A Tutorial on Adding New Instructions to the Oracle® Java HotSpot ™ Virtual Machine

Vasanth Venkatachalam
Senior Software Engineer, AMD

Introduction

Architecture-specific code generation changes for Java programs often require modifying the Java Virtual Machine (JVM) to support new target machine instructions or sequences of these instructions. This article discusses how to add this support to the Oracle® Java HotSpot Virtual Machine, using as a running example the addition of an instruction sequence for a floating-point conditional move optimization. We begin by introducing the conditional move optimization and giving an overview of the model HotSpot uses when compiling Java bytecode to native instructions. We then describe the changes required in this model to support floating-point conditional moves. By the end of this article, the reader should better understand the process of adding new instructions or instruction sequences that support their own optimizations.

This article assumes the target platform is x86, but the techniques we outline will apply to any target architecture. For consistency and readability, all the code examples presented use MASM syntax.

The Conditional Move Optimization

Most compilers generate the code for if-then-else constructs by emitting a compare instruction and a conditional branch instruction. For example, consider this code:

```
if(i < 1) {
i = i + 5;
}
```

The typical x86 native instructions emitted for the above code would be:

```
cmp edx ecx
jge elseblock  //if I >= 1, jump past the addition
add edx, 5
jmp end
end:
```

The cmp instruction compares the value of i with the scalar 1 and sets the condition flags of the rFlags register to indicate the result of the compare. The conditional jump instruction (jge) checks these flags and causes the program execution either to jump to the label marked end or to fall-through to the add instruction.

In cases in which an if construct has no else block, a compiler can sometimes emit a conditional move instruction instead of a conditional branch. The integer conditional move instruction on x86 hardware conditionally copies a value from a register or memory location into a register, based on the condition flags in the rFlags register, which were previously set by a compare instruction.

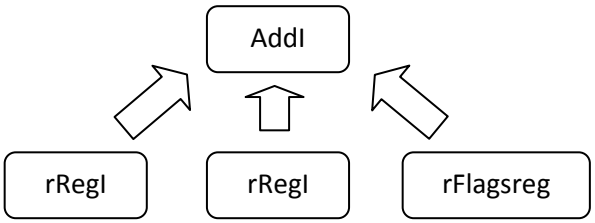
If we were to convert this code to use conditional moves instead of branches, it would become:

```
mov eax, edx
add eax, 5
cmp edx ecx
cmovge edx, eax //conditionally move the result of the addition into
edx
```

The advantage of the conditional move optimization is that it removes branches from the compiled code. The latest JDK-7 release of HotSpot emits integer cmoves for x86/x64 architectures but not floating-point cmoves. This is because the floating-point cmove instruction is a new feature on the AMD Family 15h processors, which have yet to be released. In the following sections, we will examine how HotSpot implements integer cmoves. This will give us insight into the changes needed to add support for floating-point cmoves.

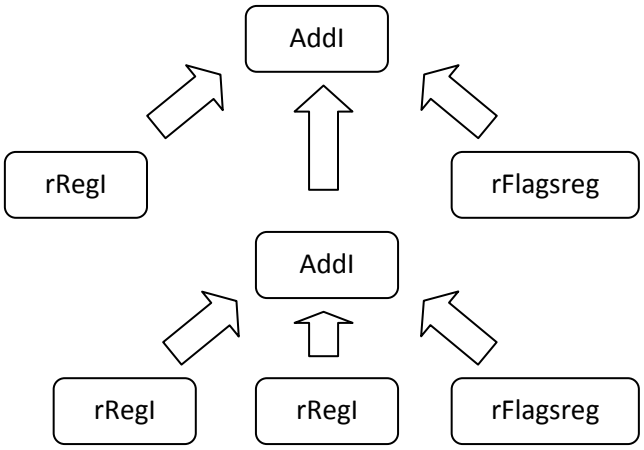
Overview of the HotSpot Compilation Process

When compiling a Java method, HotSpot first parses the bytecodes and creates an intermediate representation (IR), which represents the instructions in the method in the form of an abstract syntax tree. The nodes in the tree represent operations (e.g., addition, multiplication, cmove) and their children represent the inputs to those operations. For example, we can visually represent the sub-tree corresponding to a basic “reg-to-reg” form of the integer addition operation:



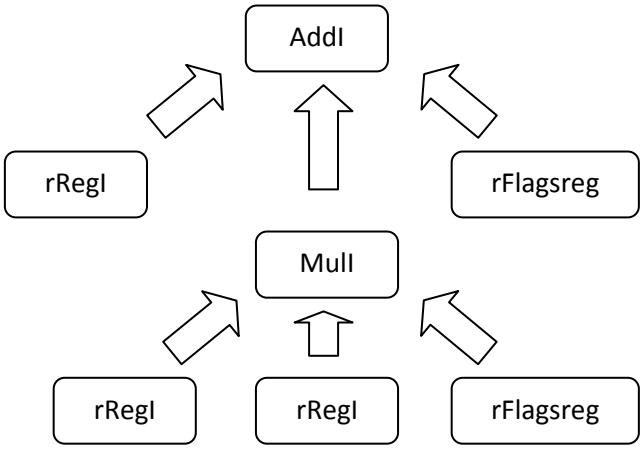
The addition operation takes three inputs, a source integer register (rRegI), a second source integer register (rRegI), and the rFlags register (rFlagsreg), which would be written to in the case of overflow.

We can also represent more complex operations in the IR. For example, the IR may contain a sub-tree that corresponds to the addition of a value in a register with the previous result of an integer addition, visually represented as:

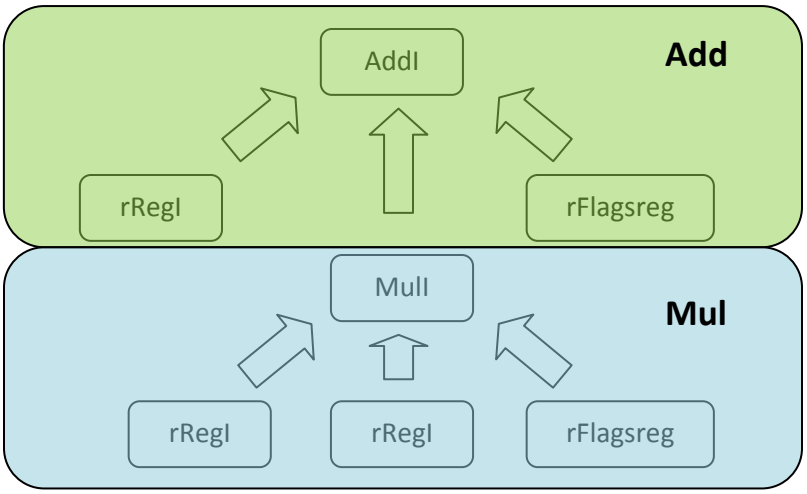


The root AddI node contains three children, as in the first example, but the second child is another AddI node.

HotSpot traverses the IR and maps specific sub-trees to sequences of native instructions. In this manner, it selects the instructions to emit for the target platform. However, there are many ways to select these instructions. Consider this sub-tree, which represents the result of adding the value in a register to the result of a multiplication:



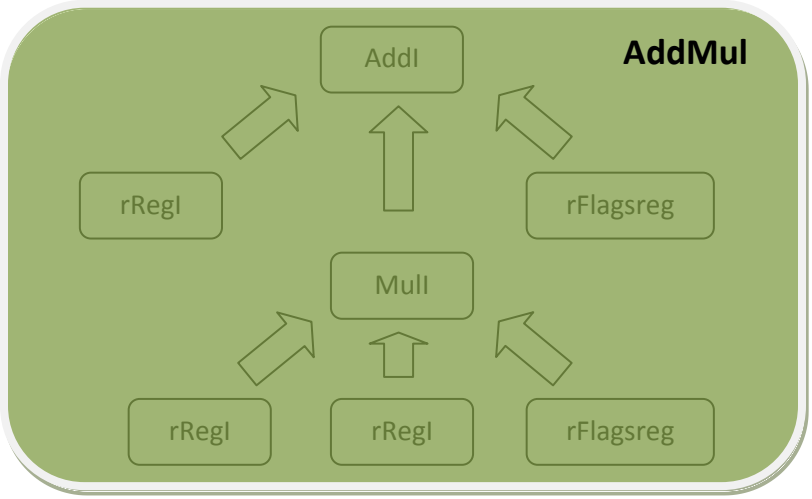
Suppose there were an instruction *AddMul* that multiplied the values in two input registers and added the result to a third register, updating rFlags as necessary. Then there would be two ways to generate native code for this syntax tree. The first would be to generate a separate mul instruction for the MulI node, and a separate add instruction for the AddI node. We can visually represent this strategy for mapping native instructions to this IR sub-tree:



This mapping eventually would generate this sequence of native instructions:

```
mul dst, src1 //dst = dst * src1
add dst, src2 //dst = dst + src2
```

An alternate approach would be to map the whole sub-tree to our fictitious AddMul instruction:



This alternate mapping would generate these native instructions:

```
AddMul dst, src1, src2 // dst = src1 * dst + src2
```

The AddMul instruction in this example is hypothetical. However, our example shows there are multiple ways to map IR nodes to sequences of native instructions. How does HotSpot decide which instruction sequence to emit in these cases? The answer lies in the architecture description language file (AD).

The AD file defines the native instructions the HotSpot code generator can emit, and it specifies for each instruction the IR sub-tree for which that instruction (or instruction sequence) would be a possible

candidate. In addition, it assigns every candidate instruction sequence a cost that HotSpot uses to determine which instruction sequences to emit in cases when multiple instruction sequences match the same IR sub-tree.

The Architecture Description File

Adding new instructions to HotSpot is a two-step process. The first step is defining the IR pattern (also called a match rule) for which we want to generate the new instruction (or instruction sequence) and defining the instruction (or instruction sequences) in the AD file. The second step is extending the HotSpot assembler with the routines that encode the new instruction(s). Examining this process for integer cmoves will give us insight into how we can do it for floating-point cmoves.

For x86 code, there are two AD files located under the directory src/cpu/x86/vm. The file x86_32.ad applies to 32-bit platforms and x86_64.ad applies to 64-bit platforms. For the purposes of this example, we will look at the section of the x86_64.ad file that defines the version of the integer cmove instruction that takes two integer registers as operands.

```
// Conditional move
instruct cmovI_reg(rRegI dst, rRegI src, rFlagsReg cr, cmpOp cop)
%{
  match(Set dst (CMoveI (Binary cop cr) (Binary dst src)));
  ins_cost(200);
  format %{ "cmovl$cop $dst, $src\t# signed, int" %}
  opcode(0x0F, 0x40);
  ins_encode(REX_reg_reg(dst, src), enc_cmov(cop), reg_reg(dst, src));
  ins_pipe(pipe_cmov_reg);
%}
```

The *instruct* construct defines a new instruction (or instruction sequence), specifies the conditions under which the instruction (or instruction sequence) should be emitted and calls routines that encode the instruction (or instruction sequence).

The first line, *instruct cmovI_reg(rRegI dst, rRegI src, rFlagsReg cr, cmpOp cop),* defines the name of the new instruction sequence (*cmovI_reg*) and lists any parameters that will be used in the body of this instruct. The actual name can be arbitrary, but it is a good practice to choose names that reflect the nature of the new instruction(s). In this case, the parameters are a source register (*src*) and destination register (*dst*), both of which store integer values (hence the *regI*), the rFlags register (*cr*) that conditional moves require, and a bool operator (*cop*) that specifies the condition code for the conditional move (e.g., move if greater-than, move if less-than).

The line *ins_cost(200)* specifies the cost of generating this instruction. As we have mentioned, the same IR sub-tree can have multiple code generation candidates. When this is the case, HotSpot chooses the candidate with the lowest cost based on the value entered in this line. For the instructions we added to HotSpot, the values chosen for ins_cost are just estimates that guarantee that HotSpot will execute these instructions. These values may not be identical to the actual cost (in cycles) of the instructions.

The format section, *format %{ "cmovl$cop $dst, $src\t# signed, int" %}* , generates the printing functions used when the command line flag *–XX:+PrintOptoAssembly* is enabled. This option allows us to see what instructions the JVM is emitting.

There are many ways to specify the encoding for an instruction sequence. In this example, the *opcode(…)* line specifies the opcode that should be emitted and the *ins_encode(…)* line is encoding the parameters for the instruction, which in this case include the register arguments and the condition code for the comparison operator *cmpOp*.

The *ins_encode* construct has two formats. In the first format, used in this example, *ins_encode* is followed by a list of comma-separated arguments called encoding classes enclosed in parenthesis. These encoding classes are macros that call assembler routines. This is no longer the preferred way of using *ins_encode*. The preferred format is the block format:

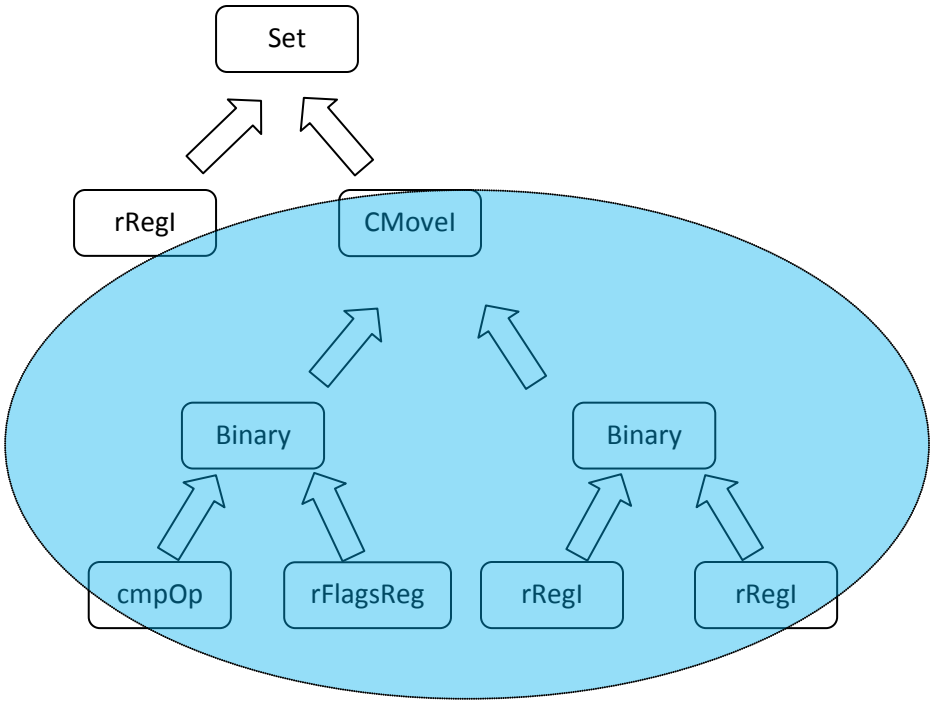
```
 ins_encode %{
stmt1;
stmt2;
stmt3;
.
.
.
stmtn;
%}
```

Where *stmt1-stmtn* are calls to assembler routines. For example, in the definition of the load_byte instruction, the *ins_encode* line is:

```
 ins_encode %{
    __ movsbl($dst$$Register, $mem$$Address); // Call to assembler
routine that encodes movsbl instruction
  %}
```

Similarly, we have used the block format of *ins_encode* to encode our new instruction sequence for floating-point conditional moves. To summarize, all of the work for encoding an instruction should happen via calls to assembler routines inside an *ins_encode %{ %}* block, instead of using encoding classes.

Finally, the line *match(Set dst (CMoveI (Binary cop cr) (Binary dst src)))* is called a match rule. It specifies the IR sub-tree for which the integer cmove instruction is a possible code generation candidate. The best way to understand how the match rule works is to study its visual representation as an in-order tree:



This rule says that for IR sub-trees that match the right-hand child of the Set node (highlighted above), the sequence of native instructions defined by the *cmovI_reg* instruct are a candidate code generation choice.

In more detail:

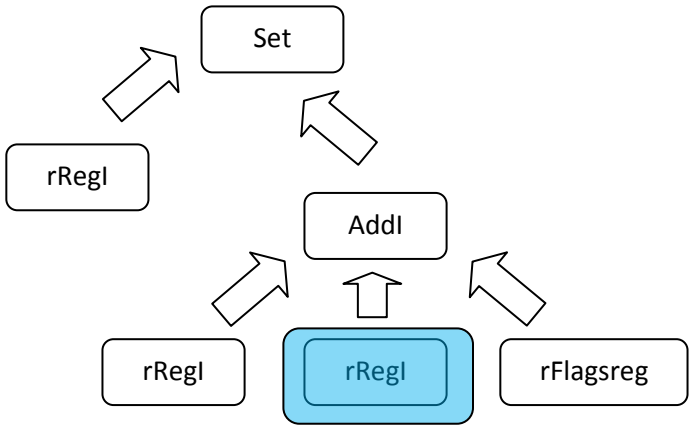
- If the IR contains a sub-tree that has a CMoveI node, whose left and right children are both Binary nodes, *and*
- The left-hand Binary node has a left-child that is a cmpOp node and a right-child that is an rFlagsReg node, *and*
- The right-hand Binary node has a left-child that is an rRegI node and a right-child that is an rRegI node, *then*
- The instruction sequence defined by the *cmovI_reg i*nstruct is one of the candidate instruction sequences that can be generated for this sub-tree.

The right-hand child of the Set node is always the *source* of the match rule; that is, it specifies the IR sub-trees that could match this rule, and thus the sub-trees for which this sequence of native instructions is a possible candidate. The left-hand child is the *destination* of the match rule. It says the result of a CMoveI operation would be stored in a node of type rRegI (an integer register).

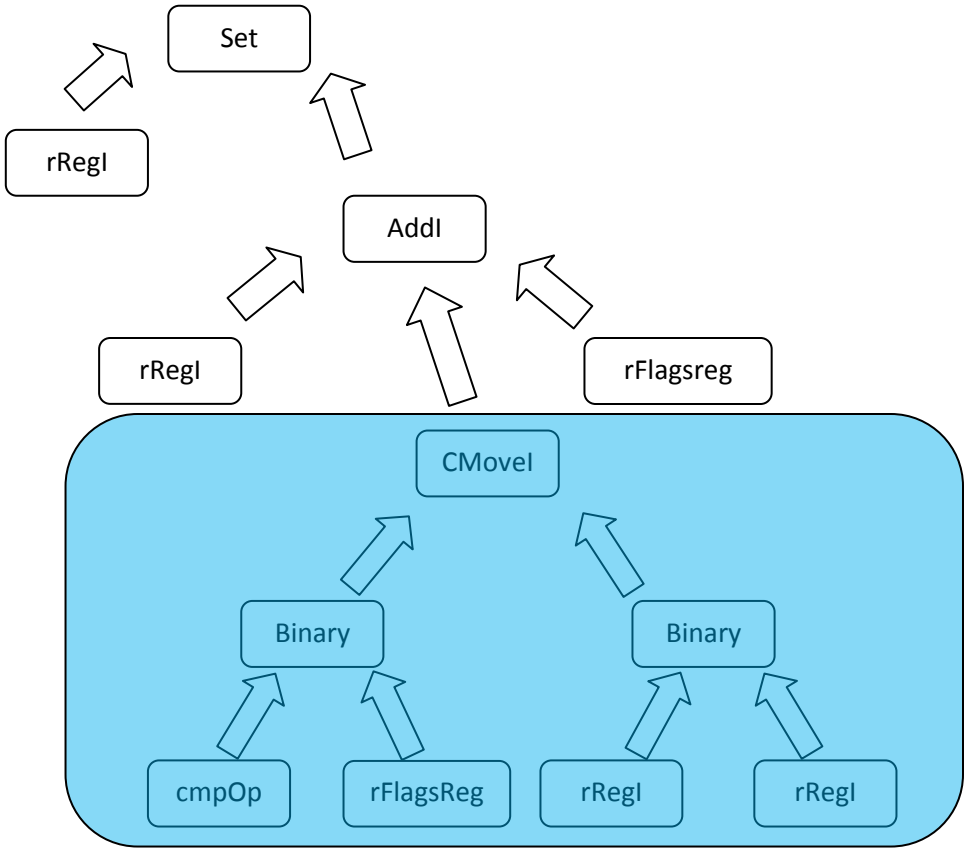
The Binary nodes are an implementation detail used to transform nodes that require more than two inputs into a binary tree prior to matching, because the IR matcher can only match binary trees. In this

example, the CMoveI node essentially requires four inputs (cmpOp, rFlagsReg, rRegI, rRegI). The Binary nodes allow formation of a tree that takes two inputs instead of four.

These observations allow us to construct more complex match rules. For example, suppose we want to define a new instruction sequence that should be emitted for any integer addition operation in which one of the source registers is the output of a previously defined CMoveI operation. The first step in deriving the match rule is to look at the match rule for a generic addition operation:



We can then derive the match rule for our new addition operation by replacing the second rRegI parameter to AddI (highlighted) with the source of the CMoveI match rule:



In AD syntax, this new match rule would be expressed as:

match (Set dst (AddI src (CMoveI (Binary (cop cr) Binary (cmove_dest cmove_source)))))

where the variables *src*, *dst*, *cmove_source* and *cmove_dest* have previously been defined in the list of parameters to the instruct.

In this manner, we can define more complex instruction sequences that depend on the results of previously defined instructions. This is an important concept we will use when defining floating-point conditional moves.

How HotSpot Treats Floating-point Conditional Moves

To decide what enhancements are needed to support floating-point cmoves, we should first see what HotSpot does in cases in which it would emit floating-point cmoves. Consider the following program:

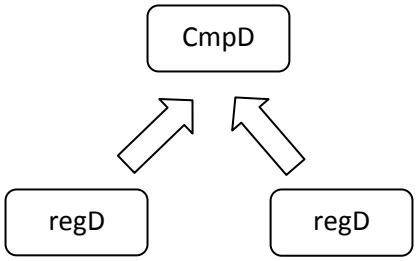
```
public class cmovd {

public static double test(double f) {
   if(f < 10.78) {
     f = 10;
   }
```

```
  return f;
}

public static void main(String[] args) {
    double f = 5.3;
    f = test(f);
}
```

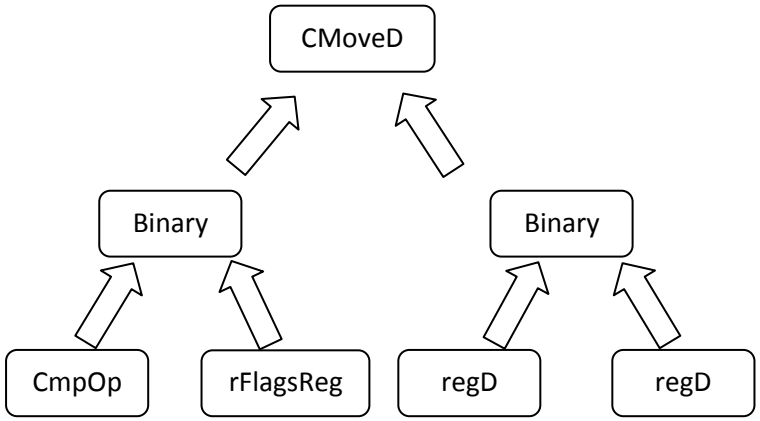
The compiled code HotSpot generates for the *test( )* routine in the above program is:

```
push   %rbp
sub    %rsp, $0x10
nop
movsd  %xmm1,   82(%rip)       // Load 10 in xmm1
movsd  %xmm2 ,  82(%rip)       // Load 10.78 in xmm2

ucomisd %xmm2, %xmm0           // Compare f to 10.78
jbe    0x00002aaaab6eda00      // If f < 10.78 fall through
movapd %xmm0,%xmm1             // Set f = 10
add    %rsp, $0x10
pop    %rbp
```

HotSpot is emitting a floating point compare (*ucomisd*), a jump if less than or equal (*jbe*), and a floating point move (*movapd*). HotSpot emits these three instructions because it first finds a sub-tree in the IR that matches the *ucomisd* instruction:



Then it finds a second sub-tree that matches the *jbe*, *movapd* sequence:



This will become apparent when you examine the parts of the AD file that are emitting the *ucomisd* instruction and *jbe*, *movapd* instructions. The lines in the AD file that define the *ucomisd* instruction are:

```
instruct cmpD_cc_reg_CF(rFlagsRegUCF cr, regD src1, regD src2) %{
  match(Set cr (CmpD src1 src2));
  ins_cost(100);
  format %{ "ucomisd $src1, $src2 test" %}
  ins_encode %{
    __ ucomisd($src1$$XMMRegister, $src2$$XMMRegister);
  %}
  ins_pipe(pipe_slow);
%}
```

The lines that decide what instruction sequence to emit for a cmovD IR node are:

```
instruct cmovD_regUCF(cmpOpUCF cop, rFlagsRegUCF cr, regD dst, regD
src) %{
  match(Set dst (CMoveD (Binary cop cr) (Binary dst src)));
  ins_cost(200);
  expand %{
    cmovD_regU(cop, cr, dst, src);
  %}
%}
```

Notice how the match rules for these two instructions correspond to the IR sub-trees depicted in the previous diagram.

The block with the keyword *expand* means the sequence of native instructions emitted for *cmovD_regUCF* are the same sequence emitted for *cmovD_regU*, another instruct that was previously defined in the same AD file:

```
instruct cmovD_regU(cmpOpU cop, rFlagsRegU cr, regD dst, regD src)
%{
  match(Set dst (CMoveD (Binary cop cr) (Binary dst src)));

  ins_cost(200);
  format %{ "jn$cop    skip\t# unsigned cmove double\n\t"
            "movsd     $dst, $src\n"
    "skip:" %}
  ins_encode(enc_cmovd_branch(cop, dst, src));
  ins_pipe(pipe_slow);
%}
```

The *cmovD_regU* instruct is calling a function *enc_cmovd_branch( )*. A quick look at that function reveals it is generating the code for a conditional jmp instruction followed by a *movapd* instruction:

```
enc_class enc_cmovd_branch(cmpOp cop, regD dst, regD src)
  %{

    // Invert sense of branch from sense of cmov
```

```
    emit_cc(cbuf, 0x70, $cop$$cmpcode ^ 1); // Generate branch
instruction
    emit_d8(cbuf, $dst$$reg < 8 && $src$$reg < 8 ? 4 : 5); // REX

    // UseXmmRegToRegMoveAll ? movapd(dst, src) : movsd(dst, src)
    emit_opcode(cbuf, UseXmmRegToRegMoveAll ? 0x66 : 0xF2);
    if ($dst$$reg < 8) {
      if ($src$$reg >= 8) {
        emit_opcode(cbuf, Assembler::REX_B);
      }
    } else {
      if ($src$$reg < 8) {
        emit_opcode(cbuf, Assembler::REX_R);
      } else {
        emit_opcode(cbuf, Assembler::REX_RB);
      }
    }
    emit_opcode(cbuf, 0x0F);
    emit_opcode(cbuf, UseXmmRegToRegMoveAll ? 0x28 : 0x10);
    emit_rm(cbuf, 0x3, $dst$$reg & 7, $src$$reg & 7);
  %}
```

Floating-point Conditional Moves

Floating-point conditional move instructions are a new feature on the AMD 15h Family of processors. They make use of a new instruction, *vpcmov*, which has the following format:

vpcmov dest, src1, src2, selector

The instruction says, "Set each bit position in *dest* to either the corresponding bit in *src1* or the corresponding bit in *src2*, based on the value of the corresponding bit in *selector*." If the *selector* bit is set to 1, the bit from *src1* is the input; otherwise, the bit from *src2* is the input.

For simplicity, we will focus on the variant of this instruction in which the arguments *dest*, *src1*, *src2*, and *selector* are all floating-point (XMM or YMM) registers. Suppose we want to move the floating-point value in xmm2 into xmm1 conditionally. The syntax for this would be:

vpcmov xmm1, xmm2, xmm1, *selector*

where *selector* is an xmm register whose binary representation contains either all 0's or all 1's. If it contains all 0's, the value of xmm2 will not be moved into xmm1. Otherwise -- that is, if it contains all 1's -- the value of xmm2 will be moved into xmm1.

This selector argument is one of the features that differentiates floating-point cmoves from integer cmoves. The integer cmove instruction reads the condition flags in the rFlags register (previously set by a compare) to determine whether the move should be executed. The floating-point cmove instruction instead reads this selector to identify which bits from the source should be moved into the destination.

As a result, the *vpcmov* instruction cannot use compare instructions that set rFlags. It instead must be preceded by a compare instruction that produces a mask of all 1's or all 0's depending on whether the compare succeeded or failed.

What We Expect Our Code to Look Like

With these observations in mind, let's take another look at our sample program and the compiled code that HotSpot generates for it:

```
public static double test(double f) {

   if(f < 10.78) {
     f = 10;
   }
  return f;
}
  push   %rbp
  sub    %rsp, $0x10
  nop
  movsd  %xmm1,   82(%rip)      // Load 10 in xmm1
  movsd  %xmm2 ,  82(%rip)      // Load 10.78 in xmm2

  ucomisd %xmm2, %xmm0          // Compare f to 10.78
  jbe    0x00002aaaab6eda00     // If f < 10.78 fall through
  movapd %xmm0,%xmm1            // Set f = 10
  add    %rsp, $0x10
  pop    %rbp
```

We want to replace the *ucomisd*, *jbe*, *movapd* instructions with a suitable floating point compare instruction followed by a *vpcmov*. We cannot re-use the *ucomisd* instruction because it sets the rFlags register. An alternative, since we are dealing with double-precision floating-point code, is to use the *cmpsd* (compare scalar double-precision instruction. The format of *cmpsd* is:

```
cmpsd dst, src, imm8
```

This instruction compares the low-order 64 bits of *dst* (which is an XMM register) with the low-order 64 bits of *src* (which can be an XMM register or a memory location) and writes a 64-bit value of all 1's (TRUE) or all 0's (FALSE) into *dst*. The *imm8* parameter is an 8-bit immediate value that specifies the condition code for the compare (e.g., compare less than, compare greater than). Using this instruction would transform our compiled code:

```
ucomisd %xmm2, %xmm0
jbe    0x00002aaaab6eda00
movapd %xmm0,%xmm1
```

```
cmpsd xmm2, xmm0, imm8-nle
vpcmov xmm0, xmm1, xmm0, xmm2
```

The *cmpsd* instruction checks that the value contained in xmm2 (10.78) is not less than or equal to the value contained in xmm1 (f). If this condition is TRUE, then it overwrites the xmm2 register with all 1's;

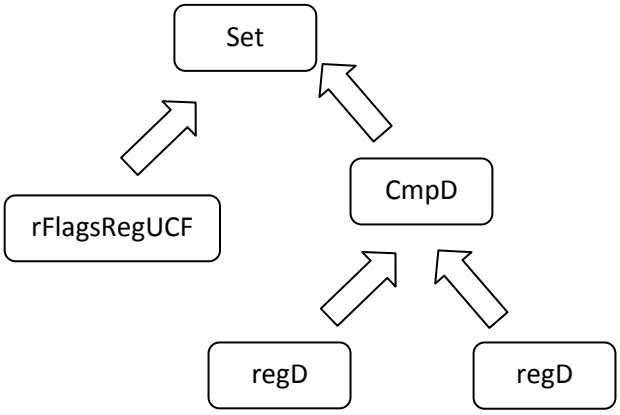
otherwise, it overwrites xmm2 with all 0's. The xmm2 register thereby becomes the selector argument for the *vpcmov* instruction, which conditionally moves the value 10 (from xmm1) into f (xmm0).

Let's consider the changes to the AD file necessary to transform the code in this manner.

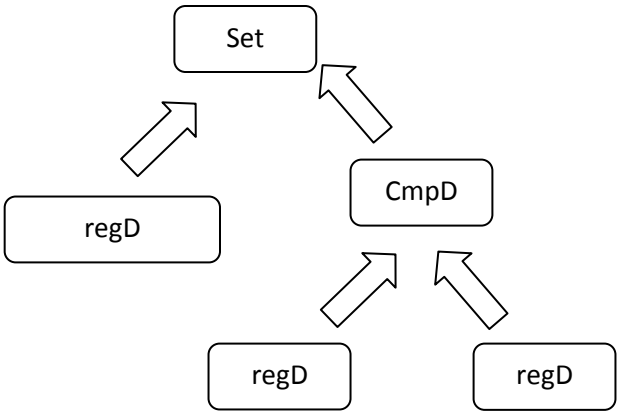
Step One: Define the Match Rule

We want to define a new instruct that emits a *cmpsd*, *vpcmov* sequence in place of *ucomisd*, *jbe*, *movapd*. The first step in this process is to work out the match rule for this instruct. We do this by looking at the match rules for *ucomisd* and *cmovd* to determine what needs to be different.

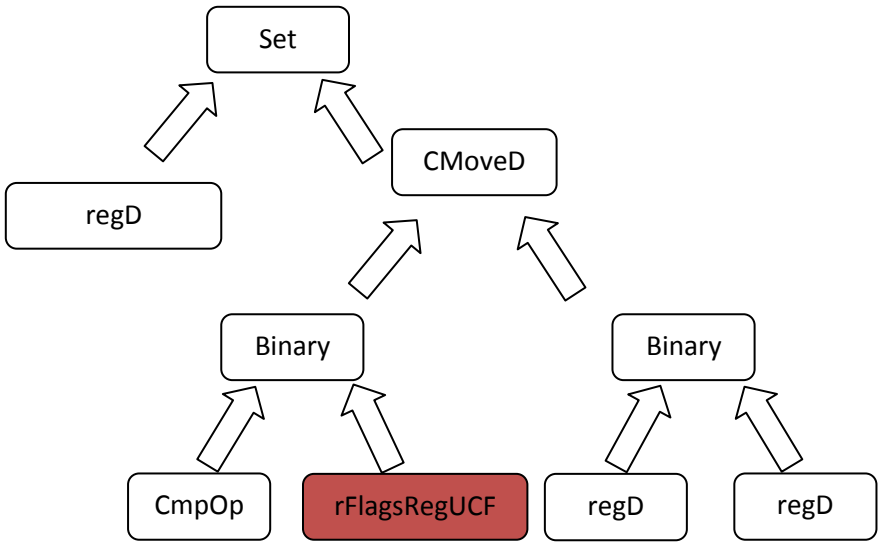
Recall the match-rule for the *ucomisd* instruction:



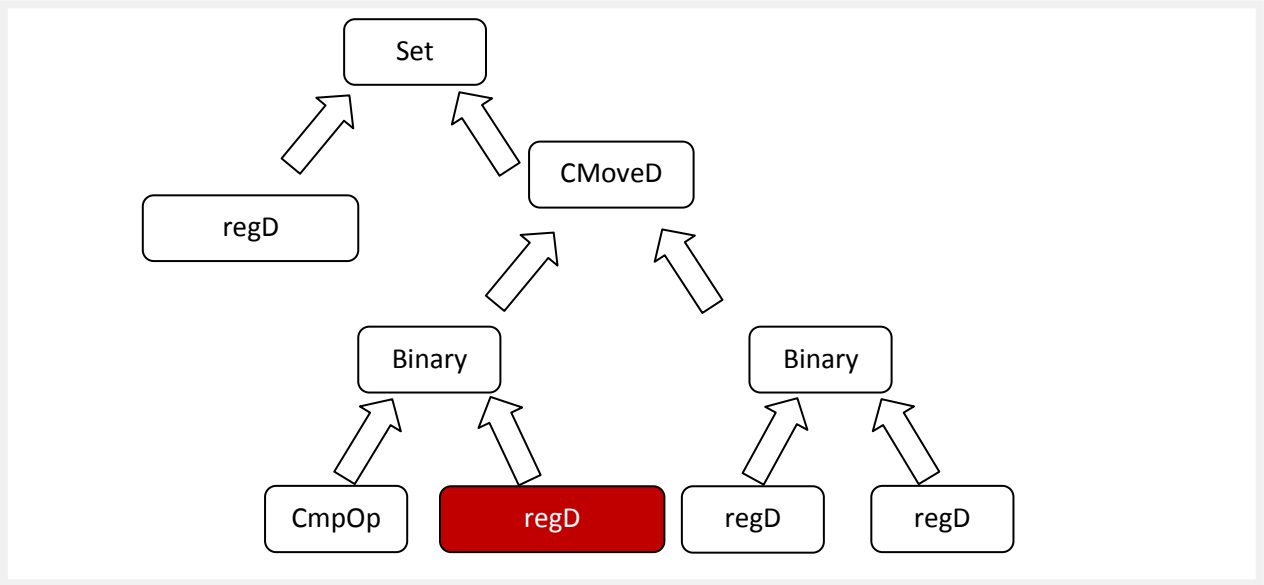
The *cmpsd* instruction writes its result to a floating-point register instead of modifying rFlagsRegUCF. Thus the destination of the *cmpsd* match rule would have regD in place of rFlagsRegUCF:



Now look at the match rule that HotSpot currently uses to identify the cases in which a floating-point (double-precision) cmove would be appropriate:

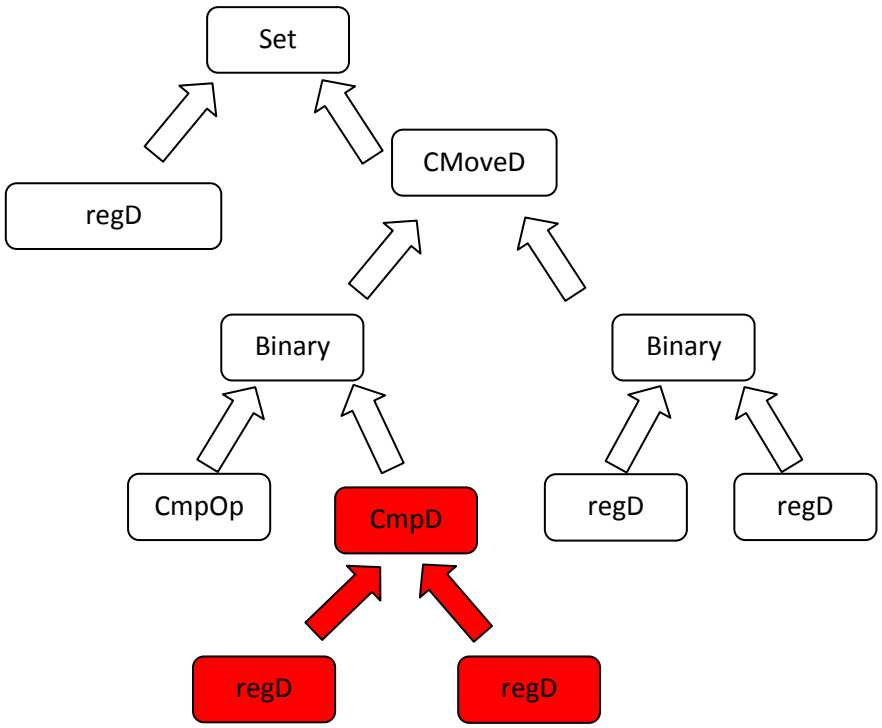


One of the inputs to the CMoveD node is an rFlagsRegUCF node, which was the result of a previous *ucomisd* instruction. If HotSpot were to generate a *cmpsd* instruction rather than an *ucomisd* instruction, then the result of that instruction would be of type regD rather than rFlagsRegUCF. Thus, to detect cases in which a *cmpsd* instruction was generated, the match rule for *vpcmov* would have to be:



At this point, we may wonder whether we can define two separate instructs, one for *cmpsd* and another for *vpcmov*. One subtlety prevents us from doing this. The *cmpsd* instruction requires an immediate value that indicates the sense of the compare (greater than, less than, etc.). This value must be derived from the CmpOp argument to CMoveD. If we were to define the *cmpsd* instruction and *vpcmov* instruction using two separate instructs in the AD file, we would not be able to pass this CmpOp operand to the function that encodes *cmpsd*. Thus we have to define a single instruct for emitting the *cmpsd*, *vpcmov* sequence. We would get the match rule for this instruct by substituting the source of the *cmpsd*

match rule for the second regD argument of the *vpcmov* match rule. With this final change, our match rule for emitting *cmpsd*, *vpcmov* becomes:



Step Two: Define the Instruct

Now that we have worked out the match rule, we are ready to define our new instruct that emits *cmpsd*, *vpcmov*:

```
instruct vpcmovD_regUCF(cmpOpUCF cop, regD cmp_src1, regD cmp_src2,
regD dst, regD src)
%{
  predicate(UseVectorCMove && ((_kids[0]->_kids[0]->_leaf->as_Bool()-
>_test._test == BoolTest::lt)||
(_kids[0]->_kids[0]->_leaf->as_Bool()->_test._test == BoolTest::eq) ||
(_kids[0]->_kids[0]->_leaf->as_Bool()->_test._test == BoolTest::le) ||
(_kids[0]->_kids[0]->_leaf->as_Bool()->_test._test == BoolTest::ne) ||
(_kids[0]->_kids[0]->_leaf->as_Bool()->_test._test == BoolTest::gt) ||
(_kids[0]->_kids[0]->_leaf->as_Bool()->_test._test == BoolTest::ge)));
  match(Set dst (CMoveD (Binary cop (CmpD cmp_src1 cmp_src2)) (Binary
dst src)));
  ins_cost(100);
  format %{ "jn$cop    skip\t# unsigned cmove double\n\t"
            "movsd     $dst, $src\n"
    "skip:" %}
  ins_encode %{
```

```
  if($cop$$cmpcode == Assembler::equal) {
   __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,0);
  }
  else if($cop$$cmpcode == Assembler::less) {
  __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,1);
 }
  else if($cop$$cmpcode == Assembler::lessEqual) {
    __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,2);
  }
  else if($cop$$cmpcode == Assembler::notEqual) {
    __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,4);
  }
  else if($cop$$cmpcode == Assembler::greater) {
   __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,6);
  }
  else if($cop$$cmpcode == Assembler::greaterEqual) {
   __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,5);
  }
  else if($cop$$cmpcode == Assembler::below) {
   __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,1);
  }
  else if($cop$$cmpcode == Assembler::belowEqual) {
   __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,2);
  }
  else if($cop$$cmpcode == Assembler::above) {
  __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,6);
  }
  else if($cop$$cmpcode == Assembler::aboveEqual) {
  __ cmpsd($cmp_src1$$XMMRegister, $cmp_src2$$XMMRegister,5);
  }
  __ vpcmov($dst$$XMMRegister, $src$$XMMRegister, $dst$$XMMRegister,
$cmp_src1$$XMMRegister);
  %}
  ins_pipe(pipe_slow);
%}
```

The *instruct* parameters include a *cmpOpUCF (cop)* parameter from which the immediate value for the *cmpsd* instruction will be derived, the two source registers to the *cmpsd* instruction (*cmp_src1*, *cmp_src2*), and the source and destination registers for the *vpcmov* instruction (*src*, *dest*).

The line that says *predicate(…)* tells the JVM to generate this instruction sequence only under specific conditions. The conditions are that the command line option *UseVectorCMove* has been enabled and the cop parameter has one of the six bool values (=,>,>=,<,<=, !=) for which the *cmpsd* instruction can be emitted. The *_kids[]* construct allows us to refer to nodes in the match tree within the predicate construct. For each node in the tree, the *_kids[]* array gives its children. For example, in this match tree for the *cmpsd*, *vpcmovd* sequence, *_kids[0]->_kids[0]* would refer to the CmpOp node, whereas *_kids[0]->_kids[1]->_kids[0]* would refer to the first regD node that is an input to the CmpD node.

To ensure our new instructions are emitted only when the *UseVectorCMove* command line option is enabled, we defined this new option in src/share/vm/runtime/globals.hpp by adding the lines:

```
  product(bool, UseVectorCMove, false,                                      \
          "Emits AVX Conditional Move on supported hw")                     \
```

The line *ins_cost(100)* assigns a cost of 100 to this instruction sequence. We assigned a cost of 100 so HotSpot chooses this sequence over the *ucomisd*, *jbe*, *movapd* instruction sequence (which has a cost of 200).

The if statements parse the cmpOp parameter to determine the immediate value that should be passed to the *cmpsd* instruction to indicate the condition code for the compare. The mapping between these immediate values and the condition codes is specified in the *AMD64 Architecture Programmer's Manual Volume 4: 128-bit Media Instructions*:

Mnemonic	**Implied value of immediate8**
CMPEQSD	0
CMPLTSD	1
CMPLESD	2
CMPUNORDSD	3
CMPNEQSD	4
CMPNLTSD	5
CMPNLESD	6

There is a subtlety here. HotSpot has its own way of mapping immediate values to condition codes, which is different from the mapping specified in this table. Thus, we cannot use the unmodified condition code of the cmpOp parameter as the immediate value for the *cmpsd* instruction. Instead, we need to find out what condition code HotSpot is emitting, and -- based on its value -- pass one of the immediate8 values from this table to the *cmpsd* instruction. This is what the if statements are doing.

The final line in the *ins_encode* block calls an assembler routine that emits the *vpcmov* instruction.

Step Three: Modify the Assembler

Once you have defined an instruction sequence and its associated match pattern in the AD file, the final step is to define the assembler routines that encode the instruction. You add these new routines to the file src/cpu/x86/vm/assembler_x86.cpp.

Here is the routine that encodes the *cmpsd* instruction:

```
void Assembler::cmpsd(XMMRegister dst, XMMRegister src, int imm8) {
  NOT_LP64(assert(VM_Version::supports_sse2(), ""));
  int encode = prefix_and_encode(dst->encoding(), src->encoding());
  emit_byte(0xF2);
  emit_byte(0x0F);
```

```
  emit_byte(0xC2);
  emit_byte(0xC0 | encode);
  emit_byte(imm8);
}
```

Similarly, we have added a routine that encodes the reg-to-reg form of the vpcmov instruction. For details on the encoding, refer to the *AMD64 Architecture Programmer's Manual Volume 6: 128-bit and 256-bit XOP and FMA4 Instructions*.

```
void Assembler::vpcmov(XMMRegister dst, XMMRegister src1, XMMRegister src2,
XMMRegister selector) {
  int XOP_RXB = 14; // R, X, AND B bits enabled.
  int XOP_X = 4;    // R and B bits disabled => dest=1000 | dest, src2 =
1000b | src2
  int XOP_XB = 6;   // R bit disabled. dest = 1000b | dest
  int XOP_RX = 12;  // B bit disabled src2 = 1000b | src2
  emit_byte(0x8F); // XOP byte
  int dstenc = dst->encoding();
  int src1enc = src1->encoding();
  int src2enc = src2->encoding();
  int selectenc = selector->encoding();
  if(dstenc < 8) {
    if(src2enc >= 8) {
      emit_byte((XOP_RX << 4) | 0x08);
    }
    else {
      emit_byte((XOP_RXB << 4) | 0x08);
    }
  }
  else {
    if(src2enc >= 8) {
      emit_byte((XOP_X << 4) | 0x08);
    }
    else {
      emit_byte((XOP_XB << 4) | 0x08);
    }
  }
  emit_byte((~src1enc&0x0f) << 3);
  emit_byte(0xA2); //opcode for vpcmovd
  emit_byte(0xC0 | dstenc << 3 | src2enc);
  emit_byte(selectenc << 4);
}
```

Step Four: Test the New Instructions

Once you have modified HotSpot to emit a new instruction sequence for a pattern in the IR, the final step is to make sure your new instructions are emitted. The best way to do this is to attach a disassembler to HotSpot and view the generated code for a test program.

The HotSpot *hsdis* disassembler can be invoked by specifying the *–XX:+PrintAssembly* option followed by the *–XX: PrintAssemblyOptions=hsdis-print-bytes* option on the command line. The first option enables the disassembler and the second option causes the disassembler to output the bytes for each instruction. This is important because the disassembler may not understand the mnemonics for new or unsupported instructions. However, you can still verify that the new instructions are being emitted by checking the bytes.

Let's do this for our test program:

```
public class cmovd {

public static double test(double f) {
   if(f < 10.78) {
     f = 10;
   }
  return f;
}

public static void main(String[] args) {
    double f = 5.3;
    f = test(f);
}
```

Here is the compiled code HotSpot previously generated for the *test( )* routine:

```
  push   %rbp
  sub    %rsp, $0x10
  nop
  movsd  %xmm1,   82(%rip)     // Load 10 in xmm1
  movsd  %xmm2 ,  82(%rip)     // Load 10.78 in xmm2

  ucomisd %xmm2, %xmm0         // Compare f to 10.78
  jbe    0x00002aaaab6eda00    // If f < 10.78 fall through
  movapd %xmm0,%xmm1           // Set f = 10
  add    %rsp, $0x10
  pop    %rbp
```

To run our test program with our new floating-point cmove optimization enabled, we would add the flag *–XX:+UseVectorCMove* to the command line. Here is the newly generated code:

```
  push   %rbp
  sub    %rsp, $0x10,
  nop
  movsd  %xmm1, 82(%rip)     // Load 10 in xmm1
  movsd  %xmm2, 82(%rip)     // Load 10.78 in xmm2
  cmpnlesd %xmm2,%xmm0       ;...f20fc2d0 06      // f < 10.78?
  pop    %rax                ;...8fe8
  jo     0x00002aaaab6edf65  ;...70a2
  shlb   (%rax), $0x48       ;...c02048
  add    %esp, $0x10         ;...83c410
  pop    %rbp                ;...5d
```

To the right of each instruction you can see the bytes being displayed. For brevity, we have included only the bytes for the code fragment that interest us. HotSpot emits a *cmpsd* instruction in place of *ucomisd*. The version of the *cmpsd* instruction it emits is:

```
// Check that xmm2 (10.78) is not less than or equal to xmm0 (f)

cmpsd xmm2, xmm0, imm8-nle
```

The following instruction it is actually a *vpcmov* instruction, even though it appears as *pop*, *jo*, *shlb*. We would verify this by examining the bytes emitted, 8fea70a2c020. These bytes indicate the encoding of:

```
// Assign xmm0 (f) the value in xmm1 (10) if f < 10.78

vpcmov xmm0, xmm1, xmm0, xmm2
```

which is the version of *vpcmov* we want.

Additional Issues

We may think we're done, but there is a catch.

In our sample program, the if clause contained the conditional *if(f < 10.78)*. What if the conditional were instead *if(f > 10.78)*? The compiled code that HotSpot generates in this case is:

```
push    %rbp
sub     %rsp, $0x10,
nop
movsd   %xmm1, 82(%rip)      // Load 10 in xmm1
movsd   %xmm2, 82(%rip)      // Load 10.78 in xmm2
cmpnlesd %xmm0,%xmm2         ;.. .f20fc2c2 06      // 10.78 < f?
pop     %rax                 ;...8fe8
jo      0x00002aaaab6edf65   ;...70a2
shlb    (%rax), $0x48        ;...c00048
add     %esp, $0x10           ;...83c410
pop     %rbp                 ;...5d
```

The highlighted code region corresponds to the instructions:

```
// Check that xmm0 (f) is not less than or equal to xmm2 (10.78)
cmpsd xmm0, xmm2, imm8-nle

// Assign xmm0 (f) the value in xmm1 (10) if f > 10.78
vpcmov xmm0, xmm1, xmm0, xmm0
```

Instead of inverting the condition code for *cmpsd* (by emitting *cmplesd* instead of *cmpnlesd*), HotSpot emits the same *cmpnlesd* instruction but swaps the order in which the parameters are passed to it. This also overwrites xmm0, which is the second source register for *vpcmov*. As a result, in cases when we want the conditional move operation to fail, the destination register (xmm0) will get the wrong result.

This is not a bug in HotSpot but a general problem we need to watch for when writing complex encodings. There are two ways to change a compare instruction from a "compare less than" to a "compare greater than." The first is to invert the condition code passed to the instruction. The second is to swap the order of the operands. Before deciding which floating point compare instruction to emit, we should have run experiments to see which of these two cases applies to HotSpot.

It turns out the solution is to use the three-operand version of *cmpsd*, which writes its result into a separate destination register instead of overwriting one of the two source registers. Here is pseudo-code that illustrates how the above code sequence might look with this change:

```
// Check that xmm0 (f) is not less than or equal to xmm2 (10.78)
 vcmpsd xmm3, xmm0, xmm2, imm8-nle

// Assign xmm0 (f) the value in xmm1 (10) if f >10.78
 vpcmov xmm0, xmm1, xmm0, xmm3
```

Emitting this three-operand *vcmpsd* instead of *cmpsd* requires minor changes to the code we have written so far. The match rule will be the same. Because the *vcmpsd* instruction has a different encoding from *cmpsd* and requires an extra parameter for a destination register, we will have to update our instruct definition and define a new assembler routine to encode *vcmpsd*. We leave it as an exercise for the reader to work out the details of these changes.

Conclusion

This paper explained how to add new instructions to the Oracle® Java HotSpot Virtual Machine, using as a running example the addition of an instruction sequence for a floating-point conditional move optimization. Adding support for new instructions requires defining the match rules that determine when these new instructions will be emitted, defining the required instruct(s) in the AD file, and defining assembler routines that encode the instruction(s). The techniques outlined in this article apply to any architecture.

Acknowledgements

We thank the engineers of the Oracle® HotSpot team, especially David Cox and Tom Rodriguez, for their timely review of this article on short notice.