Translating JVM Code to MIPS Code

Outline

Introduction

Introduction

Compilation is not necessarily done after the class file is constructed

Introduction

Compilation is not necessarily done after the class file is constructed

At “execution”, the class is loaded into the JVM and then interpreted

Introduction

Compilation is not necessarily done after the class file is constructed

At “execution”, the class 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 — code that can be directly executed by the underlying computer

Introduction

Compilation is not necessarily done after the class file is constructed

At “execution”, the class 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 — code that can be directly executed by the underlying computer

So at run time, control shifts back and forth between JVM code and native code

Introduction

Compilation is not necessarily done after the class file is constructed

At "execution", the class 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 — code that can be directly executed by the underlying computer

So at run time, control shifts back and forth between JVM code and native code

The native code runs much faster than does the interpreted JVM code

Introduction

Compilation is not necessarily done after the class file is constructed

At “execution”, the class 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 — code that can be directly executed by the underlying computer

So at run time, control shifts back and forth between JVM code and native code

The native code runs much faster than does the interpreted JVM code

Compiling JVM code to native code involves the following

- Register allocation
- Optimization
- Instruction selection
- Run-time support

Introduction

Introduction

We will translate a small subset of JVM instructions to the native code for the MIPS architecture, and execute the native code using SPIM (a MIPS simulator)

Introduction

We will translate a small subset of JVM instructions to the native code for the MIPS architecture, and execute the native code using SPIM (a MIPS simulator)

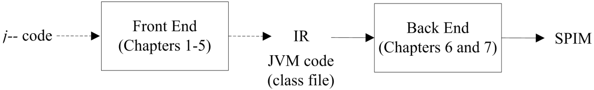
MIPS is a relatively modern reduced instruction set computer (RISC), which has a set of simple but fast instructions that operate on values in registers — for this reason it is often referred to as a register-based architecture

Introduction

We will translate a small subset of JVM instructions to the native code for the MIPS architecture, and execute the native code using SPIM (a MIPS simulator)

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

Our goal is illustrated in the following figure



We re-define what constitute the IR, the front end and the back end

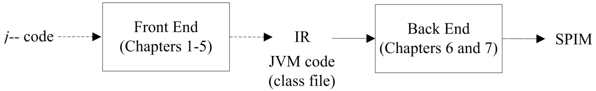
- JVM code is our new IR
- The *j*-- to JVM translator (Chapters 1 — 5) is our new front end
- The JVM to SPIM translator (Chapters 6 and 7) is our new back end

Introduction

We will translate a small subset of JVM instructions to the native code for the MIPS architecture, and execute the native code using SPIM (a MIPS simulator)

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

Our goal is illustrated in the following figure



j-- code

Front End
(Chapters 1-5)

IR

JVM code
(class file)

Back End
(Chapters 6 and 7)

SPIM

We re-define what constitute the IR, the front end and the back end

- JVM code is our new IR
- The *j*-- to JVM translator (Chapters 1 — 5) is our new front end
- The JVM to SPIM translator (Chapters 6 and 7) is our new back end

We translate enough JVM code to SPIM code to handle the *j*-- program shown in the following slide

Introduction

Introduction

```java
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--;
        }
        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’);
    }
}
```

Introduction

Introduction

We handle static methods, conditional statements, while loops, recursive method invocations, and enough arithmetic to do a few computations

Introduction

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, for example, constant strings

Introduction

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, for example, constant strings

The program above refers to an array, but doesn't do anything with it so we do not implement array objects

Introduction

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, for example, constant strings

The program above refers to an array, but doesn't do anything with it so we do not implement array objects

So, our run-time support is minimal

Introduction

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, for example, constant strings

The program above refers to an array, but doesn't do anything with it so we do not implement array objects

So, our run-time support is minimal

To determine what JVM instructions must be handled, it is worth looking at the output from running `javap` on `Factorial.class`

Introduction

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, for example, constant strings

The program above refers to an array, but doesn't do anything with it so we do not implement array objects

So, our run-time support is minimal

To determine what JVM instructions must be handled, it is worth looking at the output from running `javap` on `Factorial.class`

```
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
```

Introduction

Introduction

```
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
```

Introduction

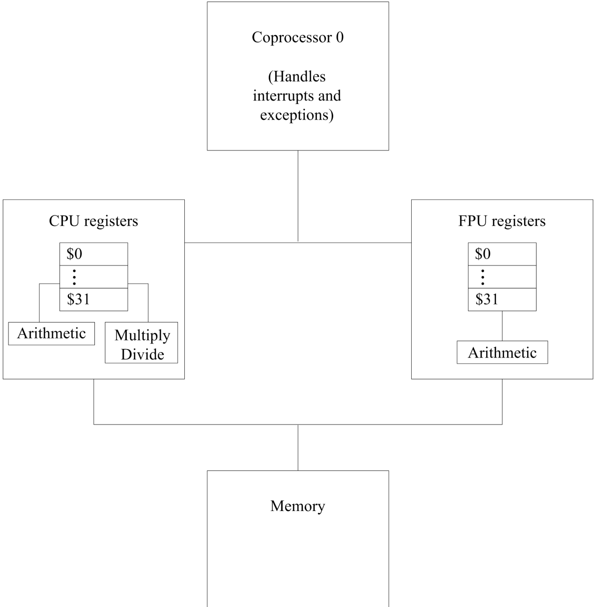
Introduction

```
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
}
```

SPIM and the MIPS Architecture

SPIM and the MIPS Architecture

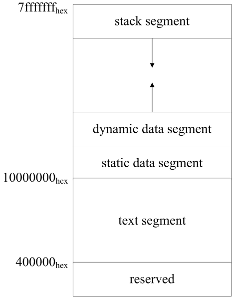
The MIPS computer organization is shown below



SPIM and the MIPS Architecture

SPIM and the MIPS Architecture

Memory organization, by convention divided into four segments, is shown below



- Text segment - The program's instructions go here
- Static data segment - Static data, which exist for the duration of the program, go here
- Dynamic data segment (aka heap) - This is where objects and arrays are dynamically allocated during execution of the program
- Like the stack 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

SPIM and the MIPS Architecture

SPIM and the MIPS Architecture

Many of the thirty two (0 – 31) 32-bit general-purpose registers, by convention are designated for special uses, and have alternative names

- $zero (0) always holds the constant 0
- $at (1) is reserved for use by the assembler
- $v0 and $v1 (2 and 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 to save them
- $s0 – $s7 (16 – 23) are meant to hold values that must be preserved across routine calls; it is up to the callee to save these registers
- $t8 and $t9 (24 and 25) are caller-saved temporaries
- $k0 and $k1 (26 and 27) are reserved for use by the operating system kernel
- $gp (28) is a global pointer 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

SPIM and the MIPS Architecture

SPIM and the MIPS Architecture

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

SPIM and the MIPS Architecture

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

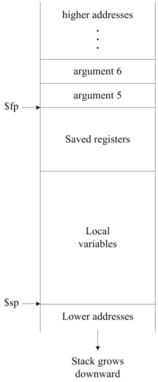
Most bookkeeping for routine invocation is recorded in a stack frame on the run-time stack segment, as is done in the JVM; but here we must also deal with registers

SPIM and the MIPS Architecture

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 run-time stack segment, as is done in the JVM; but here we must also deal with registers

The stack frame for an invoked routine is shown below

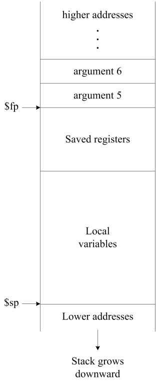


SPIM and the MIPS Architecture

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 run-time stack segment, as is done in the JVM; but here we must also deal with registers

The stack frame for an invoked routine is shown below

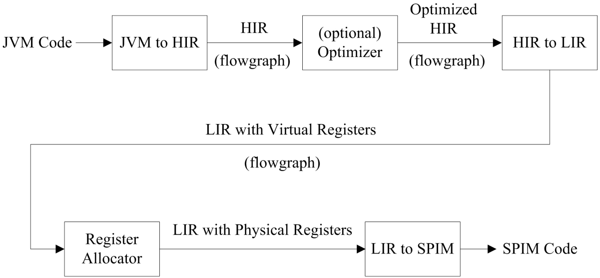


SPIM provides a set of system calls for accessing simple input and output functions

Our Translator

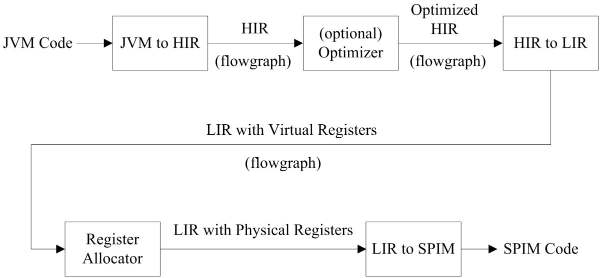
Our Translator

Phases of our JVM to SPIM translator are shown below



Our Translator

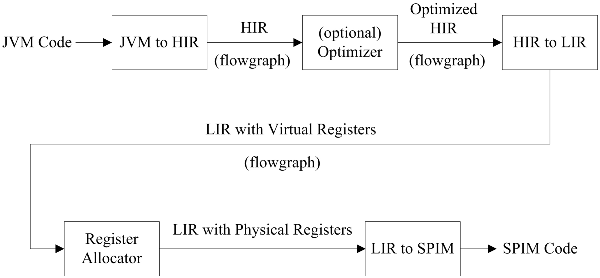
Phases of our JVM to SPIM translator are shown below



The first step is to scan through the JVM instructions and construct a flow graph of basic blocks

Our Translator

Phases of our JVM to SPIM translator are shown below



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

Our Translator

Our Translator

Consider the `computeIter()` method from our `Factorial` example

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

Our Translator

Consider the `computeIter()` method from our `Factorial` example

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

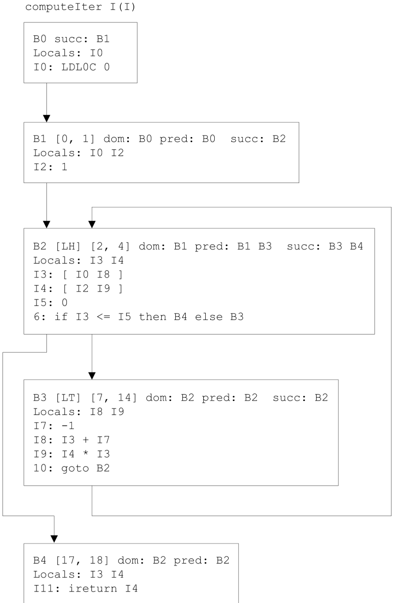
The JVM code for the method is shown below (line breaks to delineate basic blocks)

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

   0: const_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
```

Our Translator

Our Translator

The control-flow graph, expressed as a graph constructed from the basic blocks is shown below



```
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
```

Our Translator

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

The denotation `[LT]` on the first line of `B3` indicates that `B3` is a loop tail

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

The denotation `[LT]` on the first line of `B3` indicates that `B3` is a loop tail

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

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

The denotation `[LT]` on the first line of `B3` indicates that `B3` is a loop tail

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 denotation `dom` labels the basic block's immediate dominator

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

The denotation `[LT]` on the first line of `B3` indicates that `B3` is a loop tail

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 denotation `dom` labels the basic block's immediate dominator

A node d is said to dominate a node n if every path from the entry node (`B0`) to n must go through d

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

The denotation `[LT]` on the first line of `B3` indicates that `B3` is a loop tail

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 denotation `dom` labels the basic block's immediate dominator

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 is not the same as n

Our Translator

The boxes represent the basic blocks and the arrows indicate the flow of control among the blocks

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`)

We add an extra beginning block `B0` for the method's entry point

The denotation `[LH]` on the first line of `B2` indicates that `B2` is a loop header

The denotation `[LT]` on the first line of `B3` indicates that `B3` is a loop tail

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 denotation `dom` labels the basic block's immediate dominator

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 is not the same as *n*

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*, ie, it is the node on the path from the entry node to *n* that is the "closest" to *n*

Our Translator

Our Translator

Local variables are tracked in a state vector called `Locals` and are indexed in this vector by their location in the JVM stack frame

Our Translator

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`

Our Translator

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

Our Translator

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`; in `B1` the vector has two elements: the first is `I0` for `n` and the second is `I2` for result

Our Translator

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`; in `B1` the vector has two elements: the first is `I0` for `n` and the second is `I2` for result

The instruction sequence within each basic block is of a higher level than is JVM code

Our Translator

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`; in `B1` the vector has two elements: the first is `I0` for `n` and the second is `I2` for result

The instruction sequence within each basic block is of a higher level than is JVM code

For example, the *j*-- statement

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

might bre represented in HIR by

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

where `I0`, `I1` and `I2` refer to the instruction IDs labeling instructions that compute values for `x`, `y` and `z` respectively

Our Translator

Our Translator

The instruction for loading a constant is simply the constant itself; for example, the single instruction from block `B1`

```
I2: 1
```

Our Translator

The instruction for loading a constant is simply the constant itself; for example, the single instruction from block `B1`

```
I2: 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`

Our Translator

Our Translator

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

Our Translator

Our HIR employs static single assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value, which 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;
```

we might subscript our variables to distinguish different versions

x_1 = 3;

x_2 = x_1 + y_1;

Our Translator

Our HIR employs static single assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value, which 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;
```

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

Our Translator

Our HIR employs static single assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value, which 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;
```

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

Our Translator

Our HIR employs static single assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value, which 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;
```

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 two or more predecessors, the states must be merged

Our Translator

Our Translator

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

```
static int ssa(int w1) {
    if (w1 > 0) {
        w2 = 1;
    }
    else {
        w3 = 2;
    }
    return w?;
}
```

Our Translator

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

```
static int ssa(int w1) {
    if (w1 > 0) {
        w2 = 1;
    }
    else {
        w3 = 2;
    }
    return w?;
}
```

In the statement

```
return w?;
```

which `w` do we return?

Our Translator

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

```
static int ssa(int w1) {
    if (w1 > 0) {
        w2 = 1;
    }
    else {
        w3 = 2;
    }
    return w?;
}
```

In the statement

```
return w?;
```

which `w` do we return?

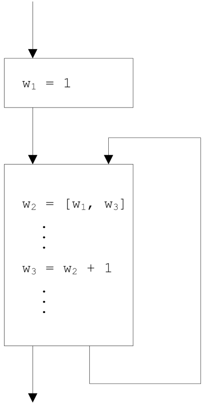
We solve this problem by using what is called a Phi function, a special HIR instruction that captures the possibility of a variable having one of several values; in our example, the final block would contain the following code

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

Our Translator

Our Translator

Another place where Phi functions are needed are in loop headers, basic blocks having at least one incoming backward branch and at least two predecessors, as illustrated below



Our Translator

Our Translator

We conservatively define Phi functions for all variables and then remove redundant Phi functions later

Our Translator

We conservatively define Phi functions for all variables and then remove redundant Phi functions later

In the first instance, $\mathtt{w}_2$ can be defined as a Phi function with operands $\mathtt{w}_1$ and $\mathtt{w}_2$

$$\mathtt{w}_2 \ = [\mathtt{w}_1 \ \mathtt{w}_2]$$

Our Translator

We conservatively define Phi functions for all variables and then remove redundant Phi functions later

In the first instance, $\mathtt{w}_2$ can be defined as a Phi function with operands $\mathtt{w}_1$ and $\mathtt{w}_2$

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_2]$

When $\mathtt{w}$ is later incremented, the second operand may be overwritten by the new $\mathtt{w}_3$

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_3]$

Our Translator

We conservatively define Phi functions for all variables and then remove redundant Phi functions later

In the first instance, $\mathtt{w}_2$ can be defined as a Phi function with operands $\mathtt{w}_1$ and $\mathtt{w}_2$

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_2]$

When $\mathtt{w}$ is later incremented, the second operand may be overwritten by the new $\mathtt{w}_3$

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_3]$

A redundant Phi function will not be changed and can be removed

Our Translator

We conservatively define Phi functions for all variables and then remove redundant Phi functions later

In the first instance, $\mathtt{w}_2$ can be defined as a Phi function with operands $\mathtt{w}_1$ and $\mathtt{w}_2$

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_2]$

When $\mathtt{w}$ is later incremented, the second operand may be overwritten by the new $\mathtt{w}_3$

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_3]$

A redundant Phi function will not be changed and can be removed

If the $\mathtt{w}$ is never modified in the loop body, the Phi function instruction takes the form

$\mathtt{w}_2 = [\mathtt{w}_1\ \mathtt{w}_2]$

Our Translator

We conservatively define Phi functions for all variables and then remove redundant Phi functions later

In the first instance, `w2` can be defined as a Phi function with operands `w1` and `w2`

`w2 = [w1 w2]`

When `w` is later incremented, the second operand may be overwritten by the new `w3`

`w2 = [w1 w3]`

A redundant Phi function will not be changed and can be removed

If the `w` is never modified in the loop body, the Phi function instruction takes the form

`w2 = [w1 w2]`

Phi functions are tightly bound to state vectors, so 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

Our Translator

Our Translator

The translation process from JVM instructions to HIR takes place in the constructor `NEmitter()` in the class `NEmitter`; for each method, the control-flow graph HIR is constructed in several steps

Our Translator

The translation process from JVM instructions to HIR takes place in the constructor `NEmitter()` in the class `NEmitter`; for each method, the control-flow graph HIR is constructed in several steps

The `NControlFlowGraph` constructor is invoked on the method, which produces the control-flow graph `cfg`; in this first step, the JVM code is translated to sequences of tuples

1. 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
2. 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 `ArrayList` called `tuples`

Our Translator

The translation process from JVM instructions to HIR takes place in the constructor `NEmitter()` in the class `NEmitter`; for each method, the control-flow graph HIR is constructed in several steps

The `NControlFlowGraph` constructor is invoked on the method, which produces the control-flow graph `cfg`; in this first step, the JVM code is translated to sequences of tuples

1. 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
2. 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 `ArrayList` called `tuples`

The method call

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

detects loop headers and loop tails

Our Translator

The translation process from JVM instructions to HIR takes place in the constructor `NEmitter()` in the class `NEmitter`; for each method, the control-flow graph HIR is constructed in several steps

The `NControlFlowGraph` constructor is invoked on the method, which produces the control-flow graph `cfg`; in this first step, the JVM code is translated to sequences of tuples

1. 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
2. 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 `ArrayList` called `tuples`

The method call

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

detects loop headers and loop tails

The method call

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

removes unreachable blocks

Our Translator

Our Translator

The method call

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

computes an immediate dominator for each basic block, that closest predecessor through which all paths must pass to reach the target block; it's a useful place to which insert invariant code that is lifted out of a loop in optimization

Our Translator

The method call

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

computes an immediate dominator for each basic block, that closest predecessor through which all paths must pass to reach the target block; it's a useful place to which insert invariant code that is lifted out of a loop in optimization

The method call

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

converts the tuples representation to HIR, stored as a sequence of HIR instructions in the array list `hir` for each block

Our Translator

The method call

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

computes an immediate dominator for each basic block, that closest predecessor through which all paths must pass to reach the target block; it's a useful place to which insert invariant code that is lifted out of a loop in optimization

The method call

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

converts the tuples representation to HIR, stored as a sequence of HIR instructions in the array list `hir` for each block

The method call

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

eliminates unnecessary Phi functions and replaces them with their simpler values

Our Translator

The method call

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

computes an immediate dominator for each basic block, that closest predecessor through which all paths must pass to reach the target block; it's a useful place to which insert invariant code that is lifted out of a loop in optimization

The method call

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

converts the tuples representation to HIR, stored as a sequence of HIR instructions in the array list `hir` for each block

The method call

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

eliminates unnecessary Phi functions and replaces them with their simpler values

The HIR is now ready for further analysis

Our Translator

Our Translator

That the HIR is in SSA form makes it amenable to several simple optimizations, which make for fewer instructions and/or faster programs

Our Translator

That the HIR is in SSA form makes it amenable to several simple optimizations, which make for fewer instructions and/or faster programs

Local optimizations are improvements made based on analysis of the linear sequence of instructions within a basic block

Our Translator

That the HIR is in SSA form makes it amenable to several simple optimizations, which make for fewer instructions and/or faster programs

Local optimizations are improvements made based on analysis of the linear sequence of instructions within a basic block

Global optimizations require an analysis of the whole control graph and involve what is called data-flow analysis

Our Translator

That the HIR is in SSA form makes it amenable to several simple optimizations, which make for fewer instructions and/or faster programs

Local optimizations are improvements made based on analysis of the linear sequence of instructions within a basic block

Global optimizations require an analysis of the whole control graph and involve what is called data-flow analysis

In some 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; for example, consider the following code

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

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

can be replaced with

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

Our Translator

Our Translator

Expressions having operands that are both constants, or variables whose values are known to be constants, can be folded, that is replaced by their constant value

Our Translator

Expressions having operands that are both constants, or variables whose values are known to be constants, can be folded, that is replaced by their constant value

For example, consider the Java method

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

and the 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
```

Our Translator

Expressions having operands that are both constants, or variables whose values are known to be constants, can be folded, that is replaced by their constant value

For example, consider the Java method

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

and the 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`

Our Translator

Our Translator

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, in the following method

```
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;
```

Our Translator

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, in the following method

```
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;
```

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];
    }
}
```

where `a`, `b`, and `c` are integer matrices

Our Translator

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, in the following method

```
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;
```

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];
    }
}
```

where `a`, `b`, and `c` are integer matrices

If `a’`, `b’`, and `c’` are their base addresses respectively, then the memory addresses of `a[i][j]`, `b[i][j]`, and `c[i][j]` are `a’ + i * 4 * 1000 + j * 4`, `b’ + i * 4 * 1000 + j * 4`, and `c’ + i * 4 * 1000 + j * 4`; eliminating the common offsets, `i * 4 * 1000 + j * 4`, can save us a lot of computation

Our Translator

Our Translator

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

Our Translator

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

For example, the following *j--* code for summing two matrices

```
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;
}
```

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;
}
```

Our Translator

Our Translator

When indexing an array, we must check that the index is within bounds; for example, in our code for matrix addition, 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 to zero, and less than 1000; and this must be done for each of `c`, `a` and `b`

Our Translator

When indexing an array, we must check that the index is within bounds; for example, in our code for matrix addition, 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 to 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`

Our Translator

When indexing an array, we must check that the index is within bounds; for example, in our code for matrix addition, 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 to 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`

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 want to make sure that `a` is non-null before computing the offset to the field `f`

Our Translator

When indexing an array, we must check that the index is within bounds; for example, in our code for matrix addition, 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 to 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`

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 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, in the following code

```
...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`

Our Translator

When indexing an array, we must check that the index is within bounds; for example, in our code for matrix addition, 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 to 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`

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 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, in the following code

```
...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`

The HIR does not present every opportunity for optimization, particularly those involving back branches (in loops); for this we would need full data-flow analysis where we compute where in the code computed values remain valid

Our Translator

Our Translator

Since the HIR is not necessarily suitable for register allocation, we translate it into a low-level intermediate representation (LIR) where

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

Our Translator

Since the HIR is not necessarily suitable for register allocation, we translate it into a low-level intermediate representation (LIR) where

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

For example, the LIR for `Factorial.computeIter()` is shown below

```
computeIter (I)I

  B0

  B1
  0: LDC [1] [V32|I]
  5: MOVE $a0 [V33|I]
  10: MOVE [V32|I] [V34|I]

  B2
  15: LDC [0] [V35|I]
  20: BRANCH [LE] [V33|I] [V35|I] B4

  B3
  25: LDC [-1] [V36|I]
  30: ADD [V33|I] [V36|I] [V37|I]
  35: MUL [V34|I] [V33|I] [V38|I]
  40: MOVE [V38|I] [V34|I]
  45: MOVE [V37|I] [V33|I]
  50: BRANCH B2

  B4
  55: MOVE [V34|I] $v0
  60: RETURN $v0
```

Our Translator

Our Translator

In the above example, seven virtual registers `v32` – `v38` are allocated to the LIR computation

Our Translator

In the above example, 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

Our Translator

In the above example, 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

We start enumerating virtual registers beginning at 32; the numbers 0 through 31 are reserved for enumerating physical registers

Our Translator

In the above example, 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

We start enumerating virtual registers beginning at 32; the numbers 0 through 31 are reserved for enumerating physical registers

LIR instructions are read from left to right

Our Translator

In the above example, 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

We start enumerating virtual registers beginning at 32; the numbers 0 through 31 are reserved for enumerating physical registers

LIR instructions are read from left to right

We enumerate the LIR instructions by multiples of five, which eases the insertion of spill (and restore) instructions, which may be required for register allocation

Our Translator

Our Translator

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

1. The `NEmitter` constructor invokes the `NControlFlowGraph` method `hirToLir()` on the control-flow graph

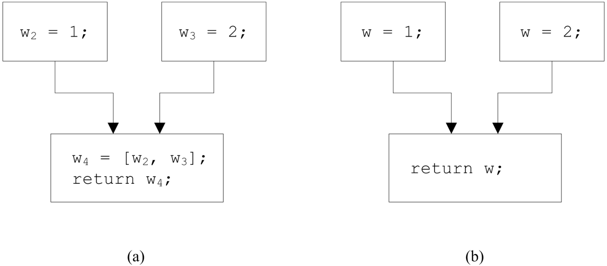
   ```
   cfg.hirToLir();
   ```

   which 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. `NEmitter` invokes the `NControlFlowGraph` method `resolvePhiFunctions()` on the control-flow graph

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

   which resolves Phi function instructions, replacing them by move instructions near the end of the predecessor blocks; for example, the Phi function from figure (a) below resolves to the moves in figure (b)



(a) (b)

Our Translator

Our Translator

A 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
5. A run-time library of code that supports the Java API

Our Translator

A 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
5. A run-time library of code that supports the Java API

We use a simple naming mechanism that takes account of just classes and their members

Our Translator

A 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
5. A run-time library of code that supports the Java API

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>`

Our Translator

A 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
5. A run-time library of code that supports the Java API

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>`

Static fields are similarly named

Our Translator

A 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
5. A run-time library of code that supports the Java API

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>`

Static fields are similarly named

String literals in the data segment have labels that suggest what they label

Our Translator

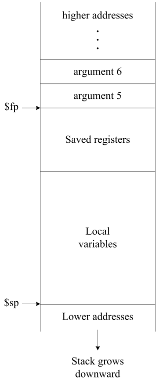
Our Translator

Our run-time stack conforms to the run-time convention described for SPIM

Our Translator

Our run-time stack conforms to the run-time convention described for SPIM

Each time a method is invoked, a new stack frame of the type shown below is pushed onto the stack; upon return from the method, the same frame is popped off from the stack



Our Translator

Our Translator

SPIM provides a set of built-in system calls for performing simple I/O tasks

Our Translator

SPIM provides a set of built-in system calls for performing simple I/O tasks

Our runtime environment includes a class `SPIM`, which is a wrapper that gives us access to these calls as a set of static methods

```java
package spim;

public class SPIM {
    public static void printInt(int value) { }
    public static void printFloat(float value) { }
    public static void printDouble(double value) { }
    public static void printString(String value) { }
    public static void printChar(char value) { }
    public static int readInt() { return 0; }
    public static float readFloat() { return 0; }
    public static double readDouble() { return 0; }
    public static String readString(int length) { return null; }
    public static char readChar() { return ' '; }
    public static int open(String filename, int flags, int mode) { return 0; }
    public static String read(int fd, int length) { return null; }
    public static int write(int fd, String buffer, int length) { return 0; }
    public static void close(int fd) { }
    public static void exit() { }
    public static void exit2(int status) { }
}
```

Our Translator

SPIM provides a set of built-in system calls for performing simple I/O tasks

Our runtime environment includes a class `SPIM`, which is a wrapper that gives us access to these calls as a set of static methods

```
package spim;

public class SPIM {
    public static void printInt(int value) { }
    public static void printFloat(float value) { }
    public static void printDouble(double value) { }
    public static void printString(String value) { }
    public static void printChar(char value) { }
    public static int readInt() { return 0; }
    public static float readFloat() { return 0; }
    public static double readDouble() { return 0; }
    public static String readString(int length) { return null; }
    public static char readChar() { return ’ ’; }
    public static int open(String filename, int flags, int mode) { return 0; }
    public static String read(int fd, int length) { return null; }
    public static int write(int fd, String buffer, int length) { return 0; }
    public static void close(int fd) { }
    public static void exit() { }
    public static void exit2(int status) { }
}
```

Since 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; for example

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

Our Translator

Our Translator

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

Our Translator

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

We iterate through the list of methods for each class; for each method, we do the following

- We generate a label for the method's entry point
- We generate code to push a new frame onto the run-time stack and then code to save all our registers; we treat all of SPIM's general purpose registers \$t0 – \$t9 and \$s0 – \$s7 as callee-saved registers
- Since all branches in the code are expressed as branches to basic blocks, a unique label for each basic block is generated into the code
- 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
- 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
- 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

Our Translator

Our Translator

After we have generated the text portion (the program instructions) for the method, we then populate a data area from the list of string literals constructed previously; any other literals that you may wish to implement would be handled in the same way

Our Translator

After we have generated the text portion (the program instructions) for the method, we then populate a data area from the list of string literals constructed previously; any other literals that you may wish to implement would be handled in the same way

Once all of the program code has been generated, we then copy out the SPIM code for implementing the `SPIM` class

Our Translator

After we have generated the text portion (the program instructions) for the method, we then populate a data area from the list of string literals constructed previously; any other literals that you may wish to implement would be handled in the same way

Once all of the program code has been generated, we then copy out the SPIM code for implementing the `SPIM` class

For example, the SPIM 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
```

Our Translator

Our Translator

```
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
```

Our Translator

```
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
```

We can perform peephole optimizations (considering just a few instuctions at a time) on the SPIM code to remove jumps to immediate instructions and simplify jumps to jumps