```
// ---------------------------------------------------------------------------
// Generate a native wrapper for a given method.  The method takes arguments
// in the Java compiled code convention, marshals them to the native
// convention (handlizes oops, etc), transitions to native, makes the call,
// returns to java state (possibly blocking), unhandlizes any result and
// returns.
//
// Critical native functions are a shorthand for the use of
// GetPrimtiveArrayCritical and disallow the use of any other JNI
// functions.  The wrapper is expected to unpack the arguments before
// passing them to the callee and perform checks before and after the
// native call to ensure that they GC_locker
// lock_critical/unlock_critical semantics are followed.  Some other
// parts of JNI setup are skipped like the tear down of the JNI handle
// block and the check for pending exceptions it's impossible for them
// to be thrown.
//
// They are roughly structured like this:
//    if (GC_locker::needs_gc())
//      SharedRuntime::block_for_jni_critical();
//    tranistion to thread_in_native
//    unpack arrray arguments and call native entry point
//    check for safepoint in progress
//    check if any thread suspend flags are set
//      call into JVM and possible unlock the JNI critical
//      if a GC was suppressed while in the critical native.
//    transition back to thread_in_Java
//    return to caller
//
nmethod* SharedRuntime::generate_native_wrapper(MacroAssembler* masm,
                                                methodHandle method,
                                                int compile_id,
                                                BasicType* in_sig_bt,
                                                VMRegPair* in_regs,
                                                BasicType ret_type) {
  if (method->is_method_handle_intrinsic()) {
    vmIntrinsics::ID iid = method->intrinsic_id();
    intptr_t start = (intptr_t)__ pc();
    int vep_offset = ((intptr_t)__ pc()) - start;
    gen_special_dispatch(masm,
                         method,
                         in_sig_bt,
                         in_regs);
    int frame_complete = ((intptr_t)__ pc()) - start;  // not complete, period
    __ flush();
    int stack_slots = SharedRuntime::out_preserve_stack_slots();  // no out slots at all,
actually
    return nmethod::new_native_nmethod(method,
                                       compile_id,
                                       masm->code(),
                                       vep_offset,
                                       frame_complete,
                                       stack_slots / VMRegImpl::slots_per_word,
                                       in_ByteSize(-1),
                                       in_ByteSize(-1),
                                       (OopMapSet*)NULL);
  }
  bool is_critical_native = true;
```

```
  address native_func = method->critical_native_function();
  if (native_func == NULL) {
    native_func = method->native_function();
    is_critical_native = false;
  }
  assert(native_func != NULL, "must have function");

  // An OopMap for lock (and class if static)
  OopMapSet *oop_maps = new OopMapSet();
  intptr_t start = (intptr_t)__ pc();

  // We have received a description of where all the java arg are located
  // on entry to the wrapper. We need to convert these args to where
  // the jni function will expect them. To figure out where they go
  // we convert the java signature to a C signature by inserting
  // the hidden arguments as arg[0] and possibly arg[1] (static method)

  const int total_in_args = method->size_of_parameters();
  int total_c_args = total_in_args;
  if (!is_critical_native) {
    total_c_args += 1;
    if (method->is_static()) {
      total_c_args++;
    }
  } else {
    for (int i = 0; i < total_in_args; i++) {
      if (in_sig_bt[i] == T_ARRAY) {
        total_c_args++;
      }
    }
  }

  BasicType* out_sig_bt = NEW_RESOURCE_ARRAY(BasicType, total_c_args);
  VMRegPair* out_regs   = NEW_RESOURCE_ARRAY(VMRegPair, total_c_args);
  BasicType* in_elem_bt = NULL;

  int argc = 0;
  if (!is_critical_native) {
    out_sig_bt[argc++] = T_ADDRESS;
    if (method->is_static()) {
      out_sig_bt[argc++] = T_OBJECT;
    }

    for (int i = 0; i < total_in_args ; i++ ) {
      out_sig_bt[argc++] = in_sig_bt[i];
    }
  } else {
    Thread* THREAD = Thread::current();
    in_elem_bt = NEW_RESOURCE_ARRAY(BasicType, total_in_args);
    SignatureStream ss(method->signature());
    for (int i = 0; i < total_in_args ; i++ ) {
      if (in_sig_bt[i] == T_ARRAY) {
        // Arrays are passed as int, elem* pair
        out_sig_bt[argc++] = T_INT;
        out_sig_bt[argc++] = T_ADDRESS;
        Symbol* atype = ss.as_symbol(CHECK_NULL);
        const char* at = atype->as_C_string();
        if (strlen(at) == 2) {
          assert(at[0] == '[', "must be");
          switch (at[1]) {
            case 'B': in_elem_bt[i]  = T_BYTE; break;
            case 'C': in_elem_bt[i]  = T_CHAR; break;
```

```
          case 'D': in_elem_bt[i]  = T_DOUBLE; break;
          case 'F': in_elem_bt[i]  = T_FLOAT; break;
          case 'I': in_elem_bt[i]  = T_INT; break;
          case 'J': in_elem_bt[i]  = T_LONG; break;
          case 'S': in_elem_bt[i]  = T_SHORT; break;
          case 'Z': in_elem_bt[i]  = T_BOOLEAN; break;
          default: ShouldNotReachHere();
        }
      }
    } else {
      out_sig_bt[argc++] = in_sig_bt[i];
      in_elem_bt[i] = T_VOID;
    }
    if (in_sig_bt[i] != T_VOID) {
      assert(in_sig_bt[i] == ss.type(), "must match");
      ss.next();
    }
  }
}

// Now figure out where the args must be stored and how much stack space
// they require.
int out_arg_slots;
out_arg_slots = c_calling_convention(out_sig_bt, out_regs, NULL, total_c_args);

// Compute framesize for the wrapper.  We need to handlize all oops in
// incoming registers

// Calculate the total number of stack slots we will need.

// First count the abi requirement plus all of the outgoing args
int stack_slots = SharedRuntime::out_preserve_stack_slots() + out_arg_slots;

// Now the space for the inbound oop handle area
int total_save_slots = 6 * VMRegImpl::slots_per_word;  // 6 arguments passed in registers
if (is_critical_native) {
  // Critical natives may have to call out so they need a save area
  // for register arguments.
  int double_slots = 0;
  int single_slots = 0;
  for ( int i = 0; i < total_in_args; i++) {
    if (in_regs[i].first()->is_Register()) {
      const Register reg = in_regs[i].first()->as_Register();
      switch (in_sig_bt[i]) {
        case T_BOOLEAN:
        case T_BYTE:
        case T_SHORT:
        case T_CHAR:
        case T_INT:  single_slots++; break;
        case T_ARRAY:  // specific to LP64 (7145024)
        case T_LONG: double_slots++; break;
        default:  ShouldNotReachHere();
      }
    } else if (in_regs[i].first()->is_XMMRegister()) {
      switch (in_sig_bt[i]) {
        case T_FLOAT:  single_slots++; break;
        case T_DOUBLE: double_slots++; break;
        default:  ShouldNotReachHere();
      }
    } else if (in_regs[i].first()->is_FloatRegister()) {
      ShouldNotReachHere();
```

```
      }
    }
    total_save_slots = double_slots * 2 + single_slots;
    // align the save area
    if (double_slots != 0) {
      stack_slots = round_to(stack_slots, 2);
    }
  }

  int oop_handle_offset = stack_slots;
  stack_slots += total_save_slots;

  // Now any space we need for handlizing a klass if static method

  int klass_slot_offset = 0;
  int klass_offset = -1;
  int lock_slot_offset = 0;
  bool is_static = false;

  if (method->is_static()) {
    klass_slot_offset = stack_slots;
    stack_slots += VMRegImpl::slots_per_word;
    klass_offset = klass_slot_offset * VMRegImpl::stack_slot_size;
    is_static = true;
  }

  // Plus a lock if needed

  if (method->is_synchronized()) {
    lock_slot_offset = stack_slots;
    stack_slots += VMRegImpl::slots_per_word;
  }

  // Now a place (+2) to save return values or temp during shuffling
  // + 4 for return address (which we own) and saved rbp
  stack_slots += 6;

  // Ok The space we have allocated will look like:
  //
  //
  // FP-> |                     |
  //      |---------------------|
  //      | 2 slots for moves   |
  //      |---------------------|
  //      | lock box (if sync)  |
  //      |---------------------| <- lock_slot_offset
  //      | klass (if static)   |
  //      |---------------------| <- klass_slot_offset
  //      | oopHandle area      |
  //      |---------------------| <- oop_handle_offset (6 java arg registers)
  //      | outbound memory     |
  //      | based arguments     |
  //      |                     |
  //      |---------------------|
  //      |                     |
  // SP-> | out_preserved_slots |
  //
  //


  // Now compute actual number of stack words we need rounding to make
  // stack properly aligned.
```

```
  stack_slots = round_to(stack_slots, StackAlignmentInSlots);

  int stack_size = stack_slots * VMRegImpl::stack_slot_size;

  // First thing make an ic check to see if we should even be here

  // We are free to use all registers as temps without saving them and
  // restoring them except rbp. rbp is the only callee save register
  // as far as the interpreter and the compiler(s) are concerned.


  const Register ic_reg = rax;
  const Register receiver = j_rarg0;

  Label hit;
  Label exception_pending;

  assert_different_registers(ic_reg, receiver, rscratch1);
  __ verify_oop(receiver);
  __ load_klass(rscratch1, receiver);
  __ cmpq(ic_reg, rscratch1);
  __ jcc(Assembler::equal, hit);

  __ jump(RuntimeAddress(SharedRuntime::get_ic_miss_stub()));

  // Verified entry point must be aligned
  __ align(8);

  __ bind(hit);

  int vep_offset = ((intptr_t)__ pc()) - start;

  // The instruction at the verified entry point must be 5 bytes or longer
  // because it can be patched on the fly by make_non_entrant. The stack bang
  // instruction fits that requirement.

  // Generate stack overflow check

  if (UseStackBanging) {
    __ bang_stack_with_offset(StackShadowPages*os::vm_page_size());
  } else {
    // need a 5 byte instruction to allow MT safe patching to non-entrant
    __ fat_nop();
  }

  // Generate a new frame for the wrapper.
  __ enter();
  // -2 because return address is already present and so is saved rbp
  __ subptr(rsp, stack_size - 2*wordSize);

  // Frame is now completed as far as size and linkage.
  int frame_complete = ((intptr_t)__ pc()) - start;

    if (UseRTMLocking) {
      // Abort RTM transaction before calling JNI
      // because critical section will be large and will be
      // aborted anyway. Also nmethod could be deoptimized.
      __ xabort(0);
    }

#ifdef ASSERT
    {
      Label L;
```

```
      __ mov(rax, rsp);
      __ andptr(rax, -16); // must be 16 byte boundary (see amd64 ABI)
      __ cmpptr(rax, rsp);
      __ jcc(Assembler::equal, L);
      __ stop("improperly aligned stack");
      __ bind(L);
    }
#endif /* ASSERT */


  // We use r14 as the oop handle for the receiver/klass
  // It is callee save so it survives the call to native

  const Register oop_handle_reg = r14;

  if (is_critical_native) {
    check_needs_gc_for_critical_native(masm, stack_slots, total_c_args, total_in_args,
                                       oop_handle_offset, oop_maps, in_regs, in_sig_bt);
  }

  //
  // We immediately shuffle the arguments so that any vm call we have to
  // make from here on out (sync slow path, jvmti, etc.) we will have
  // captured the oops from our caller and have a valid oopMap for
  // them.

  // -----------------
  // The Grand Shuffle

  // The Java calling convention is either equal (linux) or denser (win64) than the
  // c calling convention. However the because of the jni_env argument the c calling
  // convention always has at least one more (and two for static) arguments than Java.
  // Therefore if we move the args from java -> c backwards then we will never have
  // a register->register conflict and we don't have to build a dependency graph
  // and figure out how to break any cycles.
  //

  // Record esp-based slot for receiver on stack for non-static methods
  int receiver_offset = -1;

  // This is a trick. We double the stack slots so we can claim
  // the oops in the caller's frame. Since we are sure to have
  // more args than the caller doubling is enough to make
  // sure we can capture all the incoming oop args from the
  // caller.
  //
  OopMap* map = new OopMap(stack_slots * 2, 0 /* arg_slots*/);

  // Mark location of rbp (someday)
  // map->set_callee_saved(VMRegImpl::stack2reg( stack_slots - 2), stack_slots * 2, 0,
vmreg(rbp));

  // Use eax, ebx as temporaries during any memory-memory moves we have to do
  // All inbound args are referenced based on rbp and all outbound args via rsp.


#ifdef ASSERT
  bool reg_destroyed[RegisterImpl::number_of_registers];
  bool freg_destroyed[XMMRegisterImpl::number_of_registers];
  for ( int r = 0 ; r < RegisterImpl::number_of_registers ; r++ ) {
    reg_destroyed[r] = false;
  }
  for ( int f = 0 ; f < XMMRegisterImpl::number_of_registers ; f++ ) {
```

```
    freg_destroyed[f] = false;
  }

#endif /* ASSERT */

  // This may iterate in two different directions depending on the
  // kind of native it is.  The reason is that for regular JNI natives
  // the incoming and outgoing registers are offset upwards and for
  // critical natives they are offset down.
  GrowableArray<int> arg_order(2 * total_in_args);
  VMRegPair tmp_vmreg;
  tmp_vmreg.set1(rbx->as_VMReg());

  if (!is_critical_native) {
    for (int i = total_in_args - 1, c_arg = total_c_args - 1; i >= 0; i--, c_arg--) {
      arg_order.push(i);
      arg_order.push(c_arg);
    }
  } else {
    // Compute a valid move order, using tmp_vmreg to break any cycles
    ComputeMoveOrder cmo(total_in_args, in_regs, total_c_args, out_regs, in_sig_bt,
arg_order, tmp_vmreg);
  }

  int temploc = -1;
  for (int ai = 0; ai < arg_order.length(); ai += 2) {
    int i = arg_order.at(ai);
    int c_arg = arg_order.at(ai + 1);
    __ block_comment(err_msg("move %d -> %d", i, c_arg));
    if (c_arg == -1) {
      assert(is_critical_native, "should only be required for critical natives");
      // This arg needs to be moved to a temporary
      __ mov(tmp_vmreg.first()->as_Register(), in_regs[i].first()->as_Register());
      in_regs[i] = tmp_vmreg;
      temploc = i;
      continue;
    } else if (i == -1) {
      assert(is_critical_native, "should only be required for critical natives");
      // Read from the temporary location
      assert(temploc != -1, "must be valid");
      i = temploc;
      temploc = -1;
    }
#ifdef ASSERT
    if (in_regs[i].first()->is_Register()) {
      assert(!reg_destroyed[in_regs[i].first()->as_Register()->encoding()], "destroyed
reg!");
    } else if (in_regs[i].first()->is_XMMRegister()) {
      assert(!freg_destroyed[in_regs[i].first()->as_XMMRegister()->encoding()], "destroyed
reg!");
    }
    if (out_regs[c_arg].first()->is_Register()) {
      reg_destroyed[out_regs[c_arg].first()->as_Register()->encoding()] = true;
    } else if (out_regs[c_arg].first()->is_XMMRegister()) {
      freg_destroyed[out_regs[c_arg].first()->as_XMMRegister()->encoding()] = true;
    }
#endif /* ASSERT */
    switch (in_sig_bt[i]) {
      case T_ARRAY:
        if (is_critical_native) {
          unpack_array_argument(masm, in_regs[i], in_elem_bt[i], out_regs[c_arg + 1],
out_regs[c_arg]);
```

```
          c_arg++;
#ifdef ASSERT
          if (out_regs[c_arg].first()->is_Register()) {
            reg_destroyed[out_regs[c_arg].first()->as_Register()->encoding()] = true;
          } else if (out_regs[c_arg].first()->is_XMMRegister()) {
            freg_destroyed[out_regs[c_arg].first()->as_XMMRegister()->encoding()] = true;
          }
#endif
          break;
        }
      case T_OBJECT:
        assert(!is_critical_native, "no oop arguments");
        object_move(masm, map, oop_handle_offset, stack_slots, in_regs[i], out_regs[c_arg],
                    ((i == 0) && (!is_static)),
                    &receiver_offset);
        break;
      case T_VOID:
        break;

      case T_FLOAT:
        float_move(masm, in_regs[i], out_regs[c_arg]);
          break;

      case T_DOUBLE:
        assert( i + 1 < total_in_args &&
                in_sig_bt[i + 1] == T_VOID &&
                out_sig_bt[c_arg+1] == T_VOID, "bad arg list");
        double_move(masm, in_regs[i], out_regs[c_arg]);
        break;

      case T_LONG :
        long_move(masm, in_regs[i], out_regs[c_arg]);
        break;

      case T_ADDRESS: assert(false, "found T_ADDRESS in java args");

      default:
        move32_64(masm, in_regs[i], out_regs[c_arg]);
    }
  }

  int c_arg;

  // Pre-load a static method's oop into r14.  Used both by locking code and
  // the normal JNI call code.
  if (!is_critical_native) {
    // point c_arg at the first arg that is already loaded in case we
    // need to spill before we call out
    c_arg = total_c_args - total_in_args;

    if (method->is_static()) {

      //  load oop into a register
      __ movoop(oop_handle_reg, JNIHandles::make_local(method->method_holder()-
>java_mirror()));

      // Now handlize the static class mirror it's known not-null.
      __ movptr(Address(rsp, klass_offset), oop_handle_reg);
      map->set_oop(VMRegImpl::stack2reg(klass_slot_offset));

      // Now get the handle
      __ lea(oop_handle_reg, Address(rsp, klass_offset));
```

```
      // store the klass handle as second argument
      __ movptr(c_rarg1, oop_handle_reg);
      // and protect the arg if we must spill
      c_arg--;
    }
  } else {
    // For JNI critical methods we need to save all registers in save_args.
    c_arg = 0;
  }

  // Change state to native (we save the return address in the thread, since it might not
  // be pushed on the stack when we do a a stack traversal). It is enough that the pc()
  // points into the right code segment. It does not have to be the correct return pc.
  // We use the same pc/oopMap repeatedly when we call out

  intptr_t the_pc = (intptr_t) __ pc();
  oop_maps->add_gc_map(the_pc - start, map);

  __ set_last_Java_frame(rsp, noreg, (address)the_pc);


  // We have all of the arguments setup at this point. We must not touch any register
  // argument registers at this point (what if we save/restore them there are no oop?

  {
    SkipIfEqual skip(masm, &DTraceMethodProbes, false);
    // protect the args we've loaded
    save_args(masm, total_c_args, c_arg, out_regs);
    __ mov_metadata(c_rarg1, method());
    __ call_VM_leaf(
      CAST_FROM_FN_PTR(address, SharedRuntime::dtrace_method_entry),
      r15_thread, c_rarg1);
    restore_args(masm, total_c_args, c_arg, out_regs);
  }

  // RedefineClasses() tracing support for obsolete method entry
  if (RC_TRACE_IN_RANGE(0x00001000, 0x00002000)) {
    // protect the args we've loaded
    save_args(masm, total_c_args, c_arg, out_regs);
    __ mov_metadata(c_rarg1, method());
    __ call_VM_leaf(
      CAST_FROM_FN_PTR(address, SharedRuntime::rc_trace_method_entry),
      r15_thread, c_rarg1);
    restore_args(masm, total_c_args, c_arg, out_regs);
  }

  // Lock a synchronized method

  // Register definitions used by locking and unlocking

  const Register swap_reg = rax;  // Must use rax for cmpxchg instruction
  const Register obj_reg  = rbx;  // Will contain the oop
  const Register lock_reg = r13;  // Address of compiler lock object (BasicLock)
  const Register old_hdr  = r13;  // value of old header at unlock time

  Label slow_path_lock;
  Label lock_done;

  if (method->is_synchronized()) {
    assert(!is_critical_native, "unhandled");


    const int mark_word_offset = BasicLock::displaced_header_offset_in_bytes();
```

```
    // Get the handle (the 2nd argument)
    __ mov(oop_handle_reg, c_rarg1);

    // Get address of the box

    __ lea(lock_reg, Address(rsp, lock_slot_offset * VMRegImpl::stack_slot_size));

    // Load the oop from the handle
    __ movptr(obj_reg, Address(oop_handle_reg, 0));

    if (UseBiasedLocking) {
      __ biased_locking_enter(lock_reg, obj_reg, swap_reg, rscratch1, false, lock_done,
&slow_path_lock);
    }

    // Load immediate 1 into swap_reg %rax
    __ movl(swap_reg, 1);

    // Load (object->mark() | 1) into swap_reg %rax
    __ orptr(swap_reg, Address(obj_reg, 0));

    // Save (object->mark() | 1) into BasicLock's displaced header
    __ movptr(Address(lock_reg, mark_word_offset), swap_reg);

    if (os::is_MP()) {
      __ lock();
    }

    // src -> dest iff dest == rax else rax <- dest
    __ cmpxchgptr(lock_reg, Address(obj_reg, 0));
    __ jcc(Assembler::equal, lock_done);

    // Hmm should this move to the slow path code area???

    // Test if the oopMark is an obvious stack pointer, i.e.,
    //  1) (mark & 3) == 0, and
    //  2) rsp <= mark < mark + os::pagesize()
    // These 3 tests can be done by evaluating the following
    // expression: ((mark - rsp) & (3 - os::vm_page_size())),
    // assuming both stack pointer and pagesize have their
    // least significant 2 bits clear.
    // NOTE: the oopMark is in swap_reg %rax as the result of cmpxchg

    __ subptr(swap_reg, rsp);
    __ andptr(swap_reg, 3 - os::vm_page_size());

    // Save the test result, for recursive case, the result is zero
    __ movptr(Address(lock_reg, mark_word_offset), swap_reg);
    __ jcc(Assembler::notEqual, slow_path_lock);

    // Slow path will re-enter here

    __ bind(lock_done);
  }


  // Finally just about ready to make the JNI call


  // get JNIEnv* which is first argument to native
  if (!is_critical_native) {
    __ lea(c_rarg0, Address(r15_thread, in_bytes(JavaThread::jni_environment_offset())));
```

```
  }

  // Now set thread in native
  __ movl(Address(r15_thread, JavaThread::thread_state_offset()), _thread_in_native);

  __ call(RuntimeAddress(native_func));

  // Verify or restore cpu control state after JNI call
  __ restore_cpu_control_state_after_jni();

  // Unpack native results.
  switch (ret_type) {
  case T_BOOLEAN: __ c2bool(rax);            break;
  case T_CHAR   : __ movzwl(rax, rax);      break;
  case T_BYTE   : __ sign_extend_byte (rax); break;
  case T_SHORT  : __ sign_extend_short(rax); break;
  case T_INT    : /* nothing to do */        break;
  case T_DOUBLE :
  case T_FLOAT  :
    // Result is in xmm0 we'll save as needed
    break;
  case T_ARRAY:                 // Really a handle
  case T_OBJECT:                // Really a handle
      break; // can't de-handlize until after safepoint check
  case T_VOID: break;
  case T_LONG: break;
  default       : ShouldNotReachHere();
  }

  // Switch thread to "native transition" state before reading the synchronization state.
  // This additional state is necessary because reading and testing the synchronization
  // state is not atomic w.r.t. GC, as this scenario demonstrates:
  //     Java thread A, in _thread_in_native state, loads _not_synchronized and is preempted.
  //     VM thread changes sync state to synchronizing and suspends threads for GC.
  //     Thread A is resumed to finish this native method, but doesn't block here since it
  //     didn't see any synchronization is progress, and escapes.
  __ movl(Address(r15_thread, JavaThread::thread_state_offset()), _thread_in_native_trans);

  if(os::is_MP()) {
    if (UseMembar) {
      // Force this write out before the read below
      __ membar(Assembler::Membar_mask_bits(
           Assembler::LoadLoad | Assembler::LoadStore |
           Assembler::StoreLoad | Assembler::StoreStore));
    } else {
      // Write serialization page so VM thread can do a pseudo remote membar.
      // We use the current thread pointer to calculate a thread specific
      // offset to write to within the page. This minimizes bus traffic
      // due to cache line collision.
      __ serialize_memory(r15_thread, rcx);
    }
  }

  Label after_transition;

  // check for safepoint operation in progress and/or pending suspend requests
  {
    Label Continue;

    __ cmp32(ExternalAddress((address)SafepointSynchronize::address_of_state()),
             SafepointSynchronize::_not_synchronized);
```

```
    Label L;
    __ jcc(Assembler::notEqual, L);
    __ cmpl(Address(r15_thread, JavaThread::suspend_flags_offset()), 0);
    __ jcc(Assembler::equal, Continue);
    __ bind(L);

    // Don't use call_VM as it will see a possible pending exception and forward it
    // and never return here preventing us from clearing _last_native_pc down below.
    // Also can't use call_VM_leaf either as it will check to see if rsi & rdi are
    // preserved and correspond to the bcp/locals pointers. So we do a runtime call
    // by hand.
    //
    save_native_result(masm, ret_type, stack_slots);
    __ mov(c_rarg0, r15_thread);
    __ mov(r12, rsp); // remember sp
    __ subptr(rsp, frame::arg_reg_save_area_bytes); // windows
    __ andptr(rsp, -16); // align stack as required by ABI
    if (!is_critical_native) {
      __ call(RuntimeAddress(CAST_FROM_FN_PTR(address,
JavaThread::check_special_condition_for_native_trans)));
    } else {
      __ call(RuntimeAddress(CAST_FROM_FN_PTR(address,
JavaThread::check_special_condition_for_native_trans_and_transition)));
    }
    __ mov(rsp, r12); // restore sp
    __ reinit_heapbase();
    // Restore any method result value
    restore_native_result(masm, ret_type, stack_slots);

    if (is_critical_native) {
      // The call above performed the transition to thread_in_Java so
      // skip the transition logic below.
      __ jmpb(after_transition);
    }

    __ bind(Continue);
  }

  // change thread state
  __ movl(Address(r15_thread, JavaThread::thread_state_offset()), _thread_in_Java);
  __ bind(after_transition);

  Label reguard;
  Label reguard_done;
  __ cmpl(Address(r15_thread, JavaThread::stack_guard_state_offset()),
JavaThread::stack_guard_yellow_disabled);
  __ jcc(Assembler::equal, reguard);
  __ bind(reguard_done);

  // native result if any is live

  // Unlock
  Label unlock_done;
  Label slow_path_unlock;
  if (method->is_synchronized()) {

    // Get locked oop from the handle we passed to jni
    __ movptr(obj_reg, Address(oop_handle_reg, 0));

    Label done;

    if (UseBiasedLocking) {
```

```
      __ biased_locking_exit(obj_reg, old_hdr, done);
    }

    // Simple recursive lock?

    __ cmpptr(Address(rsp, lock_slot_offset * VMRegImpl::stack_slot_size),
(int32_t)NULL_WORD);
    __ jcc(Assembler::equal, done);

    // Must save rax if if it is live now because cmpxchg must use it
    if (ret_type != T_FLOAT && ret_type != T_DOUBLE && ret_type != T_VOID) {
      save_native_result(masm, ret_type, stack_slots);
    }


    // get address of the stack lock
    __ lea(rax, Address(rsp, lock_slot_offset * VMRegImpl::stack_slot_size));
    //  get old displaced header
    __ movptr(old_hdr, Address(rax, 0));

    // Atomic swap old header if oop still contains the stack lock
    if (os::is_MP()) {
      __ lock();
    }
    __ cmpxchgptr(old_hdr, Address(obj_reg, 0));
    __ jcc(Assembler::notEqual, slow_path_unlock);

    // slow path re-enters here
    __ bind(unlock_done);
    if (ret_type != T_FLOAT && ret_type != T_DOUBLE && ret_type != T_VOID) {
      restore_native_result(masm, ret_type, stack_slots);
    }

    __ bind(done);

  }
  {
    SkipIfEqual skip(masm, &DTraceMethodProbes, false);
    save_native_result(masm, ret_type, stack_slots);
    __ mov_metadata(c_rarg1, method());
    __ call_VM_leaf(
        CAST_FROM_FN_PTR(address, SharedRuntime::dtrace_method_exit),
        r15_thread, c_rarg1);
    restore_native_result(masm, ret_type, stack_slots);
  }

  __ reset_last_Java_frame(false, true);

  // Unpack oop result
  if (ret_type == T_OBJECT || ret_type == T_ARRAY) {
      Label L;
      __ testptr(rax, rax);
      __ jcc(Assembler::zero, L);
      __ movptr(rax, Address(rax, 0));
      __ bind(L);
      __ verify_oop(rax);
  }

  if (!is_critical_native) {
    // reset handle block
    __ movptr(rcx, Address(r15_thread, JavaThread::active_handles_offset()));
    __ movl(Address(rcx, JNIHandleBlock::top_offset_in_bytes()), (int32_t)NULL_WORD);
```

```
  }

  // pop our frame

  __ leave();

  if (!is_critical_native) {
    // Any exception pending?
    __ cmpptr(Address(r15_thread, in_bytes(Thread::pending_exception_offset())),
(int32_t)NULL_WORD);
    __ jcc(Assembler::notEqual, exception_pending);
  }

  // Return

  __ ret(0);

  // Unexpected paths are out of line and go here

  if (!is_critical_native) {
    // forward the exception
    __ bind(exception_pending);

    // and forward the exception
    __ jump(RuntimeAddress(StubRoutines::forward_exception_entry()));
  }

  // Slow path locking & unlocking
  if (method->is_synchronized()) {

    // BEGIN Slow path lock
    __ bind(slow_path_lock);

    // has last_Java_frame setup. No exceptions so do vanilla call not call_VM
    // args are (oop obj, BasicLock* lock, JavaThread* thread)

    // protect the args we've loaded
    save_args(masm, total_c_args, c_arg, out_regs);

    __ mov(c_rarg0, obj_reg);
    __ mov(c_rarg1, lock_reg);
    __ mov(c_rarg2, r15_thread);

    // Not a leaf but we have last_Java_frame setup as we want
    __ call_VM_leaf(CAST_FROM_FN_PTR(address, SharedRuntime::complete_monitor_locking_C), 3);
    restore_args(masm, total_c_args, c_arg, out_regs);

#ifdef ASSERT
    { Label L;
    __ cmpptr(Address(r15_thread, in_bytes(Thread::pending_exception_offset())),
(int32_t)NULL_WORD);
    __ jcc(Assembler::equal, L);
    __ stop("no pending exception allowed on exit from monitorenter");
    __ bind(L);
    }
#endif
    __ jmp(lock_done);

    // END Slow path lock

    // BEGIN Slow path unlock
    __ bind(slow_path_unlock);
```

```
    // If we haven't already saved the native result we must save it now as xmm registers
    // are still exposed.

    if (ret_type == T_FLOAT || ret_type == T_DOUBLE ) {
      save_native_result(masm, ret_type, stack_slots);
    }

    __ lea(c_rarg1, Address(rsp, lock_slot_offset * VMRegImpl::stack_slot_size));

    __ mov(c_rarg0, obj_reg);
    __ mov(r12, rsp); // remember sp
    __ subptr(rsp, frame::arg_reg_save_area_bytes); // windows
    __ andptr(rsp, -16); // align stack as required by ABI

    // Save pending exception around call to VM (which contains an EXCEPTION_MARK)
    // NOTE that obj_reg == rbx currently
    __ movptr(rbx, Address(r15_thread, in_bytes(Thread::pending_exception_offset())));
    __ movptr(Address(r15_thread, in_bytes(Thread::pending_exception_offset())),
(int32_t)NULL_WORD);

    __ call(RuntimeAddress(CAST_FROM_FN_PTR(address,
SharedRuntime::complete_monitor_unlocking_C)));
    __ mov(rsp, r12); // restore sp
    __ reinit_heapbase();
#ifdef ASSERT
    {
      Label L;
      __ cmpptr(Address(r15_thread, in_bytes(Thread::pending_exception_offset())),
(int)NULL_WORD);
      __ jcc(Assembler::equal, L);
      __ stop("no pending exception allowed on exit complete_monitor_unlocking_C");
      __ bind(L);
    }
#endif /* ASSERT */

    __ movptr(Address(r15_thread, in_bytes(Thread::pending_exception_offset())), rbx);

    if (ret_type == T_FLOAT || ret_type == T_DOUBLE ) {
      restore_native_result(masm, ret_type, stack_slots);
    }
    __ jmp(unlock_done);

    // END Slow path unlock

  } // synchronized

  // SLOW PATH Reguard the stack if needed

  __ bind(reguard);
  save_native_result(masm, ret_type, stack_slots);
  __ mov(r12, rsp); // remember sp
  __ subptr(rsp, frame::arg_reg_save_area_bytes); // windows
  __ andptr(rsp, -16); // align stack as required by ABI
  __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, SharedRuntime::reguard_yellow_pages)));
  __ mov(rsp, r12); // restore sp
  __ reinit_heapbase();
  restore_native_result(masm, ret_type, stack_slots);
  // and continue
  __ jmp(reguard_done);



  __ flush();
```

```
  nmethod *nm = nmethod::new_native_nmethod(method,
                                            compile_id,
                                            masm->code(),
                                            vep_offset,
                                            frame_complete,
                                            stack_slots / VMRegImpl::slots_per_word,
                                            (is_static ? in_ByteSize(klass_offset) :
in_ByteSize(receiver_offset)),

in_ByteSize(lock_slot_offset*VMRegImpl::stack_slot_size),
                                            oop_maps);

  if (is_critical_native) {
    nm->set_lazy_critical_native(true);
  }

  return nm;

}
```