Translating JVM Code to MIPS Code

Outline

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

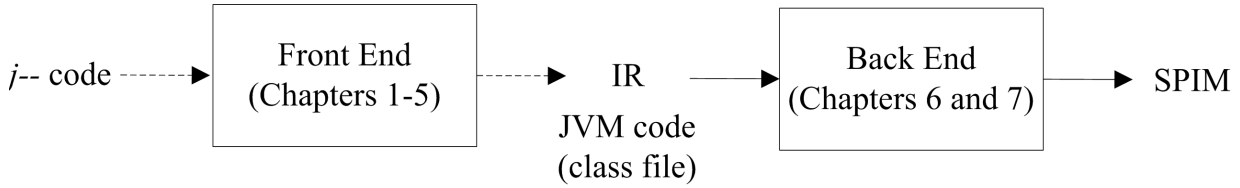
- Register allocation
- Optimization
- Instruction selection
- Run-time support

Introduction

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

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

Introduction

```
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

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

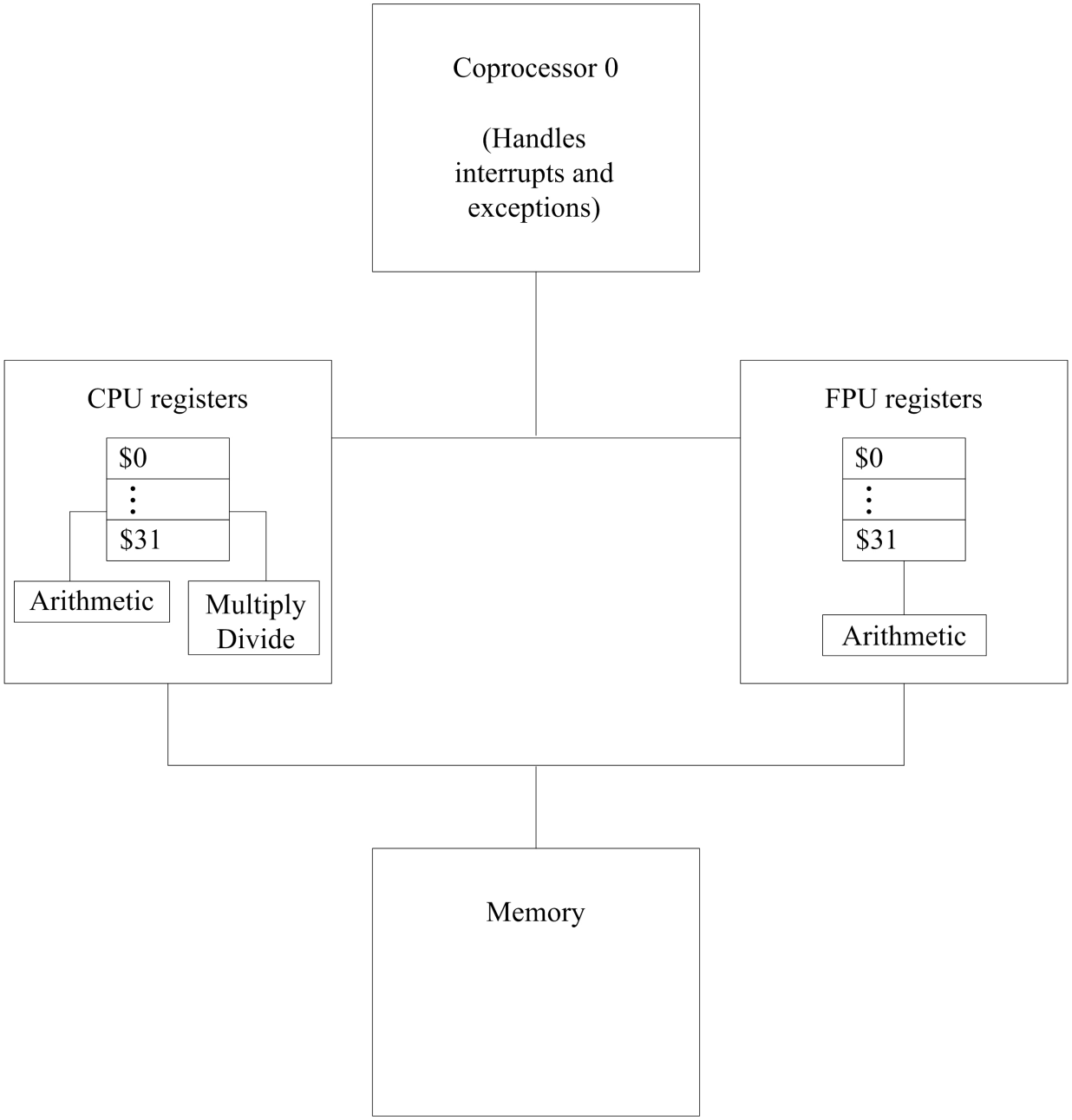
```
public static int computeRec(int);
  Code:
   Stack=3, Locals=1, Args_size=1
   0: iload_0
   1: iconst_0
   2: if_icmpgt 10
   5: iconst_1
   6: ireturn
   7: goto  19
   10: iload_0
   11: iload_0
   12: iconst_1
   13: isub
   14: invokestatic #13; //Method computeRec:(I)I
   17: imul
   18: ireturn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

```
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
}
```

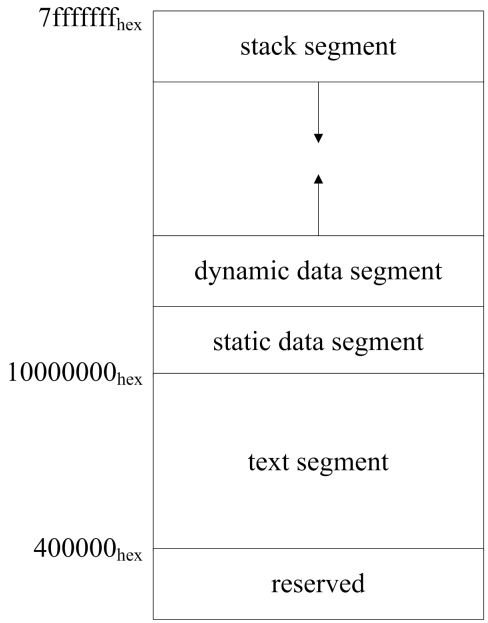
SPIM and the MIPS Architecture

The MIPS computer organization is shown below



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

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

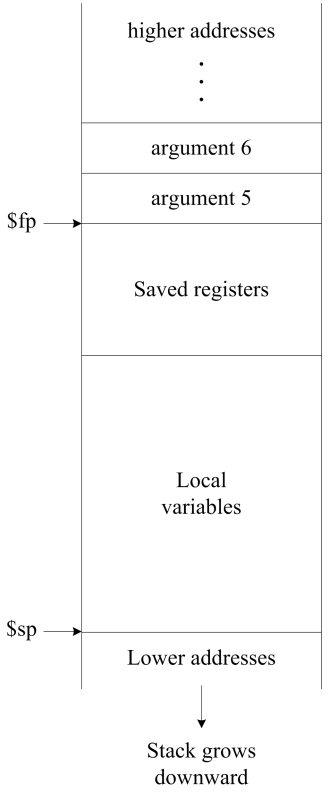
- $zero (0) always holds the constant 0
- $at (1) is reserved for use by the assembler
- $v0 and $v1 (2 and 3) are used for expression evaluation and as the results of a function
- $a0 – $a3 (4 – 7) are used for passing the first four arguments to routines; any additional arguments are passed on the 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 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

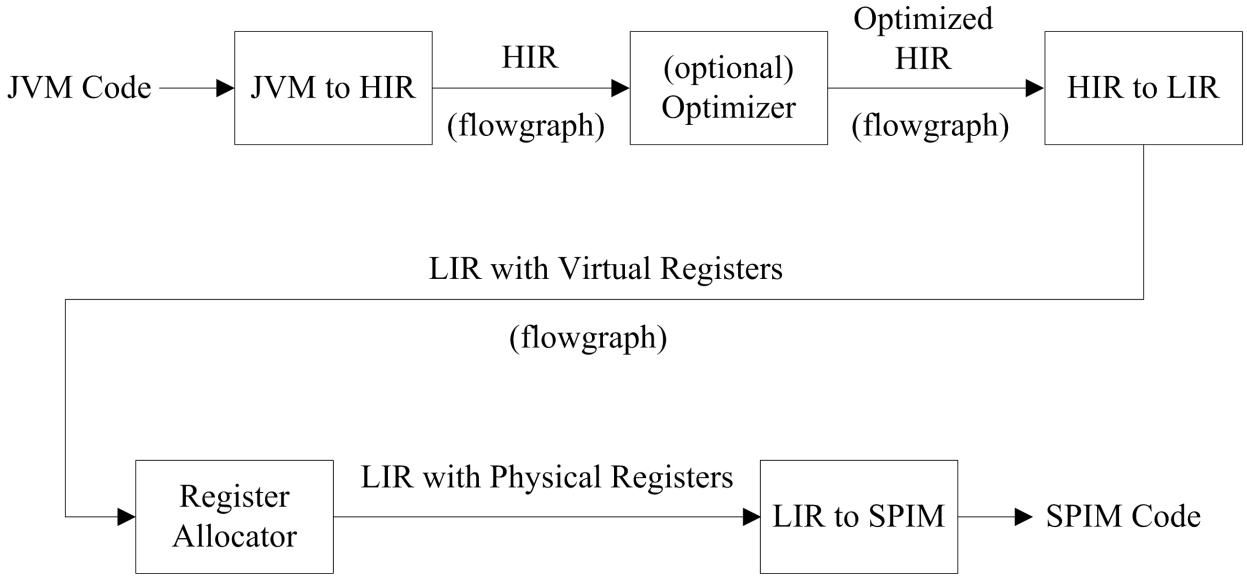
The stack frame for an invoked routine is shown below



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

Our Translator

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

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

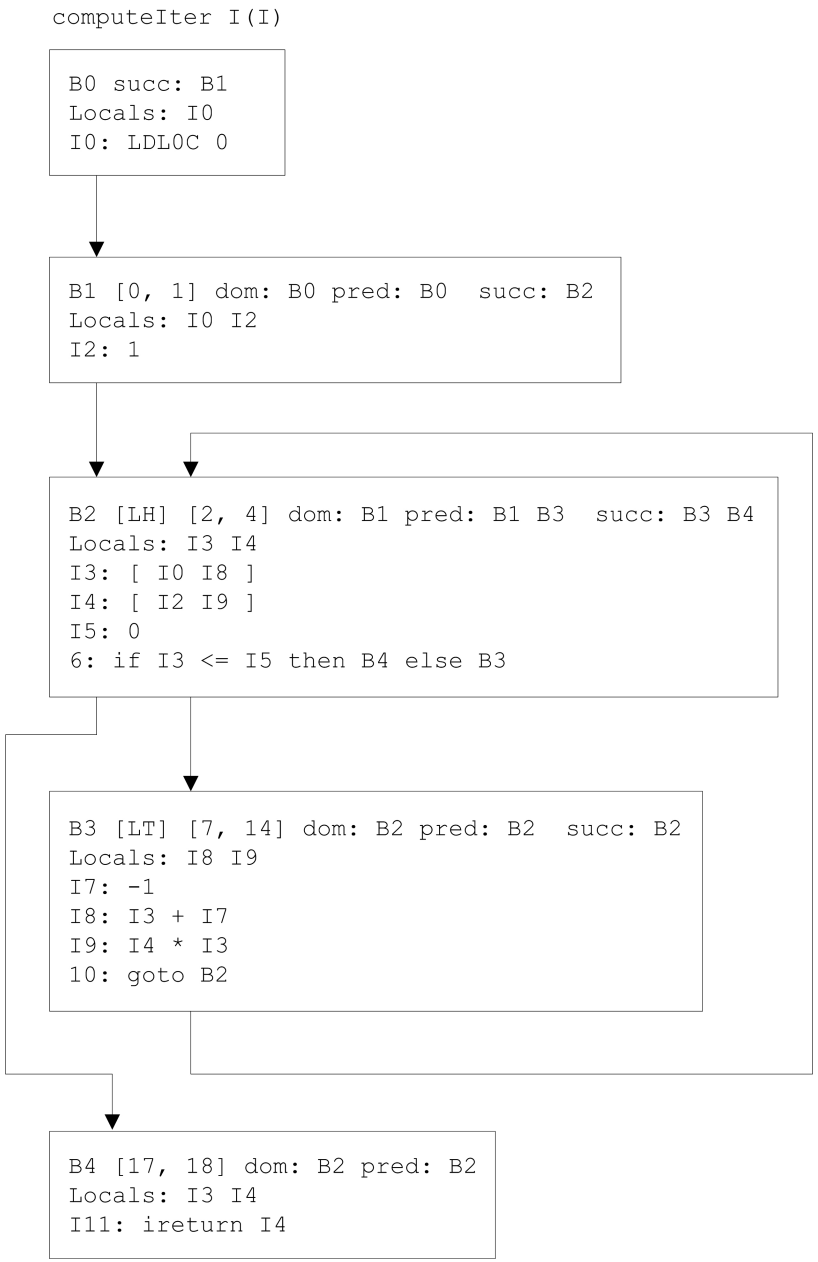
```
    public static int computeIter(int n) {
        int result = 1;
        while ( n > 0 ) {
            result = result * n--;
        }
        return result;
    }
```

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

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



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

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

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 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

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?;
```

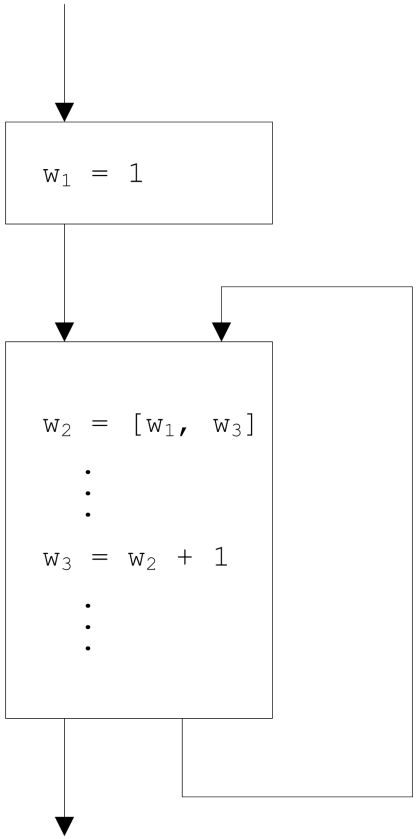
which `w` do we return?

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

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

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

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

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

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

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