中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

V8 simulator：如何调用host function

智能软件研究中心 邱吉
qiuji@iscas.ac.cn

2021/07/09

Outline

- 什么是V8的simulator
- simulator run的两种模式
- simulator如何调用host function

Outline

- 什么是V8的simulator
- simulator run的两种模式
- simulator如何调用host function

什么是V8的simulator

- 什么是V8的simulator：v8/src/execution/目录下实现的arm/arm64, mips/mips64, ppc/s390, riscv64的用户态，指令集仿真器

```
./riscv64/simulator-riscv64.cc
./riscv64/simulator-riscv64.h
```

- V8内置simulator的目的：
  - V8是多后端架构的JS引擎
  - 在X64平台上对其他指令集架构后端的开发和测试仅依赖于模拟器，而不需要真实的硬件平台

如何运行simulator run

- simulator build： https://github.com/riscv/v8/wiki/Simulator-Build

args.gn文件内容：

```
is_component_build = true
is_debug = true
symbol_level = 2
target_cpu = "x64"
v8_target_cpu = "riscv64"
use_goma = false
goma_dir = "None"
v8_enable_backtrace = true
v8_enable_fast_mksnapshot = true
v8_enable_slow_dchecks = true
v8_optimized_debug = false
v8_enable_trace_ignition = true
```

target_cpu: 编译出的d8可执行文件和libv8.so运行库是x64的
v8_targert_cpu: v8的turbofan的后端是riscv64的， 产生riscv64的snapshot(builtin) 和JIT code

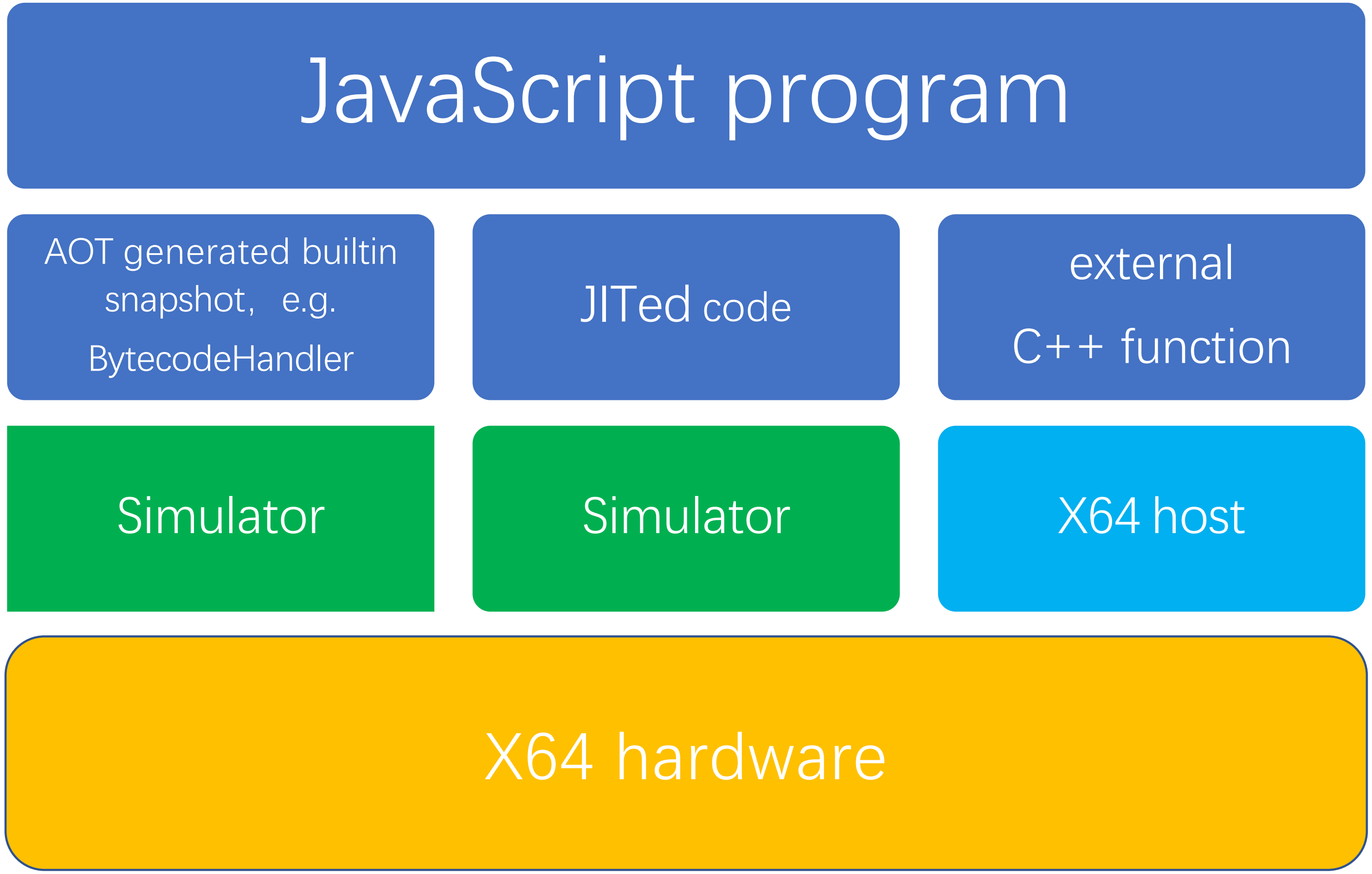
- ./d8 hello.js

hello.js:
console.log("hello")

Outline

- 什么是V8的simulator
- simulator run的两种模式
- simulator如何调用host function

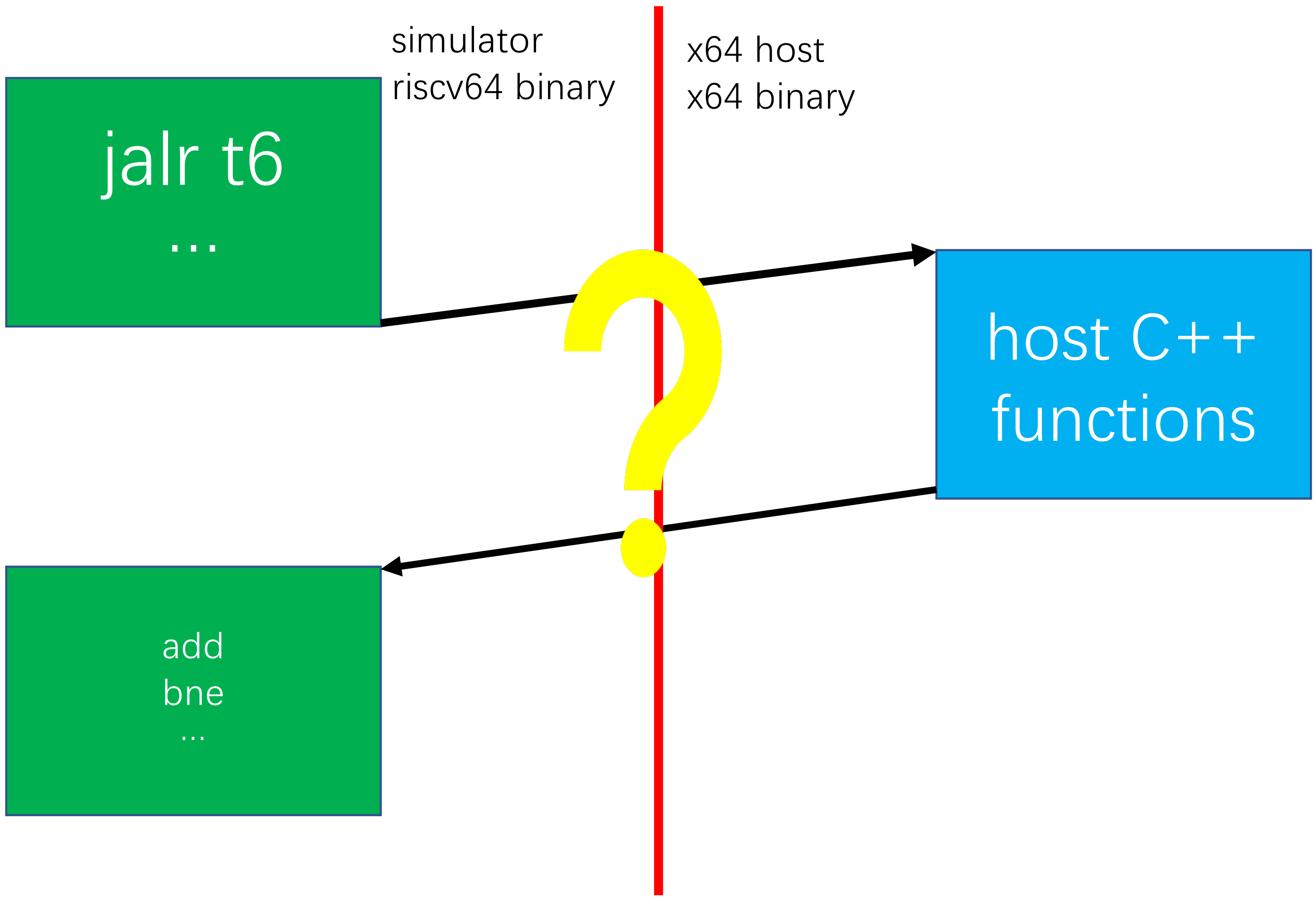
simulator run下的两种模式:
simulator & host



Outline

- 什么是V8的simulator
- simulator run的两种模式
- simulator如何调用host function
  - 解释执行的过程
  - SoftwareInterrupt如何执行
  - ExternalReferenceTable和Redirect如何建立和被使用

simulator如何调用host function



先探究一下执行的trace log

- ./d8 --trace-sim hello.js 2>&1 |tee log.txt

```
  0x7f33d96ab21c  266    00001d37      lui      s10, 0x1          0000000000001000   (266)   int64:4096 uint64:4096
  0x7f33d96ab220  267    016d0d33      add      s10, s10, s6      000055e1e0e7bfc0   (267)   int64:94428629286848 uint64:94428629286848
  0x7f33d96ab224  268    540d3f83      ld       t6, 1344(s10)     000055e1e0b163f8   (268)   int64:94428625724408 uint64:94428625724408 <-- [addr: 55e1e0e7c500]
  0x7f33d96ab228  269    00000317      auipc    t1, 0x0           00007f33d96ab228   (269)   int64:139860667707944 uint64:139860667707944
  0x7f33d96ab22c  270    f26b3423      sd       t1, -216(s6)                         (270)   int64:139860667707944 uint64:139860667707944 --> [addr: 55e1e0e7aee8]
  0x7f33d96ab230  271    f28b3023      sd       fp, -224(s6)                         (271)   int64:139860464057904 uint64:139860464057904 --> [addr: 55e1e0e7aee0]
  0x7f33d96ab234  272    000f80e7      jalr     t6                00007f33d96ab238   (272)   int64:139860667707960 uint64:139860667707960
Call to host function check_object_type at 0x7f33dafb0670 args 6a3caa22a9 , 1200000000 , ebdff69f01 , a788901599 , 7f33cd473e88 , 7f33cd274410 , 00000001 , 00000021 , 7f33cd473d7
0 , 00000000
Returned 7f33d85e7c6c : 00000000
  0x55e1e0b163f8  273    00000073      ecall
  0x7f33d96ab238  274    f20b0d13      addi     s10, s6, -224     000055e1e0e7aee0   (274)   int64:94428629282528 uint64:94428629282528
```

先探究一下执行的trace log

- ./d8 --trace-sim hello.js 2>&1 |tee log.txt

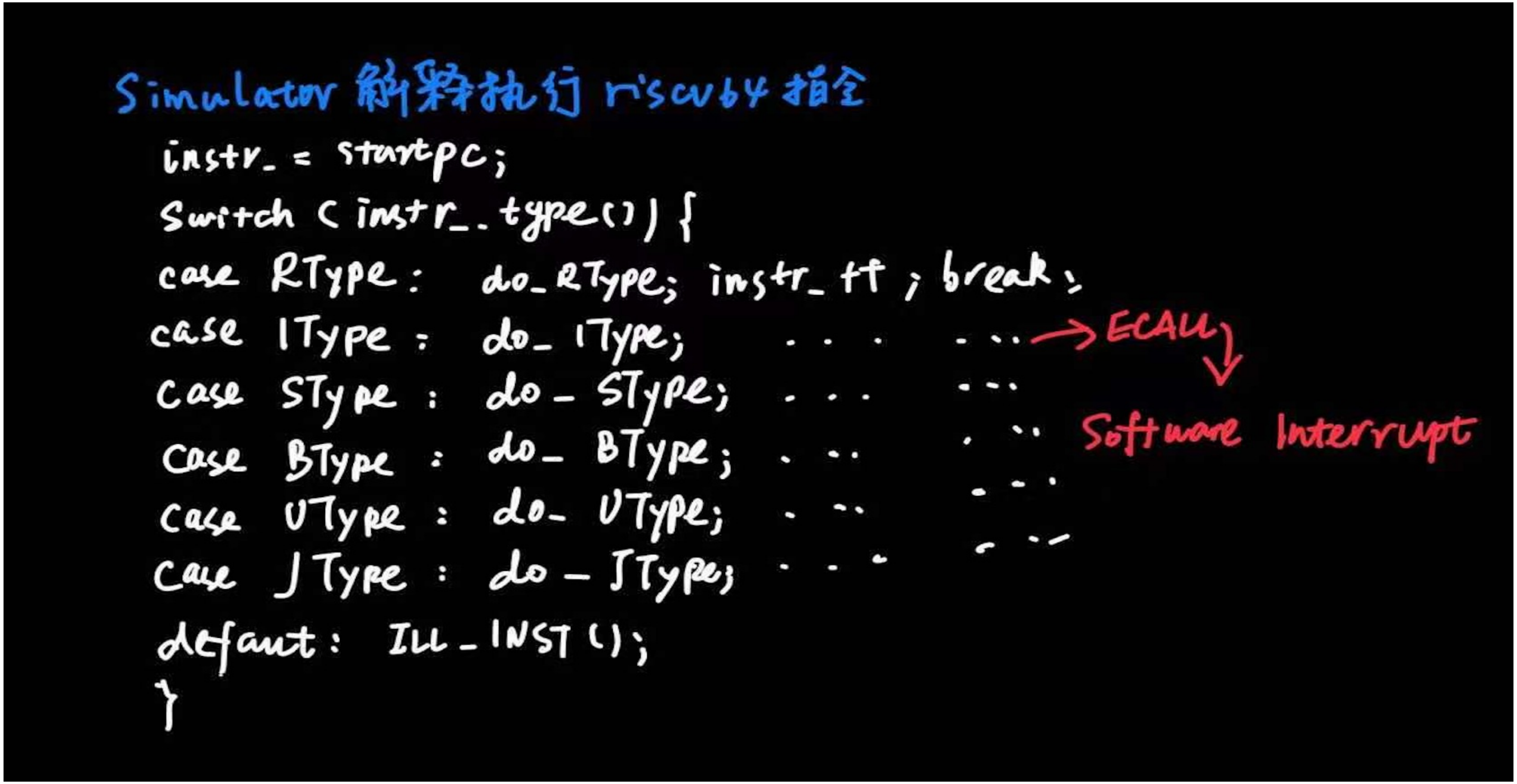
```
0x7f33d96ab21c  266  00001d37  lui    s10, 0x1         0000000000001000  (266)  int64:4096 uint64:4096
0x7f33d96ab220  267  016d0d33  add    s10, s10, s6     000055e1e0e7bfc0  (267)  int64:94428629286848 uint64:94428629286848
0x7f33d96ab224  268  540d3f83  ld     t6, 1344(s10)    000055e1e0b163f8  (268)  int64:94428625724408 uint64:94428625724408 <-- [addr: 55e1e0e7c500]
0x7f33d96ab228  269  00000317  auipc  t1, 0x0          00007f33d96ab228  (269)  int64:139860667707944 uint64:139860667707944
0x7f33d96ab22c                                                                                                              8]
0x7f33d96ab230                                                                                                              0]
0x7f33d96ab234
Call to host fun                                                                                                            f33cd473d7
0 , 00000000
Returned 7f33d85
0x55e1e0b163f8
0x7f33d96ab238
```

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

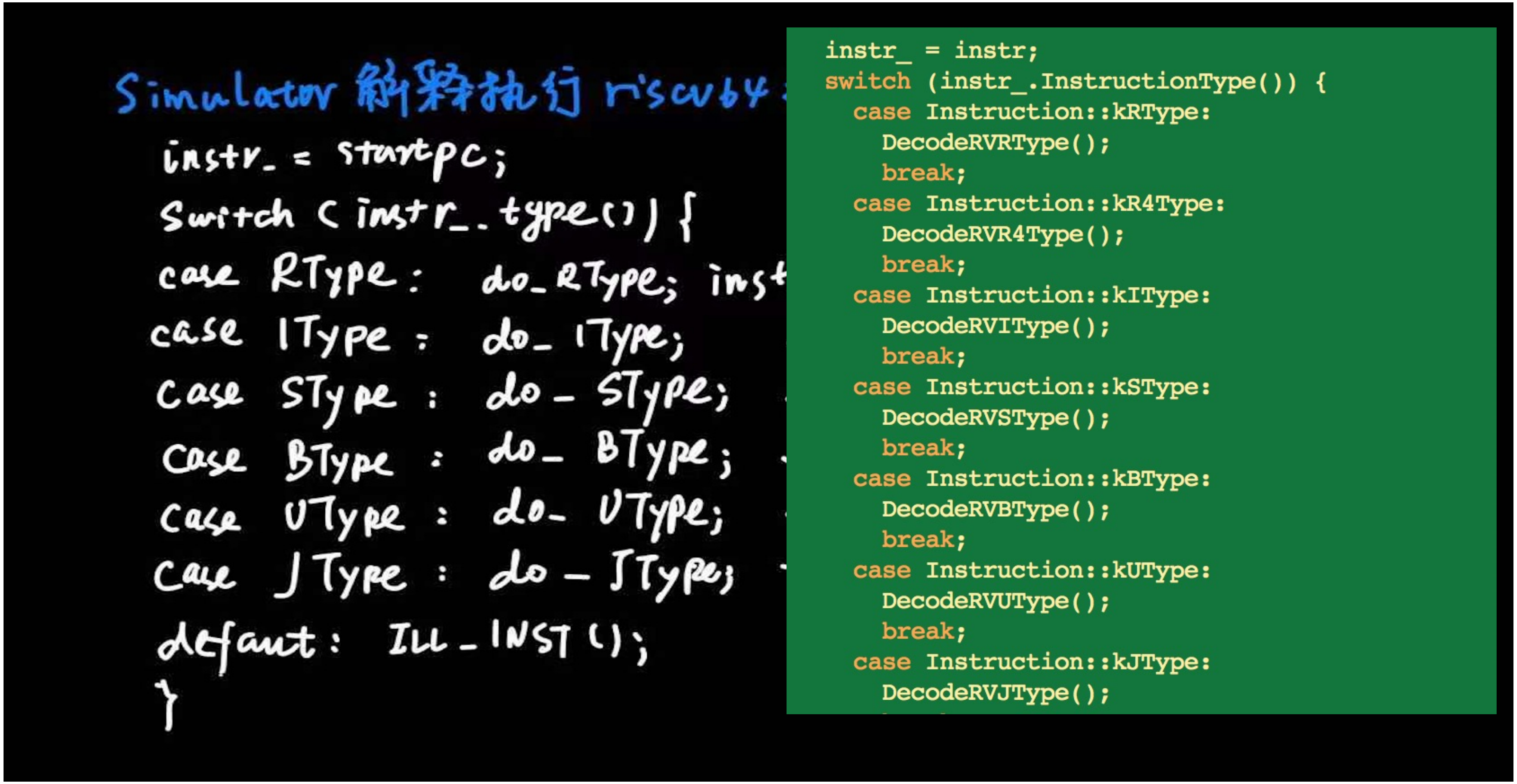
These two instructions cause a precise requested trap to the supporting execution environment.

The ECALL instruction is used to make a service request to the execution environment. The EEI will define how parameters for the service request are passed, but usually these will be in defined locations in the integer register file.

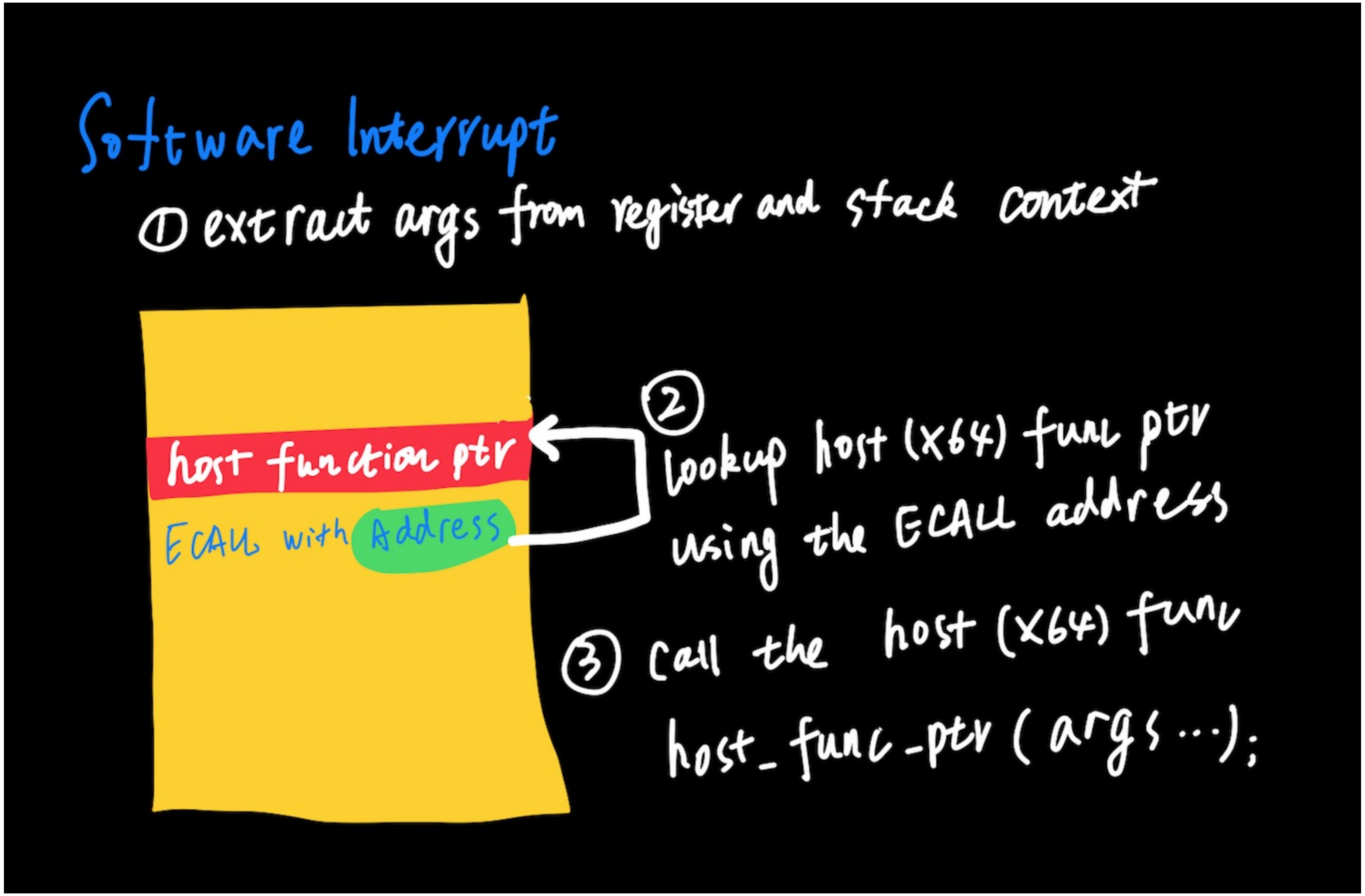
Simulator的解释执行过程



Simulator的解释执行过程



ECALL的处理流程：software interrupt



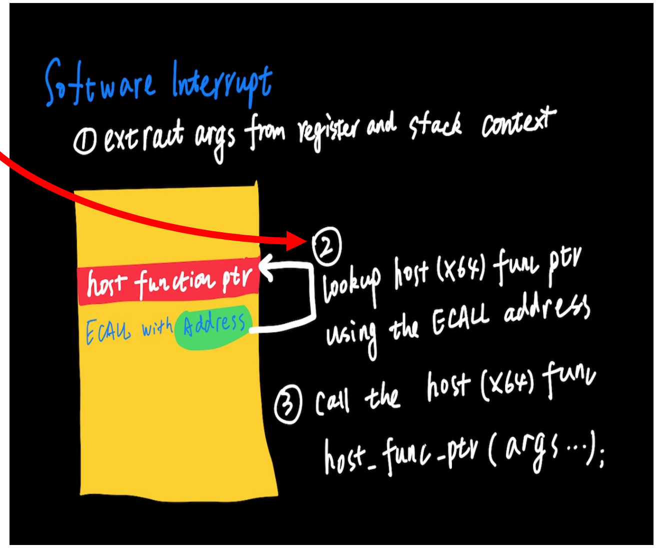
```
// Software interrupt instructions are used by the simulator to call into the
// C-based V8 runtime. They are also used for debugging with simulator.
void Simulator::SoftwareInterrupt() {
  // There are two instructions that could get us here, the ebreak or ecall
  // instructions are "SYSTEM" class opcode distinuished by Imm12Value field w/
  // the rest of instruction fields being zero
  int32_t func = instr_.Imm12Value();
  // We first check if we met a call_rt_redirected.
  if (instr_.InstructionBits() == rtCallRedirInstr) {  // ECALL
    Redirection* redirection = Redirection::FromInstruction(instr_.instr());

    int64_t* stack_pointer = reinterpret_cast<int64_t*>(get_register(sp));

    int64_t arg0 = get_register(a0);
    int64_t arg1 = get_register(a1);
    int64_t arg2 = get_register(a2);
    int64_t arg3 = get_register(a3);
    int64_t arg4 = get_register(a4);
    int64_t arg5 = get_register(a5);
    int64_t arg6 = get_register(a6);
    int64_t arg7 = get_register(a7);
    int64_t arg8 = stack_pointer[0];
    int64_t arg9 = stack_pointer[1];
    STATIC_ASSERT(kMaxCParameters == 10);

    bool fp_call =
        (redirection->type() == ExternalReference::BUILTIN_FP_FP_CALL) ||
        (redirection->type() == ExternalReference::BUILTIN_COMPARE_CALL) ||
        (redirection->type() == ExternalReference::BUILTIN_FP_CALL) ||
        (redirection->type() == ExternalReference::BUILTIN_FP_INT_CALL);

    // This is dodgy but it works because the C entry stubs are never moved.
    // See comment in codegen-arm.cc and bug 1242173.
    int64_t saved_ra = get_register(ra);

    intptr_t external =
        reinterpret_cast<intptr_t>(redirection->external_function());
```

v8/src/execution/riscv64/simulator-riscv64.cc

ExternalReference的Table和Redirect链表

- Q：模拟器jalr t6的目标地址是从哪里来的？
- A：从ExternalReferenceTable中来
- Q：ECALL处理的第二步骤中，lookup的信息是从哪里来的？
- A： 从Redirect链表中来
- Q：ERT是什么？
- A：host function index和ECALL指令的映射表
- Q：ERT是什么时候建立的？
- A：d8初始化过程中，IsolateData建立后
- Q：Redirect链表是什么
- A：host function地址和ECALL指令的组合结构体

0x7f33d96ab21c	266	00001d37	lui	s10, 0x1	0000000000001000
0x7f33d96ab220	267	016d0d33	add	s10, s10, s6	000055e1e0e7bfc0
0x7f33d96ab224	268	540d3f83	ld	t6, 1344(s10)	000055e1e0b163f8



ExternalReferenceTable和Redirect链表的建立过程: d8初始化的一部分

```
void ExternalReferenceTable::InitializeOncePerProcess() {
  int index = 0;

  // kNullAddress is preserved through serialization/deserialization.
  AddIsolateIndependent(kNullAddress, &index);
  AddIsolateIndependentReferences(&index);
  AddBuiltins(&index);
  AddRuntimeFunctions(&index);
  AddAccessors(&index);

  CHECK_EQ(kSizeIsolateIndependent, index);
}
```

Table内容分为4个部分：

- IsolateIndependentReferences
- Builtins
- RuntimeFunctions
- Accessors

接下来以IsolateIndependentReferences部分为例，说明ERT的初始化

```
void ExternalReferenceTable::AddIsolateIndependentReferences(int* index) {
  CHECK_EQ(kSpecialReferenceCount, *index);

#define ADD_EXTERNAL_REFERENCE(name, desc) \
  AddIsolateIndependent(ExternalReference::name().address(), index);
  EXTERNAL_REFERENCE_LIST(ADD_EXTERNAL_REFERENCE)
#undef ADD_EXTERNAL_REFERENCE

  CHECK_EQ(kSpecialReferenceCount + kExternalReferenceCountIsolateIndependent,
           *index);
}
```

ExternalReferenceTable的建立过程：两个要点

```
void ExternalReferenceTable::AddIsolateIndependentReferences(int* index) {
  CHECK_EQ(kSpecialReferenceCount, *index);

#define ADD_EXTERNAL_REFERENCE(name, desc) \
  AddIsolateIndependent(ExternalReference::name().address(), index);
  EXTERNAL_REFERENCE_LIST(ADD_EXTERNAL_REFERENCE)
#undef ADD_EXTERNAL_

  CHECK_EQ(kSpecialR
              *index);
}
```

```
#define EXTERNAL_REFERENCE_LIST(V)                                         \
  V(abort_with_reason, "abort_with_reason")                                \
  V(address_of_builtin_subclassing_flag, "FLAG_builtin_subclassing")       \
  V(address_of_double_abs_constant, "double_absolute_constant")            \
  V(address_of_double_neg_constant, "double_negate_constant")              \
  V(address_of_enable_experimental_regexp_engine,                          \
    "address_of_enable_experimental_regexp_engine")                        \
  V(address_of_float_abs_constant, "float_absolute_constant")              \
  V(address_of_float_neg_constant, "float_negate_constant")                \
  V(address_of_harmony_regexp_match_indices_flag,                          \
    "FLAG_harmony_regexp_match_indices")                                   \
  V(address_of_min_int, "LDoubleConstant::min_int")                        \
  V(address_of_mock_arraybuffer_allocator_flag,                            \
    "FLAG_mock_arraybuffer_allocator")                                     \
  V(address_of_one_half, "LDoubleConstant::one_half")                      \
  V(address_of_runtime_stats_flag, "TracingFlags::runtime_stats")          \
  V(address_of_the_hole_nan, "the_hole_nan")                               \
  V(address_of_uint32_bias, "uint32_bias")                                 \
  V(address_of_wasm_i8x16_swizzle_mask, "wasm_i8x16_swizzle_mask")         \
  V(address_of_wasm_i8x16_popcnt_mask, "wasm_i8x16_popcnt_mask")           \
```

两个要点：
1.AddIsolateIndependent函数
2.函数中的宏展开

ExternalReferenceTable的建立过程：

要点一：AddIsolateIndependent函数

```
void ExternalReferenceTable::AddIsolateIndependent(Address address,
                                                   int* index) {
  ref_addr_isolate_independent_[(*index)++] = address;
}
```

依次将address填入ref_addr_isolate_independent表中

ExternalReferenceTable的建立过程：
要点二：address从哪里来？Redirect

```
void ExternalReferenceTable::AddIsolateIndependentReferences(int* index) {
   AddIsolateIndependent(ExternalReference::abort_with_reason().address(), index);
   AddIsolateIndependent(ExternalReference::address_of_builtin_subclassing_flag().address(), index);
   AddIsolateIndependent(ExternalReference::address_of_double_abs_constant().address(), index);
   AddIsolateIndependent(ExternalReference::address_of_double_neg_constant().address(), index);
```

```
#define FUNCTION_REFERENCE(Name, Target)                                  \
  ExternalReference ExternalReference::Name() {                           \
    STATIC_ASSERT(IsValidExternalReferenceType<decltype(&Target)>::value); \
    return ExternalReference(Redirect(FUNCTION_ADDR(Target)));            \
  }

FUNCTION_REFERENCE(abort_with_reason, i::abort_with_reason)
```

以abort_with_reason为例说明它的ERT entry如何产生

因此，ExternalReference::abort_with_reason()展开后的代码如下，重要的点在Redirect:

```
ExternalReference ExternalReference::abort_with_reason() {
 return ExternalReference(Redirect((reinterpret_cast<v8::internal::Address>(i::abort_with_reason))))
}
```

```
396    Address address() const { return address_; }
397
```

ExternalReferenceTable的建立过程：Redirect如何进行

```
// static
Address ExternalReference::Redirect(Address address, Type type) {
#ifdef USE_SIMULATOR
  return SimulatorBase::RedirectExternalReference(address, type);
#else
  return address;
#endif
}
```

只有USE_SIMULATOR的情况下，才要对地址和Type进行特殊处理，也就是Redirect

```
Address SimulatorBase::RedirectExternalReference(Address external_function,
                                                 ExternalReference::Type type) {
  base::MutexGuard lock_guard(Simulator::redirection_mutex());
  Redirection* redirection = Redirection::Get(external_function, type);
  return redirection->address_of_instruction();
}
```

ExternalReferenceTable的建立过程：Redirect如何进行

```
Address SimulatorBase::RedirectExternalReference(Address external_function,
                                                 ExternalReference::Type type) {
  base::MutexGuard lock_guard(Simulator::redirection_mutex());
  Redirection* redirection = Redirection::Get(external_function, type);
  return redirection->address_of_instruction();
}
  return redirection->address_of_instruction();
}
```

```
Redirection* Redirection::Get(Address external_function,
                              ExternalReference::Type type) {
  Redirection* current = Simulator::redirection();
  for (; current != nullptr; current = current->next_) {
    if (current->external_function_ == external_function &&
        current->type_ == type) {
      return current;
    }
  }
  return new Redirection(external_function, type);
}
```

Get函数获得Simulator中的Redirect指针成员，遍历搜索是否已经存在地址和type与传入的参数匹配的entry，如果存在返回匹配的指针，否则new一个新的Redirection对象.

ExternalReferenceTable的建立过程：Redirect如何进行

```
Redirection::Redirection(Address external_function,
                         ExternalReference::Type type)
    : external_function_(external_function), type_(type), next_(nullptr) {
  next_ = Simulator::redirection();
  base::MutexGuard lock_guard(Simulator::i_cache_mutex());
  Simulator::SetRedirectInstruction(
      reinterpret_cast<Instruction*>(address_of_instruction()));
  Simulator::FlushICache(Simulator::i_cache(),
                         reinterpret_cast<void*>(&instruction_),
                         sizeof(instruction_));
  Simulator::set_redirection(this);
#if ABI_USES_FUNCTION_DESCRIPTORS
  function_descriptor_[0] = reinterpret_cast<intptr_t>(&instruction_);
  function_descriptor_[1] = 0;
  function_descriptor_[2] = 0;
#endif
}
```

```
class Redirection {
 private:
  Address external_function_;
  uint32_t instruction_;
  ExternalReference::Type type_;
  Redirection* next_;
#if ABI_USES_FUNCTION_DESCRIPTORS
  intptr_t function_descriptor_[3];
#endif
};
```

Redirection对象的成员：

1. external函数的地址名
2. 一条32bit的指令（会填ECALL）
3. 指向下一个Redirection的指针，用于形成链表
4. 函数描述符(RISCV64没有用)

Redirection的构造函数中关键操作是SetRedirectInstruction

```
void Simulator::SetRedirectInstruction(Instruction* instruction) {
  instruction->SetInstructionBits(rtCallRedirInstr);
}
```

设置成ECALL

ExternalReferenceTable和Redirect链表的建立过程: d8初始化的一部分

```
void ExternalReferenceTable::InitializeOncePerProcess() {
  int index = 0;

  // kNullAddress is preserved through serialization/deserialization.
  AddIsolateIndependent(kNullAddress, &index);
  AddIsolateIndependentReferences(&index);
  AddBuiltins(&index);
  AddRuntimeFunctions(&index);
  AddAccessors(&index);

  CHECK_EQ(kSizeIsolateIndependent, index);
}
```

Table内容分为4个部分：

- IsolateIndependentReferences
- Builtins
- RuntimeFunctions
- Accessors

完成后所有的外部函数都经过Redirect， Redirect后， 每个Redirect的instruction的地址（也就是ECALL指令的地址）被返回， 填入了ref_addr_isolate_independent表格

ExternalReferenceTable是如何用到的：

汇编阶段的li指令

```
void TurboAssembler::li(Register dst, ExternalReference value, LiFlags mode) {
  // TODO(jgruber,v8:8887): Also consider a root-relative load when generating
  // non-isolate-independent code. In many cases it might be cheaper than
  // embedding the relocatable value.
  if (root_array_available_ && options().isolate_independent_code) {
    IndirectLoadExternalReference(dst, value);
    return;
  }
  li(dst, Operand(v
```

```
void TurboAssemblerBase::IndirectLoadExternalReference(
    Register destination, ExternalReference reference) {
  CHECK(root_array_available_);

  if (IsAddressableThroughRootRegister(isolate(), reference)) {
    // Some external references can be efficiently loaded as an offset from
    // kRootRegister.
    intptr_t offset =
        RootRegisterOffsetForExternalReference(isolate(), reference);
    LoadRootRegisterOffset(destination, offset);
  } else {
    // Otherwise, do a memory load from the external reference table.
    LoadRootRelative(
        destination,
        RootRegisterOffsetForExternalReferenceTableEntry(isolate(), reference));
  }
}
```

ExternalReferenceTable是如何用到的：IndirectLoadExternalReference

```
void TurboAssemblerBase::IndirectLoadExternalReference(
    Register destination, ExternalReference reference) {
  CHECK(root_array_available_);

  if (IsAddressableThroughRootRegister(isolate(), reference)) {
    // Some external references can be efficiently loaded as an offset from
    // kRootRegister.
    intptr_t offset =
        RootRegisterOffsetForExternalReference(isolate(), reference);
    LoadRootRegisterOffset(destination, offset);
  } else {
    // Otherwise, do a memory load from the external reference table.
    LoadRootRelative(
        destination,
        RootRegisterOffsetForExternalReferenceTableEntry(isolate(), reference));
  }
}
```

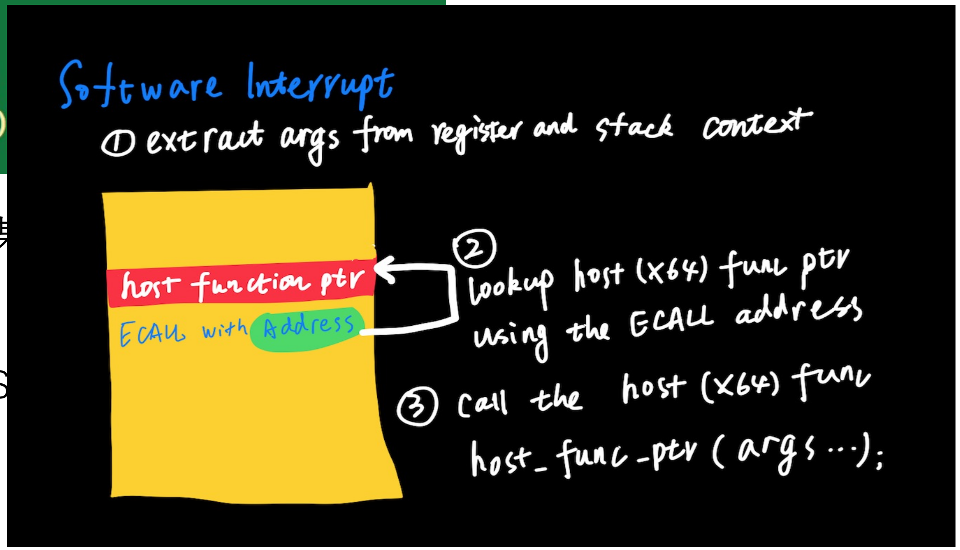
ExternalReferenceTable是如何用到的：LoadRootRelative

```
void TurboAssemblerBase::IndirectLoadExternalReference(
    Register destination, ExternalReference reference) {
  CHECK(root_array_available_);

  if (IsAddressableThroughRootRegister(isolate(), reference)) {
    // Some external references can be efficiently loaded as an offset from
    // kRootRegister.
    intptr_t offset =
        RootRegisterOffsetForExternalReference(isolate(), reference);
    LoadRootRegisterOffset(destination, offset);
  } else {
    // Otherwise, do a memory load from the external reference table.
    LoadRootRelative(
        destination,
        RootRegisterOffsetForExternalReferenceTableEntry(isolate(), reference));
  }
}
```

ExternalReferenceTable是如何用到的：RootRegisterOffsetForExternalReferenceTableEntry

```
int32_t TurboAssemblerBase::RootRegisterOffsetForExternalReferenceTableEntry(
    Isolate* isolate, const ExternalReference& reference) {
  // Encode as an index into the external reference table stored on the
  // isolate.
  ExternalReferenceEncoder encoder(isolate);
  ExternalReferenceEncoder::Value v = encoder.Encode(reference.address());
  CHECK(!v.is_from_api());

  return IsolateData::external_reference_table_offset()
         ExternalReferenceTable::OffsetOfEntry(v.index()
}
```

return的地址：ExternalEntryTable中被填入的对应着
Redirection对象的instruction_成员的地址
li addr，external_reference
jalr addr 就执行到了ECALL指令，从而进入模拟器的S
函数



总结：

- 本次课程介绍了：
  - Simulator的目的和外部函数调用的执行过程
  - ExternalReferenceTable和Redirect链表的建立过程
  - 上述结构的使用过程

中国科学院软件研究所 Institute of Software Chinese Academy of Sciences
智能软件研究中心 Intelligent Software Research Center

Thanks

2020/07/09