从零开始的RISC-V模拟器开发

第5讲 Spike篇之设备模拟

中国科学院软件研究所
PLCT实验室

李威威 liweiwei@iscas.ac.cn
王俊强 wangjunqiang@iscas.ac.cn
吴伟 wuwei2016@iscas.ac.cn

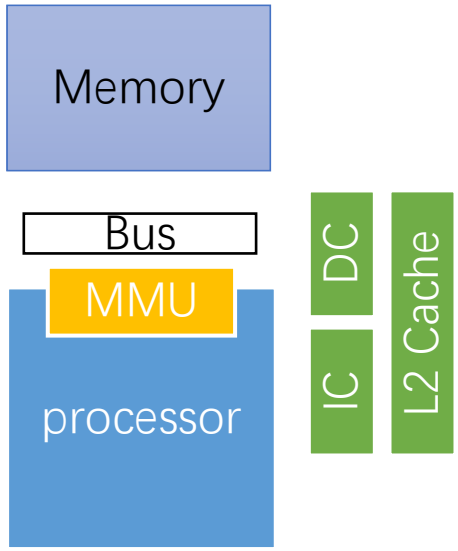
回顾

riscv/mmu.cc

```
tlb_entry_t mmu_t::fetch_slow_path(reg_t vaddr)
{
  reg_t paddr = translate(vaddr, sizeof(fetch_temp), FETCH, 0);

  if (auto host_addr = sim->addr_to_mem(paddr)) {
    return refill_tlb(vaddr, paddr, host_addr, FETCH);
  } else {
    if (!mmio_load(paddr, sizeof fetch_temp, (uint8_t*)&fetch_temp))
      throw trap_instruction_access_fault(vaddr, 0, 0);
    tlb_entry_t entry = {(char*)&fetch_temp - vaddr, paddr - vaddr};
    return entry;
  }
}
```

riscv/sim.cc

```
char* sim_t::addr_to_mem(reg_t addr) {
  if (!paddr_ok(addr))
    return NULL;
  auto desc = bus.find_device(addr);
  if (auto mem = dynamic_cast<mem_t*>(desc.second))
    if (addr - desc.first < mem->size())
      return mem->contents() + (addr - desc.first);
  return NULL;
}
```



mmio访问

riscv/mmu.cc

```
bool mmu_t::mmio_load(reg_t addr, size_t len, uint8_t* bytes)
{
  if (!mmio_ok(addr, LOAD))
    return false;

  return sim->mmio_load(addr, len, bytes);
}

bool mmu_t::mmio_store(reg_t addr, size_t len, const uint8_t*
bytes)
{
  if (!mmio_ok(addr, STORE))
    return false;

  return sim->mmio_store(addr, len, bytes);
}
```

riscv/sim.cc

```
bool sim_t::mmio_load(reg_t addr, size_t len, uint8_t* bytes)
{
  if (addr + len < addr || !paddr_ok(addr + len - 1))
    return false;
  return bus.load(addr, len, bytes);
}

bool sim_t::mmio_store(reg_t addr, size_t len, const uint8_t* bytes)
{
  if (addr + len < addr || !paddr_ok(addr + len - 1))
    return false;
  return bus.store(addr, len, bytes);
}
```

Spike设备树

设备树dts

命令： --dump-dts

```
std::string make_dts(size_t insns_per_rtc_tick, size_t cpu_hz,
                     reg_t initrd_start, reg_t initrd_end,
                     const char* bootargs,
                     std::vector<processor_t*> procs,
                     std::vector<std::pair<reg_t, mem_t*>> mems)
```

liww@liww-tm: /workspace/riscv/plct-spike/build

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

```
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    timebase-frequency = <10000000>;
    CPU0: cpu@0 {
      device_type = "cpu";
      reg = <0>;
      status = "okay";
      compatible = "riscv";
      riscv,isa = "rv64imafdc";
      mmu-type = "riscv,sv48";
      riscv,pmpregions = <16>;
      riscv,pmpgranularity = <4>;
      clock-frequency = <1000000000>;
      CPU0_intc: interrupt-controller {
        #interrupt-cells = <1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
      };
    };
  };
  memory@80000000 {
    device_type = "memory";
    reg = <0x0 0x80000000 0x0 0x80000000>;
  };
  soc {
    #address-cells = <2>;
    #size-cells = <2>;
    compatible = "ucbbar,spike-bare-soc", "simple-bus";
    ranges;
    clint@2000000 {
      compatible = "riscv,clint0";
      interrupts-extended = <&CPU0_intc 3 &CPU0_intc 7 >;
      reg = <0x0 0x2000000 0x0 0xc0000>;
    };
  };
  htif {
    compatible = "ucb,htif0";
  };
};
liww@liww-tm:build$
```

Spike设备支持

设备基础类型

riscv/devices.h

```
class abstract_device_t {
 public:
  virtual bool load(reg_t addr, size_t len, uint8_t* bytes) = 0;
  virtual bool store(reg_t addr, size_t len, const uint8_t* bytes) =
0;
  virtual ~abstract_device_t() {}
};
```

Spike设备支持

设备基础类型

riscv/devices.h

```
class abstract_device_t {
 public:
  virtual bool load(reg_t addr, size_t len, uint8_t* bytes) = 0;
  virtual bool store(reg_t addr, size_t len, const uint8_t* bytes) =
0;
  virtual ~abstract_device_t() {}
};
```

```
class processor_t : public
abstract_device_t
{
  …
};
```

Spike设备支持

设备基础类型

riscv/devices.h

```
class abstract_device_t {
 public:
  virtual bool load(reg_t addr, size_t len, uint8_t* bytes) = 0;
  virtual bool store(reg_t addr, size_t len, const uint8_t* bytes) =
0;
  virtual ~abstract_device_t() {}
};
```

- bus
- mem
- rom
- clint
- plugin

Spike设备支持

riscv/sim.cc：sim_t::sim_t(...)

```
for (auto& x : mems)
  bus.add_device(x.first, x.second);

for (auto& x : plugin_devices)
  bus.add_device(x.first, x.second);
...
//handle clic
clint.reset(new clint_t(procs, CPU_HZ / INSNS_PER_RTC_TICK,
real_time_clint));
reg_t clint_base;
if (fdt_parse_clint(fdt, &clint_base, "riscv,clint0")) {
  bus.add_device(CLINT_BASE, clint.get());
} else {
  bus.add_device(clint_base, clint.get());
}
```

```
void sim_t::set_rom()
{
 ...
  boot_rom.reset(new rom_device_t(rom));
  bus.add_device(DEFAULT_RSTVEC, boot_rom.get());
}
```

bus设备支持

riscv/devices.h

```
class bus_t : public abstract_device_t {
 public:
  bool load(reg_t addr, size_t len, uint8_t* bytes);
  bool store(reg_t addr, size_t len, const uint8_t* bytes);
  void add_device(reg_t addr, abstract_device_t* dev);

  std::pair<reg_t, abstract_device_t*> find_device(reg_t addr);

 private:
  std::map<reg_t, abstract_device_t*> devices;
};
```

riscv/devices.cc

```
void bus_t::add_device(reg_t addr, abstract_device_t* dev)
{
  devices[addr] = dev;
}
```

```
bool bus_t::load(reg_t addr, size_t len, uint8_t* bytes)
{
  // Find the device with the base address closest to but
  // less than addr (price-is-right search)
  auto it = devices.upper_bound(addr);
  if (devices.empty() || it == devices.begin()) {
    // Either the bus is empty, or there weren't
    // any items with a base address <= addr
    return false;
  }
  // Found at least one item with base address <= addr
  // The iterator points to the device after this, so
  // go back by one item.
  it--;
  return it->second->load(addr - it->first, len, bytes);
}
```

mem设备支持

```
class mem_t : public abstract_device_t {
 public:
  mem_t(reg_t size);
  mem_t(const mem_t& that) = delete;
  ~mem_t();

  bool load(reg_t addr, size_t len, uint8_t* bytes) { return load_store(addr, len, bytes, false); }
  bool store(reg_t addr, size_t len, const uint8_t* bytes) { return load_store(addr, len, const_cast<uint8_t*>(bytes),
true); }
  char* contents(reg_t addr);
  reg_t size() { return sz; }

 private:
  bool load_store(reg_t addr, size_t len, uint8_t* bytes, bool store);

  std::map<reg_t, char*> sparse_memory_map;
  reg_t sz;
};
```

```
mem_t::mem_t(reg_t size)
  : sz(size)
{
  if (size == 0 || size % PGSIZE != 0)
    throw std::runtime_error("memory size must be a positive multiple of 4
KiB");
}
```

mem设备支持

spike_main/spike.cc

```
parser.option('m', 0, 1, [&](const char* s){mems =
make_mems(s);});
...
 if (mems.empty())
   mems = make_mems("2048");
```

```
 for (auto& x : mems)
   bus.add_device(x.first, x.second);
```

命令	描述
-m<n> - m<a:m, b:n,...>	Provide <n> MiB of target memory [default 2048] Provide memory regions of size m and n bytes at base addresses a and b (with 4 KiB alignment)

```
char* sim_t::addr_to_mem(reg_t addr) {
 if (!paddr_ok(addr))
   return NULL;
 auto desc = bus.find_device(addr);
 if (auto mem = dynamic_cast<mem_t*>(desc.second))
   if (addr - desc.first < mem->size())
     return mem->contents() + (addr - desc.first);
 return NULL;
}
```

mem设备支持

```
char* mem_t::contents(reg_t addr) {
  reg_t ppn = addr >> PGSHIFT, pgoff = addr % PGSIZE;
  auto search = sparse_memory_map.find(ppn);
  if (search == sparse_memory_map.end()) {
    auto res = (char*)calloc(PGSIZE, 1);
    if (res == nullptr)
      throw std::bad_alloc();
    sparse_memory_map[ppn] = res;
    return res + pgoff;
  }
  return search->second + pgoff;
}
```

rom设备支持

riscv/devices.h

```
class rom_device_t : public abstract_device_t {
 public:
  rom_device_t(std::vector<char> data);
  bool load(reg_t addr, size_t len, uint8_t* bytes);
  bool store(reg_t addr, size_t len, const uint8_t* bytes);
  const std::vector<char>& contents() { return data; }
 private:
  std::vector<char> data;
};
```

riscv/rom.cc

```
rom_device_t::rom_device_t(std::vector<char> data)
  : data(data)
{
}

bool rom_device_t::load(reg_t addr, size_t len, uint8_t* bytes)
{
  if (addr + len > data.size())
    return false;
  memcpy(bytes, &data[addr], len);
  return true;
}

bool rom_device_t::store(reg_t addr, size_t len, const uint8_t* bytes)
{
  return false;
}
```

clint设备支持

```
class clint_t : public abstract_device_t {
 public:
  clint_t(std::vector<processor_t*>&, uint64_t freq_hz, bool real_time);
  bool load(reg_t addr, size_t len, uint8_t* bytes);
  bool store(reg_t addr, size_t len, const uint8_t* bytes);
  size_t size() { return CLINT_SIZE; }
  void increment(reg_t inc);
 private:
  typedef uint64_t mtime_t;
  typedef uint64_t mtimecmp_t;
  typedef uint32_t msip_t;
  std::vector<processor_t*>& procs;
  uint64_t freq_hz;
  bool real_time;
  uint64_t real_time_ref_secs;
  uint64_t real_time_ref_usecs;
  mtime_t mtime;
  std::vector<mtimecmp_t> mtimecmp;
};
```

```
/* 0000 msip hart 0
 * 0004 msip hart 1
 * 4000 mtimecmp hart 0 lo
 * 4004 mtimecmp hart 0 hi
 * 4008 mtimecmp hart 1 lo
 * 400c mtimecmp hart 1 hi
 * bff8 mtime lo
 * bffc mtime hi
 */
```

```
clint_t::clint_t(std::vector<processor_t*>& procs, uint64_t
freq_hz, bool real_time)
  : procs(procs), freq_hz(freq_hz), real_time(real_time), mtime(0),
mtimecmp(procs.size())
{
  struct timeval base;

  gettimeofday(&base, NULL);

  real_time_ref_secs = base.tv_sec;
  real_time_ref_usecs = base.tv_usec;
}
```

clint设备支持

```
parser.option(0, "real-time-clint", 0, [&](const char *s){real_time_clint = true;});
```

riscv/sim.cc： sim_t::sim_t(...)

```
clint.reset(new clint_t(procs, CPU_HZ / INSNS_PER_RTC_TICK, real_time_clint));
reg_t clint_base;
if (fdt_parse_clint(fdt, &clint_base, "riscv,clint0")) {
  bus.add_device(CLINT_BASE, clint.get());
} else {
  bus.add_device(clint_base, clint.get());
}
```

clint设备支持

```
void sim_t::step(size_t n)
{
 for (size_t i = 0, steps = 0; i < n; i += steps)
 {
  steps = std::min(n - i, INTERLEAVE - current_step);
  procs[current_proc]->step(steps);

  current_step += steps;
  if (current_step == INTERLEAVE)
  {
   current_step = 0;
   procs[current_proc]->get_mmu()->yield_load_reservation();
   if (++current_proc == procs.size()) {
    current_proc = 0;
    clint->increment(INTERLEAVE / INSNS_PER_RTC_TICK);
   }

   host->switch_to();
  }
 }
}
```

```
void clint_t::increment(reg_t inc)
{
 if (real_time) {
  struct timeval now;
  uint64_t diff_usecs;

  gettimeofday(&now, NULL);
  diff_usecs = ((now.tv_sec - real_time_ref_secs) * 1000000) + (now.tv_usec
- real_time_ref_usecs);
  mtime = diff_usecs * freq_hz / 1000000;
 } else {
  mtime += inc;
 }
 for (size_t i = 0; i < procs.size(); i++) {
  procs[i]->state.mip &= ~MIP_MTIP;
  if (mtime >= mtimecmp[i])
   procs[i]->state.mip |= MIP_MTIP;
 }
}
```

Spike设备扩展

Mmio plugin设备

riscv/devices.h

```
class mmio_plugin_device_t : public abstract_device_t {
 public:
  mmio_plugin_device_t(const std::string& name, const
std::string& args);
  virtual ~mmio_plugin_device_t() override;

  virtual bool load(reg_t addr, size_t len, uint8_t* bytes) override;
  virtual bool store(reg_t addr, size_t len, const uint8_t* bytes)
override;

 private:
  mmio_plugin_t plugin;
  void* user_data;
};
```

riscv/devices.cc

```
mmio_plugin_device_t::mmio_plugin_device_t(const std::string& name,
                                           const std::string& args)
  : plugin(mmio_plugin_map().at(name)),
user_data((*plugin.alloc)(args.c_str()))
{
}

mmio_plugin_device_t::~mmio_plugin_device_t()
{
  (*plugin.dealloc)(user_data);
}

bool mmio_plugin_device_t::load(reg_t addr, size_t len, uint8_t* bytes)
{
  return (*plugin.load)(user_data, addr, len, bytes);
}

bool mmio_plugin_device_t::store(reg_t addr, size_t len, const uint8_t*
bytes)
{
  return (*plugin.store)(user_data, addr, len, bytes);
}
```

Spike plugin设备接口

riscv/mmio_plugin.h

```
typedef struct {
  // Allocate user data for an instance of the plugin. The parameter is a simple
  // c-string containing arguments used to construct the plugin. It returns a
  // void* to the allocated data.
  void* (*alloc)(const char*);

  // Load a memory address of the MMIO plugin. The parameters are the user_data
  // (void*), memory offset (reg_t), number of bytes to load (size_t), and the
  // buffer into which the loaded data should be written (uint8_t*). Return true
  // if the load is successful and false otherwise.
  bool (*load)(void*, reg_t, size_t, uint8_t*);

  // Store some bytes to a memory address of the MMIO plugin. The parameters are
  // the user_data (void*), memory offset (reg_t), number of bytes to store
  // (size_t), and the buffer containing the data to be stored (const uint8_t*).
  // Return true if the store is successful and false otherwise.
  bool (*store)(void*, reg_t, size_t, const uint8_t*);

  // Deallocate the data allocated during the call to alloc. The parameter is a
  // pointer to the user data allocated during the call to alloc.
  void (*dealloc)(void*);
} mmio_plugin_t;
```

Spike plugin设备注册

riscv/devices.cc

```
// Type for holding all registered MMIO plugins by name.
using mmio_plugin_map_t = std::map<std::string,
mmio_plugin_t>;

void register_mmio_plugin(const char* name_cstr,
                          const mmio_plugin_t* mmio_plugin)
{
  std::string name(name_cstr);
  if (!mmio_plugin_map().emplace(name, *mmio_plugin).second) {
    throw std::runtime_error("Plugin \"" + name + "\" already
registered!");
  }
}
```

Spike设备扩展

以块设备为例

```
__attribute__((constructor)) static void on_load()
{
  static mmio_plugin_t test_mmio_plugin = {
      test_mmio_plugin_alloc,
      test_mmio_plugin_load,
      test_mmio_plugin_store,
      test_mmio_plugin_dealloc
  };

  register_mmio_plugin("test_mmio_plugin",
&test_mmio_plugin);
}
```

```
{
    vblock@10000000 {
    compatible = "plct,vblock";
    reg = <0x0 0x10000000 0x0 0xa00000>;
    len = <0x5000>;
  };
```

代码仓库: https://github.com/plctlab/plct-spike/tree/snapshot-develop

Spike设备扩展

以块设备为例

```
void* test_mmio_plugin_alloc(const char* args)
{
   printf("ALLOC -- ARGS=%s\n", args);
   int fd = open(args, O_RDWR, 0);
   if(fd==-1){
     printf("open block disk file failed.\n");
     exit(-1);
   }
   struct stat st; //定义文件信息结构体
   int r=fstat(fd,&st);
   if(r==-1){
     printf("get file size failed. \n");
     close(fd);
     exit(-1);
   }
   len=st.st_size;
   void *
p=mmap(NULL,len,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
   return p;
}
```

```
bool test_mmio_plugin_load(void* self, reg_t addr, size_t len,
uint8_t* bytes)
{
   memcpy(bytes, (char *)self + addr, len);
   return true;
}

bool test_mmio_plugin_store(void* self, reg_t addr, size_t len,
const uint8_t* bytes)
{
   memcpy((char *)self + addr, bytes, len);
   msync((char *)self + addr,len,0);
   return true;
}

void test_mmio_plugin_dealloc(void* self)
{
   munmap(self, len);
   printf("DEALLOC -- SELF=%p\n", self);
}
```

HTIF

Host/Target Interface

- riscv-fesvr runs on your host computer and communicates with the target design
- a non-standard tool for Berkeley processors

```
class sim_t : public htif_t, public simif_t
{
  ...
}
```

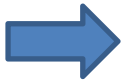
```
int sim_t::run()
{
  host = context_t::current();
  target.init(sim_thread_main, this);
  return htif_t::run();
}
```

HTIF初始化

```
htif_t::htif_t(int argc, char** argv) : htif_t()
{
  parse_arguments(argc, argv);
  register_devices();
}
```

HTIF参数解析

```
htif_t::htif_t(int argc, char** argv) : htif_t()
{
  parse_arguments(argc, argv);
  register_devices();
}
```



```
void htif_t::parse_arguments(int argc, char ** argv)
{
    ...
    if (c == -1) break;
 retry:
   switch (c) {
    case 'h': usage(argv[0]);
      throw std::invalid_argument("User queried htif_t help text");
    case HTIF_LONG_OPTIONS_OPTIND:
      if (optarg) dynamic_devices.push_back(new rfb_t(atoi(optarg)));
      else        dynamic_devices.push_back(new rfb_t);
      break;
    case HTIF_LONG_OPTIONS_OPTIND + 1:
      // [TODO] Remove once disks are supported again
      throw std::invalid_argument("--disk/+disk unsupported (use a
ramdisk)");
      dynamic_devices.push_back(new disk_t(optarg));
      Break;
    ...
 }
```

HTIF选项

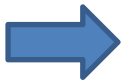
```
#define HTIF_USAGE_OPTIONS \
"HOST OPTIONS\n\
  -h, --help             Display this help and exit\n\
  +h,  +help\n\
      +permissive        The host will ignore any unparsed options up until\n\
                        +permissive-off (Only needed for VCS)\n\
      +permissive-off    Stop ignoring options. This is mandatory if using\n\
                        +permissive (Only needed for VCS)\n\
     --rfb=DISPLAY       Add new remote frame buffer on display DISPLAY\n\
      +rfb=DISPLAY         to be accessible on 5900 + DISPLAY (default = 0)\n\
     --signature=FILE    Write torture test signature to FILE\n\
      +signature=FILE\n\
     --signature-granularity=VAL          Size of each line in signature.\n\
      +signature-granularity=VAL\n\
     --chroot=PATH       Use PATH as location of syscall-servicing binaries\n\
      +chroot=PATH\n\
     --payload=PATH      Load PATH memory as an additional ELF payload\n\
      +payload=PATH\n\
\n\
HOST OPTIONS (currently unsupported)\n\
     --disk=DISK         Add DISK device. Use a ramdisk since this isn't\n\
      +disk=DISK           supported\n\
\n\
```

HTIF设备注册

```
htif_t::htif_t(int argc, char** argv) : htif_t()
{
  parse_arguments(argc, argv);
  register_devices();
}
```

```
void htif_t::register_devices()
{

device_list.register_device(&syscall_proxy);
  device_list.register_device(&bcd);
  for (auto d : dynamic_devices)
    device_list.register_device(d);
}
```

```
  device_list_t device_list;
  syscall_t syscall_proxy;
  bcd_t bcd;
  std::vector<device_t*> dynamic_devices;
```



```
void htif_t::parse_arguments(int argc, char ** argv)
{
     ...
     if (c == -1) break;
 retry:
   switch (c) {
     case 'h': usage(argv[0]);
       throw std::invalid_argument("User queried htif_t help text");
     case HTIF_LONG_OPTIONS_OPTIND:
       if (optarg) dynamic_devices.push_back(new rfb_t(atoi(optarg)));
       else        dynamic_devices.push_back(new rfb_t);
       break;
     case HTIF_LONG_OPTIONS_OPTIND + 1:
       // [TODO] Remove once disks are supported again
       throw std::invalid_argument("--disk/+disk unsupported (use a
ramdisk)");
       dynamic_devices.push_back(new disk_t(optarg));
       Break;
     ...
 }
```

device list

```
class device_list_t
{
 public:
  device_list_t();
  void register_device(device_t* dev);
  void handle_command(command_t cmd);
  void tick();

 private:
  std::vector<device_t*> devices;
  null_device_t null_device;
  size_t num_devices;
};
```

```
void device_list_t::register_device(device_t* dev)
{
  num_devices++;
  assert(num_devices < command_t::MAX_DEVICES);
  devices[num_devices-1] = dev;
}

void device_list_t::handle_command(command_t cmd)
{
  devices[cmd.device()]->handle_command(cmd);
}

void device_list_t::tick()
{
  for (size_t i = 0; i < num_devices; i++)
    devices[i]->tick();
}
```

HTIF运行

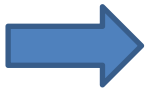
```
int htif_t::run()
{
  start();

  auto enq_func = [](std::queue<reg_t>* q, uint64_t x) { q->push(x); };
  std::queue<reg_t> fromhost_queue;
  std::function<void(reg_t)> fromhost_callback =
  std::bind(enq_func, &fromhost_queue, std::placeholders::_1);

  if (tohost_addr == 0) {
    while (true)
      idle();
  }

  while (!signal_exit && exitcode == 0)
  {
    if (auto tohost = from_target(mem.read_uint64(tohost_addr))) {
      mem.write_uint64(tohost_addr, target_endian<uint64_t>::zero);
      command_t cmd(mem, tohost, fromhost_callback);
      device_list.handle_command(cmd);
    } else {
      idle();
    }
    device_list.tick();

    if (!fromhost_queue.empty() && !mem.read_uint64(fromhost_addr)) {
      mem.write_uint64(fromhost_addr, to_target(fromhost_queue.front()));
      fromhost_queue.pop();
    }
  }

  stop();

  return exit_code();
}
```

HTIF运行

```
int htif_t::run()
{
  start();

  auto enq_func = [](std::queue<reg_t>* q, uint64_t x) { q->push(x); };
  std::queue<reg_t> fromhost_queue;
  std::function<void(reg_t)> fromhost_callback =
  std::bind(enq_func, &fromhost_queue, std::placeholders::_1);

  if (tohost_addr == 0) {
    while (true)
      idle();
  }

  while (!signal_exit && exitcode == 0)
  {
    if (auto tohost = from_target(mem.read_uint64(tohost_addr))) {
      mem.write_uint64(tohost_addr, target_endian<uint64_t>::zero);
      command_t cmd(mem, tohost, fromhost_callback);
      device_list.handle_command(cmd);
    } else {
      idle();
    }
    device_list.tick();
    if (!fromhost_queue.empty() && !mem.read_uint64(fromhost_addr)) {
      mem.write_uint64(fromhost_addr, to_target(fromhost_queue.front()));
      fromhost_queue.pop();
    }
  }

  stop();

  return exit_code();
}
```

HTIF设备处理及注册命令

```
void device_list_t::handle_command(command_t cmd)
{
  devices[cmd.device()]->handle_command(cmd);
}
```



```
void device_t::handle_command(command_t cmd)
{
  command_handlers[cmd.cmd()](cmd);
}
```

```
void device_t::register_command(size_t cmd, command_func_t handler, const char* name)
{
  assert(cmd < command_t::MAX_COMMANDS);
  assert(strlen(name) < IDENTITY_SIZE);
  command_handlers[cmd] = handler;
  command_names[cmd] = name;
}
```

HTIF设备结构

```
class device_t
{
 public:
  device_t();
  virtual ~device_t() {}
  virtual const char* identity() = 0;
  virtual void tick() {}

  void handle_command(command_t cmd);

 protected:
  typedef std::function<void(command_t)> command_func_t;
  void register_command(size_t, command_func_t, const char*);

 private:
  device_t& operator = (const device_t&); // disallow
  device_t(const device_t&); // disallow
  static const size_t IDENTITY_SIZE = 64;
  void handle_null_command(command_t cmd);
  void handle_identify(command_t cmd);

  std::vector<command_func_t> command_handlers;
  std::vector<std::string> command_names;
};
```

fesvr/device.h

HTIF命令结构

```
class device_t
{
 public:
  device_t();
  virtual ~device_t() {}
  virtual const char* identity() = 0;
  virtual void tick() {}

  void handle_command(command_t cmd);

 protected:
  typedef std::function<void(command_t)> command_func_t;
  void register_command(size_t, command_func_t, const char*);

 private:
  device_t& operator = (const device_t&); // disallow
  device_t(const device_t&); // disallow
  static const size_t IDENTITY_SIZE = 64;
  void handle_null_command(command_t cmd);
  void handle_identify(command_t cmd);

  std::vector<command_func_t> command_handlers;
  std::vector<std::string> command_names;
};
```

fesvr/device.h

```
class command_t
{
 public:
  typedef std::function<void(uint64_t)> callback_t;
  command_t(memif_t& memif, uint64_t tohost, callback_t cb)
    : _memif(memif), tohost(tohost), cb(cb) {}

  memif_t& memif() { return _memif; }
  uint8_t device() { return tohost >> 56; }
  uint8_t cmd() { return tohost >> 48; }
  uint64_t payload() { return tohost << 16 >> 16; }
  void respond(uint64_t resp) { cb((tohost >> 48 << 48) | (resp << 16 >> 16)); }

  static const size_t MAX_COMMANDS = 256;
  static const size_t MAX_DEVICES = 256;

 private:
  memif_t& _memif;
  uint64_t tohost;
  callback_t cb;
};
```

HTIF系统调用代理设备

fesvr/syscall.h

```
class syscall_t : public device_t
{
 public:
  syscall_t(htif_t*);

  void set_chroot(const char* where);

 private:
  const char* identity() { return "syscall_proxy"; }

  htif_t* htif;
  memif_t* memif;
  std::vector<syscall_func_t> table;
  fds_t fds;

  void handle_syscall(command_t cmd);
  void dispatch(addr_t mm);

  std::string chroot;
  std::string do_chroot(const char* fn);
  std::string undo_chroot(const char* fn);
```

```
  reg_t sys_exit(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_openat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_read(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_pread(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_write(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_pwrite(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_close(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_lseek(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_fstat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_lstat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_fstatat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_faccessat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_fcntl(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_ftruncate(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_renameat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_linkat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_unlinkat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_mkdirat(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_getcwd(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_getmainvars(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
  reg_t sys_chdir(reg_t, reg_t, reg_t, reg_t, reg_t, reg_t, reg_t);
};
```

HTIF系统调用代理设备功能

```
syscall_t::syscall_t(htif_t* htif)
  : htif(htif), memif(&htif->memif()), table(2048)
{
  table[17] = &syscall_t::sys_getcwd;
  table[25] = &syscall_t::sys_fcntl;
  table[34] = &syscall_t::sys_mkdirat;
  table[35] = &syscall_t::sys_unlinkat;
  table[37] = &syscall_t::sys_linkat;
  table[38] = &syscall_t::sys_renameat;
  table[46] = &syscall_t::sys_ftruncate;
  table[48] = &syscall_t::sys_faccessat;
  table[49] = &syscall_t::sys_chdir;
  table[56] = &syscall_t::sys_openat;
  table[57] = &syscall_t::sys_close;
  table[62] = &syscall_t::sys_lseek;
  table[63] = &syscall_t::sys_read;
  table[64] = &syscall_t::sys_write;
  table[67] = &syscall_t::sys_pread;
  table[68] = &syscall_t::sys_pwrite;
  ...
  register_command(0, std::bind(&syscall_t::handle_syscall,
this, _1), "syscall");
...
}
```

fesvr/syscall.cc

```
void syscall_t::handle_syscall(command_t cmd)
{
  if (cmd.payload() & 1) // test pass/fail
  {
    htif->exitcode = cmd.payload();
    if (htif->exit_code())
      std::cerr << "*** FAILED *** (tohost = " << htif->exit_code() << ")" << std::endl;
    return;
  }
  else // proxied system call
    dispatch(cmd.payload());

  cmd.respond(1);
}
```

htif bcd设备

fesvr/device.h

```
class bcd_t : public device_t
{
 public:
  bcd_t();
  const char* identity() { return "bcd"; }
  void tick();

 private:
  void handle_read(command_t cmd);
  void handle_write(command_t cmd);

  std::queue<command_t> pending_reads;
};
```

```
bcd_t::bcd_t()
{
  register_command(0, std::bind(&bcd_t::handle_read, this, _1), "read");
  register_command(1, std::bind(&bcd_t::handle_write, this, _1), "write");
}

void bcd_t::handle_read(command_t cmd)
{
  pending_reads.push(cmd);
}

void bcd_t::handle_write(command_t cmd)
{
  canonical_terminal_t::write(cmd.payload());
}

void bcd_t::tick()
{
  int ch;
  if (!pending_reads.empty() && (ch = canonical_terminal_t::read()) != -1)
  {
    pending_reads.front().respond(0x100 | ch);
    pending_reads.pop();
  }
}
```

谢谢各位

欢迎提问、讨论、交流合作