**********
***12867 Fri Oct 28 01:27:08 2011***
***new/src/cpu/sparc/vm/vm_version_sparc.cpp***
**********

```
   1 /*
   2  * Copyright (c) 1997, 2011, Oracle and/or its affiliates. All rights reserved.
   2  * Copyright (c) 1997, 2010, Oracle and/or its affiliates. All rights reserved.
   3  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
   4  *
   5  * This code is free software; you can redistribute it and/or modify it
   6  * under the terms of the GNU General Public License version 2 only, as
   7  * published by the Free Software Foundation.
   8  *
   9  * This code is distributed in the hope that it will be useful, but WITHOUT
  10  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
  11  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
  12  * version 2 for more details (a copy is included in the LICENSE file that
  13  * accompanied this code).
  14  *
  15  * You should have received a copy of the GNU General Public License version
  16  * 2 along with this work; if not, write to the Free Software Foundation,
  17  * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
  18  *
  19  * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
  20  * or visit www.oracle.com if you need additional information or have any
  21  * questions.
  22  *
  23  */

  25 #include "precompiled.hpp"
  26 #include "assembler_sparc.inline.hpp"
  27 #include "memory/resourceArea.hpp"
  28 #include "runtime/java.hpp"
  29 #include "runtime/stubCodeGenerator.hpp"
  30 #include "vm_version_sparc.hpp"
  31 #ifdef TARGET_OS_FAMILY_linux
  32 # include "os_linux.inline.hpp"
  33 #endif
  34 #ifdef TARGET_OS_FAMILY_solaris
  35 # include "os_solaris.inline.hpp"
  36 #endif

  38 int VM_Version::_features = VM_Version::unknown_m;
  39 const char* VM_Version::_features_str = "";

  41 void VM_Version::initialize() {
  42   _features = determine_features();
  43   PrefetchCopyIntervalInBytes = prefetch_copy_interval_in_bytes();
  44   PrefetchScanIntervalInBytes = prefetch_scan_interval_in_bytes();
  45   PrefetchFieldsAhead         = prefetch_fields_ahead();

  47   assert(0 <= AllocatePrefetchInstr && AllocatePrefetchInstr <= 1, "invalid valu
  48   if( AllocatePrefetchInstr < 0 ) AllocatePrefetchInstr = 0;
  49   if( AllocatePrefetchInstr > 1 ) AllocatePrefetchInstr = 0;

  51   // Allocation prefetch settings
  52   intx cache_line_size = prefetch_data_size();
  53   if( cache_line_size > AllocatePrefetchStepSize )
  54     AllocatePrefetchStepSize = cache_line_size;

  56   assert(AllocatePrefetchLines > 0, "invalid value");
  57   if( AllocatePrefetchLines < 1 )     // set valid value in product VM
  58     AllocatePrefetchLines = 3;
  59   assert(AllocateInstancePrefetchLines > 0, "invalid value");
  60   if( AllocateInstancePrefetchLines < 1 ) // set valid value in product VM
  61     AllocateInstancePrefetchLines = 1;
```



```
  63   AllocatePrefetchDistance = allocate_prefetch_distance();
  64   AllocatePrefetchStyle    = allocate_prefetch_style();

  66   assert((AllocatePrefetchDistance % AllocatePrefetchStepSize) == 0 &&
  67          (AllocatePrefetchDistance > 0), "invalid value");
  68   if ((AllocatePrefetchDistance % AllocatePrefetchStepSize) != 0 ||
  69       (AllocatePrefetchDistance <= 0)) {
  70     AllocatePrefetchDistance = AllocatePrefetchStepSize;
  71   }

  73   if (AllocatePrefetchStyle == 3 && !has_blk_init()) {
  74     warning("BIS instructions are not available on this CPU");
  75     FLAG_SET_DEFAULT(AllocatePrefetchStyle, 1);
  76   }

  78   if (has_v9()) {
  79     assert(ArraycopySrcPrefetchDistance < 4096, "invalid value");
  80     if (ArraycopySrcPrefetchDistance >= 4096)
  81       ArraycopySrcPrefetchDistance = 4064;
  82     assert(ArraycopyDstPrefetchDistance < 4096, "invalid value");
  83     if (ArraycopyDstPrefetchDistance >= 4096)
  84       ArraycopyDstPrefetchDistance = 4064;
  85   } else {
  86     if (ArraycopySrcPrefetchDistance > 0) {
  87       warning("prefetch instructions are not available on this CPU");
  88       FLAG_SET_DEFAULT(ArraycopySrcPrefetchDistance, 0);
  89     }
  90     if (ArraycopyDstPrefetchDistance > 0) {
  91       warning("prefetch instructions are not available on this CPU");
  92       FLAG_SET_DEFAULT(ArraycopyDstPrefetchDistance, 0);
  93     }
  94   }

  96   UseSSE = 0; // Only on x86 and x64

  98   _supports_cx8 = has_v9();

 100   if (is_niagara()) {
 101     // Indirect branch is the same cost as direct
 102     if (FLAG_IS_DEFAULT(UseInlineCaches)) {
 103       FLAG_SET_DEFAULT(UseInlineCaches, false);
 104     }
 105     // Align loops on a single instruction boundary.
 106     if (FLAG_IS_DEFAULT(OptoLoopAlignment)) {
 107       FLAG_SET_DEFAULT(OptoLoopAlignment, 4);
 108     }
 109     // When using CMS, we cannot use memset() in BOT updates because
 110     // the sun4v/CMT version in libc_psr uses BIS which exposes
 111     // "phantom zeros" to concurrent readers. See 6948537.
 112     if (FLAG_IS_DEFAULT(UseMemSetInBOT) && UseConcMarkSweepGC) {
 113       FLAG_SET_DEFAULT(UseMemSetInBOT, false);
 114     }
 115 #ifdef _LP64
 116     // 32-bit oops don't make sense for the 64-bit VM on sparc
 117     // since the 32-bit VM has the same registers and smaller objects.
 118     Universe::set_narrow_oop_shift(LogMinObjAlignmentInBytes);
 119 #endif // _LP64
 120 #ifdef COMPILER2
 121     // Indirect branch is the same cost as direct
 122     if (FLAG_IS_DEFAULT(UseJumpTables)) {
 123       FLAG_SET_DEFAULT(UseJumpTables, true);
 124     }
 125     // Single-issue, so entry and loop tops are
 126     // aligned on a single instruction boundary
 127     if (FLAG_IS_DEFAULT(InteriorEntryAlignment)) {
```

```
128       FLAG_SET_DEFAULT(InteriorEntryAlignment, 4);
129     }
130     if (is_niagara_plus()) {
131       if (has_blk_init() && UseTLAB &&
132           FLAG_IS_DEFAULT(AllocatePrefetchInstr)) {
133         // Use BIS instruction for TLAB allocation prefetch.
134         FLAG_SET_ERGO(intx, AllocatePrefetchInstr, 1);
135         if (FLAG_IS_DEFAULT(AllocatePrefetchStyle)) {
136           FLAG_SET_ERGO(intx, AllocatePrefetchStyle, 3);
137         }
138         if (FLAG_IS_DEFAULT(AllocatePrefetchDistance)) {
139           // Use smaller prefetch distance with BIS
140           FLAG_SET_DEFAULT(AllocatePrefetchDistance, 64);
141         }
142       }
143       if (is_T4()) {
144         // Double number of prefetched cache lines on T4
145         // since L2 cache line size is smaller (32 bytes).
146         if (FLAG_IS_DEFAULT(AllocatePrefetchLines)) {
147           FLAG_SET_ERGO(intx, AllocatePrefetchLines, AllocatePrefetchLines*2);
148         }
149         if (FLAG_IS_DEFAULT(AllocateInstancePrefetchLines)) {
150           FLAG_SET_ERGO(intx, AllocateInstancePrefetchLines, AllocateInstancePre
151         }
152       }
153       if (AllocatePrefetchStyle != 3 && FLAG_IS_DEFAULT(AllocatePrefetchDistance
154         // Use different prefetch distance without BIS
155         FLAG_SET_DEFAULT(AllocatePrefetchDistance, 256);
156       }
157       if (AllocatePrefetchInstr == 1) {
158         // Need a space at the end of TLAB for BIS since it
159         // will fault when accessing memory outside of heap.
160
161         // +1 for rounding up to next cache line, +1 to be safe
162         int lines = AllocatePrefetchLines + 2;
163         int step_size = AllocatePrefetchStepSize;
164         int distance = AllocatePrefetchDistance;
165         _reserve_for_allocation_prefetch = (distance + step_size*lines)/(int)Hea
166       }
167     }
168 #endif
169   }
170
171   // Use hardware population count instruction if available.
172   if (has_hardware_popc()) {
173     if (FLAG_IS_DEFAULT(UsePopCountInstruction)) {
174       FLAG_SET_DEFAULT(UsePopCountInstruction, true);
175     }
176   } else if (UsePopCountInstruction) {
177     warning("POPC instruction is not available on this CPU");
178     FLAG_SET_DEFAULT(UsePopCountInstruction, false);
179   }
180
181   // T4 and newer Sparc cpus have new compare and branch instruction.
182   if (has_cbcond()) {
183     if (FLAG_IS_DEFAULT(UseCBCond)) {
184       FLAG_SET_DEFAULT(UseCBCond, true);
185     }
186   } else if (UseCBCond) {
187     warning("CBCOND instruction is not available on this CPU");
188     FLAG_SET_DEFAULT(UseCBCond, false);
189   }
190
191   assert(BlockZeroingLowLimit > 0, "invalid value");
192   if (has_block_zeroing()) {
193     if (FLAG_IS_DEFAULT(UseBlockZeroing)) {
```



```
194       FLAG_SET_DEFAULT(UseBlockZeroing, true);
195     }
196   } else if (UseBlockZeroing) {
197     warning("BIS zeroing instructions are not available on this CPU");
198     FLAG_SET_DEFAULT(UseBlockZeroing, false);
199   }
200
201   assert(BlockCopyLowLimit > 0, "invalid value");
202   if (has_block_zeroing()) { // has_blk_init() && is_T4(): core's local L2 cache
203     if (FLAG_IS_DEFAULT(UseBlockCopy)) {
204       FLAG_SET_DEFAULT(UseBlockCopy, true);
205     }
206   } else if (UseBlockCopy) {
207     warning("BIS instructions are not available or expensive on this CPU");
208     FLAG_SET_DEFAULT(UseBlockCopy, false);
209   }
210
211 #ifdef COMPILER2
212   // T4 and newer Sparc cpus have fast RDPC.
213   if (has_fast_rdpc() && FLAG_IS_DEFAULT(UseRDPCForConstantTableBase)) {
214     FLAG_SET_DEFAULT(UseRDPCForConstantTableBase, true);
215 //    FLAG_SET_DEFAULT(UseRDPCForConstantTableBase, true);
216   }
217
218   // Currently not supported anywhere.
219   FLAG_SET_DEFAULT(UseFPUForSpilling, false);
220
221   assert((InteriorEntryAlignment % relocInfo::addr_unit()) == 0, "alignment is n
222 #endif
223
224   assert((CodeEntryAlignment % relocInfo::addr_unit()) == 0, "alignment is not a
225   assert((OptoLoopAlignment % relocInfo::addr_unit()) == 0, "alignment is not a
226
227   char buf[512];
228   jio_snprintf(buf, sizeof(buf), "%s%s%s%s%s%s%s%s%s%s%s%s%s%s",
229                (has_v9() ? ", v9" : (has_v8() ? ", v8" : "")),
230                (has_hardware_popc() ? ", popc" : ""),
231                (has_vis1() ? ", vis1" : ""),
232                (has_vis2() ? ", vis2" : ""),
233                (has_vis3() ? ", vis3" : ""),
234                (has_blk_init() ? ", blk_init" : ""),
235                (has_cbcond() ? ", cbcond" : ""),
236                (is_ultra3() ? ", ultra3" : ""),
237                (is_sun4v() ? ", sun4v" : ""),
238                (is_niagara_plus() ? ", niagara_plus" : (is_niagara() ? ", niagar
239                (is_sparc64() ? ", sparc64" : ""),
240                (!has_hardware_mul32() ? ", no-mul32" : ""),
241                (!has_hardware_div32() ? ", no-div32" : ""),
242                (!has_hardware_fsmuld() ? ", no-fsmuld" : ""));
243
244   // buf is started with ", " or is empty
245   _features_str = strdup(strlen(buf) > 2 ? buf + 2 : buf);
246
247   // UseVIS is set to the smallest of what hardware supports and what
248   // the command line requires.  I.e., you cannot set UseVIS to 3 on
249   // older UltraSparc which do not support it.
250   if (UseVIS > 3) UseVIS=3;
251   if (UseVIS < 0) UseVIS=0;
252   if (!has_vis3()) // Drop to 2 if no VIS3 support
253     UseVIS = MIN2((intx)2,UseVIS);
254   if (!has_vis2()) // Drop to 1 if no VIS2 support
255     UseVIS = MIN2((intx)1,UseVIS);
256   if (!has_vis1()) // Drop to 0 if no VIS1 support
257     UseVIS = 0;
258
259 #ifndef PRODUCT
```

```
 259   if (PrintMiscellaneous && Verbose) {
 260     tty->print("Allocation");
 261     if (AllocatePrefetchStyle <= 0) {
 262       tty->print_cr(": no prefetching");
 263     } else {
 264       tty->print(" prefetching: ");
 265       if (AllocatePrefetchInstr == 0) {
 266           tty->print("PREFETCH");
 267       } else if (AllocatePrefetchInstr == 1) {
 268           tty->print("BIS");
 269       }
 270       if (AllocatePrefetchLines > 1) {
 271         tty->print_cr(" at distance %d, %d lines of %d bytes", AllocatePrefetchD
 272       } else {
 273         tty->print_cr(" at distance %d, one line of %d bytes", AllocatePrefetchD
 274       }
 275     }
 276     if (PrefetchCopyIntervalInBytes > 0) {
 277       tty->print_cr("PrefetchCopyIntervalInBytes %d", PrefetchCopyIntervalInByte
 278     }
 279     if (PrefetchScanIntervalInBytes > 0) {
 280       tty->print_cr("PrefetchScanIntervalInBytes %d", PrefetchScanIntervalInByte
 281     }
 282     if (PrefetchFieldsAhead > 0) {
 283       tty->print_cr("PrefetchFieldsAhead %d", PrefetchFieldsAhead);
 284     }
 285   }
 286 #endif // PRODUCT
 287 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   27118 Fri Oct 28 01:27:09 2011
new/src/share/vm/opto/machnode.cpp
**********************************************************
______unchanged_portion_omitted_
 478 #endif


 481 //=============================================================================
 482 int MachConstantNode::constant_offset() {
 483   int offset = _constant.offset();
 484   // Bind the offset lazily.
 485   if (offset == -1) {
 486     Compile::ConstantTable& constant_table = Compile::current()->constant_table(
 487     // If called from Compile::scratch_emit_size assume the worst-case
 488     // for load offsets: half the constant table size.
 489     // NOTE: Don't return or calculate the actual offset (which might
 490     // be zero) because that leads to problems with e.g. jumpXtnd on
 491     // some architectures (cf. add-optimization in SPARC jumpXtnd).
 492     if (Compile::current()->in_scratch_emit_size())
 493       return constant_table.size() / 2;
 494 #endif /* ! codereview */
 495     offset = constant_table.table_base_offset() + constant_table.find_offset(_co
 496     _constant.set_offset(offset);
 497   }
 498   return offset;
 499 }


 502 //=============================================================================
 503 #ifndef PRODUCT
 504 void MachNullCheckNode::format( PhaseRegAlloc *ra_, outputStream *st ) const {
 505   int reg = ra_->get_reg_first(in(1)->in(_vidx));
 506   tty->print("%s %s", Name(), Matcher::regName[reg]);
 507 }
 508 #endif

 510 void MachNullCheckNode::emit(CodeBuffer &cbuf, PhaseRegAlloc *ra_) const {
 511   // only emits entries in the null-pointer exception handler table
 512 }
 513 void MachNullCheckNode::label_set(Label* label, uint block_num) {
 514   // Nothing to emit
 515 }
 516 void MachNullCheckNode::save_label( Label** label, uint* block_num ) {
 517   // Nothing to emit
 518 }

 520 const RegMask &MachNullCheckNode::in_RegMask( uint idx ) const {
 521   if( idx == 0 ) return RegMask::Empty;
 522   else return in(1)->as_Mach()->out_RegMask();
 523 }

 525 //=============================================================================
 526 const Type *MachProjNode::bottom_type() const {
 527   if( _ideal_reg == fat_proj ) return Type::BOTTOM;
 528   // Try the normal mechanism first
 529   const Type *t = in(0)->bottom_type();
 530   if( t->base() == Type::Tuple ) {
 531     const TypeTuple *tt = t->is_tuple();
 532     if (_con < tt->cnt())
 533       return tt->field_at(_con);
 534   }
 535   // Else use generic type from ideal register set
 536   assert((uint)_ideal_reg < (uint)_last_machine_leaf && Type::mreg2type[_ideal_r
 537   return Type::mreg2type[_ideal_reg];
 538 }
```



```
540 const TypePtr *MachProjNode::adr_type() const {
541   if (bottom_type() == Type::MEMORY) {
542     // in(0) might be a narrow MemBar; otherwise we will report TypePtr::BOTTOM
543     const TypePtr* adr_type = in(0)->adr_type();
544     #ifdef ASSERT
545     if (!is_error_reported() && !Node::in_dump())
546       assert(adr_type != NULL, "source must have adr_type");
547     #endif
548     return adr_type;
549   }
550   assert(bottom_type()->base() != Type::Memory, "no other memories?");
551   return NULL;
552 }

554 #ifndef PRODUCT
555 void MachProjNode::dump_spec(outputStream *st) const {
556   ProjNode::dump_spec(st);
557   switch (_ideal_reg) {
558   case unmatched_proj:  st->print("/unmatched");                          break;
559   case fat_proj:        st->print("/fat"); if (WizardMode) _rout.dump(); break;
560   }
561 }
562 #endif

564 //=============================================================================
565 #ifndef PRODUCT
566 void MachIfNode::dump_spec(outputStream *st) const {
567   st->print("P=%f, C=%f",_prob, _fcnt);
568 }
569 #endif

571 //=============================================================================
572 uint MachReturnNode::size_of() const { return sizeof(*this); }

574 //-------------------------------Registers-------------------------------------
575 const RegMask &MachReturnNode::in_RegMask( uint idx ) const {
576   return _in_rms[idx];
577 }

579 const TypePtr *MachReturnNode::adr_type() const {
580   // most returns and calls are assumed to consume & modify all of memory
581   // the matcher will copy non-wide adr_types from ideal originals
582   return _adr_type;
583 }

585 //=============================================================================
586 const Type *MachSafePointNode::bottom_type() const {  return TypeTuple::MEMBAR;

588 //-------------------------------Registers-------------------------------------
589 const RegMask &MachSafePointNode::in_RegMask( uint idx ) const {
590   // Values in the domain use the users calling convention, embodied in the
591   // _in_rms array of RegMasks.
592   if( idx < TypeFunc::Parms ) return _in_rms[idx];

594   if (SafePointNode::needs_polling_address_input() &&
595       idx == TypeFunc::Parms &&
596       ideal_Opcode() == Op_SafePoint) {
597     return MachNode::in_RegMask(idx);
598   }

600   // Values outside the domain represent debug info
601   return *Compile::current()->matcher()->idealreg2spillmask[in(idx)->ideal_reg()
602 }
```

```
605 //=============================================================================

607 uint MachCallNode::cmp( const Node &n ) const
608 { return _tf == ((MachCallNode&)n)._tf; }
609 const Type *MachCallNode::bottom_type() const { return tf()->range(); }
610 const Type *MachCallNode::Value(PhaseTransform *phase) const { return tf()->rang

612 #ifndef PRODUCT
613 void MachCallNode::dump_spec(outputStream *st) const {
614   st->print("# ");
615   tf()->dump_on(st);
616   if (_cnt != COUNT_UNKNOWN)  st->print(" C=%f",_cnt);
617   if (jvms() != NULL)  jvms()->dump_spec(st);
618 }
619 #endif


622 bool MachCallNode::return_value_is_used() const {
623   if (tf()->range()->cnt() == TypeFunc::Parms) {
624     // void return
625     return false;
626   }

628   // find the projection corresponding to the return value
629   for (DUIterator_Fast imax, i = fast_outs(imax); i < imax; i++) {
630     Node *use = fast_out(i);
631     if (!use->is_Proj()) continue;
632     if (use->as_Proj()->_con == TypeFunc::Parms) {
633       return true;
634     }
635   }
636   return false;
637 }


640 //------------------------------Registers--------------------------------------
641 const RegMask &MachCallNode::in_RegMask( uint idx ) const {
642   // Values in the domain use the users calling convention, embodied in the
643   // _in_rms array of RegMasks.
644   if (idx < tf()->domain()->cnt())  return _in_rms[idx];
645   // Values outside the domain represent debug info
646   return *Compile::current()->matcher()->idealreg2debugmask[in(idx)->ideal_reg()
647 }

649 //=============================================================================
650 uint MachCallJavaNode::size_of() const { return sizeof(*this); }
651 uint MachCallJavaNode::cmp( const Node &n ) const {
652   MachCallJavaNode &call = (MachCallJavaNode&)n;
653   return MachCallNode::cmp(call) && _method->equals(call._method);
654 }
655 #ifndef PRODUCT
656 void MachCallJavaNode::dump_spec(outputStream *st) const {
657   if (_method_handle_invoke)
658     st->print("MethodHandle ");
659   if (_method) {
660     _method->print_short_name(st);
661     st->print(" ");
662   }
663   MachCallNode::dump_spec(st);
664 }
665 #endif

667 //------------------------------Registers--------------------------------------
668 const RegMask &MachCallJavaNode::in_RegMask(uint idx) const {
669   // Values in the domain use the users calling convention, embodied in the
670   // _in_rms array of RegMasks.
```



```
671   if (idx < tf()->domain()->cnt())  return _in_rms[idx];
672   // Values outside the domain represent debug info
673   Matcher* m = Compile::current()->matcher();
674   // If this call is a MethodHandle invoke we have to use a different
675   // debugmask which does not include the register we use to save the
676   // SP over MH invokes.
677   RegMask** debugmask = _method_handle_invoke ? m->idealreg2mhdebugmask : m->ide
678   return *debugmask[in(idx)->ideal_reg()];
679 }

681 //=============================================================================
682 uint MachCallStaticJavaNode::size_of() const { return sizeof(*this); }
683 uint MachCallStaticJavaNode::cmp( const Node &n ) const {
684   MachCallStaticJavaNode &call = (MachCallStaticJavaNode&)n;
685   return MachCallJavaNode::cmp(call) && _name == call._name;
686 }

688 //----------------------------uncommon_trap_request----------------------------
689 // If this is an uncommon trap, return the request code, else zero.
690 int MachCallStaticJavaNode::uncommon_trap_request() const {
691   if (_name != NULL && !strcmp(_name, "uncommon_trap")) {
692     return CallStaticJavaNode::extract_uncommon_trap_request(this);
693   }
694   return 0;
695 }

697 #ifndef PRODUCT
698 // Helper for summarizing uncommon_trap arguments.
699 void MachCallStaticJavaNode::dump_trap_args(outputStream *st) const {
700   int trap_req = uncommon_trap_request();
701   if (trap_req != 0) {
702     char buf[100];
703     st->print("(%s)",
704               Deoptimization::format_trap_request(buf, sizeof(buf),
705                                                   trap_req));
706   }
707 }

709 void MachCallStaticJavaNode::dump_spec(outputStream *st) const {
710   st->print("Static ");
711   if (_name != NULL) {
712     st->print("wrapper for: %s", _name );
713     dump_trap_args(st);
714     st->print(" ");
715   }
716   MachCallJavaNode::dump_spec(st);
717 }
718 #endif

720 //=============================================================================
721 #ifndef PRODUCT
722 void MachCallDynamicJavaNode::dump_spec(outputStream *st) const {
723   st->print("Dynamic ");
724   MachCallJavaNode::dump_spec(st);
725 }
726 #endif
727 //=============================================================================
728 uint MachCallRuntimeNode::size_of() const { return sizeof(*this); }
729 uint MachCallRuntimeNode::cmp( const Node &n ) const {
730   MachCallRuntimeNode &call = (MachCallRuntimeNode&)n;
731   return MachCallNode::cmp(call) && !strcmp(_name,call._name);
732 }
733 #ifndef PRODUCT
734 void MachCallRuntimeNode::dump_spec(outputStream *st) const {
735   st->print("%s ",_name);
736   MachCallNode::dump_spec(st);
```

```
737 }
738 #endif
739 //=============================================================================
740 // A shared JVMState for all HaltNodes.  Indicates the start of debug info
741 // is at TypeFunc::Parms.  Only required for SOE register spill handling -
742 // to indicate where the stack-slot-only debug info inputs begin.
743 // There is no other JVM state needed here.
744 JVMState jvms_for_throw(0);
745 JVMState *MachHaltNode::jvms() const {
746   return &jvms_for_throw;
747 }

749 //=============================================================================
750 #ifndef PRODUCT
751 void labelOper::int_format(PhaseRegAlloc *ra, const MachNode *node, outputStream
752   st->print("B%d", _block_num);
753 }
754 #endif // PRODUCT

756 //=============================================================================
757 #ifndef PRODUCT
758 void methodOper::int_format(PhaseRegAlloc *ra, const MachNode *node, outputStrea
759   st->print(INTPTR_FORMAT, _method);
760 }
761 #endif // PRODUCT
```