7. Translating JVM Code to MIPS Code

7.1 There is Life for a Compiler After the JVM

7.1.1 What Happens to JVM Code

Compilation is not necessarily over with after the class file is constructed. At "execution", the class encoded in that file is loaded into the JVM and then interpreted. In the Oracle HotSpot™ VM, once a method has been executed several times, it is compiled to native code (that is code that can be directly executed by the underlying computer, which is executing the VM). There are other times JVM code is compiled to native code such as regions of code that are executed many times. Once these "hotspots" in the code are compiled, the native code is cached so that it may be re-used in subsequent invocations of the method. So at run time, control shifts back and forth between JVM code and native code. Of course the native code runs much faster than does the interpreted JVM code. This regime takes advantage of the fact that nearly all programs spend most of their time executing small regions of code.

This scenario is further complicated by the existence of at least two run-time environments. The VM that runs on a server performs many optimizations on the native code. While these optimizations are expensive they produce very fast code that, on a server, runs over and over again. On the other hand, the VM that runs on a client computer, such as a user's personal workstation, performs fewer optimizations and so minimizes the one-time cost of compiling the method to native code.

Compiling JVM code to native code involves the following:

- **Register allocation.** In the JVM code, all local variables, transients and instruction operands are stored on the run-time stack. On the other hand, the computer architectures we are compiling to have a fixed number of (often eight or thirty-two) high-speed registers. The native instructions often require that operands are in registers and assigning as many local variables and transients to registers makes for faster running programs. When all registers are full and the computation requires another, one register must be spilled to a memory location, perhaps on the runtime stack. Register spilling in turn degrades performance and so is to be minimized. So register allocation becomes extremely important.
- **Optimization.** The code produced can be improved so as to run faster. In some instances, invocations of methods having short bodies can be in lined, that is replaced by the bodies' code
- **Instruction selection.** We must choose and generate the native code instructions sufficient to perform the computations
- **Run-time support.** A certain amount of functional support is required at run time. For example, we need support for creating new objects on the heap. We also

implement a SPIM class that gives us basic input and output functionality for testing our programs.

One might reasonably ask, “Why don’t we simply compile the entire JVM program to native code at the very start?” One reason is the semantics of Java; the dynamic quality of Java means we are not exactly sure which code will be needed at compile time. Further, new classes may be loaded while a Java program is executing. That HotSpot compilation can compete with static compilation in performance is counter-intuitive to programmers. But Oracle has benchmarks that demonstrate that the HotSpot VM often outperforms programs that have been statically compiled to native code.

7.1.2 What We Will Do Here and Why

Here we shall translate a certain subset of JVM instructions to the native code for the MIPS architecture. MIPS is a relatively modern reduced instruction set computer (RISC), which has a set of simple but fast instructions that operate on values in registers; for this reason it is often referred to as a register-based architecture – it relies on loads and stores for moving values between memory and its 32 general purpose registers. The RISC architecture differs from the traditional complex instruction set computer (CISC), which has fewer but more complex (and so slower) instructions with a wider range of operand addressing capabilities; CISC operands can be expressed as registers, memory locations or a combination of both (allowing indexing) and so CISC architectures normally have fewer (for example, 8) general purpose registers. A popular CISC architecture is Intel’s family of i86x computers.

More accurately, we will target the MIPS *assembly language*, which is directly interpreted by James Larus’s SPIM simulator (Larus, 2000-2010), which is readily available for many environments. We call this assembly language SPIM code.

Assembly code is a symbolic language for expressing a native code program. It captures the native code program nicely because there is a one-to-one correspondence between it and the bit pattern representing each individual instruction but is meaningful to the reader.

Normally, a compiler will produce this assembly code and then will use an *assembler*, that is a simple translator, for translating the assembly code to the bit representation of native code.

But the SPIM interpreter interprets the MIPS assembly code directly. This serves our purpose just fine; we can produce sequences of MIPS instructions that are both readable and can be executed directly for testing the code we generate.

Our goal here is illustrated in Figure 7.1. The work that we have already done is shown using the dashed line. The work we intend to do here in this chapter and the next is shown using the solid line.

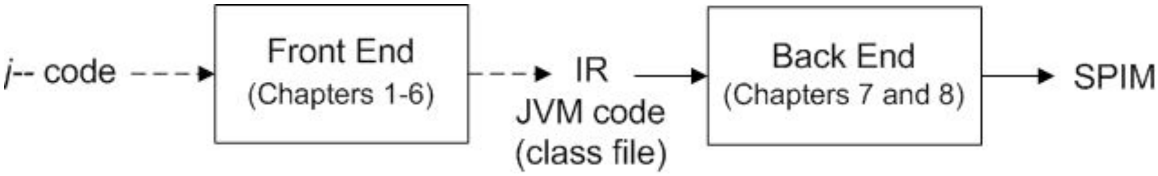


Figure 7.1 Our *j*-- to SPIM Compiler

In this regime, we can redefine what constitutes the IR, the front end and the back end:

- JVM code is our new IR.
- The *j*-- to JVM translator (discussed in chapters 1 – 6) is our new front end.
- The JVM to SPIM translator (discussed in this chapter and the next) is our new back end.

7.1.3 The Scope of Our Work

We translate a sufficient subset of the JVM to SPIM code to give the student a taste of native code generation, some of the possibilities for optimization, and register allocation. More precisely, we translate enough JVM code to SPIM code to handle the following *j*-- program[1].

```
import spim.SPIM;

// Prints factorial of a number computed using recursive and iterative
// algorithms.

public class Factorial
{
    // Return the factorial of the given number computed recursively.

    public static int computeRec(int n)
    {
        if (n <= 0) {
            return 1;
        }
        else {
            return n * computeRec(n - 1);
        }
    }

    // Return the factorial of the given number computed iteratively.

    public static int computeIter(int n)
    {
        int result = 1;
        while ( n > 0 ) {
            result = result * n--;
        }
```

[1] We also compile a few more programs, which are included in the code tree but are not discussed further here: **`Fibonacci`**, **`GCD`**, **`Formals`** and **`HelloWorld`**.

```
        return result;
    }

    // Entry point; print factorial of a number computed using
    // recursive and iterative algorithms.

    public static void main(String[] args)
    {
        int n = 7;
        SPIM.printInt(Factorial.computeRec(n));
        SPIM.printChar('\n');
        SPIM.printInt(Factorial.computeIter(n));
        SPIM.printChar('\n');
    }
}
```

We handle static methods, conditional statements, while loops, recursive method invocations, and enough arithmetic to do a few computations. We must deal with some objects, e.g. constant strings. Although the program above refers to an array, it does not really do anything with it so we do not implement array objects. Our runtime support is minimal.

In order to determine just what JVM instructions must be handled, it is worth looking at the output from running **`javap`** on the class file produced for this program.

```
public class Factorial extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
… <the constant pool is elided here> …

{
public Factorial();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial     #8; //Method java/lang/Object."<init>":()V
   4: return

public static int computeRec(int);
  Code:
   Stack=3, Locals=1, Args_size=1
   0: iload_0
   1: iconst_0
   2: if_icmpgt   10
   5: iconst_1
   6: ireturn
   7: goto  19
   10:iload_0
   11:iload_0
   12:iconst_1
   13:isub
   14:invokestatic      #13; //Method computeRec:(I)I
   17:imul
   18:ireturn
   19:nop

public static int computeIter(int);
  Code:
   Stack=2, Locals=2, Args_size=1
   0: iconst_1
   1: istore_1
   2: iload_0
   3: iconst_0
   4: if_icmple   17
   7: iload_1
   8: iload_0
   9: iinc  0, -1
   12:imul
   13:istore_1
   14:goto  2
   17:iload_1
   18:ireturn

public static void main(java.lang.String[]);
  Code:
   Stack=1, Locals=2, Args_size=1
   0: bipush      7
   2: istore_1
   3: iload_1
   4: invokestatic      #13; //Method computeRec:(I)I
   7: invokestatic      #22; //Method spim/SPIM.printInt:(I)V
   10:bipush      10
   12:invokestatic      #26; //Method spim/SPIM.printChar:(C)V
   15:iload_1
   16:invokestatic      #28; //Method computeIter:(I)I
   19:invokestatic      #22; //Method spim/SPIM.printInt:(I)V
   22:bipush      10
   24:invokestatic      #26; //Method spim/SPIM.printChar:(C)V
   27:return
}
```

All methods (other than the default constructor) are static; thus, the plethora of `invokestatic` instructions. Otherwise, we see lots of loads, stores, a few arithmetic operations and both conditional and unconditional branches.

Translating additional instructions is left as a set of exercises.

7.2 SPIM and the MIPS Architecture

SPIM and the MIPS computer it simulates are both nicely described by SPIM's author in (Larus, 2009). Our brief description here is based on that.

7.2.1 MIPS Organization

The MIPS computer organization is sketched out in Figure 7.2.

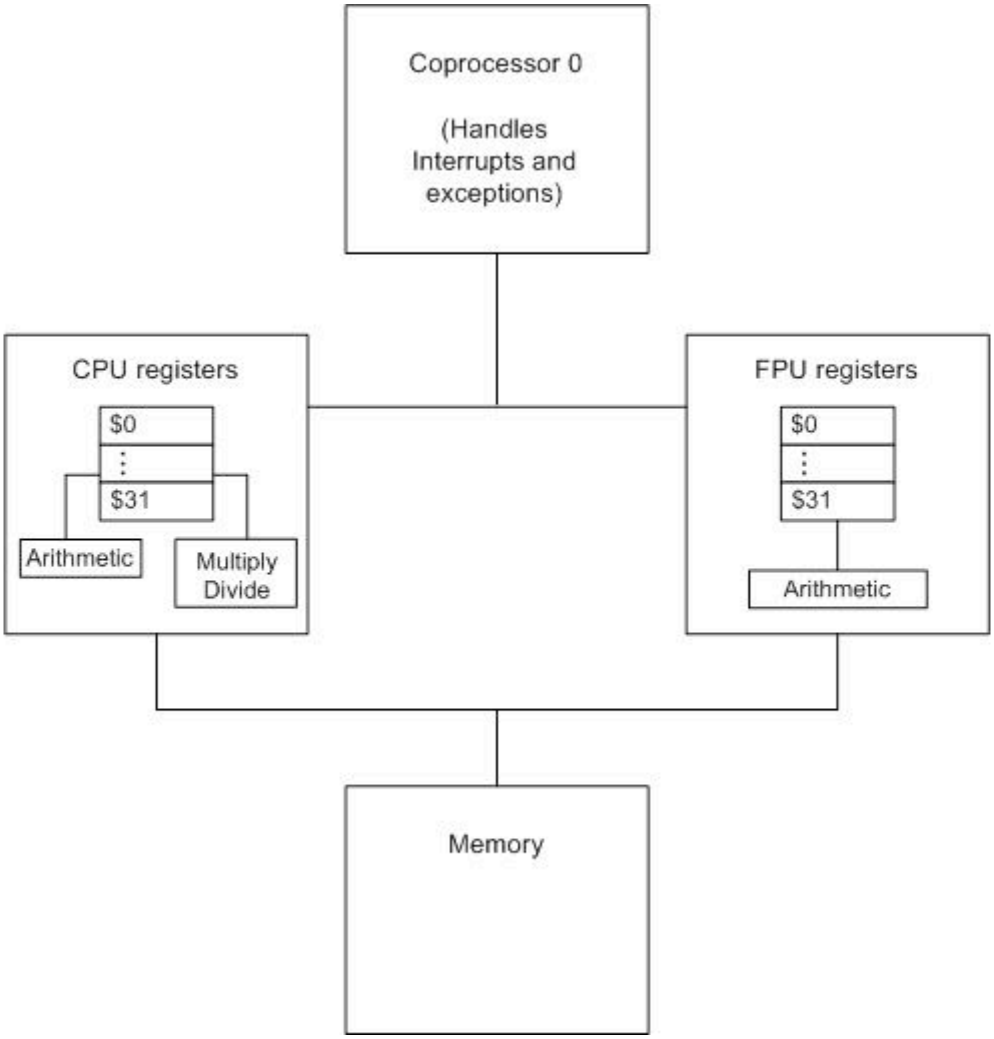


Figure 7.2 The MIPS Computer Organization

It has an integer central processing unit (CPU), which operates on 32 general purpose registers (numbered $0 - $31); a separate floating point coprocessor 1 (FPU), with its own 32 registers ($f0 - $f31) for doing single precision (32 bit) and double precision (64 bit) floating point arithmetic, and coprocessor 0 – for handling exceptions and interrupts, also with its own set of registers, including a status register. There are instructions for moving values from one register set to another. Our translation will focus on the integer-processing unit; exercises may involve the reader's dealing with the other coprocessors.

Programming the raw MIPS computer can be quite complicated given its (time) delayed branches, delayed loads, caching and memory latency. Fortunately, the MIPS assembler models a virtual machine that both hides these timing issues and uses pseudo-instructions to provide a slightly richer instruction set. By default, SPIM simulates this virtual machine although it can be configured to model the more complicated raw MIPS computer. We will make use of the simpler default assembly language.

7.2.2 Memory Organization

Memory is by convention divided into four segments, as illustrated in Figure 7.3 and derived from (Larus, 2009).

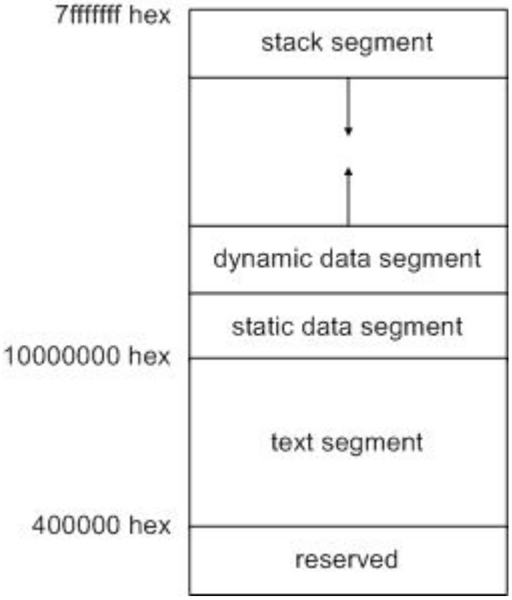


Figure 7.3 SPIM Memory Organization

1. **Text segment**. The program's instructions go here, near the bottom end of memory, starting at 400000_{hex}. (The memory location below 400000_{hex} are reserved.)
2. **Static data segment**. Static data, which exist for the duration of the program, go here, starting at 1000000_{hex}. This would be a good place to put constants (including constant strings) and static fields. To access these values conveniently, MIPS designates one of its 32 registers $gp (register number 28), which points to the middle of a 64K block data in this segment. Individual memory locations may be addressed at fixed (positive or negative) offsets from $gp.
3. **Dynamic data segment.** Often called the heap, this is where objects and arrays are dynamically allocated during execution of the program. Many systems employ a garbage collection to reclaim the space allocated to objects that are no longer of use. This segment starts just above the static data segment and grows upward, towards the runtime stack.
4. **Stack segment.** The stack is like that for the JVM. Every time a routine is called, a new stack frame is pushed onto the stack; every time a return is executed, a frame is popped off. The stack starts at the top of memory and grows downward toward the heap. That the dynamic data segment and stack segment start as far apart as possible and grow toward each other leads to an effective use dynamic

memory. We use the $fp register (30) to point to the current stack frame and the $sp register (29) to point to the last element on the stack.

MIPS and SPIM require that quantities be aligned on byte addresses that are multiples of their size; e.g., 32-bit words must be at addresses that are multiples of four and 64-bit doubles must be at addresses that are multiples of eight. Fortunately, the MIPS assembler (and, so SPIM) provides program directives for correctly aligning data quantities. For example, in

```
b:    .byte 10
w:    .word 324
```

the byte 10 is aligned on a byte boundary and may be addressed using the label b, and the word 324 is aligned on a 4-byte boundary and may be addressed as w; w is guaranteed to address a word whose address is evenly divisible by 4. There are assembler directives for denoting and addressing data items of several types in the static data segment. See Appendix E and (Larus, 2009).

Interestingly, SPIM does not define a byte order for quantities of several bytes; rather, it adopts the byte order of the machine on which it is running. For example, on an Intel x86 machine (Windows or Mac OS X on an Intel x86) the order is said to be little-endian, meaning that the bytes in a word are ordered from right to left. On Sparc or a Power PC (Solaris or Mac OS X on a Sparc) the order is said to be big-endian, meaning that the bytes in a word are ordered from left to right. For example, given the directive,

```
w:    .byte 0, 1, 2, 3
```

if we address w as a *word*, the 0 will appear in the lowest-order byte on a little-endian machine (as in Figure 7.4a) and in the highest order byte on a big-endian machine (as in Figure 7.4b).

3
2
1
0
little-endian
(a)
0
1
2
3
big-endian
(b)

Figure 7.4 Little-endian vs. Big-endian

For most of our work, the endian-ness of our underlying machine on which SPIM is running should not matter.

7.2.3 Registers

Many of the 32 32-bit general-purpose registers, by convention are designated for special uses, and have alternative names that remind us of these uses:

- $zero (register number 0) always holds the constant 0.
- $at (1) is reserved for use by the assembler and should not be used by programmers or in code generated by compilers.
- $v0 (2) and $v1 (3) are used for expression evaluation and as the results of a function.
- $a0 - $a3 (4 – 7) are used for passing the first four arguments to routines; any additional arguments are passed on the stack.
- $t0 - $t7 (8 – 15) are meant to hold temporary values that need not be preserved across routine calls. If they must be preserved, it is up to the caller (the routine making the call) to save them; hence they are called *caller-saved registers*.
- $s0 - $s7 (16 – 23) are meant to hold values that must be preserved across routine calls. It is up to the callee (the routine being called) to save these registers; hence they are called *callee-saved registers*.
- $t8 and $t9 (24 and 25) are caller-saved temporaries.
- $k0 (26) and $k1 (27) are reserved for use by the operating system kernel and so should not be used by programmers or in code generated by compilers.
- $gp (28) is a global pointer; it points to the middle of a 64K block of memory in the static data segment.
- $sp (29) is the stack pointer, pointing to the last location on the stack.
- $fp (30) is the stack frame pointer, pointing to the latest frame on the stack.
- $ra (31) is the return address register, holding the address to which execution should continue upon return from the latest routine.

It is a good idea to follow these conventions when generating code; this allows our code to cooperate with code produced elsewhere.

7.2.4 Routine Call and Return Convention

SPIM assumes we follow a particular protocol in implementing routine calls, when one routine (the caller) invokes another routine (the callee).

Most bookkeeping for routine invocation is recorded in a stack frame on the runtime stack segment, much as like is done in the JVM; but here we must also deal with registers. The stack frame for an invoked (callee) routine is illustrated in Figure 7.5.

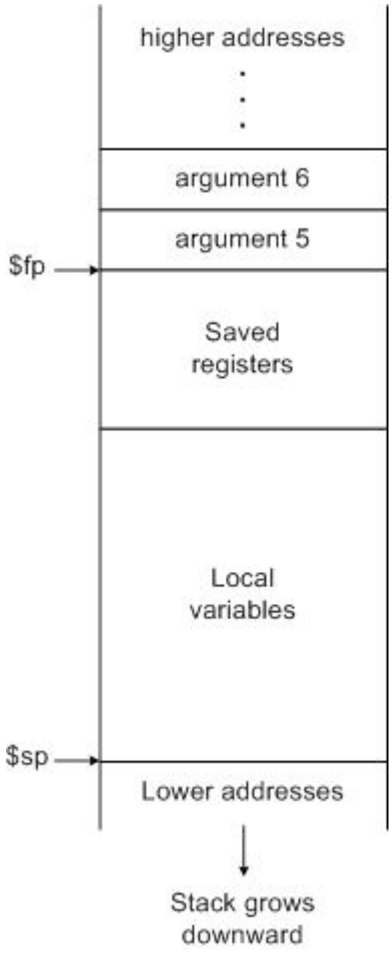


Figure 7.5 A Stack Frame

SPIM has pretty well defined protocols for preparing for a call in the calling code, making a call, saving registers in the frame at the start of a method, restoring the saved registers from the frame at the end of a method, and executing a return to the calling code. We follow these protocols closely in our run-time stack management and they are described where we discuss our run-time stack in section 7.3.5.3.

7.2.5 Input and Output

The MIPS computer is said to have memory-mapped I/O, meaning the input and output device registers are referred to as special, reserved addresses in memory. Although SPIM does simulate this for a simple console, it also provides a set of simple system calls for accessing simple input and output functions. We shall use these system calls for our work.

7.3 Our Translator

7.3.1 The Organization of our Translator

Our JVM to SPIM translator is based on those described by Christian Wimmer (Wimmer, 2004) and Hanspeter Mössenböck (Mössenböck, 2000), which are in turn versions of the Sun (now Oracle) HotSpot Client Compiler. Our translator differs in several ways from those described by Wimmer and Mössenböck, and while theirs produce native code for the Intel 86x, ours targets SPIM, a simulator for the MIPS32 architecture.

Roughly, the phases proceed as illustrated in Figure 7.6.

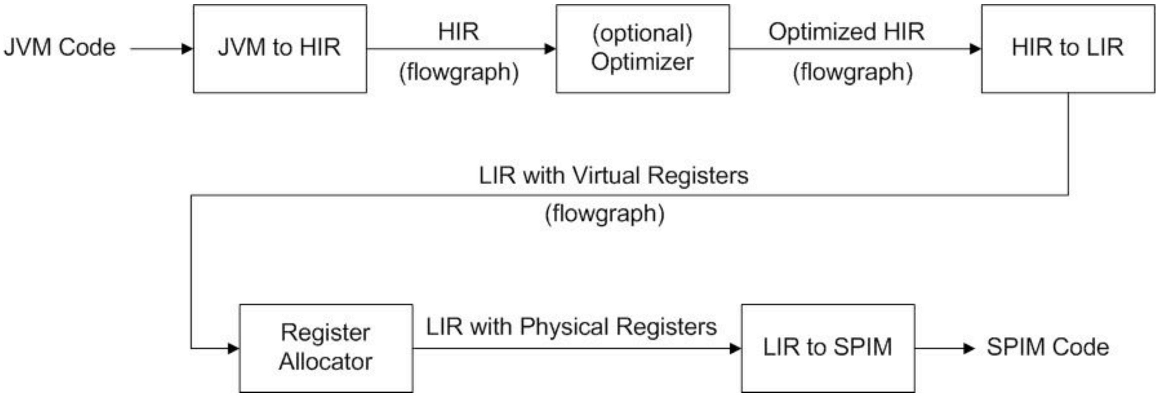


Figure 7.6 Phases of the JVM to SPIM Translator

In the first phase, the JVM code is parsed and translated to a *control flow graph*, composed of basic blocks. A basic block consists of a linear sequence of instructions with just one entry point at the start of the block, and one exit point at the end; otherwise, there are no branches into or out of the block. The instructions are of a high level and preserve the tree structure of expressions; following Wimmer, we call this a high-level intermediate representation (HIR). Organizing the code in this way makes it more amenable to analysis.

In the second (optional) phase, various optimizations may be applied to improve the code, making it faster and often smaller.

In the third phase, a lower level representation of the code is constructed, assigning explicit virtual registers to the operands of the instructions; there is no limit to the number of virtual registers used. Following Wimmer, we call this a low-level intermediate representation (LIR). The instructions are very close to the instructions in our target instruction set, SPIM.

In the fourth phase, we perform register allocation. The 32 physical registers in the MIPS architecture (and so in SPIM) are assigned to take the place of virtual registers. Register allocation is discussed in Chapter 8.

In the fifth and final phase, we generate SPIM code – our goal.

The names of all classes participating in the translation from JVM code to SPIM code begin with the letter **`N`** (the **`N`** stands for Native). The translation is directed by the driver, **`NEmitter`**; most of the steps are directed by its constructor **`NEmitter()`**.

`NEmitter()` iterates through the classes and methods for each class, constructing the control flow graph of HIR instructions for each method, doing any optimizations, rewriting the HIR as LIR and performing register allocation.

SPIM code is emitted by the driver (**`Main`** or **`JavaCCMain`**) by invoking **`NEmitter`**'s **`write()`** method.

7.3.2 The HIR Control Flow Graph

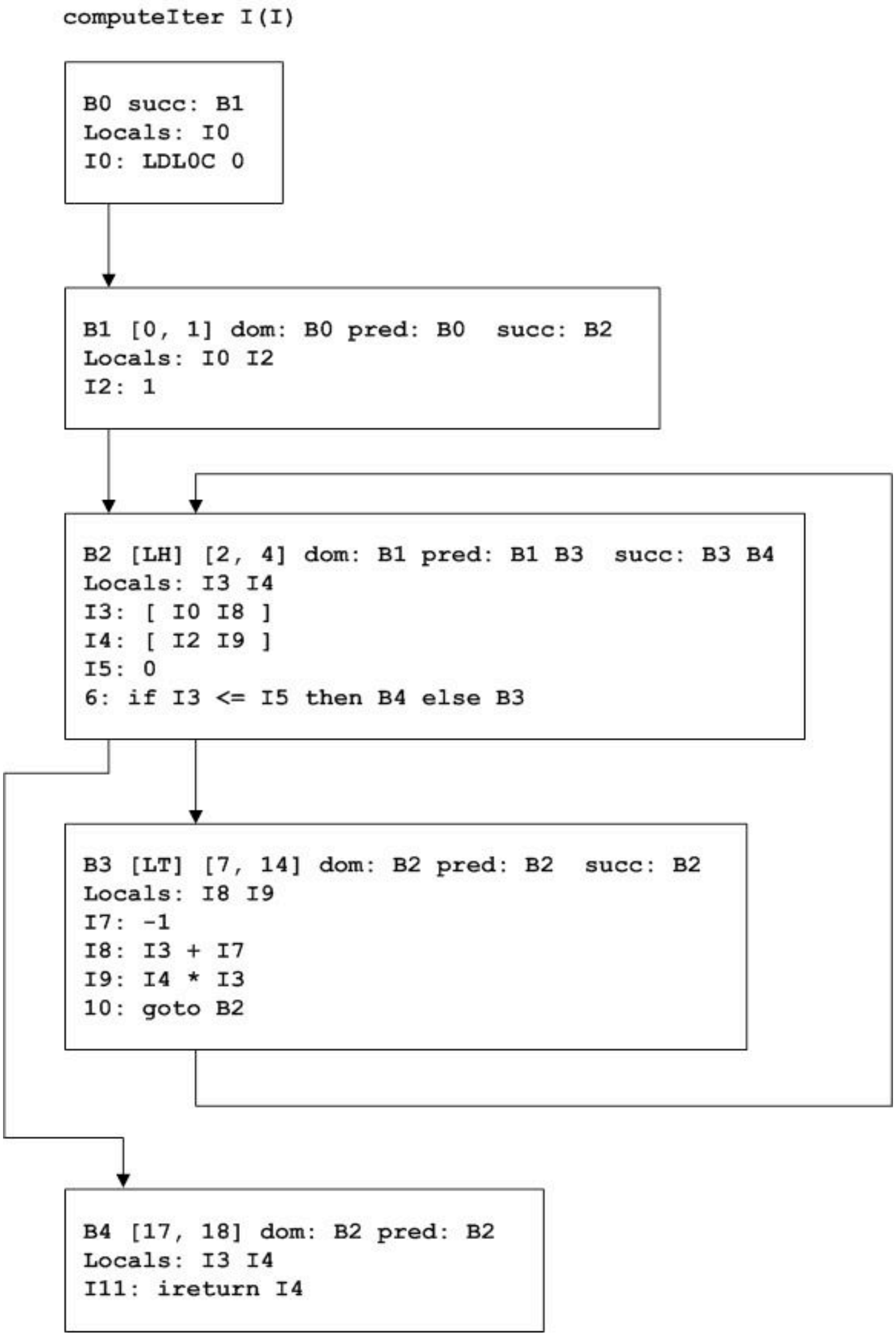
7.3.2.1 The Control Flow Graph

The first step is to scan through the JVM instructions and construct a flow graph of basic blocks. A *basic block* is a sequence of instructions with just one entry point at the start and one exit point at the end; otherwise, there are no branches into or out of the instruction sequence.

To see what happens here, consider the JVM code for the method **`computeIter()`** from our **`Factorial`** example.

```
public static int computeIter(int);
  Code:
   Stack=2, Locals=2, Args_size=1

   0: iconst_1
   1: istore_1

   2: iload_0
   3: iconst_0
   4: if_icmple   17

   7: iload_1
   8: iload_0
   9: iinc  0, -1
   12:imul
   13:istore_1
   14:goto  2

   17:iload_1
   18:ireturn
```

We've inserted line breaks to delineate basic blocks. The entry point is the **`iconst_1`** instruction at location 0 so that begins a basic block; let's call it B1. The **`iload_0`** at location 2 is the target of the **`goto`** instruction (at 14) so that also must start a basic block; we'll call it B2. B2 extends through the **`if_icmple`** instruction; the block must end there since it is a branch. The next basic block (B3) begins at location 7 and extends to the **`goto`** at location 14. Finally, the **`iload_1`** at location 17 is the target of the **`if_icmple`** branch at location 4 so it starts basic block B4. B4 extends to the end of the method and is terminated by the **`ireturn`** instruction.



```
computeIter I(I)

B0 succ: B1
Locals: I0
I0: LDLOC 0

B1 [0, 1] dom: B0 pred: B0  succ: B2
Locals: I0 I2
I2: 1

B2 [LH] [2, 4] dom: B1 pred: B1 B3  succ: B3 B4
Locals: I3 I4
I3: [ I0 I8 ]
I4: [ I2 I9 ]
I5: 0
6: if I3 <= I5 then B4 else B3

B3 [LT] [7, 14] dom: B2 pred: B2  succ: B2
Locals: I8 I9
I7: -1
I8: I3 + I7
I9: I4 * I3
10: goto B2

B4 [17, 18] dom: B2 pred: B2
Locals: I3 I4
I11: ireturn I4
```

Figure 7.7 HIR Flow Graph for Factorial.computeIter()

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks; indeed all control flow information is captured in the graph. Notice that the boxes and arrows are not really necessary to follow the flow of control among blocks; we put them there in the figure only for emphasis. You will notice that the first line of text within each box identifies the block, a list of any successor blocks (labeled by `succ:`) and a list of any predecessor blocks (labeled by `pred:`). For example, block B3 has a predecessor B2 indicating that control flows from the end of block B2 into the start of B3; B3 also has a successor B2 indicating that control flows from the end of B3 to the start of block B3. Notice that we add an extra beginning block B0 for the method's entry point; this makes certain analysis operations, done later, easier.

The denotation `[LH]` on the first line of B2 indicates that B2 is a *loop header* – the first block in a loop. The denotation `[LT]` on the first line of B3 indicates that B3 is a *loop tail* – the last block in a loop (and a predecessor of the loop header). This information is used later for identifying loops for performing optimizations and ordering blocks for optimal register allocation.

The pairs of numbers within square brackets, for example `[7, 14]` on the first line of B3, denote the ranges of JVM instructions captured in the block.

The keyword **dom:** labels the basic block's *immediate dominator.* In our graph, a node d is said to *dominate* a node n if every path from the entry node (B0) to n must go through d. A node d *strictly dominates* n if it dominates n but isn't the same as n. Finally, node d is an immediate dominator of node n if d strictly dominates n but does not dominate any other node that strictly dominates n. That is, it's the node on the path from the entry node to n that is the "closest" to n. Dominators are useful for certain optimizations, including the lifting of loop-invariant code (see section 7.3.3.5).

7.3.2.2 The State Vector

Local variables are tracked in a *state vector* called `locals` and are indexed in this vector by their location in the JVM stack frame. The current state of this vector *at the end of a block's instruction sequence* is printed on the block's second line and labeled with `Locals:`. The values are listed in positional order and each value is represented by the instruction id for the instruction that computes it.

For example, in B0 this vector has just one element, corresponding to the method's formal argument `n`. `I0` identifies the instruction `LDLOC 0` that loads the value (for `n`) from position 0 on the stack.[2]; notice the instruction is typed to be integer by the `I` in `I0`. In B1 the vector has two elements: the first is `I0` for `n` and the second is `I2` for `result`.

[2] The argument actually will be in register $a0 in the SPIM code, following SPIM convention.

7.3.2.3 HIR Instructions

The instruction sequence within each basic block is of a higher level than is JVM. These sequences capture the expression trees from the original source. We follow (Wimmer, 2004) in calling this a high-level intermediate representation (HIR). For example, the Java statement,

```
w = x + y + z;
```

might be represented in HIR by

```
  I8: I0 + I1
  I9: I8 + I2
```

The `I0`, `I1` and `I2` refer to the instruction ids labeling instructions that compute values for `x`, `y` and `z` respectively. So the instruction labeled `I8` computes the sum of the values for `x` and `y`, and then the instruction labeled `I9` sums that with the value for `z` to produce a value for `w`. Think of this as an abstract syntax tree such as that illustrated in Figure 7.8. The `I`'s are type labels, indicating the types are (in this case) integers.

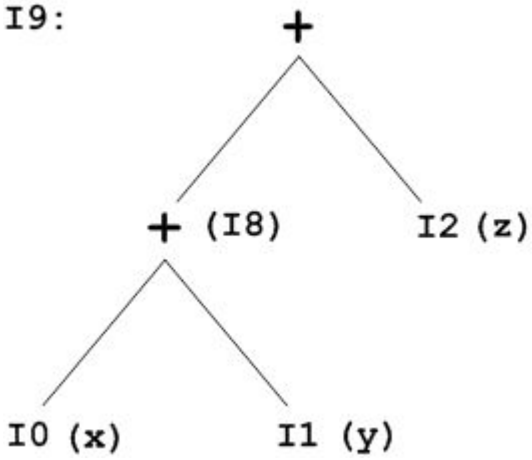


Figure 7.8 (HIR) AST for `w = x + y + z`

The instruction for loading a constant is simply the constant itself. For example, consider block B1 in Figure 7.7. It has just the single instruction

```
  I2: 1
```

Which is to say, the temporary `I2` (later, `I2` will be allocated a register) is loaded with the constant `1`. `1` is that instruction that loads the constant `1`.

Not all instructions generate values. For example, the instruction

```
6: if I3 <= I5 then B4 else B3
```

in block B2 produces no value but transfers control to either B4 or B3. Of course, the `6:` has no type associated with it.

As we shall see in section 7.2.3, HIR lends itself to various optimizations.

7.3.2.4 Static Single Assignment (SSA) Form

Our HIR employs single static assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value. This means that when a variable is re-assigned in the method, one must create a new version for it.

For example, given the simple sequence,

```
x = 3;
x = x + y;
```

the second assignment to x requires that we create a new version. For example, we might subscript our variables to distinguish different versions.

```
X_1 = 3;
X_2 = X_1 + y_1;
```

In the HIR we represent a variable's value by the instruction that computed it and we track these values in the state vector. The value in a state vector's element may change as we sequence through the block's instructions. If the next block has just one predecessor, it can copy the predecessor's state vector at its start; if there are more than two predecessors, the states must be merged.

For example, consider the following *j--* method, where the variables are in SSA form.

```
static int ssa( int w_1 ) {
    if (w_1 > 0)
        w_2 = 1;
    else
        w_3 = 2;
    return w_?;
}
```

In the statement,

```
return w_?;
```

which **w** do we return, $\mathbf{w}_2$ or $\mathbf{w}_3$? Well, it depends on which of the two paths taken through the if-then-else statement. In terms of basic blocks, it is as illustrated in Figure 7.9.

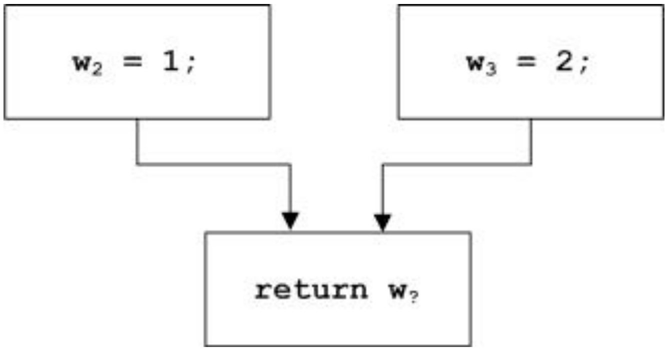


Figure 7.9 The SSA Merge Problem

We solve this problem by using what is called a Phi function, a special HIR instruction that captures the possibility of a variable having one of several values.
In our example, the final block would contain the following code.

```
w4 =  [w2 w3];
return w4;
```

The `[w2 w3]` represents a Phi function with two operands; the operand w_2 captures the possibility that one branch of the if-then-else was taken and the w_3 captures the possibility that the other branch was taken. Of course, the target SPIM has no representation for this but we shall remove these special Phi instructions before attempting to generate SPIM instructions. We can still do analysis on it. The data flow is illustrated in Figure 7.10.

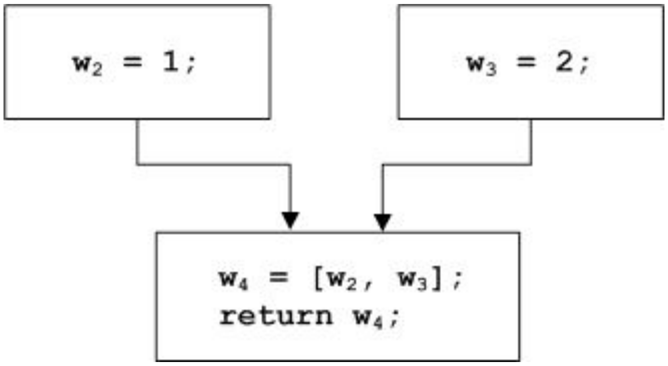


Figure 7.10 Phi Functions Solve the SSA Merge Problem

Another place where Phi functions are needed are in loop headers. Recall, a loop header is a basic block having at least one incoming backward branch and at least two predecessors, as is illustrated in Figure 7.11.

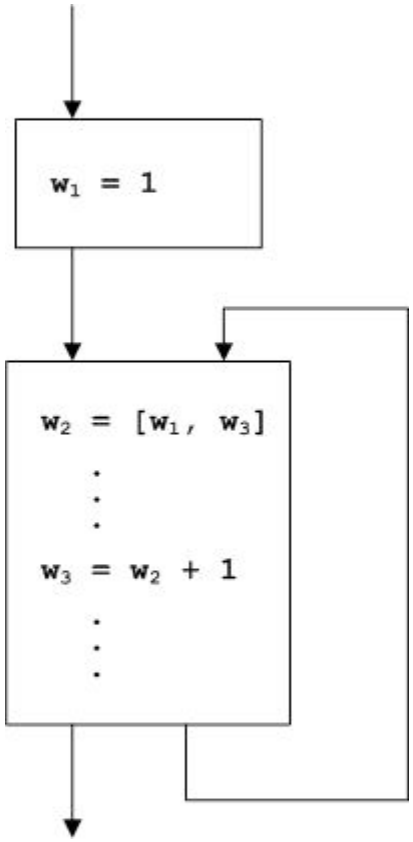


Figure 7.11 Phi Functions in Loop Headers

Unfortunately, a changed variable flowing in from a backward branch is known only after the block has been fully processed. For this reason, at loop headers, we conservatively define Phi functions for all variables and then remove redundant Phi functions later. In the first instance, w_2 can be defined as a Phi function with operands w_1 and w_2.

```
w2 = [w1 w2]
```

Then, when **w** is later incremented, the second operand may be overwritten by the new w_3.

```
w2 = [w1 w3]
```

Of course, a redundant Phi function will not be changed. If the **w** is never modified in the loop body, the Phi function instruction takes the form,

```
w2 = [w1 w2]
```

and can be removed as it is apparent that **w** has not been changed.

Phi functions are tightly bound to state vectors. When a block is processed:

- If the block has just a single predecessor, then it may inherit the state vector of that predecessor; the states are simply copied.
- If the block has more than one predecessor, then those states in the vectors that differ must be merged using Phi functions.
- For loop headers we conservatively create Phi functions for all variables, and then later remove redundant Phi functions.

In block B2 in the code for Factorial (Figure 7.7), **`I3`** and **`I4`** identify the instructions:

```
I3: [ I0 I8 ]
I4: [ I2 I9 ]
```

These are Phi function instructions capturing the local variable **`n`** and **`result`** respectively. **`I3`** is a Phi function with operands **`I0`** and **`I8`**; **`I4`** is a Phi function with operands **`I2`** and **`I9`**.

7.3.2.5 Control Flow Graph Construction

The best place to look at the translation process is the constructor **`NEmitter()`** in class **`NEmitter`**. For each method, the control flow graph of HIR is constructed in several steps:

1. The **`NControlFlowGraph`** constructor is invoked on the method. This produces the control flow graph **`cfg`**. In this first step, the JVM code is translated to sequences of tuples.
    a. Objects of type **`NBasicBlock`** represent the basic blocks in the control flow graph. The control flow is captured by the links, **`successors`** in each block. There are also, the links **`predecessors`** for analysis.
    b. The JVM code is first translated to a list of tuples, corresponding to the JVM instructions. Each block stores its sequence of tuples in an array list, **`tuples`**. For example, the blocks of tuples for our **`Factorial`** example, **`computeIter`** are as follows.

```
B0

B1
0: iconst_1
1: istore_1

B2
2: iload_0
3: iconst_0
4: if_icmple   0   13

B3
7: iload_1
8: iload_0
9: iinc   0   255
12: imul
```

```
13: istore_1
14: goto   255   244

B4
17: iload_1
18: ireturn
```

This use of tuples isn’t strictly necessary but it makes the translation to HIR easier.

2. The message expression,

   ```
   cfg.detectLoops(cfg.basicBlocks.get(0), null);
   ```

   detects loop headers and loop tails. This information may be used for lifting invariant code from loops during the optimization phase and for ordering blocks for register allocation.

3. The message expression,

   ```
   cfg.removeUnreachableBlocks();
   ```

   removes unreachable blocks, for example, blocks resulting from jumps that come after a return instruction.

4. The message expression,

   ```
   cfg.computeDominators(cfg.basicBlocks.get(0), null);
   ```

   computes a dominator for each basic block. A *dominator* for a block is that closest predecessor through which all paths must pass to reach the target block. It is a useful place to which insert invariant code that is lifted out of a loop in optimization.

5. The message expression,

   ```
   cfg.tuplesToHir();
   ```

   converts the tuples representation to HIR, stored as a sequence of HIR instructions in the array list, `hir` for each block. As the tuples are scanned, their execution is simulated, using a stack to keep track of newly created values and instruction ids. The HIR is in SSA form, with (sometimes redundant) Phi function instructions.

6. The message expression,

   ```
   cfg.eliminateRedundantPhiFunctions();
   ```

eliminates unnecessary Phi functions and replaces them with their simpler values as discussed above.

The HIR is now ready for further analysis.

7.3.3 Optimizations

7.3.3.1 Overview

One property of the HIR is that it is amenable to the application of various optimizations. These include

- **constant folding**, where operations with constant operands may be folded into a single constant value;
- **inlining**, where the invocations of methods having small bodies of code are replaced by the bodies, saving the overhead of method calls;
- **common subexpression elimination** (CSE), where an expression that is equivalent to a previous expression may be replaced by the result computed in the first;
- **lifting of invariant expressions from loops**, where expressions based on variables that don't change in the loop iteration may be lifted out of the loop and so executed just once;
- **strength reduction**, where costly operations may be replaced by less costly ones;
- **null check elimination,** where a pointer is provably non-null and so needn't be checked;
- **array bounds check elimination**, where an array index is provably within bounds (for example, as shown by a previous equivalent check); and
- **control flow optimizations**, where control flow, e.g. jumps to jumps, may be simplified.

Some of these are easily implemented; others require additional infrastructure and analysis.

7.3.3.2 Constant Folding

Expressions having operands that are both constants, e.g.

```
3 + 4
```

or even variables whose values are known to be constants, e.g.

```
int i = 3;
int j = 4;
… i + j …
```

can be *folded*, that is replaced by their constant value.

For example, consider the Java method

```
static void foo1() {
    int i = 1;
    int j = 2;
    int k = i + j + 3;
}
```

and corresponding HIR code

```
  B0 succ: B1
  Locals:
    0: 0
    1: 1
    2: 2

  B1 [0, 10] dom: B0 pred: B0
  Locals:
    0: I3
    1: I4
    2: I7
  I3: 1
  I4: 2
  I5: I3 + I4
  I6: 3
  I7: I5 + I6
  8: return
```

The instruction `I3 + I4` at `I5`: can be replaced by the constant `3` and the `I5 + I6` at `I7:` can replaced by the constant `6`.

7.3.3.3 Inlining

The cost of calling a routine (for invoking a method) can be considerable. Besides the control flow management and stack frame management, register values must be saved on the stack. But in certain cases, the code of a callee's body can replace the call sequence in the caller's code, saving the overhead of a routine call. We call this *inlining*.

Inlining usually makes sense for methods whose only purpose is to access a field as in getters and setters.

For example, consider the following two methods, defined within a class Getter.

```
static int getA() {
    return Getter.a;
}

static void foo() {
    int i;
    i = getA();
}
```

The **a** in **Getter.a** refers to a static field declared in **Getter**. We can replace the **getA()** with the field selection directly:

```
static void foo() {
    int i;
    i = Getter.a;
}
```

Inlining makes sense only when the callee's code is relatively short. Also, when modeling a virtual method call, we must be able to determine the routine that should be called at compile-time. One also should be on the lookout for nested recursive calls; repeated inlining could go on indefinitely.

7.3.3.4 Common Subexpression Elimination (CSE)

Another optimization one may make is common subexpression elimination, where we identify expressions that are re-evaluated even if their operands are unchanged.

For example, consider the following method, unlikely as it may be.

```
void foo(int i) {
    int j = i * i * i;
    int k = i * i * i;
}
```

We can replace

```
int k = i * i * i;
```

in **foo()** with the more efficient,

```
int k = j;
```

To see how we might recognize common subexpressions, consider the HIR code for the original version of **foo()**.

```
  B0 succ: B1
  Locals: I0
  I0: LDLOC 0

  B1 [0, 12] dom: B0 pred: B0
  Locals: I0 I4 I6
  I3: I0 * I0
  I4: I3 * I0
  I5: I0 * I0
  I6: I5 * I0
  7: return
```

We sequence through the instructions, registering each in a hash table indexed by the instruction and its operand(s); the value stored is the instruction id.

At `I6:`, the reference to `I5` can replaced by `I3`. More importantly, any subsequent reference to `I6` could be replaced by `I4`.

Of course, we are not likely to see such a method with such obvious common subexpressions. But common subexpressions do arise in places one might not expect them. For example, consider the following C Language fragment.

```
for (i = 0; i < 1000; i++) {
    for (j = 0; j < 1000; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

The C compiler represents the matrices as linear byte sequences, one row of elements after another; this is called *row major form*[3]. If a, b, and c are integer matrices, and the base address of c is c’, then the byte memory address of c[i][j] is

c’ + i * 4 * 1000 + j * 4

The factor i is there because it’s the (counting from zero) ith row; the factor 1000 is there because there are 1000 elements in a row; and the factor 4 is there (twice) because there are 4 bytes in an integer word. Likewise, the addresses of a[i][j] and b[i][j] are

a’ + i * 4 * 1000 + j * 4

and

b’ + i * 4 * 1000 + j * 4

respectively. So, eliminating the common offsets, i * 4 * 1000 + j * 4 can save us a lot of computation, particularly since the inner loop is executed a million times!

But in Java, matrices are not laid out this way. Rather, in the expression

a[i][j]

the subexpression a[i] yields an integer array object, call it ai, from which we can index the jth element,

ai[j]

[3] In FORTRAN, matrices are stored as a sequence of columns, which is in *column major form.*

So, we cannot expect the same savings. On the other hand, the expression a[i] never changes in the inner loop; only the j is being incremented. We say that, within the inner loop, the expression a[i] is *invariant*; that is, it is a (inner) *loop invariant expression*. We deal with these in the next section.

And as we shall see below, there is still much to be gained from common sub-expression elimination when it comes to dealing with Java arrays.

7.3.3.5 Lifting Loop Invariant Code

Loop invariant expressions can be lifted out of the loop and computed in the predecessor block to the loop header.

For example, consider the *j*-- code for summing the two matrices from the previous section[4].

```
int i = 0;
while (i <= 999) {
    int j = 0;
    while (j <= 999) {
        c[i][j] = a[i][j] + b[i][j];
        j = j + 1;;
    }
    i = i + 1;
}
```

This can be rewritten as

```
int i = 0;
while (i <= 999) {
    int[] ai = a[i];
    int[] bi = b[i];
    int[] ci = c[i];
    int j = 0;
    while (j <= 999)
    {
        ci[j] = ai[j] + bi[j];
        j = j + 1;;
    }
    i = i + 1;
}
```

Thus a[i], b[i], and c[i] need be evaluated just 1000 times instead of 1000000 times.

This optimization is more difficult than the others. Basically, the expression

[4] The for-loop is not part of the *j*-- language so we use the equivalent while-loop notation. If you have implemented for-loops in your compiler by rewriting them as while-loops then you should produce similar HIR.

```
a[i]
```

is invariant if

1. **`i`** is constant,
2. or, all the definitions of **`i`** that reach the expression are outside the loop,
3. or only one definition of reaches the expression, and the definition is loop-invariant.

But that the HIR conforms to SSA makes it possible. Any code that is lifted must be lifted to an additional block, which is inserted before the loop header.

There is still a potential for common sub-expression elimination, even after invariants are lifted. For example the vector offsets are a product of the index and the number of bytes in each element. For example, the j * 4 must be computed for each of a, b and c; this can be computed just once and re-used across a, b and c.

But this may leave the

… j * 4…

to be computed each time around the inner loop; that is, the multiplication will have to be done a million times. We address this in the next section.

7.3.3.6 Strength Reduction

Strength reduction involves replacing one operation by another less costly operation. The savings of doing this are slight, and so only pay off when the operations appear in loops.

For example, consider the following *j*-- fragment.

```
int i = 0;
while (i <= 999) {
    int j = 0;
    while (j <= 999) {
        …(j * 4)…
        j = j + 1;;
    }
    i = i + 1;
}
```

For example, the j * 4 might be the byte offset to the j^th element of the i^th row. Because it appears in the inner loop, and because the offset must be computed for each of a, b and c in the matrix addition, it could be executed 3,000,000 times![5]

[5] Common subexpression elimination will reduce this to *just* one million.

But by fiddling with the increment of j, adding 4 instead of 1, each time around the loop, then we can use j as the byte offset.

```
int i = 0;
while (i <= 999) {
    int j = 0;
    while (j <= 3996) {
        …j…
        j = j + 4;;
    }
    i = i + 1;
}
```

Notice that in this case, we have replaced one addition by another addition, and we have eliminated the multiplication altogether![6] The same technique may be applied to the outer loop variable, **`i`**.

7.3.3.7 Null Check Elimination

Every time we send a message to an object, or access a field of an object, we must insure that the object is not the special null object. For example, in

```
...a.f...
```

we will want to make sure that a is non-null before computing the offset to the field f. But we may know that a is non-null; for example, we may have already checked it as in the following.

```
...a.f...
...a.g...
...a.h...
```

Once we've done the null-check for

```
...a.f...
```

there is no reason to do it again for either **`a.g or a.h.`**

If our optimizer is really clever[7], we can take advantage of fragments such as

```
if (a != null) {
    …a.f…
}
```

[6] Of course, as if have seen, invariant code lifting is applicable here also.

[7] That is, if it is cleverly programmed.

and even not have do the null check for **`a.f`**.

7.3.3.8 Array Bounds Check Elimination

Array bound checks may be similarly treated.

When indexing an array, we must check that the index is within bounds. For example, in our example of matrix multiplication, in the assignment

```
c[i][j] = a[i][j] + b[i][j];
```

The i and j must be tested to make sure each is greater than or equal than zero, and less than 1000. And this must be done for each of c, a and b. But if we know that a, b, and c are all of like dimensions, then once the check is done for a, it need not be repeated for b and c. Given that the inner loop, in our matrix multiplication example in section 7.3.3.5, is executed one million times, we have saved two million checks!

Again a clever compiler can analyze the context in which the statement,

```
c[i][j] = a[i][j] + b[i][j];
```

is executed (for example, within a nested for-loop) and show that the indices are never out of bounds.

7.3.3.9 Control Flow Optimization

Both the basic control flow graph construction, as well as surgery performed on the graph in the optimization process, may produce instances of jumps to jumps or jumps following return instructions. Of course, these may be simplified or removed altogether.

7.3.3.10 Summary

At a panel discussion on compiler optimization, Kathleen Knobe (Knobe, 1990) observed that, rather than to put all of one's efforts into just one or two optimizations, it is best to attempt all of them, even if they cannot be done perfectly.

We have not implemented any of these optimizations, but leave them as exercises.

Of course, one of the greatest optimizations on can make is to replace references to local variables by references to registers by allocating registers for holding local variable values. We discuss register allocation in Chapter 8.

7.3.4 The Low Level Intermediate Representation (LIR)

The HIR lends itself to optimization but is not necessarily suitable for register allocation. For this reason we translate it into a low-level intermediate representation (LIR) where

- Phi functions are removed from the code and replaced by explicit moves, and
- instruction operands are expressed as explicit virtual registers.

For example, the LIR for **`Factorial.computeIter()`** is given below; notice that we retain the control flow graph from the HIR. Notice also that we have allocated virtual registers to the computation. One may have an arbitrary number of virtual registers. In register allocation, we shall map these virtual registers to actual physical registers on the MIPS architecture. Since there might be many more virtual registers than physical registers, the mapping may require that some physical registers be spilled to stack locations and reloaded when the spilled values are needed again. Notice that each instruction is addressed at an even location; this leaves room for spills and loads (at the intervening odd locations).

```
computeIter (I)I

  B0

  B1
  0: LDC [1] [V32|I]
  2: MOVE $a0 [V33|I]
  4: MOVE [V32|I] [V34|I]

  B2
  6: LDC [0] [V35|I]
  8: BRANCH [LE] [V33|I] [V35|I] B4

  B3
  10: LDC [-1] [V36|I]
  12: ADD [V33|I] [V36|I] [V37|I]
  14: MUL [V34|I] [V33|I] [V38|I]
  16: MOVE [V38|I] [V34|I]
  18: MOVE [V37|I] [V33|I]
  20: BRANCH B2

  B4
  22: MOVE [V34|I] $v0
  24: RETURN $v0
```

In this case, seven virtual registers **`V32 - V38`** are allocated to the LIR computation. Two physical registers, **`$a0`** for the argument n and **`$v0`** for the return value, are referred to by their symbolic names; that we use these physical registers indicates that SPIM expects values to be stored in them. Notice that LIR instructions are read from left to right. For example, the load constant instruction

```
  0: LDC [1] [V32|I]
```

in block B1, loads the constant integer 1 into the integer virtual register **`V32`**; the **`|I`** notation types the virtual register as an integer. Notice that we start enumerating virtual registers beginning at 32; the numbers 0 through 31 are reserved for enumerating physical registers. The next instruction,

```
  2: MOVE $a0 [V33|I]
```

copies the value from the integer physical register **`$a0`** into the integer virtual register **`V33`**. The add instruction in block B3,

```
  12: ADD [V33|I] [V36|I] [V37|I]
```

adds the integer values from registers **`V33`** and **`V36`** and puts the sum in **`V37.`**

The process of translating HIR to LIR is relatively straightforward and is a two-step process.

1. First, the **`NEmitter`** constructor invokes the **`NControlFlowGraph`** method **`hirToLir()`** on the control flow graph:

   ```
   cfg.hirToLir();
   ```

   The method **`hitToLir()`** iterates through the array of HIR instructions for the control flow graph translating each to an LIR instruction, relying on a method **`toLir()`**, which is defined for each HIR instruction.

2. Then **`NEmitter`** invokes the **`NControlFlowGraph`** method **`resolvePhiFunctions()`** on the control flow graph:

   ```
   cfg.resolvePhiFunctions();
   ```

   This method resolves Phi function instructions, replacing them by move instructions near the end of the predecessor blocks. For example, the Phi function from Figure 7.10 and now repeated in Figure 7.12 (a) resolves to the moves in Figure 7.12 (b). One must make sure to place any move instructions before any branches in the predecessor blocks.

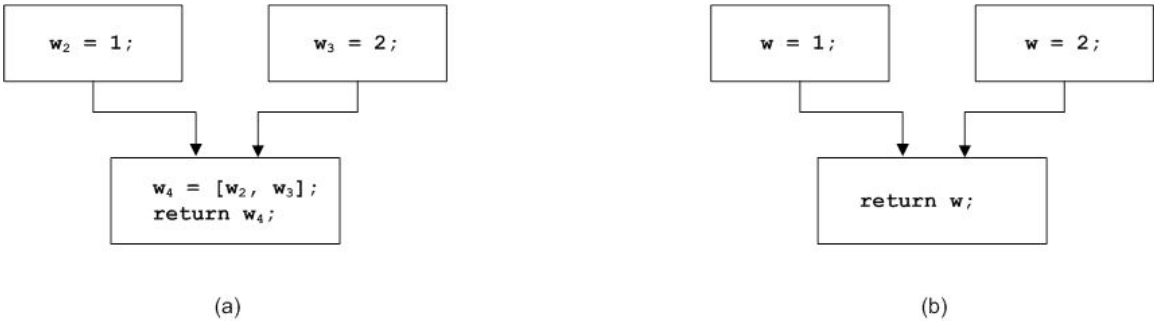


Figure 7.12 Resolving Phi Functions

7.3.6 A Simple Runtime Environment

7.3.5.1 Introduction

An actual run-time environment supporting code produced for Java would require:

1. A naming convention.
2. A run-time stack.
3. A representation for arrays and objects
4. A heap, that is an area of memory from which arrays and objects may be dynamically allocated. The heap should offer some sort of garbage collection, making objects that are no longer in use available for (re-)allocation.
5. A run-time library of code that supports the Java API.

To do all of this here would be beyond the scope of this text. Given that our goal is to have a taste of native code generation, including register allocation, we do much less here. We implement just enough of the run-time environment for supporting our running `Factorial` example (and a few more examples, which are included in our code tree).

7.3.5.2 A Naming Convention

We use a simple naming mechanism that takes account of just classes and their members. Methods are represented in SPIM by routines with assembly language names of the form **`<class>.<name>`** where **`<name>`** names a method in a class **`<class>`**. For example, the **`computeIter()`** method in class `Factorial` would have the entry point

```
Factorial.computeIter:
```

Static fields are similarly named; for example, a static integer field **`staticIntField`** in class **`Factorial`** would translate to SPIM assembly language as,

```
Factorial.staticIntField:     .word       0
```

String literals in the data segment have labels that suggest what they label, for example **`Constant..String1:`**.

7.3.5.3 A Runtime Stack

Our run-time stack conforms to the run-time convention described for SPIM in (Larus, 2009) and our section 7.2.4. Each time a method is invoked, a new stack frame of the type illustrated in Figure 7.5 is pushed onto the stack. Upon return from the method, the same frame is popped off from the stack.

When the caller wants to invoke a callee, it does the following:

1. Argument passing. The first four arguments are passed in registers $a0 - $a3. Any additional arguments are pushed onto the stack and so will appear at the beginning of the callee's stack frame.

2. Caller-saved registers. Registers $a0 - $a3 and $t0 - $t9 can be used by the callee without its having to save their values, so it is up to the caller to save any registers whose values it expects to use after the call, within its own stack frame before executing the call.
3. Executing the call. After arguments have passed and caller registers have been saved, we execute the `jal` instruction, which jumps to the callee's first instruction and saves the return address in register $ra.

Once a routine (the callee) has been invoked, it must first do the following:

1. Push stack frame. Allocate memory for its new stack frame by subtracting the frame's size from the stack pointer $sp. (Recall, the stack grows downward in memory.)
2. Callee-saved registers. The callee must save any of $s0 - $s7, $fp and $ra before altering any of them because the caller expects their values will be preserved. We must always save $fp. $ra must be saved only if the callee invokes another routine.
3. The frame pointer. The new $fp = $sp + stack frame size - 4.

The stack frame for an invoked (callee) routine is illustrated in Figure 7.13.

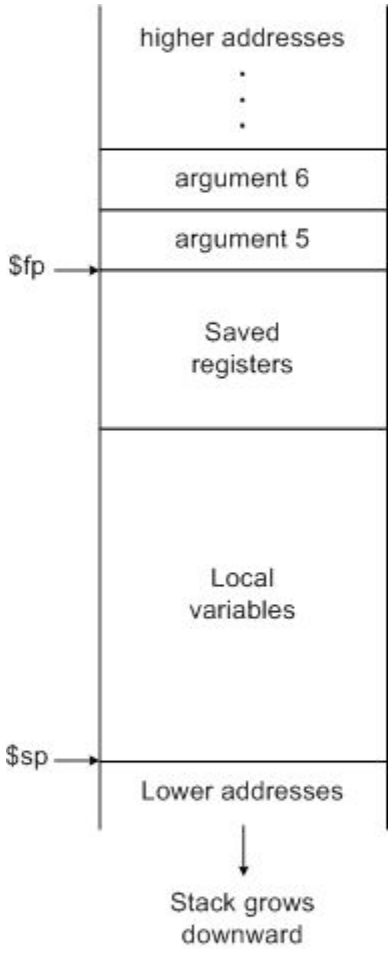


Figure 7.13 A Stack Frame

Once it has done its work, the callee returns control to the caller by doing the following:

1. Return value. If the callee returns a value, it must put that value in register $v0.
2. Callee-saved registers. All callee-saved registers that were saved at the start of the routine are restored.
3. Stack frame. The stack frame is popped from the stack, restoring register $fp from its saved location in the frame, and adding the frame size to register $sp.
4. Return. Execute the return instruction, which causes a jump to the address in register $ra.

7.3.5.4 A Layout for Objects

Although, we mean only to support static methods, objects do leak into our implementation. For example, method **`main()`** takes as its sole argument an array of strings.

Although, we don't really support objects, doing so needn't be difficult. (Corliss and Lewis, 2007) propose layouts, which we use here. For example, an arbitrary object might be organized as in Figure 7.14.

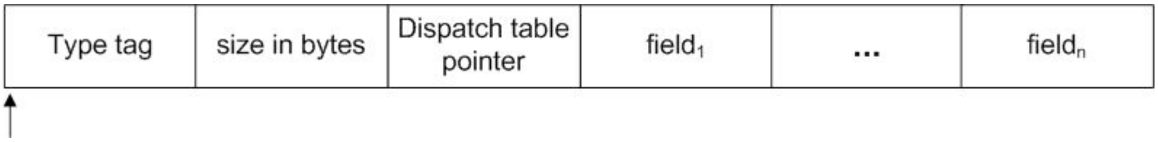


Figure 7.14 Layout for an Object

The *type tag* identifies the type (that is class) of the object. The *size in bytes* is the number of bytes that must be allocated to represent the object; this number is rounded up to a multiple of 4 so that objects begin on word boundaries. The dispatch table pointer is the address of a dispatch table, which contains the address of each method for the object's class.

The dispatch table may be allocated in the data segment and keeps track of both inherited and defined methods. When code is generated for the class, entries in the table are first copied from the superclass. Then entries for newly defined method are added; overriding method entries replace the overridden entries in the table and other method entries are simply added at the end. Entries are added in the order they are declared in the class declaration. In this way, the dispatch table for a class keeps track of the proper (inherited, overriding, or newly defined) methods that are applicable to each class. A dispatch table for class <class> may be labeled (and addressed) in the data segment using the name

```
<class>..Dispatch:
```

The double period distinguishes this name from a method named **`Dispatch()`**.

The order of fields in an object is also important because a class inherits fields from a superclass. In constructing the format of an object, first any inherited fields are allocated. Then any newly defined fields are allocated. Java semantics require that space is allocated to all newly defined fields, even those with the same names as inherited fields; the "overridden" fields can are always available through casting an object to its superclass.

For each class, we also maintain a class template in the data segment, which among other things may include a typical copy of an object of that class that can be copied to the heap during allocation. We name this,

```
<class>..Template:
```

For example, consider the following *j--* code, which defines two classes: **`Foo`** and **`Bar`**.

```
public class Foo {
    int field1 = 1;
    int field2 = 2;

    int f() {
        return field1 + field2;
    }

    int foo() {
        return field1;
    }
}

class Bar extends Foo {
    int field3 = 3;
    int field1 = 4;

    int f() {
        return field1 + field2 + field3;
    }

    int bar() {
        return field1;
    }
}
```

Consider an object of class **`Foo`**, illustrated in Figure 7.15.

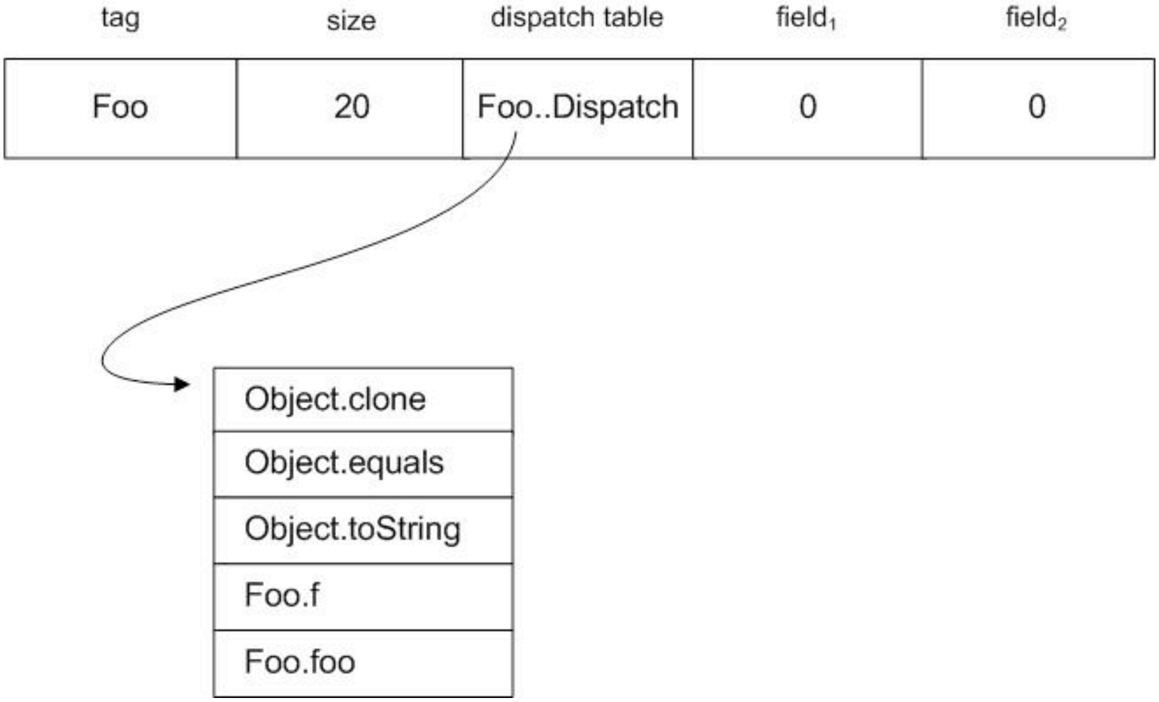


Figure 7.15 Layout and Dispatch Table for Foo

Assuming the type tag for **Foo** is 4[8],

```
Foo..Tag:   .word 4
```

The SPIM code for a template for **Foo** might look like

```
Foo..Template:
      .word Foo..Tag
      .word 20
      .word Foo..Dispatch
      .word 0
      .word 0
```

The fields have initial values of zero. It is the responsibility of the **<init>** routine for **Foo** to initialize them to 1 and 2 respectively.

The construction of the dispatch table for **Foo** follows these steps:

1. First the methods for **Foo**'s superclass (**Object**) are copied into the table. (Here we assume we've given **Object** just the three methods **clone()**, **equals()** and **toString()**).
2. Then we add the methods defined in **Foo** to the dispatch table: **f()** and **foo()**.

The SPIM code for this dispatch table might look something like the following.

```
Foo..Dispatch:
      .word 5 # the number of entries in our table
      .word Object.clone
      .word Object.equals
      .word Object.toString
      .word Foo.f
      .word Foo.foo
```

The layout and dispatch table for class **Bar**, which extends class **Foo**, is illustrated in Figure 7.16.

[8] This assumes, 1 represents a string, 2 represents an array and 3 represents an **Object**.

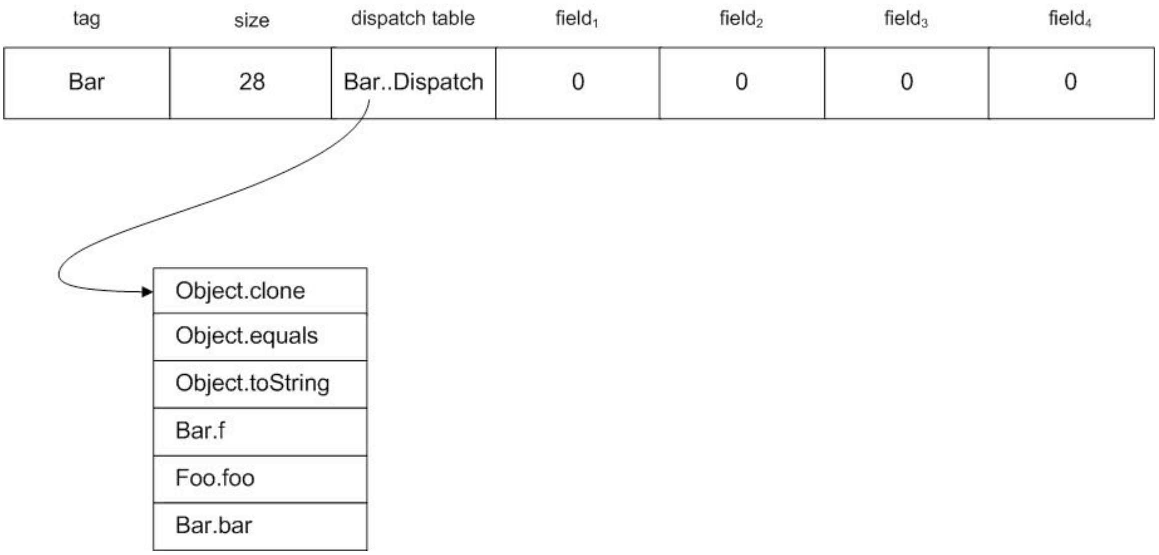


Figure 7.16 Layout and Dispatch Table for Bar

Assuming the type tag for **`Bar`** is 5,

```
Bar..Tag:   .word 5
```

The SPIM code for a template for **`Foo`** might look like

```
Bar..Template:
      .word Bar..Tag
      .word 28
      .word Bar..Dispatch
      .word 0
      .word 0
      .word 0
      .word 0
```

Again, the fields have initial values of zero. It is the responsibility of the **`<init>`** routine for **`Foo`** and **`Bar`** to initialize them to 1,2, 3, and 4 respectively.

The construction of the dispatch table for Foo follows these steps:

1. First the methods for **`Bar`**'s superclass (**`Foo`**) are copied into the table.
2. Then we add the methods defined in **`Bar`** to the dispatch table. The method **`Bar.f()`** overrides, and so replaces **`Foo.f()`** in the table; the new **`Bar:bar()`** is simply added to the table.

The SPIM code for this dispatch table might look something like the following.

```
Bar..Dispatch:
      .word 6 # the number of entries in our table
      .word Object.clone
      .word Object.equals
      .word Object.toString
      .word Bar.f
      .word Foo.foo
      .word Bar.bar
```

Arrays are a special kind of object; they are dynamically allocated on the heap but lie outside of the Object hierarchy. A possible layout for an array is illustrated in Figure 7.17.

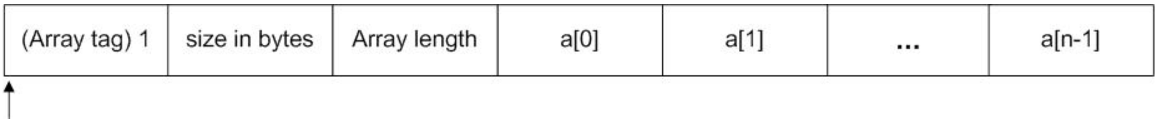


Figure 7.17 Layout for an Array

Arrays have a type tag of 1. The *size in bytes* is the size of the array object in bytes, that is 12 + n * bytes where n is the number of elements in the array and bytes is the number of bytes in each element. The array length is the number of elements in the array; any implementation of the JVM **`arraylength`** instruction must compute this value. The array elements follow. The size of the array object should be rounded up to a multiple of 4 to make sure the next object is word aligned. Notice that an array indexing expression **`a[i]`** would translate to the heap address a+12+i*bytes where a is the address of the array object on the heap and bytes is the number of bytes in a single element.

Strings are also special objects; while they are part of the Object hierarchy they are final and so they may not be sub-classed. This means we can do without a dispatch table pointer in string objects; we can maintain one dispatch table for all string operations:

```
String..Dispatch:
      .word m # m is the number of entries in the table
      .word String.clone
      .word String.equals
      .word String.toString
      … addresses of methods for Strings…
```

Which **`String`** methods we implement are arbitrary and are left as an exercise.

A possible layout for a string is illustrated in Figure 7.18.

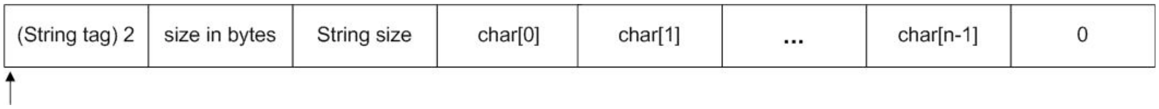


Figure 7.18 Layout for a String

The type tag for a string is 2. Its size is bytes is the number of bytes that must be allocated to represent the string object; it is 12+size+1 rounded up to a multiple of 4, where size is the number of characters in the string. String size is the number of characters in the string. The sequence of characters is terminated with an extra null character, with Unicode representation 0; the null character is necessary so that our strings are compatible with SPIM strings.

Constant strings may be allocated in the data segment, which is to say, we may generate constant strings in the data segment; strings in the data segment will then look exactly like those on the heap (the dynamic segment). For example, the constant **‟Hello World!″** might be represented at the label **Constant..String1:**

```
Constant..String1:
      .word 2     # tag 2 indicates a string.
      .word 28    # size of object in bytes.
      .word 12    # string length (not including null terminator).
      .asciiz  ‟Hello World!″ # string terminated by null character 0.
      .align 2    # next object is on a word boundary.
```

The reader will have noticed that all objects on the heap share three words (twelve bytes) of information at the start of each object. This complicates the addressing of components in the object. For example, the first element of an array a is at address a+12; the first character of a string s is at s+12; and the first field of an object o is at address o+12. That 12 must be added to an object's address each time we want an array's element, a string's character or an object's field can be expensive. We can do away with this expense by having our pointer, which we use to reference the object, always point into the object 12 bytes as illustrated in Figure 7.19.

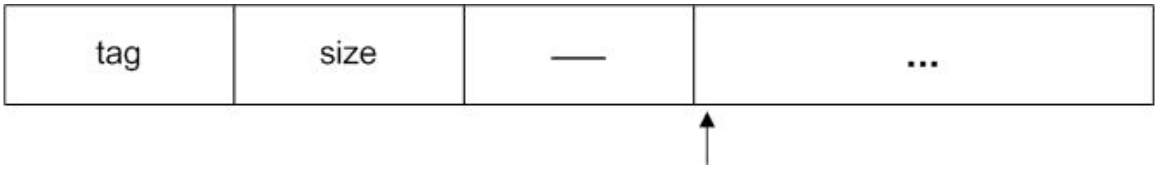


Figure 7.19 An Alternative Addressing Scheme for Objects on the Heap

7.3.5.5 Dynamic Allocation

The logic for allocating free space on the heap is pretty simple in our (oversimplified) model.

```
obj = heapPointer;
heapPointer += <object size>;
if (heapPointer >= stackPointer) goto freakOut;
<copy object template to obj>;
```

When allocating a new object, we simply increment a free pointer by the appropriate size and if we've run out of space (that is, when the heap pointer meets the stack pointer), we freak out. A more robust heap management system might perform garbage collection.

Once we have allocated the space for the object, we copy its template onto the heap at the object's location. Proper initialization is left to the SPIM routines that model the constructors.

7.3.5.6 The SPIM Class for I/O

SPIM provides a set of built-in system calls for performing simple i/o tasks. Our run time environment includes a class SPIM, which is a wrapper that gives us access to these calls as a set of static methods. The wrapper is defined as follows.

```
package spim;

/**
 * This is a Java wrapper class for the SPIM runtime object SPIM.s.
 * Any j-- program that's compiled for the SPIM target must import
 * this class for (file and console) IO operations. Note that the
 * functions have no implementations here which means that if the
 * programs using this class are compiled using j--, they will
 * compile fine but won't function as desired when run against
 * the JVM. Such programs must be compiled using the j-- compiler
 * for the SPIM target and must be run against the SPIM simulator.
 */

public class SPIM
{
    /** Wrapper for SPIM.printInt(). */

    public static void printInt(int value) { }

    /** Wrapper for SPIM.printFloat(). */

    public static void printFloat(float value) { }

    /** Wrapper for SPIM.printDouble(). */

    public static void printDouble(double value) { }

    /** Wrapper for SPIM.printString(). */

    public static void printString(String value) { }

    /** Wrapper for SPIM.printChar(). */

    public static void printChar(char value) { }

    /** Wrapper for SPIM.readInt(). */

    public static int readInt() { return 0; }

    /** Wrapper for SPIM.readFloat(). */

    public static float readFloat() { return 0; }

    /** Wrapper for SPIM.readDouble(). */
    public static double readDouble() { return 0; }

    /** Wrapper for SPIM.readString(). */

    public static String readString(int length) { return null; }

    /** Wrapper for SPIM.readChar(). */

    public static char readChar() { return ' '; }

    /** Wrapper for SPIM.open(). */

    public static int open(String filename, int flags, int mode)
    { return 0; }

    /** Wrapper for SPIM.read(). */

    public static String read(int fd, int length) { return null; }

    /** Wrapper for SPIM.write(). */

    public static int write(int fd, String buffer, int length)
    { return 0; }

    /** Wrapper for SPIM.close(). */

    public static void close(int fd) { }

    /** Wrapper for SPIM.exit(). */

    public static void exit() { }

    /** Wrapper for SPIM.exit2(). */

    public static void exit2(int status) { }
}
```

Because the **SPIM** class is defined in the package **spim**, that package name is part of the label for the entry point to each **SPIM** method, e.g.,

```
spim.SPIM.printInt:
```

7.3.6 Generating SPIM Code

After LIR with virtual registers has been generated, we invoke register allocation for mapping the virtual registers to physical registers. In this chapter we use a *naïve* register allocation scheme where physical registers are arbitrarily assigned to virtual registers (see section 8.2). Register allocation is discussed in greater detail in Chapter 8.

Once virtual registers have been mapped to physical registers, translating LIR to SPIM code is pretty straightforward.

1. We iterate through the list of methods for each class; for each method, we do the following:
    a. We generate a label for the method's entry point, e.g.,

    ```
    Factorial.computeIter:
    ```

    b. We generate code to push a new frame onto the run-time stack (remember this stack grows downward in memory) and then code to save all our registers. In this compiler, we treat all of SPIM's general purpose registers \$t0 - \$t9 and \$s0 - \$s7 as callee-saved registers; that is, we make it the responsibility of the invoked method to save any registers it uses. As we shall see in Chapter 8, some register allocation schemes do otherwise; that is, the caller saves just those registers that contain meaningful values when call is encountered.

    For **computeIter()**, the LIR uses seven general purpose registers, so we generate the following:

    ```
        subu    $sp,$sp,36   # Stack frame is 36 bytes long
        sw      $ra,32($sp)  # Save return address
        sw      $fp,28($sp)  # Save frame pointer
        sw      $t0,24($sp)  # Save register $t0
        sw      $t1,20($sp)  # Save register $t1
        sw      $t2,16($sp)  # Save register $t2
        sw      $t3,12($sp)  # Save register $t3
        sw      $t4,8($sp)   # Save register $t4
        sw      $t5,4($sp)   # Save register $t5
        sw      $t6,0($sp)   # Save register $t6
        addiu   $fp,$sp,32   # Save frame pointer
    ```

    c. Because all branches in the code are expressed as branches to basic blocks, a unique label for each basic block is generated into the code; e.g.,

    ```
    Factorial.computeIter.2:
    ```

    d. We then iterate through the LIR instructions for the block, invoking a method **toSpim()**, which is defined for each LIR instruction; there is a one-to-one translation from each LIR instruction to its SPIM equivalent. For example, the (labeled) code for block B2 is

    ```
    Factorial.computeIter.2:
        li $t3,0
        ble $t1,$t3,Factorial.computeIter.4
        j Factorial.computeIter.3
    ```

e. Any string literals that are encountered in the instructions are put into a list, together with appropriate labels. These will be emitted into a data segment at the end of the method (see step 2 below).

f. At the end of the method, we generate code to restore those registers that had been saved at the start. This code also does a jump to that instruction following the call in the calling code, which had been stored in the $ra register (ra is a mnemonic for return address) at the call. This code is labeled so that, once a return value has been placed in $v0, execution may branch to it to affect the return. For example, the register-restoring code for **`computeIter()`** is

```
Factorial.computeIter.restore:
    lw      $ra,32($sp)  # Restore return address
    lw      $fp,28($sp)  # Restore frame pointer
    lw      $t0,24($sp)  # Restore register $t0
    lw      $t1,20($sp)  # Restore register $t1
    lw      $t2,16($sp)  # Restore register $t2
    lw      $t3,12($sp)  # Restore register $t3
    lw      $t4,8($sp)   # Restore register $t4
    lw      $t5,4($sp)   # Restore register $t5
    lw      $t6,0($sp)   # Restore register $t6
    addiu   $sp,$sp,36   # Pop stack
    jr      $ra          # Return to caller
```

2. After we have generated the text portion (the program instructions) for the method (notice the **`.text`** directive at the start of code for each method), we then populate a data area, beginning with the .data directive, from the list of string literals constructed in step 1.e. Any other literals that you may wish to implement would be handled in the same way. Notice that **`computerIter()`** has no string literals; its data segment is empty.

3. Once all of the program code has been generated, we then copy out the SPIM code for implementing the SPIM class. This is where any further system code, which you may wish to implement, should go.

For example, the code for **`Factorial.computeIter()`** is as follows.

```
.text

Factorial.computeIter:
    subu    $sp,$sp,36   # Stack frame is 36 bytes long
    sw      $ra,32($sp)  # Save return address
    sw      $fp,28($sp)  # Save frame pointer
    sw      $t0,24($sp)  # Save register $t0
    sw      $t1,20($sp)  # Save register $t1
    sw      $t2,16($sp)  # Save register $t2
    sw      $t3,12($sp)  # Save register $t3
    sw      $t4,8($sp)   # Save register $t4
    sw      $t5,4($sp)   # Save register $t5
    sw      $t6,0($sp)   # Save register $t6
    addiu   $fp,$sp,32   # Save frame pointer

Factorial.computeIter.0:

Factorial.computeIter.1:
    li $t0,1
    move $t1,$a0
    move $t2,$t0

Factorial.computeIter.2:
    li $t3,0
    ble $t1,$t3,Factorial.computeIter.4
    j Factorial.computeIter.3

Factorial.computeIter.3:
    li $t4,-1
    add $t5,$t1,$t4
    mul $t6,$t2,$t1
    move $t2,$t6
    move $t1,$t5
    j Factorial.computeIter.2

Factorial.computeIter.4:
    move $v0,$t2
    j Factorial.computeIter.restore

Factorial.computeIter.restore:
    lw      $ra,32($sp)  # Restore return address
    lw      $fp,28($sp)  # Restore frame pointer
    lw      $t0,24($sp)  # Restore register $t0
    lw      $t1,20($sp)  # Restore register $t1
    lw      $t2,16($sp)  # Restore register $t2
    lw      $t3,12($sp)  # Restore register $t3
    lw      $t4,8($sp)   # Restore register $t4
    lw      $t5,4($sp)   # Restore register $t5
    lw      $t6,0($sp)   # Restore register $t6
    addiu   $sp,$sp,36   # Pop stack
    jr      $ra          # Return to caller
```

This code is emitted, together with the code for other Factorial methods and the SPIM class to a file ending in **`.s`**; in this case, the file would be named **`Factorial.s`**. This file can be loaded into any SPIM interpreter and executed.

You will notice, there is lots of moving values around among registers. In Chapter 8, we address how that might be minimized.

7.3.7 Peephole Optimization of the SPIM Code

You might have noticed in the SPIM code generated in the previous section, that some jumps were superfluous. For example, at the end of the SPIM code above, the jump at the end of block B4 simply jumps to the very next instruction! There might also be jumps to jumps. Such code can often be simplified. Jumps to the next instruction can be removed, and jumps to jumps can be simplified, (sometimes) removing the intervening jump.

We call such simplification *peephole optimization* because we need consider just a few instructions at a time as we pass over the program; we need not do any global data flow analysis.

To do peephole optimization on the SPIM code, it would be easier, if we were to keep a list (perhaps an array list) of SPIM labels and instructions that we emit, representing the instructions in some easily analyzable way. Then, once the simplifications were done, we could spit out the SPIM instructions to a file.

Further Reading

James Larus's SPIM simulator (Larus, 2000-2010) may be freely obtained on the WWW at http://sourceforge.net/projects/spimsimulator/files/. QtSpim is the interpreter of choice these days and is easily installed on Windows, Linux and Mac OSX. Larus maintains a resource page at http://pages.cs.wisc.edu/~larus/spim.html. Finally, an excellent introduction to SPIM and the MIPS may be found at (Larus, 2009).

Our JVM to SPIM translator is based on that described in Christian Wimmer's Master's thesis (Wimmer, 2004); this is a good overview of the Oracle HotSpot compiler. Another report on a register allocator for the HotSpot compiler is (Mössenböck, 2000).

Our proposed layout of objects is based on that of (Corliss and Lewis, 2007).

(Wilson, 1994) is a good introduction to classical garbage collection techniques. (Jones, 1996 and 1999) also presents many memory management strategies.

Exercises

As we have stated in the narrative, we implement enough of the JVM to SPIM translator to implement a small subset of the JVM. The following exercises ask one to expand on this.

7.1 Implement all of the relational and equality JVM instructions in the HIR, the LIR and SPIM. (Notice that some of this has been done.) Test these.

7.2 Assuming you have implemented `/` and `mod` in your *j*-- compiler, implement them in the HIR, LIR and SPIM. Test these.

7.3 Implement the bitwise JVM instructions in the HIR, LIR and SPIM. Test these.

7.4 In many places in the HIR, we refer to the instruction 0, which means 0 is an operand to some computation. In generating the LIR, that constant 0 is loaded into some virtual register. But SPIM has the register \$0, which always contains the constant 0 and so may replace the virtual register in the LIR code. Modify the translation of HIR to LIR to take advantage of this special case.

7.5 The JVM instructions **`getstatic`** and **`putstatic`** are implemented in the HIR and LIR but not in SPIM. Implement these instructions in SPIM and test them.

7.6 Implement the **`Object`** type, supporting the methods **`clone()`**, **`equals()`** and **`toString()`**.

7.7 Implement the **`String`** type, which is a subclass of **`Object`** and implements the methods **`charAt()`**, **`concat()`**, **`length()`**, and **`substring()`** with two arguments.

7.8 In section 7.3.5.4, we come up with a naming convention for the SPIM routines for methods that doesn't deal with overloaded method names, that is methods having the same name but different signatures. Propose an extension to our convention that deals with overloaded methods.

7.9 In Figure 7.19, we suggest an alternative addressing scheme where a pointer to an object actually points 12 bytes into the object, making the addressing of components (fields, array elements or string characters) simpler. Implement this scheme, changing both the runtime code and the code generated for addressing components.

7.10 Implement the JVM instructions **`invokevirtual`** and **`invokespecial`**. Test them.

7.11 Implement arrays in the HIR (**`iaload`** and **`iastore`** have been implemented in the HIR), the LIR and SPIM. Implement the **`arraylength`** instruction. Test these.

7.12 Implement the instance field operations **`getfield`** and **`putfield`** in the HIR, the LIR and SPIM. Test these.

7.13 Implement the remaining JVM load and store instructions and test them.

7.14 In the current implementation, all general-purpose registers are treated as callee-saved registers; they are saved by the method being invoked. Modify the compiler so that it instead treats all general purpose registers as caller-saved, generating code to save registers before a call to another method, and restoring those same registers after the call.

The following exercises implement various optimizations, which are discussed in section 7.3.3. The code already defines a method, **`optimize()`**, in the class

`NControlFlowGraph`. That is a good place to put calls to any methods that perform optimizations on the HIR.

7.15 Implement *inlining* in the optimizations phase. Use an arbitrary instruction count in deciding what methods to inline. Be careful in dealing with virtual methods.

7.16 Implement *constant folding* in the optimizations phase.

7.17 Implement *common sub-expression elimination* in the optimization phase.

7.18 (Difficult) Implement the *lifting of invariant code* in the optimization phase.

7.19 Design and implement a strategy for performing various forms of *strength reduction* in the optimization phase.

7.20 Implement *redundant array bounds check elimination* in the optimization phase.

7.21 Implement *redundant null check elimination* in the optimization phase.

7.22 Implement a simple version of peephole optimizations, simplifying some of the branches discussed in section 7.3.7.

7.23 (Involved[9]) Implement the JVM instructions **`lookupswitch`** and **`lookuptable`** in the HIR, the LIR and SPIM. Test these. Notice that these flow-of-control instructions introduce multiple predecessors and multiple successors to basic blocks in the flow graph.

7.24 (Involved) Introduce the long types of constants, variables and operations into the HIR, the LIR and SPIM. This assumes you have already added them to *j--*. Notice that this will complicate all portions of your JVM to SPIM translation, including register allocation since longs require two registers and complicate their operations in SPIM.

7.25 (Involved) Introduce the **`float`** and /or **`double`** types of constants, variables and operations into the HIR, the LIR and SPIM. This assumes you have already added them to *j--*. Notice that this will complicate all portions of your JVM to SPIM translation and will require using the special floating-point processor and registers, which are fully documented in (Larus, 2009).

7.26 (Involved) Implement exception handling in the HIR, the LIR and SPIM. This assumes you have already added exception handling to *j--*. Notice that this will complicate all portions of your JVM to SPIM translation and will require using

[9] Meaning not necessarily difficult but involving some work.

the special exception handling instructions, which are fully documented in (Larus, 2009).

7.27 (Involved) Read up on and implement a (relatively) simple copy form of garbage collection. See the section on Further Reading for starting points to learning about garbage collection.