```
class StubGenerator: public StubCodeGenerator {
 private:

#ifdef PRODUCT
#define inc_counter_np(counter) ((void)0)
#else
  void inc_counter_np_(int& counter) {
    // This can destroy rscratch1 if counter is far from the
code cache
    __ incrementl(ExternalAddress((address)&counter));
  }
#define inc_counter_np(counter) \
  BLOCK_COMMENT("inc_counter " #counter); \
  inc_counter_np_(counter);
#endif

  // Call stubs are used to call Java from C
  //
  // Linux Arguments:
  //    c_rarg0:   call wrapper address
address
  //    c_rarg1:   result
address
  //    c_rarg2:   result type
BasicType
  //    c_rarg3:   method
Method*
  //    c_rarg4:   (interpreter) entry point
address
  //    c_rarg5:   parameters
intptr_t*
  //    16(rbp): parameter size (in words)              int
  //    24(rbp): thread
Thread*
  //
  //     [ return_from_Java     ] <--- rsp
  //     [ argument word n      ]
  //      ...
  // -12 [ argument word 1      ]
  // -11 [ saved r15            ] <--- rsp_after_call
  // -10 [ saved r14            ]
  //  -9 [ saved r13            ]
  //  -8 [ saved r12            ]
  //  -7 [ saved rbx            ]
  //  -6 [ call wrapper         ]
  //  -5 [ result               ]
  //  -4 [ result type          ]
  //  -3 [ method               ]
  //  -2 [ entry point          ]
  //  -1 [ parameters           ]
  //   0 [ saved rbp            ] <--- rbp
  //   1 [ return address       ]
  //   2 [ parameter size       ]
  //   3 [ thread               ]
  //
  // Windows Arguments:
  //    c_rarg0:   call wrapper address
address
  //    c_rarg1:   result
address
  //    c_rarg2:   result type
BasicType
  //    c_rarg3:   method
Method*
  //    48(rbp): (interpreter) entry point
address
  //    56(rbp): parameters
intptr_t*
  //    64(rbp): parameter size (in words)              int
  //    72(rbp): thread
Thread*
  //
  //     [ return_from_Java     ] <--- rsp
  //     [ argument word n      ]
  //      ...
  // -28 [ argument word 1      ]
  // -27 [ saved xmm15          ] <--- rsp_after_call
  //     [ saved xmm7-xmm14     ]
  //  -9 [ saved xmm6           ] (each xmm register takes 2
slots)
  //  -7 [ saved r15            ]
  //  -6 [ saved r14            ]
  //  -5 [ saved r13            ]
  //  -4 [ saved r12            ]
  //  -3 [ saved rdi            ]
  //  -2 [ saved rsi            ]
  //  -1 [ saved rbx            ]
  //   0 [ saved rbp            ] <--- rbp
  //   1 [ return address       ]
  //   2 [ call wrapper         ]
  //   3 [ result               ]
  //   4 [ result type          ]
  //   5 [ method               ]
  //   6 [ entry point          ]
  //   7 [ parameters           ]
  //   8 [ parameter size       ]
  //   9 [ thread               ]
  //
  //    Windows reserves the callers stack space for
arguments 1-4.
  //    We spill c_rarg0-c_rarg3 to this space.

  // Call stub stack layout word offsets from rbp
  enum call_stub_layout {
#ifdef _WIN64
    xmm_save_first     = 6,  // save from xmm6
    xmm_save_last      = 15, // to xmm15
    xmm_save_base      = -9,
    rsp_after_call_off = xmm_save_base - 2 * (xmm_save_last -
xmm_save_first), // -27
    r15_off            = -7,
    r14_off            = -6,
    r13_off            = -5,
    r12_off            = -4,
    rdi_off            = -3,
    rsi_off            = -2,
    rbx_off            = -1,
    rbp_off            =  0,
    retaddr_off        =  1,
    call_wrapper_off   =  2,
    result_off         =  3,
    result_type_off    =  4,
    method_off         =  5,
    entry_point_off    =  6,
    parameters_off     =  7,
    parameter_size_off =  8,
    thread_off         =  9
#else
    rsp_after_call_off = -12,
    mxcsr_off          = rsp_after_call_off,
    r15_off            = -11,
    r14_off            = -10,
    r13_off            = -9,
    r12_off            = -8,
    rbx_off            = -7,
    call_wrapper_off   = -6,
    result_off         = -5,
    result_type_off    = -4,
    method_off         = -3,
    entry_point_off    = -2,
    parameters_off     = -1,
    rbp_off            =  0,
    retaddr_off        =  1,
    parameter_size_off =  2,
    thread_off         =  3
#endif
  };

#ifdef _WIN64
  Address xmm_save(int reg) {
    assert(reg >= xmm_save_first && reg <= xmm_save_last,
"XMM register number out of range");
    return Address(rbp, (xmm_save_base - (reg -
xmm_save_first) * 2) * wordSize);
  }
```

```
#endif

  address generate_call_stub(address& return_address) {
    assert((int)frame::entry_frame_after_call_words == -
(int)rsp_after_call_off + 1 &&
           (int)frame::entry_frame_call_wrapper_offset ==
(int)call_wrapper_off,
           "adjust this code");
    StubCodeMark mark(this, "StubRoutines", "call_stub");
    address start = __ pc();

    // same as in generate_catch_exception()!
    const Address rsp_after_call(rbp, rsp_after_call_off *
wordSize);

    const Address call_wrapper  (rbp, call_wrapper_off   *
wordSize);
    const Address result        (rbp, result_off         *
wordSize);
    const Address result_type   (rbp, result_type_off    *
wordSize);
    const Address method        (rbp, method_off         *
wordSize);
    const Address entry_point   (rbp, entry_point_off    *
wordSize);
    const Address parameters    (rbp, parameters_off     *
wordSize);
    const Address parameter_size(rbp, parameter_size_off *
wordSize);

    // same as in generate_catch_exception()!
    const Address thread        (rbp, thread_off         *
wordSize);

    const Address r15_save(rbp, r15_off * wordSize);
    const Address r14_save(rbp, r14_off * wordSize);
    const Address r13_save(rbp, r13_off * wordSize);
    const Address r12_save(rbp, r12_off * wordSize);
    const Address rbx_save(rbp, rbx_off * wordSize);

    // stub code
    __ enter();
    __ subptr(rsp, -rsp_after_call_off * wordSize);

    // save register parameters
#ifndef _WIN64
    __ movptr(parameters,   c_rarg5); // parameters
    __ movptr(entry_point,  c_rarg4); // entry_point
#endif

    __ movptr(method,       c_rarg3); // method
    __ movl(result_type,  c_rarg2);   // result type
    __ movptr(result,       c_rarg1); // result
    __ movptr(call_wrapper, c_rarg0); // call wrapper

    // save regs belonging to calling function
    __ movptr(rbx_save, rbx);
    __ movptr(r12_save, r12);
    __ movptr(r13_save, r13);
    __ movptr(r14_save, r14);
    __ movptr(r15_save, r15);
#ifdef _WIN64
    for (int i = 6; i <= 15; i++) {
      __ movdqu(xmm_save(i), as_XMMRegister(i));
    }

    const Address rdi_save(rbp, rdi_off * wordSize);
    const Address rsi_save(rbp, rsi_off * wordSize);

    __ movptr(rsi_save, rsi);
    __ movptr(rdi_save, rdi);
#else
    const Address mxcsr_save(rbp, mxcsr_off * wordSize);
    {
      Label skip_ldmx;
      __ stmxcsr(mxcsr_save);
      __ movl(rax, mxcsr_save);
      __ andl(rax, MXCSR_MASK);    // Only check control and
mask bits
```

```
      ExternalAddress
mxcsr_std(StubRoutines::addr_mxcsr_std());
      __ cmp32(rax, mxcsr_std);
      __ jcc(Assembler::equal, skip_ldmx);
      __ ldmxcsr(mxcsr_std);
      __ bind(skip_ldmx);
    }
#endif

    // Load up thread register
    __ movptr(r15_thread, thread);
    __ reinit_heapbase();

#ifdef ASSERT
    // make sure we have no pending exceptions
    {
      Label L;
      __ cmpptr(Address(r15_thread,
Thread::pending_exception_offset()), (int32_t)NULL_WORD);
      __ jcc(Assembler::equal, L);
      __ stop("StubRoutines::call_stub: entered with pending
exception");
      __ bind(L);
    }
#endif

    // pass parameters if any
    BLOCK_COMMENT("pass parameters if any");
    Label parameters_done;
    __ movl(c_rarg3, parameter_size);
    __ testl(c_rarg3, c_rarg3);
    __ jcc(Assembler::zero, parameters_done);

    Label loop;
    __ movptr(c_rarg2, parameters);       // parameter
pointer
    __ movl(c_rarg1, c_rarg3);            // parameter
counter is in c_rarg1
    __ BIND(loop);
    __ movptr(rax, Address(c_rarg2, 0));// get parameter
    __ addptr(c_rarg2, wordSize);       // advance to next
parameter
    __ decrementl(c_rarg1);             // decrement counter
    __ push(rax);                       // pass parameter
    __ jcc(Assembler::notZero, loop);

    // call Java function
    __ BIND(parameters_done);
    __ movptr(rbx, method);             // get Method*
    __ movptr(c_rarg1, entry_point);    // get entry_point
    __ mov(r13, rsp);                   // set sender sp
    BLOCK_COMMENT("call Java function");
    __ call(c_rarg1);

    BLOCK_COMMENT("call_stub_return_address:");
    return_address = __ pc();

    // store result depending on type (everything that is not
    // T_OBJECT, T_LONG, T_FLOAT or T_DOUBLE is treated as
T_INT)
    __ movptr(c_rarg0, result);
    Label is_long, is_float, is_double, exit;
    __ movl(c_rarg1, result_type);
    __ cmpl(c_rarg1, T_OBJECT);
    __ jcc(Assembler::equal, is_long);
    __ cmpl(c_rarg1, T_LONG);
    __ jcc(Assembler::equal, is_long);
    __ cmpl(c_rarg1, T_FLOAT);
    __ jcc(Assembler::equal, is_float);
    __ cmpl(c_rarg1, T_DOUBLE);
    __ jcc(Assembler::equal, is_double);

    // handle T_INT case
    __ movl(Address(c_rarg0, 0), rax);

    __ BIND(exit);

    // pop parameters
    __ lea(rsp, rsp_after_call);

#ifdef ASSERT
    // verify that threads correspond
    {
      Label L, S;
      __ cmpptr(r15_thread, thread);
      __ jcc(Assembler::notEqual, S);
      __ get_thread(rbx);
      __ cmpptr(r15_thread, rbx);
      __ jcc(Assembler::equal, L);
      __ bind(S);
      __ jcc(Assembler::equal, L);
      __ stop("StubRoutines::call_stub: threads must
correspond");
      __ bind(L);
    }
#endif

    // restore regs belonging to calling function
#ifdef _WIN64
    for (int i = 15; i >= 6; i--) {
      __ movdqu(as_XMMRegister(i), xmm_save(i));
    }
#endif
    __ movptr(r15, r15_save);
    __ movptr(r14, r14_save);
    __ movptr(r13, r13_save);
    __ movptr(r12, r12_save);
    __ movptr(rbx, rbx_save);

#ifdef _WIN64
    __ movptr(rdi, rdi_save);
    __ movptr(rsi, rsi_save);
#else
    __ ldmxcsr(mxcsr_save);
#endif

    // restore rsp
    __ addptr(rsp, -rsp_after_call_off * wordSize);

    // return
    __ pop(rbp);
    __ ret(0);

    // handle return types different from T_INT
    __ BIND(is_long);
    __ movq(Address(c_rarg0, 0), rax);
    __ jmp(exit);

    __ BIND(is_float);
    __ movflt(Address(c_rarg0, 0), xmm0);
    __ jmp(exit);

    __ BIND(is_double);
    __ movdbl(Address(c_rarg0, 0), xmm0);
    __ jmp(exit);

    return start;
  }
```