

Native Code Generation

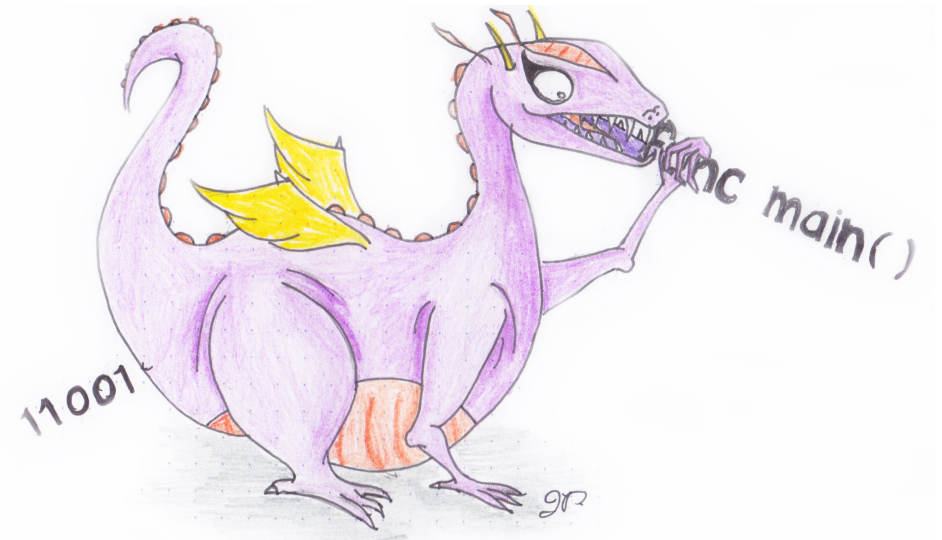
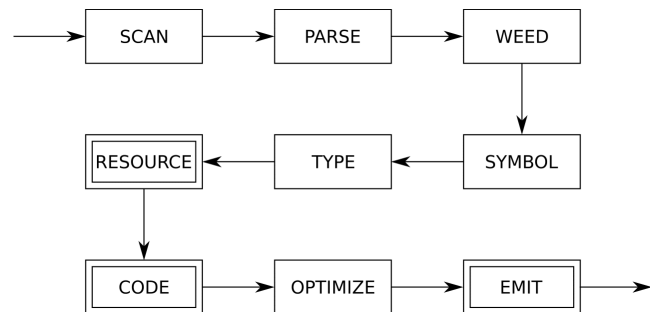
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 9:30-10:30, TR 1080

<http://www.cs.mcgill.ca/~cs520/2018/>



Bob, from Accounting

Announcements (Wednesday, March 28th)

Milestones

- Milestone 4 due: Sunday, April 8th 11:59 PM
- Peephole due: Sunday, April 8th 11:59 PM
- Final report due: Sunday, April 15th 11:59 PM

Executing JOOS Code

The JOOS compiler translates a subset of Java to bytecode, but bytecode cannot be executed on any machine directly. Execution occurs thanks to either

1. An interpreter;
2. An Ahead-Of-Time (AOT) compiler; or
3. A Just-In-Time (JIT) compiler.

In all cases, bytecode must be implicitly or explicitly translated into native code suitable for the host architecture before execution.

Interpreters

Execute bytecode one instruction at a time, by simulating the progression on a virtual machine

- “Easy” to implement;
- Can be very portable since they are target independent; but
- Suffer an inherent inefficiency.

Interpreters

```
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

- Typically run on the developer's machine;
- Translate the low-level intermediate form into native code;
- Create object files to be linked and 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 interpreters with traditional compilation techniques;
- 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.

Just-in-Time Compilers

Efficiency

For a JIT compiler to be worthwhile

- It must be *fast*, because the compilation occurs at runtime (Just-In-Time is really Just-Too-Late);
- It may concurrently interpret and compile a method (Better-Late-Than-Never); and
- It may have several levels of optimization, and recompile long-running methods.

Cutting corners

Since we require high performance

- It does not necessarily generate optimized code;
- It does not necessarily compile every instruction into native code, but relies on the runtime library for complex instructions; and
- It need not necessarily compile every method;

Generating Native Code

When generating native code, there are 4 important problems to solve

- **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 to VirtualRISC

In this class we focus on generating VirtualRISC from JVM bytecode. In a JIT compiler this requires

- Mapping the Java local stack into registers and memory;
- Instruction selection on the fly;
- Register allocation on the fly; and
- Branch target allocation on the fly.

This is successfully done in the Kaffe system.

Compiling JVM Bytecode to VirtualRISC

The general algorithm for generating native code consists of

- Finding the local stack height for each bytecode;
- Determining the number of slots in frame: locals limit + stack limit + #temps;
- Emitting prologue;
- Emitting native code for each bytecode; and
- Fixing up branches.

For VirtualRISC, we assume that input parameters arrive in registers $R0$ through R_n , and the return value must be in $R0$

Mapping Locals/Stack to the Frame

The first task before generating native code is to place variables in memory

name	offset	location	register
a	1	[fp-4]	
stack	0		R1
stack	1		R2
scratch	0		R3

Register allocation

Problem: Find the mapping scheme which keeps as many variables in registers as possible

Register allocation maps directly to graph colouring, and is therefore NP-complete. In class we will show 3 basic schemes used for generating native code

- Naïve (no allocation);
- Fixed; and
- Basic block.

Naïve Allocation

The naïve approach assumes that registers are very volatile, where the value stored in a register from one bytecode cannot be accessed from another.

```
iload_0    // loads value into R0
ineg       // cannot assume anything about R0
```

i.e. Even though `iload_0` loads the value into `R0`, then the next instruction can make no assumption about the contents of `R0`. To transfer data, we must use memory.

Generating code

- 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.

Naïve Allocation Example

Input code

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

Procedure

- Compute frame size = $4 + 2 + 0 = 6$;
- Find stack height for each bytecode;
- Emit prologue; and
- Emit native code for each bytecode.

Generated bytecode

```
.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  
.end method
```

Naïve Allocation Example

Assignment of frame slots

Native code generation

name	offset	location
a	1	[fp-32]
b	2	[fp-36]
c	3	[fp-40]
stack	0	[fp-44]
stack	1	[fp-48]

a = 1;

iconst_1

save sp, -136, sp

mov 1, R1

st R1, [fp-44]

istore_1

ld [fp-44], R1

st R1, [fp-32]

b = 13;

ldc 13

mov 13, R1

st R1, [fp-44]

istore_2

ld [fp-44], R1

st R1, [fp-36]

c = a + b;

iload_1

ld [fp-32], R1

st R1, [fp-44]

iload_2

ld [fp-36], R1

st R1, [fp-48]

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

Naïve Allocation

- Clearly very slow;
- Many unnecessary loads and stores, which
- Are the *most* expensive operations.

We wish to replace repeated loads and stores with register operations

Improved Allocation

We wish to replace loads and stores

<code>c = a + b;</code>	<code>iload_1</code>	<code>ld [fp-32], R1</code>
		<code>st R1, [fp-44]</code>
	<code>iload_2</code>	<code>ld [fp-36], R1</code>
		<code>st R1, [fp-48]</code>
	<code>iadd</code>	<code>ld [fp-48], R1</code>
		<code>ld [fp-44], R2</code>
		<code>add R2, R1, R1</code>
		<code>st R1, [fp-44]</code>
	<code>istore_3</code>	<code>ld [fp-44], R1</code>
		<code>st R1, [fp-40]</code>

by registers operations

<code>c = a + b;</code>	<code>iload_1</code>	<code>ld [fp-32], R1</code>
	<code>iload_2</code>	<code>ld [fp-36], R2</code>
	<code>iadd</code>	<code>add R1, R2, R1</code>
	<code>istore_3</code>	<code>st R1, [fp-40]</code>

where R1 and R2 represent the stack.

Announcements (Wednesday, April 4th)

Milestones

- Milestone 4 due: Tuesday, April 10th 11:59 PM
- Peephole due: Friday, April 13th 11:59 PM
- Final report due: Sunday, April 15th 11:59 PM

Also, submit a course evaluation!

Fixed Register Allocation

- 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

Given 6 registers ($m = n = k = 2$), show the register allocation table

name	offset	location	register
a	1		R1
b	2		R2
c	3	[fp-40]	
stack	0		R3
stack	1		R4
scratch	0		R5
scratch	1		R6

Fixed Register Allocation Example

Memory allocation map

name	offset	location	register
a	1		R1
b	2		R2
c	3	[fp-40]	
stack	0		R3
stack	1		R4
...			

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

Fixed Register Allocation

- Registers are allocated once; and
- The allocation does not change within a method.

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.

Advantages

- It's simple to do the allocation; and
- No problems with different control flow paths.

Disadvantages

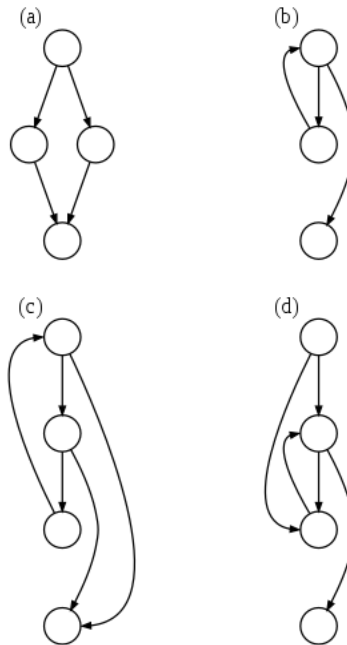
- Assumes the first locals and stack locations are most important; and
- May waste registers within a region of a method.

Basic Blocks

A *basic block* is a sequence of instructions that

- Are linear;
- Have no incoming or outgoing branches except at boundaries; and thus
- Have only one entry point (the start) and only one exit point (the end).

Basic blocks form the nodes of a *control flow graph*.



Basic Block Register Allocation

- 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 $\{ \perp, \text{mem}, R_i, \text{mem} \& R_i \}$

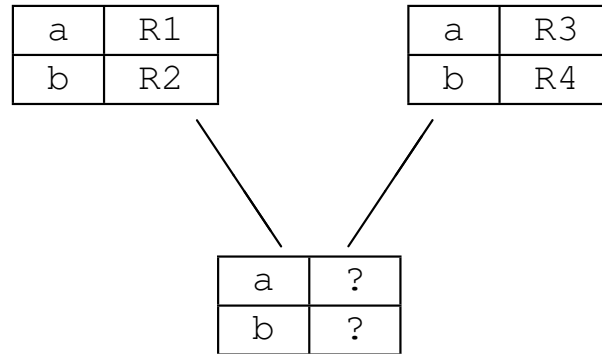
a	R2
b	mem
c	mem&R4
s_0	R1
s_1	\perp

We also maintain the inverse register map

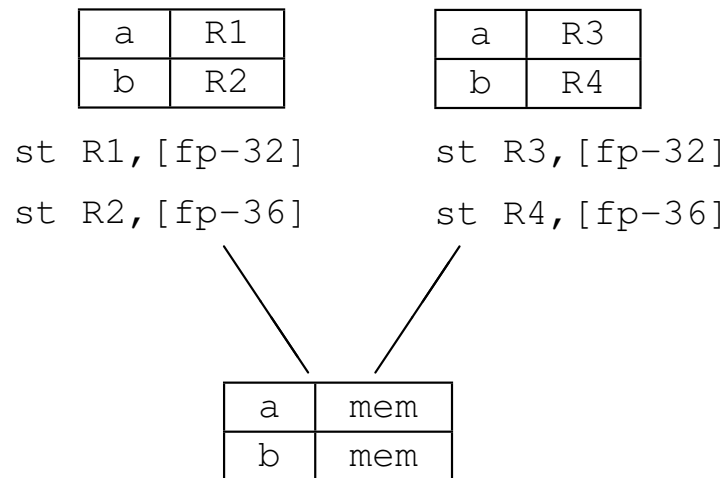
R1	s_0
R2	a
R3	\perp
R4	c
R5	\perp

Basic Block Register Allocation

In a control flow graph, divergent paths can merge at basic block boundaries (i.e. loops, if statements, etc.)



To correctly merge the paths, registers must be spilled after basic blocks



At the beginning of a basic block, all slots are therefore assumed to be in memory.

Basic Block Register Allocation Example

	save sp,-136,sp	<table><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>⊥</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table><tr><td>a</td><td>mem</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	⊥	R3	⊥	R4	⊥	R5	⊥	a	mem	b	mem	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	⊥																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	mem																					
b	mem																					
c	mem																					
s_0	⊥																					
s_1	⊥																					
iconst_1	mov 1,R1	<table><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>⊥</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table><tr><td>a</td><td>mem</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	s_0	R2	⊥	R3	⊥	R4	⊥	R5	⊥	a	mem	b	mem	c	mem	s_0	R1	s_1	⊥
R1	s_0																					
R2	⊥																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	mem																					
b	mem																					
c	mem																					
s_0	R1																					
s_1	⊥																					
istore_1	mov R1,R2	<table><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	a	R3	⊥	R4	⊥	R5	⊥	a	R2	b	mem	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	a																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	R2																					
b	mem																					
c	mem																					
s_0	⊥																					
s_1	⊥																					
ldc 13	mov 13,R1	<table><tr><td>R1</td><td>s_0</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>⊥</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>mem</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>R1</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	s_0	R2	a	R3	⊥	R4	⊥	R5	⊥	a	R2	b	mem	c	mem	s_0	R1	s_1	⊥
R1	s_0																					
R2	a																					
R3	⊥																					
R4	⊥																					
R5	⊥																					
a	R2																					
b	mem																					
c	mem																					
s_0	R1																					
s_1	⊥																					
istore_2	mov R1,R3	<table><tr><td>R1</td><td>⊥</td></tr><tr><td>R2</td><td>a</td></tr><tr><td>R3</td><td>b</td></tr><tr><td>R4</td><td>⊥</td></tr><tr><td>R5</td><td>⊥</td></tr></table> <table><tr><td>a</td><td>R2</td></tr><tr><td>b</td><td>R3</td></tr><tr><td>c</td><td>mem</td></tr><tr><td>s_0</td><td>⊥</td></tr><tr><td>s_1</td><td>⊥</td></tr></table>	R1	⊥	R2	a	R3	b	R4	⊥	R5	⊥	a	R2	b	R3	c	mem	s_0	⊥	s_1	⊥
R1	⊥																					
R2	a																					
R3	b																					
R4	⊥																					
R5	⊥																					
a	R2																					
b	R3																					
c	mem																					
s_0	⊥																					
s_1	⊥																					

iload_1 mov R2,R1

R1	s_0
R2	a
R3	b
R4	⊥
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	⊥

iload_2 mov R3,R4

R1	s_0
R2	a
R3	b
R4	s_1
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	R4

iadd add R1,R4,R1

R1	s_0
R2	a
R3	b
R4	⊥
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	⊥

istore_3 st R1,R4

R1	⊥
R2	a
R3	b
R4	c
R5	⊥

a	R2
b	R3
c	R4
s_0	⊥
s_1	⊥

return st R2,[fp-32]
 st R3,[fp-36]
 st R4,[fp-40]

R1	⊥
R2	⊥
R3	⊥
R4	⊥
R5	⊥

a	mem
b	mem
c	mem
s_0	⊥
s_1	⊥

restore
ret

Basic Block vs. Fixed Register Allocation

So far, this is actually no better than the fixed scheme, in fact, it is worse (3 memory stores from spilling vs. 1 memory store). But if we add the statement

```
c = c * c + c;
```

Then the fixed scheme and basic block schemes generate

	Fixed	Basic block
iload_3	ld [fp-40],R3	mov R4,R1
iload_3	ld [fp-40],R4	mov R4,R5
imul	mul R3,R4,R3	mul R1,R5,R1
iload_3	ld [fp-40],R4	mov R4,R5
iadd	add R3,R4,R3	add R1,R5,R1
istore_3	st R3,[fp-40]	mov R1,R4

Note: When comparing both approaches we must consider the cost of spilling

Basic Block Register Allocation

- 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.

Further Optimizations

The schemes up until this point explicitly modelled the stack. If we remove this requirement, the code gets even faster

```
save sp,-136,sp
```

```
mov 1,R1  
mov R1,R2
```

```
mov 13,R1  
mov R1,R3
```

```
mov R2,R1  
mov R3,R4  
add R1,R4,R1  
st R1,[fp-40]
```

```
restore  
ret
```

```
save sp,-136,sp
```

```
mov 1,R2
```

```
mov 13,R3
```

```
add R2,R3,R1  
st R1,[fp-40]
```

```
restore  
ret
```

Further Optimizations

Peephole optimization

```

mov 1, R3       $\implies$   mov 1, R1
mov R3, R1

```

The optimization is unsound if R3 is used in a later instruction

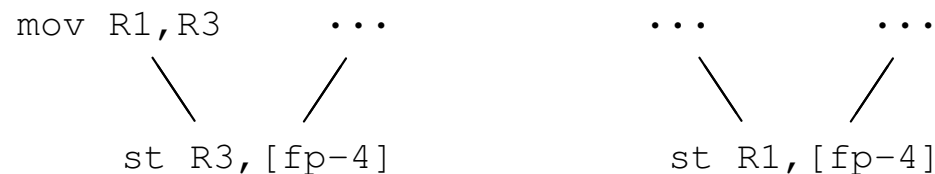
```

mov 1, R3       $\implies$   mov 1, R1
mov R3, R1
:
:
mov R3, R4      mov R3, R4

```

Control flow

Just like basic block allocation, care must also be taken when merging divergent control flow.



Such optimizations require dataflow analysis (COMP 621).

Method Invocation

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.

Up until now we ignored the scratch registers since all operations were performed on locals/stack locations. This is where scratch registers come into play

Method Invocation Example

Consider a method invocation

```
c = t.foo(a, b);
```

Memory map

name	offset	location	register
a	1	[fp-60]	R3
b	2	[fp-56]	R4
c	3	[fp-52]	
t	4	[fp-48]	R2
stack	0	[fp-36]	R1
stack	1	[fp-40]	R5
stack	2	[fp-44]	R6
scratch	0	[fp-32]	R7
scratch	1	[fp-28]	R8

Method Invocation Example

```
aload_4  
iload_1  
iload_2  
invokevirtual foo
```

```
mov R2,R1  
mov R3,R5  
mov R4,R6  
// 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  
st R0,[fp-32] // spill result  
ld R2,[fp-44] // R2 := stack_2  
ld R1,[fp-40] // R1 := stack_1  
ld R0,[fp-36] // R0 := stack_0  
ld R7,[fp-32] // reload result  
jmp [R7] // call method
```

Method Invocation

- Long and costly; and
- The lack of dataflow analysis causes massive spills within basic blocks.

Handling Branches

Native code is generated linearly, from first instruction to last. When branches jump forward in the code

- The target address is not yet known (the code has not been generated);
- Assemblers normally handle this; but
- The JIT compiler produces binary code directly in memory.

Example

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

To generate branch targets

- Previously encountered branch targets are already known;
- Keep unresolved branches in a table; and
- Patch targets when the code is eventually generated.