Compiler

Native Code Generation

Compile Time

- SJC programs are compiled into bytecode.
- This bytecode can be executed using:
  - an interpreter;
  - an Ahead-Of-Time (AOT) compiler; or
  - a Just-In-Time (JIT) compiler.
- Bytecode is compiled into native code.

Interpreters

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

Interpreters

```
pc = code.start;
while(true)
  {  npc = pc + instruction_length(code[pc]);
     switch (opcode(code[pc]))
       {  ILOAD_1: push(local[1]);                  break;
          ILOAD:   push(local[code[pc+1]]);         break;
          ISTORE:  t = pop();
                   local[code[pc+1]] = t;           break;
          IADD:    t1 = pop();  t2 = pop();
                   push(t1 + t2);                   break;
          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 SJC:
  - 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

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 calls

- determine calling conventions

branch handling

- allocate branch targets

A Model RISC Machine

- VirtualRISC is a simple RISC machine with:
  - memory;
  - registers;
  - condition codes; and
  - execution unit.
- In this model we ignore:
  - caches;
  - pipelines;
  - branch prediction units; and
  - advanced features.

VirtualRISC Memory

- a stack
  - used for function call frames;
- a heap
  - used for dynamically allocated memory;
- a global pool
  - used to store global variables; and
- a code segment
  - used to store VirtualRISC instructions.

VirtualRISC Registers

- unbounded number of general purpose registers;
- the stack pointer (`sp`) which points to the top of the stack;
- the frame pointer (`fp`) which points to the current stack frame; and
- the program counter (`pc`) which points to the current instruction.

VirtualRISC Condition Codes

- stores the result of last instruction that can set condition codes (used for branching).

VirtualRISC Execution Unit

- reads the VirtualRISC instruction at the current pc, decodes the instruction and executes it;
- this may change the state of the machine (memory, registers, condition codes);
- the pc is automatically incremented after executing an instruction; but
- function calls and branches explicitly change the pc.

Memory/Register Instructions

```
st Ri,[Rj]          [Rj] := Ri
st Ri,[Rj+C]        [Rj+C] := Ri

ld [Ri],Rj          Rj := [Ri]
ld [Ri+C],Rj        Rj := [Ri+C]
```

Register/Register Instructions

```
mov Ri,Rj          Rj := Ri
add Ri,Rj,Rk       Rk := Ri + Rj
sub Ri,Rj,Rk       Rk := Ri - Rj
mul Ri,Rj,Rk       Rk := Ri * Rj
div Ri,Rj,Rk       Rk := Ri / Rj
```

Constants may be used in place of register values: `mov 5,R1`

Condition Instructions

Instructions that set the condition codes:

```
cmp    Ri,Rj
```

Instructions to branch:

```
b    L
bg   L
bge  L
bl   L
ble  L
bne  L
```

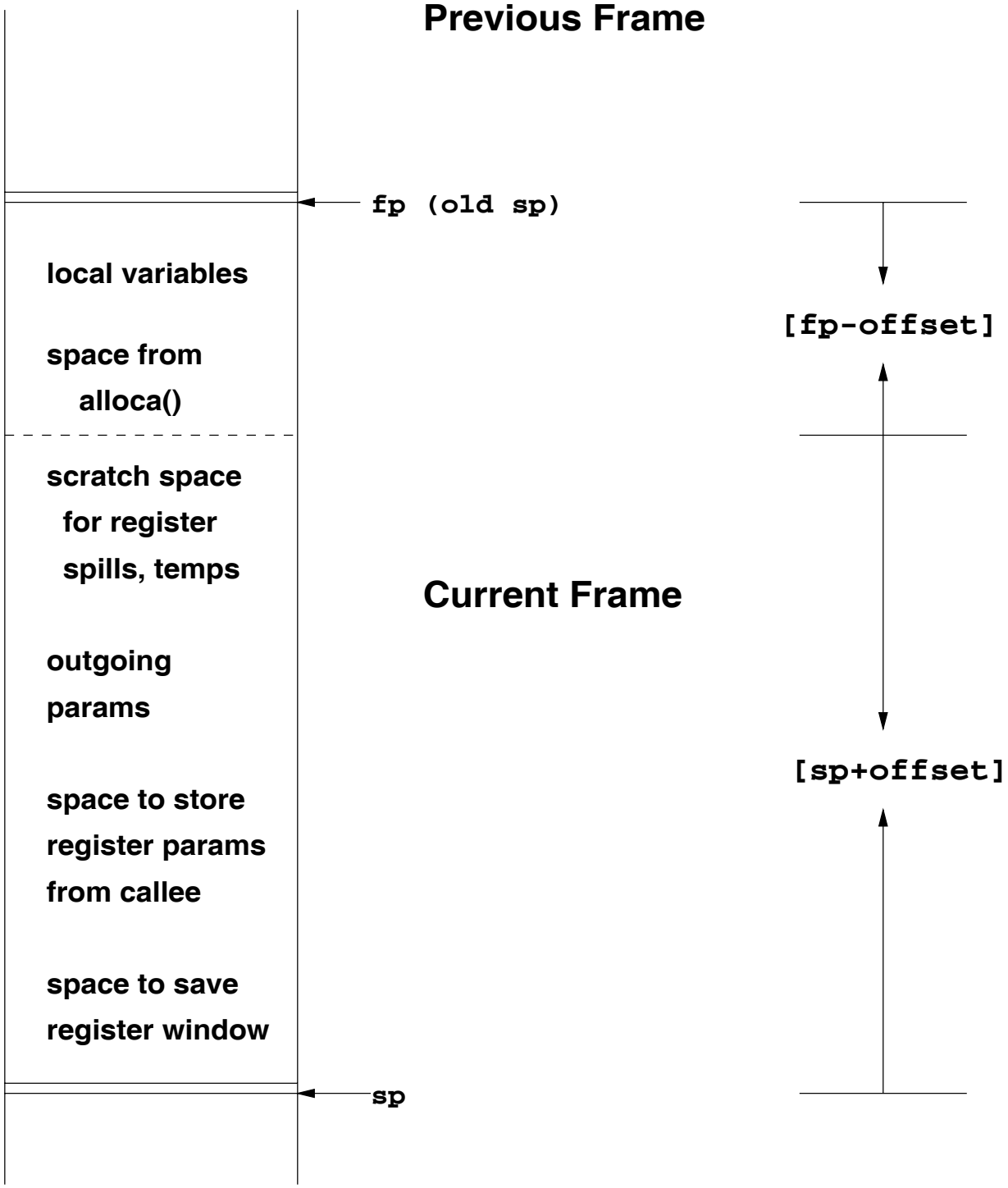
To express: `if R1 <= 9 goto L1`

we code:

```
cmp R1,9
ble L1
```

Call Related Functions

`save sp,-C,sp`	save registers, allocating `C` bytes on the stack
`call L`	`R15:=pc; pc:=L`
`restore`	restore registers
`ret`	`pc:=R15+8`
`nop`	do nothing


Previous Frame
fp (old sp)
local variables
space from
alloca()
scratch space
for register
spills, temps
Current Frame
outgoing
params
space to store
register params
from callee
space to save
register window
sp
[fp-offset]
[sp+offset]

Stack Frames

- stores function activations;
- `sp` and `fp` point to stack frames;
- when a function is called a new stack frame is created:

```
push fp; fp := sp; sp := sp + C;
```

- when a function returns, the top stack frame is popped:

```
sp := fp;  fp = pop;
```

- local variables are stored relative to `fp`
- the figure shows additional features of the SPARC architecture.

Example C Code

```
int fact(int n) {
  int i, sum;
  sum = 1;
  i = 2;
  while (i <= n){
    sum = sum * i;
    i = i + 1;
  }
  return sum;
}
```

Example VirtualRISC Code

```
_fact:
   save sp,-112,sp     // save stack frame
   st R0,[fp+68]       // save arg n in caller frame
   mov 1,R0            // R0 := 1
   st R0,[fp-16]       // sum is in [fp-16]
   mov 2,R0            // RO := 2
   st RO,[fp-12]       // i is in [fp-12]
L3:
   ld [fp-12],R0       // load i into R0
   ld [fp+68],R1       // load n into R1
   cmp R0,R1           // compare R0 to R1
   ble L5              // if R0 <= R1 goto L5
   b L4                // goto L4
```

Example VirtualRISC Code

```
L5:
   ld [fp-16],R0     // load sum into R0
   ld [fp-12],R1     // load i into R1
   mul R0,R1,R0      // R0 := R0 * R1
   st R0,[fp-16]     // store R0 into sum
   ld [fp-12],R0     // load i into R0
   add R0,1,R1       // R1 := R0 + 1
   st R1,[fp-12]     // store R1 into i
   b L3              // goto L3
L4:
   ld [fp-16],R0     // put return value into R0
   restore           // restore register window
   ret               // return from function
```

JVM to 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.

Several real JITs do this (e.g., Kaffe)

CodeGen 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 : Source

```
public void foo() {
  int a,b,c;

  a = 1;
  b = 13;
  c = a + b;
}
```

Example : 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
```

- compute frame size
  - 4 + 2 + 0 = 6;
- find stack height for each bytecode;
- emit prologue; and
- emit native code for each bytecode.

Example : Frame Slot Assignments

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

```
                                    save sp,-136,sp
a = 1;          iconst_1            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 Code is Slow

- Many unnecessary loads and stores
  - Which are the *most* expensive operations
- We would like to replace load/store operations with register operations
  - Requires that we assign registers to JVM local stack locations

Naïve Code is Slow

```
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]
```

Becomes

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

Register Allocation

A *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):

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

Register Allocation Improves Code

```
                                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
```

Register Allocation Improves Code

- 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

- 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.

Basic Block Register Allocation

The *basic block* register allocation scheme:

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

Basic Block Register Allocation

For our example:

a	R2
b	mem
c	mem&R4
s_0	R1
s_1	_

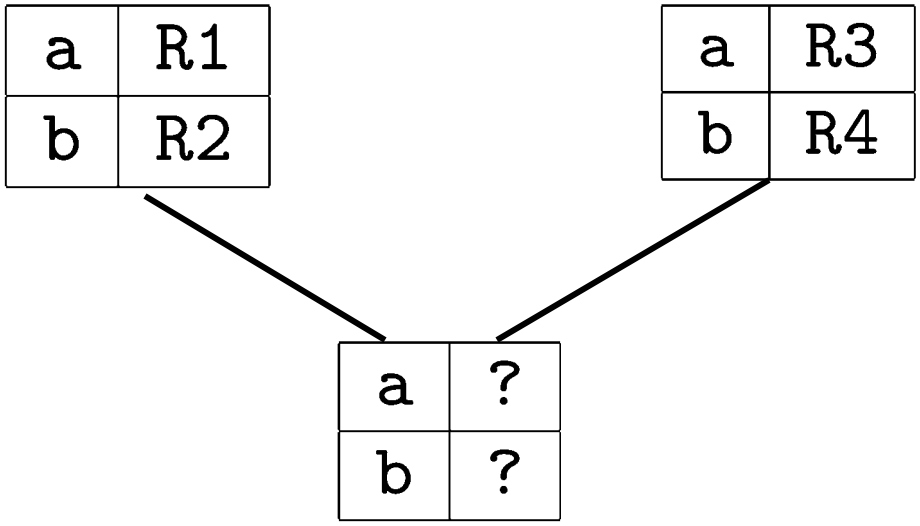
We also keep the inverse register map:

R1	s_0
R2	a
R3	_
R4	c
R5	_

Treating Control Flow

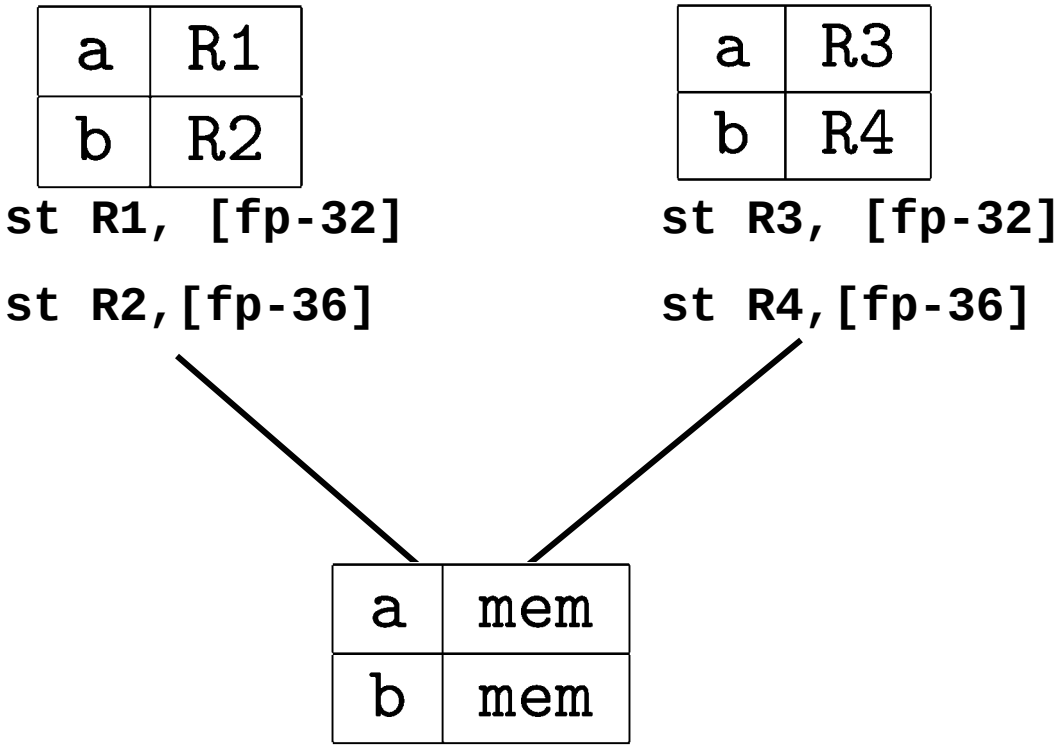
At the beginning of a block, all slots are in memory.

Basic blocks are merged by control paths:



Treating Control Flow

Registers must be spilled after basic blocks:



Example : BB Register Allocation

		R1	_		a	mem
		R2	_		b	mem
		R3	_		c	mem
		R4	_		s_0	_
		R5	_		s_1	_
	save sp,-136,sp					
		R1	s_0		a	mem
		R2	_		b	mem
		R3	_		c	mem
		R4	_		s_0	R1
		R5	_		s_1	_
iconst_1	mov 1,R1					
		R1	_		a	R2
		R2	a		b	mem
		R3	_		c	mem
		R4	_		s_0	_
		R5	_		s_1	_
istore_1	mov R1,R2					

Example : BB Register Allocation

ldc 13 mov 13,R1

R1	s_0
R2	a
R3	_
R4	_
R5	_

a	R2
b	mem
c	mem
s_0	R1
s_1	_

istore_2 mov R1,R3

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	_

Example : BB Register Allocation

iload_2	mov R3,R4	R1	s_0	a	R2
		R2	a	b	R3
		R3	b	c	mem
		R4	s_1	s_0	R1
		R5	_	s_1	R4
iadd	add R1,R4,R1	R1	s_0	a	R2
		R2	a	b	R3
		R3	b	c	mem
		R4	_	s_0	R1
		R5	_	s_1	_
istore_3	mov R1,R4	R1	_	a	R2
		R2	a	b	R3
		R3	b	c	R4
		R4	c	s_0	_
		R5	_	s_1	_

Example : BB Register Allocation

```
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	_

```
return          restore
                ret
```

No Improvement

But if we add the statement:

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

then the fixed scheme generates:

```
ld [fp-40],R3
ld [fp-40],R4
mul R3,R4,R3
ld [fp-40],R4
add R3,R4,R3
st R3,[fp-40]
```

whereas the basic block scheme continues:

```
mul R4,R4,R1
add R1,R4,R1
st R1,R4
```

which is vastly better.

Summary of BB 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 Optimization

If we skip modeling the local stack

```
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
```

Optimization

Unfortunately, this cannot be done safely on the fly by a pattern-based optimizer.

The optimization:

```
mov 1,R3          ⟶   mov 1,R1
mov R3,R1
```

is unsound if R3 is used in a later instruction:

```
mov 1,R3          ⟶   mov 1,R1
mov R3,R1

...                   ...
mov R3,R4             mov R3,R4
```

Such optimizations require *dataflow analysis*.

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.

Example

Consider a method invocation

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

where the memory map is:

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

```
aload_4                mov R2,R1
iload_1                mov R3,R5
iload_2                mov R4,R6
invokevirtual foo      // soft call to get address
                       ld [R2+4],R7
                       ld [R7+52],R8
                       // 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 in R0
                       // put args in R2, R1, and R0
                       ld [fp-44],R2 \\ R2 := stack_2
                       ld [fp-40],R1  \\ R1 := stack_1
                       st R0,[fp-32]  \\ spill result
                       ld [fp-36],R0  \\ R0 := stack\_0
                       ld [fp-32],R4  \\ reload result
                       jmp [R4]  \\ call method
```

Innefficient

- This is long and costly; and
- The lack of *dataflow analysis* causes massive spills within the 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.