin home page

  ```
  .source .class .super .limit .method
  .field static .end .var .line
  ```

- Example: gcd.j (Slide 413)

Jasmin File Structure (syntax.bnf)

```
Jasmin Syntax                                   Jonathan Meyer, April 1996

This file contains a simplified BNF version of the Jasmin syntax.

    jasmin_file ::=
        '.class' [ <access> ] <name> <break>
        '.super' <name> <break>
        [ <fields> ]
        [ <methods> ]


    <fields> ::= <field> [ <field> ... ]

    <field> ::=
        '.field' <access> <name> <signature> [ = <default> ] <break>

    <default> ::=  <int> | <quoted_string> | <float>


    <methods> ::= <method> [ <method> ... ]
```

```
<method> ::=
    '.method' <access> <name> <break>
        [ <statements> ]
    '.end' 'method' <break>

<statements> ::= <statement> [ <statement> ... ]

<statement> ::=
    <directive> <break>
    |
    <instruction> <break>
    |
    <label> ':' <break>

<directive> ::=
    '.limit' 'stack' <val>
    |
    '.limit' 'locals' <val>
    |
    '.throws' <classname>
    |
    '.catch' <classname> 'from' <label1> 'to' <label2> 'using' <label3>
```

```
<instruction> ::= <simple_instruction> | <complex_instruction>

<simple_instruction> ::=
    <insn>
    |
    <insn> <int> <int>
    |
    <insn> <int>
    |
    <insn> <num>
    |
    <insn> <word>
    |
    <insn> <word> <int>
    |
    <insn> <word> <word>
    |
    <insn> <quoted_string>

<complex_instruction> ::=
    <lookupswitch>
    |
    <tableswitch>
```

```
<lookupswitch> ::=
    lookupswitch <nl>
        <int> : <label> <nl>
        <int> : <label> <nl>
        ...
        default : <label>

<tableswitch> ::=
    tableswitch <low>  <nl>
        <label> <nl>
        <label> <nl>
        ...
        default : <label>

<access> ::= <access_item> [ <access_item> ... ]

<access_item> ::=
    'public' | 'private' | 'protected' | 'static' | 'final' |
    'synchronized' | 'volatile' | 'transient' | 'native' |
    'interface' | 'abstract'
```

Understanding Jasmin Assembly Language

1. Read syntax.bnf to understand Jasmin's syntax
2. Read Jasmin User Guide to Jasmin's syntax
3. Read Jasmin instruction reference manual to understand its instructions (1-to-1 mapped to JVM instructions)
4. To under a particular feature, do the following:
   (a) Design a Java program Test.java
   (b) Run javac -g Test.java (-g turns on all debugging info)
   (c) Run jasmind Test.class (jasmind runs java JasminVisitor Test.class) or javap -c Test
   (d) Read Jasmin code in Test.j

Reading (in Order of Increasing Importance)

- on-line JVM instructions
  `http://cs.au.dk/~mis/d0vs/jvmspec/ref-Java.html`
- Play around the tools mentioned in this lecture:
  - All available in the class account
  - Install them on your PC if you have one
- The JVM Spec Book
  - Chapter 3 (instructions)
  - Chapter 7 (more examples on compiling Java)
- "Inside the JVM" book (Chapter 5)

Next Class: Java Bytecode Generation