

## COMPILING TO NATIVE CODE

=====

Example Java source code:

```
public static void foo(int a[], int b[], int i) {
    a[i] = (2 * a[i] + b[i]);
}
```

Corresponding JVM code:

Method void foo(int[], int[], int)

```
0 aload_0    ; a
1 iload_2    ; i
2 iconst_2
3 aload_0    ; a
4 iload_2    ; i
5 iaload     ; a[i]
6 imul      ; a[i] * 2
7 aload_1    ; b
8 iload_2    ; i
9 iaload     ; b[i]
10 iadd      ; a[i] * 2 + b[i];
11 iastore   ; a[i]
12 return
```

Maximum stack height = 5

Native code generation for a pretend machine:

Assume RISC-like load/store architecture, with registers  $r_1, r_2, r_3, \dots, fp, sp$ .

```
Instructions: move r,r
              add r,r/c,r
              sll c,r,r
              cmp r,r/c
              br{eq/neq/leu/...} lab
              load c(r),r
              store r,c(r)
```

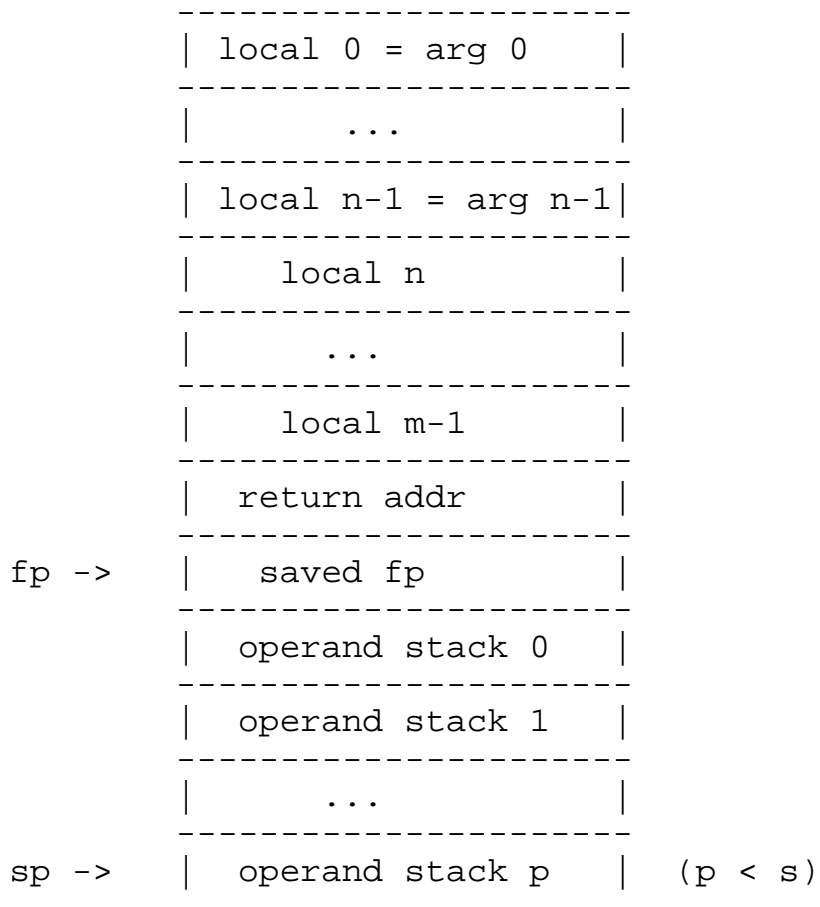
Note: this is similar to SPARC code, but not identical.

Assume constants have been resolved prior to code generation.

## STACK LAYOUT

=====

Combined data/control stack layout for procedure with  $m$  locals, of which  $n$  are args; and max operand stack depth  $s$ .



Stack pointer  $sp$  must always be honest, i.e., we must assume that anything below  $sp$  might be garbaged at any time (e.g. by a signal).

Note: this layout is not very convenient for passing arguments.

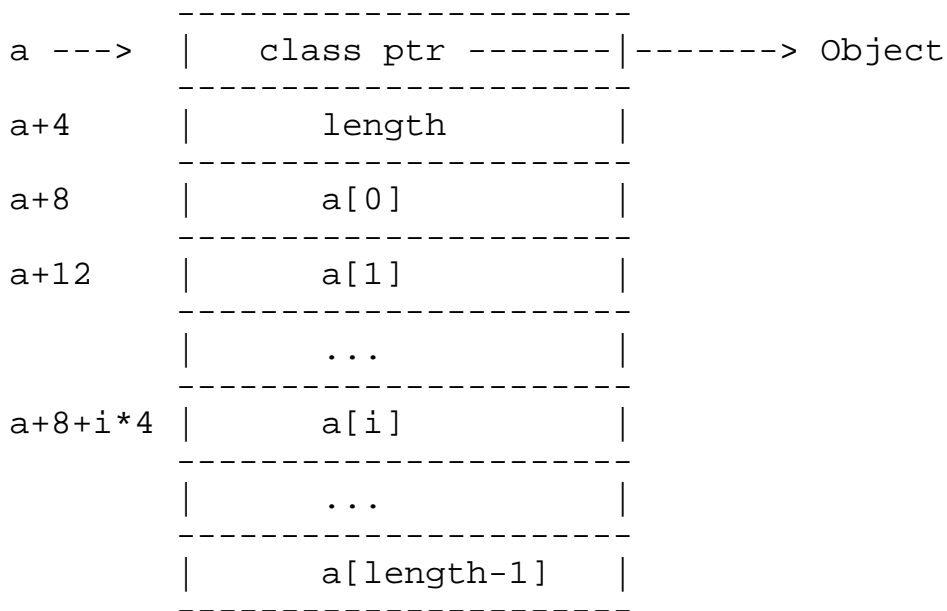
## STACK LAYOUT EXAMPLE

=====

For routine foo:

fp+16		arg 0 (a)		
fp+12		arg 1 (b)		
fp+8		arg 2 (i)		
fp+4		return addr		
fp --->		saved fp		<---
fp-4		operand stack 0		
fp-8		operand stack 1		possible
fp-12		operand stack 2		sp
fp-16		operand stack 3		values
fp-20		operand stack 4		<---

## Integer Arrays:



Recall that there are other alternatives, e.g. having `a -> a[0]` and `length` at `a-4`.

NATIVE CODE VERSION 0

=====

Very naive; just lay down code equivalent to what's executed by interpreter.

```

foo:      store fp,(sp)      ; save old frame pointer
          move sp,fp        ; new frame pointer
;aload_0
          load 16(fp),r0    ; fetch a
          add sp,#-4,sp     ; push
          store r0,(sp)    ; a
;iload_2
          load 8(fp),r0     ; fetch i
          add sp,#-4,sp     ; push
          store r0,(sp)    ; i
;iconst_2
          move #2,r0
          add sp,#-4,sp     ; push
          store r0,(sp)    ; 2
;aload_0
          load 16(fp),r0    ; fetch a
          add sp,#-4,sp     ; push
          store r0,(sp)    ; a
;iload_2
          load 8(fp),r0     ; fetch i
          add sp,#-4,sp     ; push
          store r0,(sp)    ; i
;iaload
          load 4(sp),r0     ; fetch array ptr
          cmp r0,#0         ; null check (use MMU instead?)
          bneq L1          ; branch if ok
          ...raise exception...
L1:      load 4(r0),r1      ; array length
          load (sp),r2     ; index
          cmp r2,r1        ; bounds check
          brltu L2         ; branch if ok (note standard trick)
          ... raise exception ...
L2:      sll #2,r2,r2      ; calculate byte offset
          add r0,r2,r2     ; add array base
          load 8(r2),r2    ; fetch a[i]
          add sp,#4,sp     ; pop one
          store r2,(sp)    ; push a[i] (replace)
;imul
          load (sp),r0     ; fetch a[i]
          load 4(sp),r1    ; 2
          mul r0,r1,r1     ; a[i] * 2
          add sp,#4,sp     ; pop one
          store r1,-12(sp) ; push a[i]*2 (replace)
;aload_1

```

```

    load 12(fp),r0      ; fetch b
    add sp,#-4,sp      ; push
    store r0,(sp)      ; b
; iload_2
    load 8(fp),r0      ; fetch i
    add sp,#-4,sp      ; push
    store r0,(sp)      ; i
; iaload
    load 4(sp),r0      ; fetch array ptr
    cmp r0,#0          ; null check
    bneq L3            ; branch if ok
    ...raise exception...
L3: load 4(r0),r1      ; array length
    load (sp),r2       ; index
    cmp r2,r1          ; bounds check
    brltu L4           ; branch if ok
    ... raise exception ...
L4: sll #2,r2,r2      ; calculate byte offset
    add r0,r2,r2       ; add array base
    load 8(r2),r2      ; fetch b[i]
    add sp,#4,sp       ; pop one
    store r2,(sp)      ; push b[i] (replace)
; iadd
    load 4(sp),r0      ; fetch 2*a[i]
    load (sp),r1       ; fetch b[i]
    add r0,r1,r1       ; 2*a[i] + b[i]
    add sp,#4,sp       ; pop one
    store r1,(sp)      ; push 2*a[i] + b[i] (replace)
; iastore
    load 8(sp),r0      ; fetch array ptr
    cmp r0,#0          ; null check
    bneq L5            ; branch if ok
    ...raise exception...
L5: load 4(r0),r1      ; array length
    load 4(sp),r2      ; index
    cmp r2,r1          ; bounds check
    brltu L6           ; branch if ok
    ... raise exception ...
L6: sll #2,r2,r2      ; calculate byte offset
    add r0,r2,r2       ; add array base
    load (sp),r0       ; fetch value
    store r0,8(r2)     ; store a[i]
    add sp,#12,sp      ; pop three
; return
    load (sp),fp       ; restore frame pointer
    ret

```

71 instructions; 23 loads; 13 stores; 3 registers.

NATIVE CODE VERSION 1

=====

In JVM code, we always know current stack depth. So, can treat operand stack as an array, saving the cost of adjusting sp each time we push or pop.

```

foo:          store fp,(sp)      ; save old frame pointer
              move sp,fp        ; new frame pointer
              add sp,#-20,sp     ; make space for operand stack

;aload_0
              load 16(fp),r0     ; fetch a
              store r0,-4(fp)    ; push a

;iload_2
              load 8(fp),r0      ; fetch i
              store r0,-8(fp)    ; push i

;iconst_2
              move #2,r0
              store r0,-12(fp)   ; push 2

;aload_0
              load 16(fp),r0     ; fetch a
              store r0,-16(fp)   ; push a

;iload_2
              load 8(fp),r0      ; fetch i
              store r0,-20(fp)   ; push i

;iaload
              load -16(fp),r0    ; fetch array ptr
              cmp r0,#0          ; null check
              bneq L1           ; branch if ok
              ...raise exception...
L1: load 4(r0),r1                ; array length
              load -20(fp),r2    ; index
              cmp r2,r1          ; bounds check
              brltu L2          ; branch if ok
              ... raise exception ...
L2: sll #2,r2,r2                ; calculate byte offset
              add r0,r2,r2       ; add array base
              load 8(r2),r2      ; fetch a[i]
              store r2,-16(fp)   ; push

;imul
              load -16(fp),r0    ; fetch a[i]
              load -12(fp),r1    ; 2
              mul r0,r1,r1       ; a[i] * 2
              store r1,-12(fp)   ; push a[i]*2

;aload_1
              load 12(fp),r0     ; fetch b
              store r0,-16(fp)   ; push b

;iload_2
              load 8(fp),r0      ; fetch i
              store r0,-20(fp)   ; push i

```

```

;iaload
        load -16(fp),r0    ; fetch array ptr
        cmp r0,#0         ; null check
        bneq L3          ; branch if ok
        ...raise exception...
L3:     load 4(r0),r1      ; array length
        load -20(fp),r2   ; index
        cmp r2,r1        ; bounds check
        brltu L4         ; branch if ok
        ... raise exception ...
L4:     sll #2,r2,r2      ; calculate byte offset
        add r0,r2,r2     ; add array base
        load 8(r2),r2     ; fetch b[i]
        store r2,-16(fp) ; push

; iadd
        load -12(fp),r0   ; fetch 2*a[i]
        load -16(fp),r1   ; fetch b[i]
        add r0,r1,r1     ; 2*a[i] + b[i]
        store r1,-12(fp) ; push

; iastore
        load -4(fp),r0    ; fetch array ptr
        cmp r0,#0         ; null check
        bneq L5          ; branch if ok
        ...raise exception...
L5:     load 4(r0),r1      ; array length
        load -8(fp),r2    ; index
        cmp r2,r1        ; bounds check
        brltu L6         ; branch if ok
        ... raise exception ...
L6:     sll #2,r2,r2      ; calculate byte offset
        add r0,r2,r2     ; add array base
        load -12(fp),r0   ; fetch value
        store r0,8(r2)   ; store a[i]

; return
        move fp,sp        ; pop operand stack frame
        load (sp),fp     ; restore frame pointer
        ret

```

61 instructions; 23 loads; 13 stores; 3 registers.

NATIVE CODE VERSION 2 (similar to quick1.c)  
 =====

Cache stack and local variables in registers, spilling (not shown here) if necessary.

Note: This approach may not buy too much on a machine with few registers, because of greatly increased spilling.

Register assignments:

Stack slots 0,...,4 correspond to registers: r0,...,r4

Local variable slots 0,...,2 correspond to registers: r10,r11,r12

Scratch register: r20

In this code, we assume that arguments must be loaded into registers from the stack the first time they are used. In reality, arguments are likely to be in registers already, so those local variables can just be assumed to be in place.

```
foo:          store fp,(sp)      ; save old frame pointer
              move sp,fp        ; new frame pointer
;aload_0
              load 16(fp),r10    ; fetch a (first time used)
              move r10,r0        ; push a
;iload_2
              load 8(fp),r12     ; fetch i (first time used)
              move r12,r1        ; push i
;iconst_2
              move #2,r2         ; push 2
;aload_0
              move r10,r3        ; push a
;iload_2
              move r12,r4        ; push i
;iaload
              cmp r3,#0          ; null check
              bneq L1           ; branch if ok
              ...raise exception...
L1: load 4(r3),r20               ; array length
              cmp r4,r20         ; bounds check
              brltu L2           ; branch if ok
              ... raise exception ...
L2: sll #2,r4,r4                 ; calculate byte offset
              add r3,r4,r4        ; add array base
              load 8(r4),r3       ; fetch and push a[i]
;imul
              mul r2,r3,r2        ; push a[i]*2
;aload_1
```

```
    load 12(fp),r11    ; fetch b (first time used)
    move r11,r3        ; push b
;iload_2
    move r12,r4        ; push i
;iaload
    cmp r3,#0          ; null check
    bneq L3            ; branch if ok
    ...raise exception...
L3: load 4(r3),r20      ; array length
    cmp r4,r20         ; bounds check
    brltu L4           ; branch if ok
    ... raise exception ...
L4: sll #2,r4,r4       ; calculate byte offset
    add r3,r4,r4       ; add array base
    load 8(r4),r3      ; fetch and push b[i]
; iadd
    add r2,r3,r2       ; push a[i]*2 + b[i]
; iastore
    cmp r0,#0          ; null check
    bneq L5            ; branch if ok
    ...raise exception...
L5: load 4(r0),r20      ; array length
    cmp r1,r20         ; bounds check
    brltu L6           ; branch if ok
    ... raise exception ...
L6: sll #2,r1,r1       ; calculate byte offset
    add r0,r1,r1       ; add array base
    store r2,8(r1)     ; store a[i]
; return
    load (sp),fp       ; restore frame pointer
    ret
```

41 instructions; 9 loads; 2 stores; 9 registers.

NATIVE CODE VERSION 3 (similar to quickla.c)

=====

Use more elaborate descriptors for stack slots:

- register (either stack temporary or local variable)
- constant (small enough for direct use as operand)

to achieve constant propagation and avoid register-register moves.

Stack registers: r0,... (allocate as needed)

Local variable registers: r10,r11,r12

Scratch registers: r20

```
foo:      store fp,(sp)      ; save old frame pointer
          move sp,fp        ; new frame pointer

;aload_0
          load 16(fp),r10    ; fetch a (first ref)      stack[0] = r10
;iload_2
          load 8(fp),r12     ; fetch i (first ref)      stack[1] = r12
;iconst_2
          ;                  ;                          stack[2] = 2
;aload_0
          ;                  ;                          stack[3] = r10
;iload_2
          ;                  ;                          stack[4] = r12
;iaload

          cmp r10,#0         ; null check
          bneq L1            ; branch if ok
          ...raise exception...
L1:      load 4(r10),r20     ; array length
          cmp r12,r20       ; bounds check
          brltu L2          ; branch if ok
          ... raise exception ...
L2:      sll #2,r12,r20     ; calculate byte offset
          add r10,r20,r20   ; add array base
          load 8(r20),r0    ; fetch a[i]                stack[3] =r0 (tos=3)
;imul
          mul r0,2,r0       ; a[i]*2                    stack[2] =r0 (tos=2)
;aload_1
          load 12(fp),r11   ; fetch b (first ref)      stack[3] = r11
;iload_2
          ;                  ;                          stack[4] = r12
;iaload

          cmp r11,#0         ; null check
          bneq L3            ; branch if ok
          ...raise exception...
L3:      load 4(r11),r20     ; array length
          cmp r12,r20       ; bounds check
          brltu L4          ; branch if ok
          ... raise exception ...
L4:      sll #2,r12,r20     ; calculate byte offset
```

```
    add r11,r20,r20    ; add array base
    load 8(r20),r1     ; fetch b[i]           stack[3] =r1 (tos=3)
; iadd
    add r0,r1,r0       ; a[i]*2 + b[i]       stack[2] =r0 (tos=2)
; iastore
    cmp r10,#0         ; null check
    bneq L5           ; branch if ok
    ...raise exception...
L5: load 4(r10),r20    ; array length
    cmp r12,r20       ; bounds check
    brltu L6         ; branch if ok
    ... raise exception ...
L6: sll #2,r12,r20    ; calculate byte offset
    add r10,r20,r20    ; add array base
    store r0,8(r20)    ; store a[i]
; return
    load (sp),fp      ; restore frame pointer
    ret
```

33 instructions; 9 loads; 2 stores; 6 registers.

## SCHEME 3 AND STORES

=====

Note that we must be careful not to store to a local variable register if that register currently contains a stack value -- we must first move the value to a temporary register (updating the stack description)

Because Scheme 3 assigns the local variables to fixed registers, the generated code can still have lots of 'move' instructions corresponding to 'store's from stack registers.

Some possible solutions:

- Allow local variables to move, i.e. allocate registers to them on an as-needed basis. This is messy in practice (more below).
- Keep local vars fixed, but use on-the-fly peephole optimization to elide unnecessary 'move' instructions.

## BEYOND BASIC BLOCKS

=====

Scheme 3, unlike the previous ones, only works within a single basic block.

At join points, stack slot descriptors may be incompatible:

Example Java code:

```
public static void foo (int a, int b) {
    a = (a > b) ? 0 : 1;
}
```

Bytecode:

```
0:   iload_0
1:   iload_1
2:   if_icmple      9
5:   iconst_0
6:   goto          10
9:   iconst_1
10:  istore_0      <-- join point: what is in stack[0] ?
11:  return
```

To extend the approach beyond basic blocks, we must reconcile stack slot descriptors (and local variable descriptors, if they can move) at control flow join points.

- Awkward, because we'd like to emit code for one branch before examining the other.

- One way to do this is to "normalize" the descriptors at every join point, e.g. by moving all values into standard stack registers. Details are rather messy, especially if we let local variables move too.

- Simple approach (for homework): only handle code for which stack is \*empty\* at every join point. Also, don't let locals move. Hence, there's never anything to reconcile.

- This isn't really very restrictive: javac normally produces code in which the stack is empty after each statement -- hence only when a single statement generates an internal join point can we get join points with non-empty stacks. Probably the :? operator is the way to produce such things.