Native code generation
JOOS programs are compiled into bytecode.

This bytecode can be executed thanks to either:

- an interpreter;
- an Ahead-Of-Time (AOT) compiler; or
- a Just-In-Time (JIT) compiler.

Regardless, bytecode must be implicitly or explicitly translated into native code suitable for the host architecture before execution.
Interpreters:

- are easier to implement;
- can be very portable; but
- suffer an inherent inefficiency:

```java
pc = code.start;
while(true)
    { npc = pc + instruction_length(code[pc]);
      switch (opcode(code[pc]))
      { case ILOAD_1: push(local[1]);
          break;
        case ILOAD: push(local[code[pc+1]]);
          break;
        case ISTORE: t = pop();
          local[code[pc+1]] = t;
          break;
        case IADD: t1 = pop(); t2 = pop();
          push(t1 + t2);
          break;
        case IFEQ: t = pop();
          if (t == 0) npc = code[pc+1];
          break;
        ...
      }
    pc = npc;
  }
```
Ahead-of-Time compilers:

- translate the low-level intermediate form into native code;
- create all object files, which are then linked, and finally executed.

This is not so useful for Java and JOOS:

- method code is fetched as it is needed;
- from across the internet; and
- from multiple hosts with different native code sets.
Just-in-Time compilers:

- merge interpreting with traditional compilation;
- have the overall structure of an interpreter; but
- method code is handled differently.

When a method is invoked for the first time:

- the bytecode is fetched;
- it is translated into native code; and
- control is given to the newly generated native code.

When a method is invoked subsequently:

- control is simply given to the previously generated native code.
Features of a JIT compiler:

- it must be *fast*, because the compilation occurs at run-time (Just-In-Time is really Just-Too-Late);

- it does not generate optimized code;

- it does not compile every instruction into native code, but relies on the runtime library for complex instructions;

- it need not compile every method; and

- it may concurrently interpret and compile a method (Better-Late-Than-Never).
Problems in generating native code:

- **instruction selection:**
  choose the correct instructions based on the native code instruction set;

- **memory modelling:**
  decide where to store variables and how to allocate registers;

- **method calling:**
  determine calling conventions; and

- **branch handling:**
  allocate branch targets.
Compiling JVM bytecode into VirtualRISC:

- map the Java local stack into registers and memory;
- do instruction selection on the fly;
- allocate registers on the fly; and
- allocate branch targets on the fly.

This is successfully done in the Kaffe system.
The general algorithm:

- determine number of slots in frame: 
  locals limit + stack limit + #temps;
- find starts of basic blocks;
- find local stack height for each bytecode;
- emit prologue;
- emit native code for each bytecode; and
- fix up branches.
Naïve approach:

- each local and stack location is mapped to an offset in the native frame;
- each bytecode is translated into a series of native instructions, which
- constantly move locations between memory and registers.

This is similar to the native code generated by a non-optimizing compiler.
Example:

```java
public void foo() {
    int a,b,c;
    a = 1;
    b = 13;
    c = a + b;
}
```

Generated bytecode:

```java
.method public foo()V
    .limit locals 4
    .limit stack 2
    iconst_1  ; 1
    istore_1  ; 0
    ldc 13    ; 1
    istore_2  ; 0
    iload_1   ; 1
    iload_2   ; 2
    iadd      ; 1
    istore_3  ; 0
    return    ; 0
```

- compute frame size = \(4 + 2 + 0 = 6\);
- find stack height for each bytecode;
- emit prologue; and
- emit native code for each bytecode.
Assignment of frame slots:

<table>
<thead>
<tr>
<th>name</th>
<th>offset</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>[fp-32]</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>[fp-36]</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>[fp-40]</td>
</tr>
<tr>
<td>stack</td>
<td>0</td>
<td>[fp-44]</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
<td>[fp-48]</td>
</tr>
</tbody>
</table>

Native code generation:

```
save sp,-136,sp

a = 1;        icontst.1     mov 1,R1
              st R1,[fp-44]

b = 13;       ldc 13       mov 13, R1
              st R1,[fp-32]

istore.1      ld [fp-44], R1
              st R1,[fp-32]

istore.2      ld [fp-44], R1
              st R1,[fp-36]

iadd           ld [fp-48], R1
              ld [fp-44], R2
              add R2,R1,R1
              st R1,[fp-44]

istore.3      ld [fp-44], R1
              st R1,[fp-40]

return         restore
               ret
```
The naïve code is very slow:

- many unnecessary loads and stores, which
- are the *most* expensive operations.

We wish to replace loads and stores:

\[
\begin{align*}
    \text{c = a + b;} & \quad \text{iload}_1 \quad \text{ld} \ [\text{fp-32}], \text{R1} \\
                    & \phantom{\text{iload}_1} \text{st} \ \text{R1},[\text{fp-44}] \\
    \text{iload}_2 & \quad \text{ld} \ [\text{fp-36}], \text{R1} \\
                    & \phantom{\text{iload}_2} \text{st} \ \text{R1},[\text{fp-48}] \\
    \text{iadd} & \quad \text{ld} \ [\text{fp-48}], \text{R1} \\
                    & \phantom{\text{iadd}} \text{ld} \ [\text{fp-44}], \text{R2} \\
                    & \phantom{\text{iadd}} \text{add} \ \text{R2}, \text{R1}, \text{R1} \\
                    & \phantom{\text{iadd}} \text{st} \ \text{R1},[\text{fp-44}] \\
    \text{istore}_3 & \quad \text{ld} \ [\text{fp-44}], \text{R1} \\
                    & \phantom{\text{istore}_3} \text{st} \ \text{R1},[\text{fp-40}] \\
\end{align*}
\]

by registers operations:

\[
\begin{align*}
    \text{c = a + b;} & \quad \text{iload}_1 \quad \text{ld} \ [\text{fp-32}], \text{R1} \\
                    & \phantom{\text{iload}_1} \text{iload}_2 \quad \text{ld} \ [\text{fp-36}], \text{R2} \\
                    & \phantom{\text{iload}_1 \text{iload}_2} \text{iadd} \quad \text{add} \ \text{R1}, \text{R2}, \text{R1} \\
                    & \phantom{\text{iload}_1 \text{iload}_2 \text{iadd}} \text{istore}_3 \quad \text{st} \ \text{R1},[\text{fp-40}] \\
\end{align*}
\]

where \text{R1} and \text{R2} represent the stack.
The fixed register allocation scheme:

- assign $m$ registers to the first $m$ locals;
- assign $n$ registers to the first $n$ stack locations;
- assign $k$ scratch registers; and
- spill remaining locals and locations into memory.

Example for 6 registers ($m = n = k = 2$):

<table>
<thead>
<tr>
<th>name</th>
<th>offset</th>
<th>location</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td>R1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
<td>R2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>[fp-40]</td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>0</td>
<td></td>
<td>R3</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
<td></td>
<td>R4</td>
</tr>
<tr>
<td>scratch</td>
<td>0</td>
<td></td>
<td>R5</td>
</tr>
<tr>
<td>scratch</td>
<td>1</td>
<td></td>
<td>R6</td>
</tr>
</tbody>
</table>
Improved native code generation:

```
 save sp,-136,sp
 a = 1;    iconst_1  mov 1,R3
           istore_1  mov R3,R1
 b = 13;   ldc 13    mov 13,R3
           istore_2  mov R3,R2
 c = a + b; iload_1  mov R1,R3
           iload_2  mov R2,R4
           iadd    add R3,R4,R3
           istore_3 st R3,[fp-40]
 return    restore
           ret
```

This works quite well if:

- the architecture has a large register set;
- the stack is small most of the time; and
- the first locals are used most frequently.
Summary of fixed register allocation scheme:

- registers are allocated once; and
- the allocation does not change within a method.

Advantages:

- it’s simple to do the allocation; and
- no problems with different control flow paths.

Disadvantages:

- assumes the first locals and stack locations are most important; and
- may waste registers within a region of a method.
The *basic block* register allocation scheme:

- assign frame slots to registers on demand within a basic block; and
- update *descriptors* at each bytecode.

The descriptor maps a slot to an element of the set \{⊥, mem, Ri, mem&Ri\}:

<table>
<thead>
<tr>
<th>a</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>mem</td>
</tr>
<tr>
<td>c</td>
<td>mem&amp;R4</td>
</tr>
<tr>
<td>s_0</td>
<td>R1</td>
</tr>
<tr>
<td>s_1</td>
<td>⊥</td>
</tr>
</tbody>
</table>

We also maintain the inverse register map:

<table>
<thead>
<tr>
<th>R1</th>
<th>s_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>a</td>
</tr>
<tr>
<td>R3</td>
<td>⊥</td>
</tr>
<tr>
<td>R4</td>
<td>c</td>
</tr>
<tr>
<td>R5</td>
<td>⊥</td>
</tr>
</tbody>
</table>
At the beginning of a basic block, all slots are in memory.

Basic blocks are merged by control paths:

Registers must be spilled after basic blocks:
```plaintext
| R1 | ⊥ | a  | mem |
| R2 | ⊥ | b  | mem |
| R3 | ⊥ | c  | mem |
| R4 | ⊥ | s.0| ⊥ |
| R5 | ⊥ | s.1| ⊥ |
```

**save sp, -136, sp**

```plaintext
| R1 | s.0 | a  | mem |
| R2 | ⊥   | b  | mem |
| R3 | ⊥   | c  | mem |
| R4 | ⊥   | s.0| R1 |
| R5 | ⊥   | s.1| ⊥ |
```

**iconst_1 mov 1, R1**

```plaintext
| R1 | ⊥   | a  | R2 |
| R2 | a   | b  | mem|
| R3 | ⊥   | c  | mem|
| R4 | ⊥   | s.0| ⊥ |
| R5 | ⊥   | s.1| ⊥ |
```

**istore_1 mov R1, R2**

```plaintext
| R1 | s.0 | a  | R2 |
| R2 | a   | b  | mem|
| R3 | ⊥   | c  | mem|
| R4 | ⊥   | s.0| ⊥ |
| R5 | ⊥   | s.1| ⊥ |
```

**ldc 13 mov 13, R1**

```plaintext
| R1 | s.0 | a  | R2 |
| R2 | a   | b  | mem|
| R3 | ⊥   | c  | mem|
| R4 | ⊥   | s.0| R1 |
| R5 | ⊥   | s.1| ⊥ |
```

**istore_2 mov R1, R3**

```plaintext
| R1 | ⊥   | a  | R2 |
| R2 | a   | b  | R3 |
| R3 | b   | c  | mem|
| R4 | ⊥   | s.0| ⊥ |
| R5 | ⊥   | s.1| ⊥ |
```
### Native code generation (20)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Native Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iload_1</code></td>
<td><code>mov R2,R1</code></td>
</tr>
<tr>
<td><code>iload_2</code></td>
<td><code>mov R3,R4</code></td>
</tr>
<tr>
<td><code>iadd</code></td>
<td><code>add R1,R4,R1</code></td>
</tr>
<tr>
<td><code>istore_3</code></td>
<td><code>st R1,R4</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>restore</code></td>
</tr>
</tbody>
</table>

```
  | R1 | s.0 | a | R2 |
  | R2 | a   | b | R3 |
  | R3 | b   | c | mem |
  | R4 | ⊥   | s.0 | R1 |
  | R5 | ⊥   | s.1 | ⊥ |

  | R1 | s.0 | a | R2 |
  | R2 | a   | b | R3 |
  | R3 | b   | c | mem |
  | R4 | s.1 | s.0 | R1 |
  | R5 | ⊥   | s.1 | R4 |
```

```
  | R1 | s.0 | a | R2 |
  | R2 | a   | b | R3 |
  | R3 | b   | c | mem |
  | R4 | ⊥   | s.0 | ⊥ |
  | R5 | ⊥   | s.1 | ⊥ |
```

```
  | R1 | ⊥   | a | R2 |
  | R2 | a   | b | R3 |
  | R3 | b   | c | R4 |
  | R4 | c   | s.0 | ⊥ |
  | R5 | ⊥   | s.1 | ⊥ |
```

```
  | R1 | ⊥   | a | mem |
  | R2 | ⊥   | b | mem |
  | R3 | ⊥   | c | mem |
  | R4 | ⊥   | s.0 | ⊥ |
  | R5 | ⊥   | s.1 | ⊥ |
```

```
return restore ret
```
So far, this is actually no better than the fixed scheme.

But if we add the statement:

\[ c = c \times c + c; \]

then the fixed scheme and basic block scheme generate:

<table>
<thead>
<tr>
<th></th>
<th>Fixed</th>
<th>Basic block</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload_3</td>
<td>ld [fp-40],R3</td>
<td>mv R4, R1</td>
</tr>
<tr>
<td>dup</td>
<td>ld [fp-40],R4</td>
<td>mv R4, R5</td>
</tr>
<tr>
<td>imul</td>
<td>mul R3,R4,R3</td>
<td>mul R1, R5, R1</td>
</tr>
<tr>
<td>iload_3</td>
<td>ld [fp-40],R4</td>
<td>mv R4, R5</td>
</tr>
<tr>
<td>iadd</td>
<td>add R3,R4,R3</td>
<td>add R1, R5, R1</td>
</tr>
<tr>
<td>istore_3</td>
<td>st R3,[fp-40]</td>
<td>mv R1, R4</td>
</tr>
</tbody>
</table>
Summary of basic block register allocation scheme:

• registers are allocated on demand; and
• slots are kept in registers within a basic block.

Advantages:

• registers are not wasted on unused slots; and
• less spill code within a basic block.

Disadvantages:

• much more complex than the fixed register allocation scheme;
• registers must be spilled at the end of a basic block; and
• we may spill locals that are never needed.
We can optimize further:

save sp,-136,sp  save sp,-136,sp
mov 1,R1         mov 1,R2
mov R1,R2        mov 13,R3
mov 13,R1        mov R1,R3
mov R1,R3        mov R2,R1
mov R2,R1        mov R3,R4
add R1,R4,R1     add R2,R3,R1
st R1,[fp-40]    st R1,[fp-40]
restore          restore
ret               ret

by not explicitly modelling the stack.
Unfortunately, this cannot be done safely on the fly by a peephole optimizer.

The optimization:

\[
\begin{align*}
\text{mov} & \ 1, R3 & \rightarrow & \text{mov} & \ 1, R1 \\
\text{mov} & \ R3, R1 & & & \\
\end{align*}
\]

is unsound if \( R3 \) is used in a later instruction:

\[
\begin{align*}
\text{mov} & \ 1, R3 & \rightarrow & \text{mov} & \ 1, R1 \\
\text{mov} & \ R3, R1 & & & \\
\vdots & & & & \vdots \\
\text{mov} & \ R3, R4 & \rightarrow & \text{mov} & \ R3, R4 \\
\end{align*}
\]

Such optimizations require dataflow analysis.
Invoking methods in bytecode:

- evaluate each argument leaving results on the stack; and
- emit `invokevirtual` instruction.

Invoking methods in native code:

- call library routine `soft_get_method_code` to perform the method lookup;
- generate code to load arguments into registers; and
- branch to the resolved address.
Consider a method invocation:

c = t.foo(a, b);

where the memory map is:

<table>
<thead>
<tr>
<th>name</th>
<th>offset</th>
<th>location</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>[fp-60]</td>
<td>R3</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>[fp-56]</td>
<td>R4</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>[fp-52]</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>4</td>
<td>[fp-48]</td>
<td>R2</td>
</tr>
<tr>
<td>stack</td>
<td>0</td>
<td>[fp-36]</td>
<td>R1</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
<td>[fp-40]</td>
<td>R5</td>
</tr>
<tr>
<td>stack</td>
<td>2</td>
<td>[fp-44]</td>
<td>R6</td>
</tr>
<tr>
<td>scratch</td>
<td>0</td>
<td>[fp-32]</td>
<td>R7</td>
</tr>
<tr>
<td>scratch</td>
<td>1</td>
<td>[fp-28]</td>
<td>R8</td>
</tr>
</tbody>
</table>
Generating native code:

```
aload_4                      mov R2,R1
iload_1                      mov R3,R5
iload_2                      mov R4,R6
invokevirtual foo            // soft call to get address
                           ld R7,[R2+4]
                           ld R8,[R7+52]
                           // spill all registers
                           st R3,[fp-60]
                           st R4,[fp-56]
                           st R2,[fp-48]
                           st R6,[fp-44]
                           st R5,[fp-40]
                           st R1,[fp-36]
                           st R7,[fp-32]
                           st R8,[fp-28]
                           // make call
                           mov R8,R0
                           call soft_get_method_code
                           // result is in R0
                           // put args in R2, R1, and R0
                           ld R2,[fp-44] // R2 := stack_2
                           ld R1,[fp-40] // R1 := stack_1
                           st R0,[fp-32] // spill result
                           ld R0,[fp-36] // R0 := stack_0
                           ld R4,[fp-32] // reload result
                           jmp [R4] // call method
```

- this is long and costly; and

- the lack of dataflow analysis causes massive spills within basic blocks.
Handling branches:

- the only problem is that the target address is not known;
- assemblers normally handle this; but
- the JIT compiler produces binary code directly in memory.

Generating native code:

```
if (a < b)  iload_1   ld R1,[fp-44]
    iload_2   ld R2,[fp-48]
    if_icmpge 17  sub R1,R2,R3
    bge ??
```

How to compute the branch targets:

- previously encountered branch targets are already known;
- keep unresolved branches in a table; and
- patch targets when the bytecode is eventually reached.