# 从零开始的RISC-V模拟器开发

## 第4讲 Spike篇之内存模拟

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

- 指令执行流程： riscv/execute.cc：step()

```
auto ic_entry = _mmu->access_icache(pc);

#define ICACHE_ACCESS(i) { \
insn_fetch_t fetch = ic_entry->data; \
pc = execute_insn(this, pc, fetch); \
ic_entry = ic_entry->next; \
if (i == mmu_t::ICACHE_ENTRIES-1) break; \
if (unlikely(ic_entry->tag != pc)) break; \
if (unlikely(instret+1 == n)) break; \
    instret++; \
    state.pc = pc; \
}
```

```
static reg_t execute_insn(processor_t* p, reg_t pc,
insn_fetch_t fetch)
{
  ...
    npc = fetch.func(p, fetch.insn, pc);
  …
}
```

```
struct insn_fetch_t
{
  insn_func_t func;
  insn_t insn;
};
```

- riscv/mmu.h

```
inline icache_entry_t* access_icache(reg_t addr)
{
  icache_entry_t* entry = &icache[icache_index(addr)];
  if (likely(entry->tag == addr))
    return entry;
  return refill_icache(addr, entry);
}
```

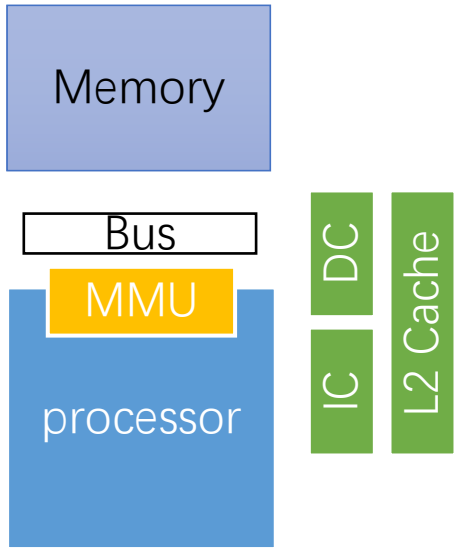
```
inline icache_entry_t* refill_icache(reg_t addr, icache_entry_t* entry)
{
  auto tlb_entry = translate_insn_addr(addr);
  insn_bits_t insn = from_le(*(uint16_t*)(tlb_entry.host_offset + addr));
  int length = insn_length(insn);
   ...
  insn_fetch_t fetch = {proc->decode_insn(insn), insn};
  entry->tag = addr;
  entry->next = &icache[icache_index(addr + length)];
  entry->data = fetch;
  ...
  return entry;
}
```

# Spike内存系统层次结构

spike_main/spike.cc

```
  std::unique_ptr<icache_sim_t> ic;
  std::unique_ptr<dcache_sim_t> dc;
  std::unique_ptr<cache_sim_t> l2;
  ...
  sim_t s(isa, priv, varch, nprocs, halted, real_time_clint, initrd_start,
initrd_end, bootargs, start_pc, mems, plugin_devices, htif_args,
std::move(hartids), dm_config, log_path, dtb_enabled, dtb_file);
  ...
  if (ic && l2) ic->set_miss_handler(&*l2);
  if (dc && l2) dc->set_miss_handler(&*l2);
  ...
  for (size_t i = 0; i < nprocs; i++)
  {
    if (ic) s.get_core(i)->get_mmu()->register_memtracer(&*ic);
    if (dc) s.get_core(i)->get_mmu()->register_memtracer(&*dc);
    ...
  }
```

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

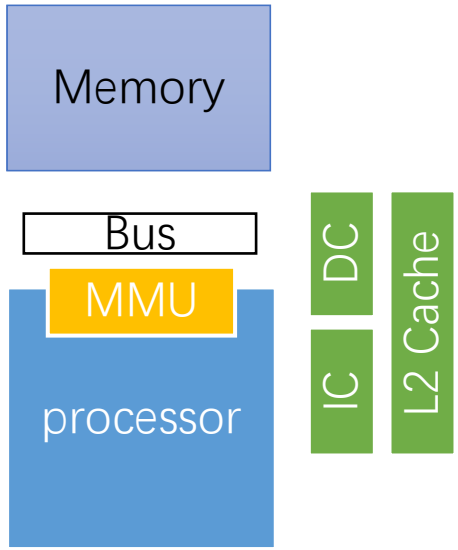


# Spike内存系统—IC和DC

## riscv/cachesim.h

```
class icache_sim_t : public cache_memtracer_t
{
 public:
  icache_sim_t(const char* config) : cache_memtracer_t(config, "I$") {}
  bool interested_in_range(uint64_t begin, uint64_t end, access_type type)
  {
    return type == FETCH;
  }
  void trace(uint64_t addr, size_t bytes, access_type type)
  {
    if (type == FETCH) cache->access(addr, bytes, false);
  }
};
```

```
class dcache_sim_t : public cache_memtracer_t
{
 public:
  dcache_sim_t(const char* config) :
cache_memtracer_t(config, "D$") {}
  bool interested_in_range(uint64_t begin, uint64_t end,
access_type type)
  {
    return type == LOAD || type == STORE;
  }
  void trace(uint64_t addr, size_t bytes, access_type type)
  {
    if (type == LOAD || type == STORE) cache->access(addr,
bytes, type == STORE);
  }
};
```

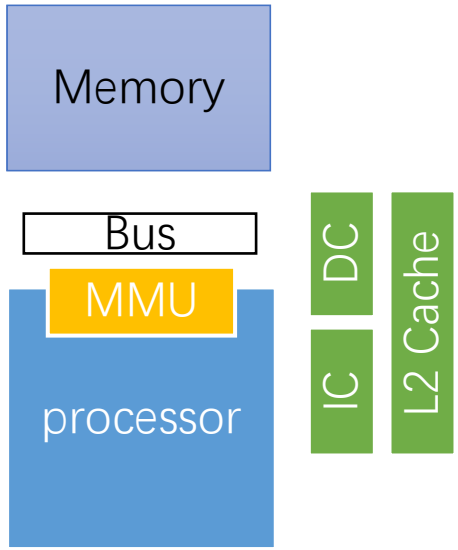


# Spike内存系统—cache_memtracer_t

riscv/cachesim.h

```
class cache_memtracer_t : public memtracer_t
{
 public:
  cache_memtracer_t(const char* config, const char* name)
  {
    cache = cache_sim_t::construct(config, name);
  }
  ~cache_memtracer_t()
  {
    delete cache;
  }
  void set_miss_handler(cache_sim_t* mh)
  {
    cache->set_miss_handler(mh);
  }
  void set_log(bool log)
  {
    cache->set_log(log);
  }
 protected:
  cache_sim_t* cache;
};
```

```
class memtracer_t
{
 public:
  memtracer_t() {}
  virtual ~memtracer_t() {}
  virtual bool interested_in_range(uint64_t begin, uint64_t end, access_type
type) = 0;
  virtual void trace(uint64_t addr, size_t bytes, access_type type) = 0;
};
```



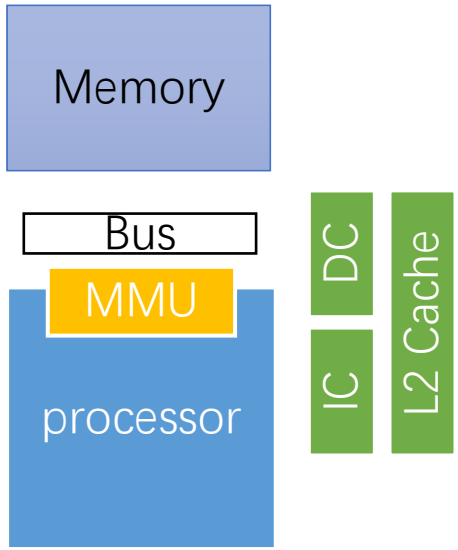

riscv/memtracer.h

# Spike内存系统—cache_sim_t及L2

## riscv/cachesim.h

```
class cache_sim_t
{
 public:
  cache_sim_t(size_t sets, size_t ways, size_t linesz, const char* name);
  cache_sim_t(const cache_sim_t& rhs);
  virtual ~cache_sim_t();

  void access(uint64_t addr, size_t bytes, bool store);
  void print_stats();
  void set_miss_handler(cache_sim_t* mh) { miss_handler = mh; }
  void set_log(bool _log) { log = _log; }

  static cache_sim_t* construct(const char* config, const char* name);

 protected:
  static const uint64_t VALID = 1ULL << 63;
  static const uint64_t DIRTY = 1ULL << 62;

  virtual uint64_t* check_tag(uint64_t addr);
  virtual uint64_t victimize(uint64_t addr);
```

```
  lfsr_t lfsr;
  cache_sim_t* miss_handler;
  size_t sets;
  size_t ways;
  size_t linesz;
  size_t idx_shift;
  uint64_t* tags;
  uint64_t read_accesses;
  uint64_t read_misses;
  uint64_t bytes_read;
  uint64_t write_accesses;
  uint64_t write_misses;
  uint64_t bytes_written;
  uint64_t writebacks;

  std::string name;
  bool log;

  void init();
};
```

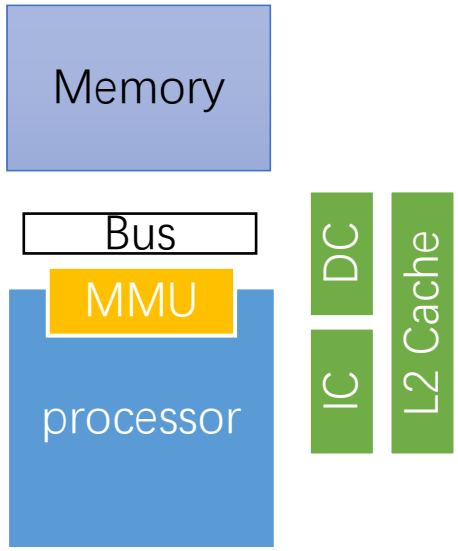


# Spike内存系统—cache access

## riscv/cachesim.cc

```
void cache_sim_t::access(uint64_t addr, size_t bytes, bool store)
{
  store ? write_accesses++ : read_accesses++;
  (store ? bytes_written : bytes_read) += bytes;

  uint64_t* hit_way = check_tag(addr);
  if (likely(hit_way != NULL))
  {
    if (store)
      *hit_way |= DIRTY;
    return;
  }

  store ? write_misses++ : read_misses++;
  if (log)
  {
    std::cerr << name << " "
              << (store ? "write" : "read") << " miss 0x"
              << std::hex << addr << std::endl;
  }
```

```
  uint64_t victim = victimize(addr);

  if ((victim & (VALID | DIRTY)) == (VALID | DIRTY))
  {
    uint64_t dirty_addr = (victim & ~(VALID | DIRTY)) << idx_shift;
    if (miss_handler)
      miss_handler->access(dirty_addr, linesz, true);
    writebacks++;
  }

  if (miss_handler)
    miss_handler->access(addr & ~(linesz-1), linesz, false);

  if (store)
    *check_tag(addr) |= DIRTY;
}
```

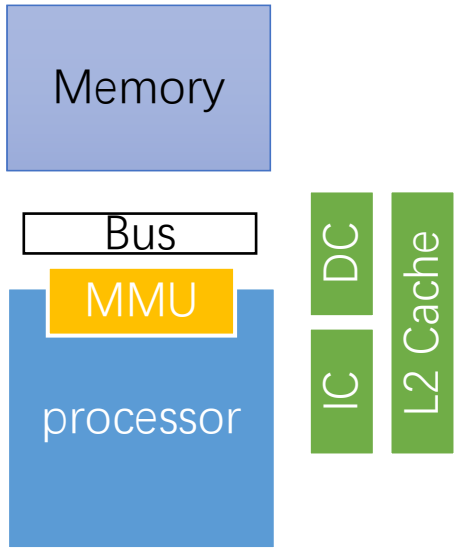


# Spike内存系统—MMU

riscv/mmu.h

```
class mmu_t
{
private:
  std::map<reg_t, reg_t> alloc_cache;
  std::vector<std::pair<reg_t, reg_t >> addr_tbl;
...
  memtracer_list_t tracer;
...
  // implement an instruction cache for simulator performance
  icache_entry_t icache[ICACHE_ENTRIES];
...
void register_memtracer(memtracer_t*);
...
}
```

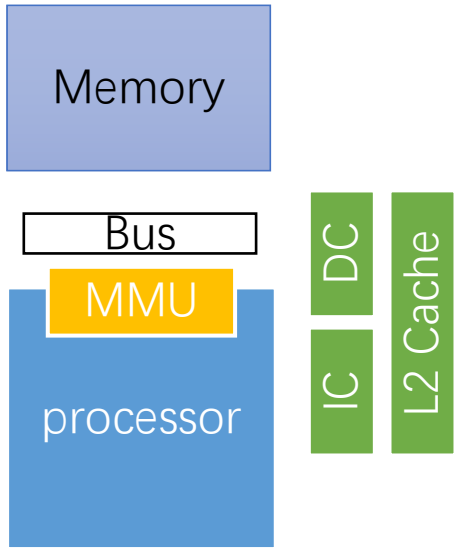
```
void mmu_t::register_memtracer(memtracer_t* t)
{
  flush_tlb();
  tracer.hook(t);
}
```



# Spike内存系统—memtracer_list_t

riscv/mmu.h

```
class memtracer_list_t : public memtracer_t
{
 public:
  bool empty() { return list.empty(); }
  bool interested_in_range(uint64_t begin, uint64_t end, access_type type)
  {
    for (std::vector<memtracer_t*>::iterator it = list.begin(); it != list.end(); ++it)
      if ((*it)->interested_in_range(begin, end, type))
        return true;
    return false;
  }
  void trace(uint64_t addr, size_t bytes, access_type type)
  {
    for (std::vector<memtracer_t*>::iterator it = list.begin(); it != list.end(); ++it)
      (*it)->trace(addr, bytes, type);
  }
  void hook(memtracer_t* h)
  {
    list.push_back(h);
  }
 private:
  std::vector<memtracer_t*> list;
};
```



# 快速指令执行流程

- 指令执行流程： riscv/execute.cc：step()

```
auto ic_entry = _mmu->access_icache(pc);

#define ICACHE_ACCESS(i) { \
insn_fetch_t fetch = ic_entry->data; \
pc = execute_insn(this, pc, fetch); \
ic_entry = ic_entry->next; \
if (i == mmu_t::ICACHE_ENTRIES-1) break; \
if (unlikely(ic_entry->tag != pc)) break; \
if (unlikely(instret+1 == n)) break; \
  instret++; \
  state.pc = pc; \
}
```

```
static reg_t execute_insn(processor_t* p, reg_t pc,
insn_fetch_t fetch)
{
 ...
  npc = fetch.func(p, fetch.insn, pc);
 …
}
```

```
struct insn_fetch_t
{
 insn_func_t func;
 insn_t insn;
};
```

- riscv/mmu.h

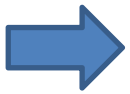
```
inline icache_entry_t* access_icache(reg_t addr)
{
 icache_entry_t* entry = &icache[icache_index(addr)];
 if (likely(entry->tag == addr))
   return entry;
 return refill_icache(addr, entry);
}
```

```
inline icache_entry_t* refill_icache(reg_t addr, icache_entry_t* entry)
{
 auto tlb_entry = translate_insn_addr(addr);
 insn_bits_t insn = from_le(*(uint16_t*)(tlb_entry.host_offset + addr));
 int length = insn_length(insn);
  ...
 insn_fetch_t fetch = {proc->decode_insn(insn), insn};
 entry->tag = addr;
 entry->next = &icache[icache_index(addr + length)];
 entry->data = fetch;
 ...
 return entry;
}
```

# 慢速指令执行流程

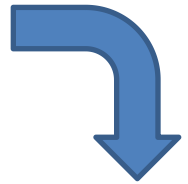
- 指令执行流程： riscv/execute.cc：step()

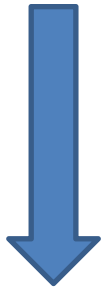
```
insn_fetch_t fetch = mmu->load_insn(pc);
if (debug && !state.serialized)
  disasm(fetch.insn);
pc = execute_insn(this, pc, fetch);
advance_pc();
```

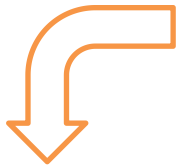


- riscv/mmu.h

```
inline insn_fetch_t load_insn(reg_t addr)
{
  icache_entry_t entry;
  return refill_icache(addr, &entry)->data;
}
```







```
struct insn_fetch_t
{
  insn_func_t func;
  insn_t insn;
};
```

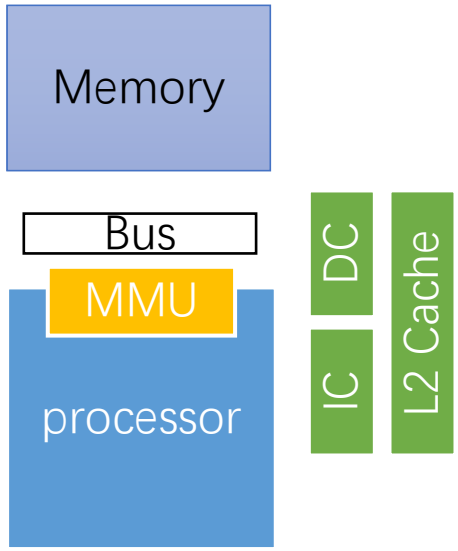
```
static reg_t execute_insn(processor_t* p, reg_t pc,
insn_fetch_t fetch)
{
  ...
    npc = fetch.func(p, fetch.insn, pc);
  …
}
```

```
inline icache_entry_t* refill_icache(reg_t addr, icache_entry_t* entry)
{
  auto tlb_entry = translate_insn_addr(addr);
  insn_bits_t insn = from_le(*(uint16_t*)(tlb_entry.host_offset + addr));
  int length = insn_length(insn);
   ...

  insn_fetch_t fetch = {proc->decode_insn(insn), insn};
  entry->tag = addr;
  entry->next = &icache[icache_index(addr + length)];
  entry->data = fetch;

  ...
  return entry;
}
```

# Spike内存系统—取指流程

riscv/mmu.h

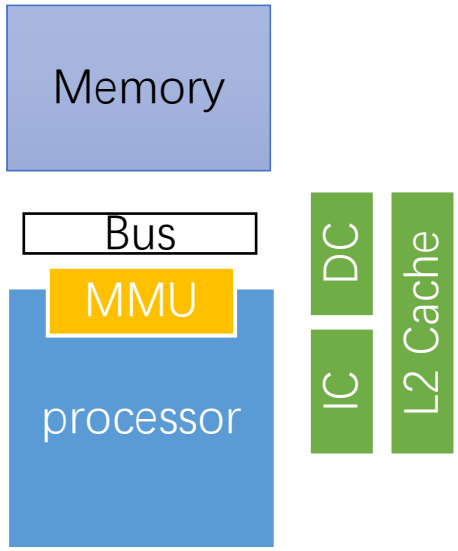
```
inline icache_entry_t* refill_icache(reg_t addr, icache_entry_t* entry)
{
  auto tlb_entry = translate_insn_addr(addr);
  insn_bits_t insn = from_le(*(uint16_t*)(tlb_entry.host_offset +
addr));
  ...
  insn_fetch_t fetch = {proc->decode_insn(insn), insn};
  entry->tag = addr;
  entry->next = &icache[icache_index(addr + length)];
  entry->data = fetch;

  reg_t paddr = tlb_entry.target_offset + addr;;
  if (tracer.interested_in_range(paddr, paddr + 1, FETCH)) {
    entry->tag = -1;
    tracer.trace(paddr, length, FETCH);
  }
  return entry;
}
```



# Spike内存系统—地址转换

riscv/mmu.h

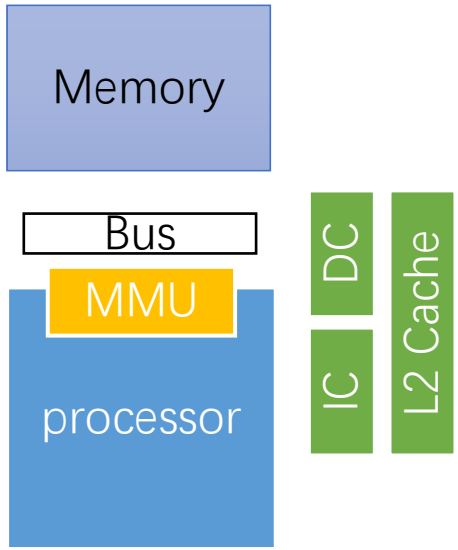
```
inline tlb_entry_t translate_insn_addr(reg_t addr) {
  reg_t vpn = addr >> PGSHIFT;
  if (likely(tlb_insn_tag[vpn % TLB_ENTRIES] == vpn))
    return tlb_data[vpn % TLB_ENTRIES];
  tlb_entry_t result;
  if (unlikely(tlb_insn_tag[vpn % TLB_ENTRIES] != (vpn |
TLB_CHECK_TRIGGERS))) {
    result = fetch_slow_path(addr);
  } else {
    result = tlb_data[vpn % TLB_ENTRIES];
  }
  if (unlikely(tlb_insn_tag[vpn % TLB_ENTRIES] == (vpn |
TLB_CHECK_TRIGGERS))) {
    uint16_t* ptr = (uint16_t*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr);
    int match = proc->trigger_match(OPERATION_EXECUTE, addr, from_le(*ptr));
    if (match >= 0) {
      throw trigger_matched_t(match, OPERATION_EXECUTE, addr, from_le(*ptr));
    }
  }
  return result;
}
```



# Spike内存系统扩展—取指转换慢速路径

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;
  }
}
```



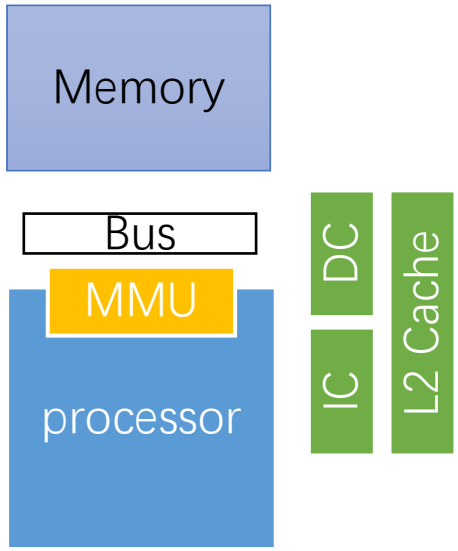
# Spike内存系统扩展—取指转换慢速路径

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;
  }
}
```

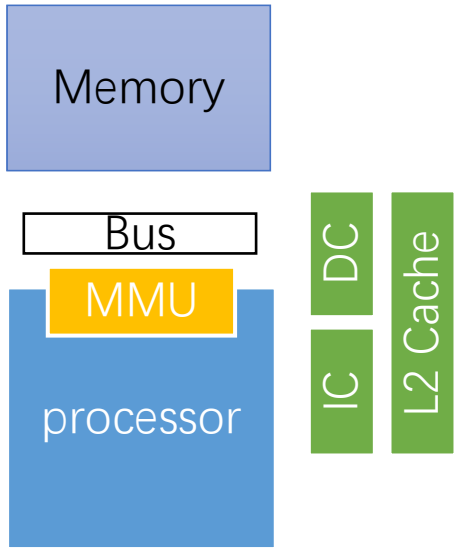
riscv/sim.cc

```
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;
}
```



# Spike内存系统扩展—填充tlb

```
tlb_entry_t mmu_t::refill_tlb(reg_t vaddr, reg_t paddr, char* host_addr, access_type type)
{
  reg_t idx = (vaddr >> PGSHIFT) % TLB_ENTRIES;
  reg_t expected_tag = vaddr >> PGSHIFT;
  if ((tlb_load_tag[idx] & ~TLB_CHECK_TRIGGERS) != expected_tag)
    tlb_load_tag[idx] = -1;
  if ((tlb_store_tag[idx] & ~TLB_CHECK_TRIGGERS) != expected_tag)
    tlb_store_tag[idx] = -1;
  if ((tlb_insn_tag[idx] & ~TLB_CHECK_TRIGGERS) != expected_tag)
    tlb_insn_tag[idx] = -1;

  if ((check_triggers_fetch && type == FETCH) ||
      (check_triggers_load && type == LOAD) ||
      (check_triggers_store && type == STORE))
    expected_tag |= TLB_CHECK_TRIGGERS;

  if (pmp_homogeneous(paddr & ~reg_t(PGSIZE - 1), PGSIZE)) {
    if (type == FETCH) tlb_insn_tag[idx] = expected_tag;
    else if (type == STORE) tlb_store_tag[idx] = expected_tag;
    else tlb_load_tag[idx] = expected_tag;
  }
  tlb_entry_t entry = {host_addr - vaddr, paddr - vaddr};
  tlb_data[idx] = entry;
  return entry;
}
```



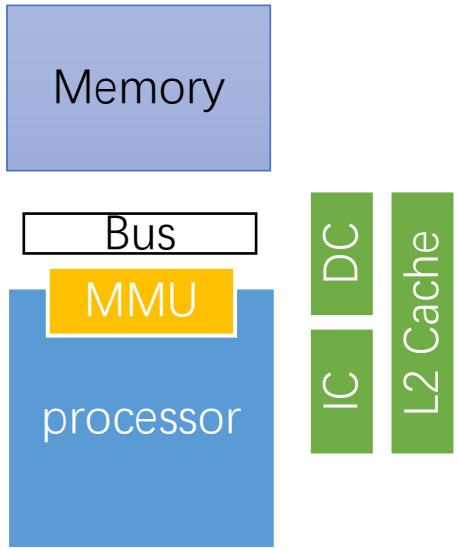
# Spike内存系统—加载内存

```
WRITE_RVC_RS2S(MMU.load_int32(RVC_RS1S + insn.rvc_lw_imm()));
```

```
load_func(int16, load, 0)
load_func(int32, load, 0)
```

```
#define load_func(type, prefix, xlate_flags) \
  inline type##_t prefix##_##type(reg_t addr, bool require_alignment = false) { \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
    if (unlikely(addr & (sizeof(type##_t)-1))) { \
      if (require_alignment) load_reserved_address_misaligned(addr); \
      else return misaligned_load(addr, sizeof(type##_t)); \
    } \
    reg_t vpn = addr >> PGSHIFT; \
    size_t size = sizeof(type##_t); \
    if (likely(tlb_load_tag[vpn % TLB_ENTRIES] == vpn)) { \
      if (proc) READ_MEM(addr, size); \
      return from_target(*(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr)); \
    } \
    if (unlikely(tlb_load_tag[vpn % TLB_ENTRIES] == (vpn | TLB_CHECK_TRIGGERS))) { \
```

```
      type##_t data = from_target(*(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr)); \
      if (!matched_trigger) { \
        matched_trigger = trigger_exception(OPERATION_LOAD, addr, data); \
        if (matched_trigger) \
          throw *matched_trigger; \
      } \
      if (proc) READ_MEM(addr, size); \
      return data; \
    } \
    target_endian<type##_t> res; \
    load_slow_path(addr, sizeof(type##_t), (uint8_t*)&res, (xlate_flags)); \
    if (proc) READ_MEM(addr, size); \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
    return from_target(res); \
  }
```



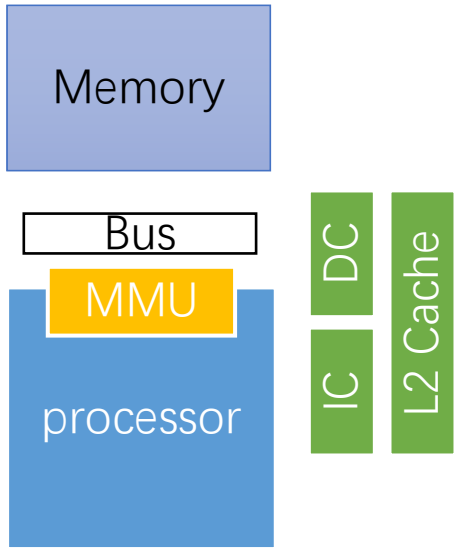
# Spike内存系统—加载内存

```
WRITE_RVC_RS2S(MMU.load_int32(RVC_RS1S + insn.rvc_lw_imm()));
```

```
load_func(int16, load, 0)
load_func(int32, load, 0)
```

```
#define load_func(type, prefix, xlate_flags) \
  inline type##_t prefix##_##type(reg_t addr, bool require_alignment = false) { \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
    if (unlikely(addr & (sizeof(type##_t)-1))) { \
      if (require_alignment) load_reserved_address_misaligned(addr); \
      else return misaligned_load(addr, sizeof(type##_t)); \
    } \
    reg_t vpn = addr >> PGSHIFT; \
    size_t size = sizeof(type##_t); \
    if (likely(tlb_load_tag[vpn % TLB_ENTRIES] == vpn)) { \
      if (proc) READ_MEM(addr, size); \
      return from_target(*(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr)); \
    } \
    if (unlikely(tlb_load_tag[vpn % TLB_ENTRIES] == (vpn | TLB_CHECK_TRIGGERS))) { \
```

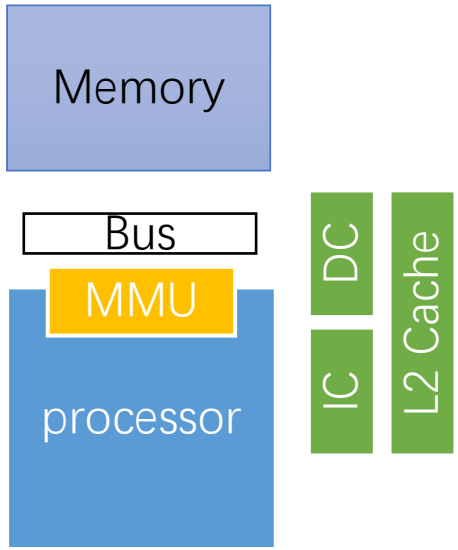
```
      type##_t data = from_target(*(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr)); \
      if (!matched_trigger) { \
        matched_trigger = trigger_exception(OPERATION_LOAD, addr, data); \
        if (matched_trigger) \
          throw *matched_trigger; \
      } \
      if (proc) READ_MEM(addr, size); \
      return data; \
    } \
    target_endian<type##_t> res; \
    load_slow_path(addr, sizeof(type##_t), (uint8_t*)&res, (xlate_flags)); \
    if (proc) READ_MEM(addr, size); \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
    return from_target(res); \
  }
```



# Spike内存系统—加载数据慢速路径

riscv/mmu.cc

```
void mmu_t::load_slow_path(reg_t addr, reg_t len, uint8_t* bytes, uint32_t
xlate_flags)
{
 reg_t paddr = translate(addr, len, LOAD, xlate_flags);

 if (auto host_addr = sim->addr_to_mem(paddr)) {
  memcpy(bytes, host_addr, len);
  if (tracer.interested_in_range(paddr, paddr + PGSIZE, LOAD))
   tracer.trace(paddr, len, LOAD);
  else
   refill_tlb(addr, paddr, host_addr, LOAD);
 } else if (!mmio_load(paddr, len, bytes)) {
  throw trap_load_access_fault((proc) ? proc->state.v : false, addr, 0, 0);
 }

 if (!matched_trigger) {
  reg_t data = reg_from_bytes(len, bytes);
  matched_trigger = trigger_exception(OPERATION_LOAD, addr, data);
  if (matched_trigger)
   throw *matched_trigger;
 }
}
```

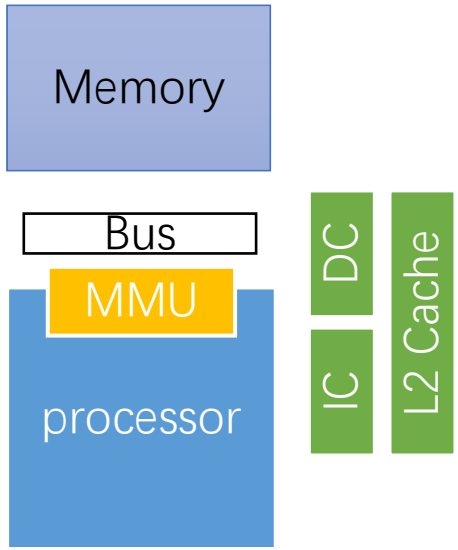


# Spike内存系统—存储内存

```
MMU.store_uint32(RVC_RS1S + insn.rvc_lw_imm(), RVC_RS2S);
```

```
store_func(uint8, store, 0)
store_func(uint16, store, 0)
store_func(uint32, store, 0)
store_func(uint64, store, 0)
```

```
#define store_func(type, prefix, xlate_flags) \
  void prefix##_##type(reg_t addr, type##_t val) { \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
    if (unlikely(addr & (sizeof(type##_t)-1))) \
      return misaligned_store(addr, val, sizeof(type##_t)); \
    reg_t vpn = addr >> PGSHIFT; \
    size_t size = sizeof(type##_t); \
    if (likely(tlb_store_tag[vpn % TLB_ENTRIES] == vpn)) { \
      if (proc) WRITE_MEM(addr, val, size); \
      *(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr) = to_target(val); \
    } \
    else if (unlikely(tlb_store_tag[vpn % TLB_ENTRIES] == (vpn | TLB_CHECK_TRIGGERS))) { \
      if (!matched_trigger) { \
        matched_trigger = trigger_exception(OPERATION_STORE, addr, val); \
      if (matched_trigger) \
        throw *matched_trigger; \
      } \
      if (proc) WRITE_MEM(addr, val, size); \
      *(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr) = to_target(val); \
    } \
    else { \
      target_endian<type##_t> target_val = to_target(val); \
      store_slow_path(addr, sizeof(type##_t), (const uint8_t*)&target_val, (xlate_flags)); \
      if (proc) WRITE_MEM(addr, val, size); \
    } \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
  }
```

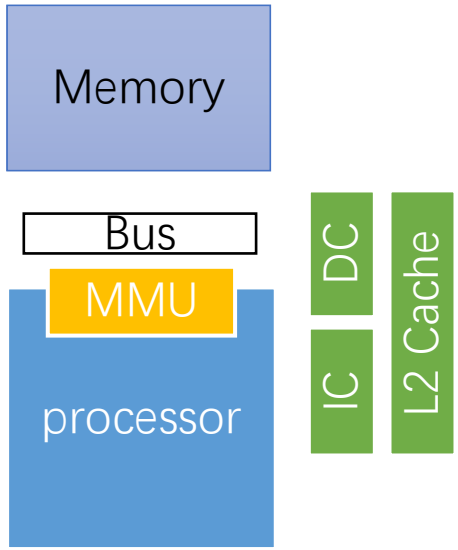


# Spike内存系统—存储内存

```
MMU.store_uint32(RVC_RS1S + insn.rvc_lw_imm(), RVC_RS2S);
```

```
#define store_func(type, prefix, xlate_flags) \
  void prefix##_##type(reg_t addr, type##_t val) { \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
    if (unlikely(addr & (sizeof(type##_t)-1))) \
      return misaligned_store(addr, val, sizeof(type##_t)); \
    reg_t vpn = addr >> PGSHIFT; \
    size_t size = sizeof(type##_t); \
    if (likely(tlb_store_tag[vpn % TLB_ENTRIES] == vpn)) { \
      if (proc) WRITE_MEM(addr, val, size); \
      *(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr) = to_target(val); \
    } \
    else if (unlikely(tlb_store_tag[vpn % TLB_ENTRIES] == (vpn | TLB_CHECK_TRIGGERS))) { \
      if (!matched_trigger) { \
        matched_trigger = trigger_exception(OPERATION_STORE, addr, val); \
      if (matched_trigger) \
        throw *matched_trigger; \
      } \
      if (proc) WRITE_MEM(addr, val, size); \
      *(target_endian<type##_t>*)(tlb_data[vpn % TLB_ENTRIES].host_offset + addr) = to_target(val); \
    } \
    else { \
      target_endian<type##_t> target_val = to_target(val); \
      store_slow_path(addr, sizeof(type##_t), (const uint8_t*)&target_val, (xlate_flags)); \
      if (proc) WRITE_MEM(addr, val, size); \
    } \
    if ((xlate_flags) != 0) \
      flush_tlb(); \
  }
```

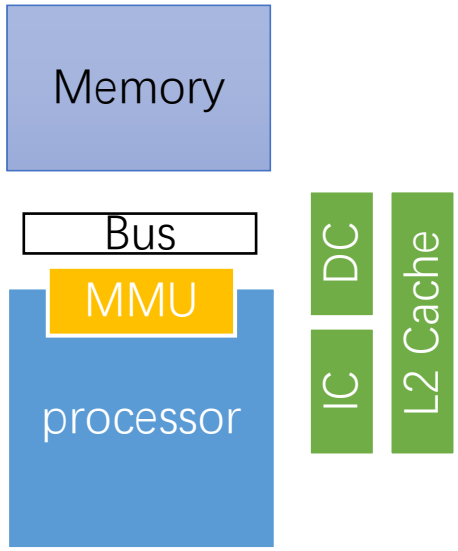
```
store_func(uint8, store, 0)
store_func(uint16, store, 0)
store_func(uint32, store, 0)
store_func(uint64, store, 0)
```



# Spike内存系统—存储慢速路径

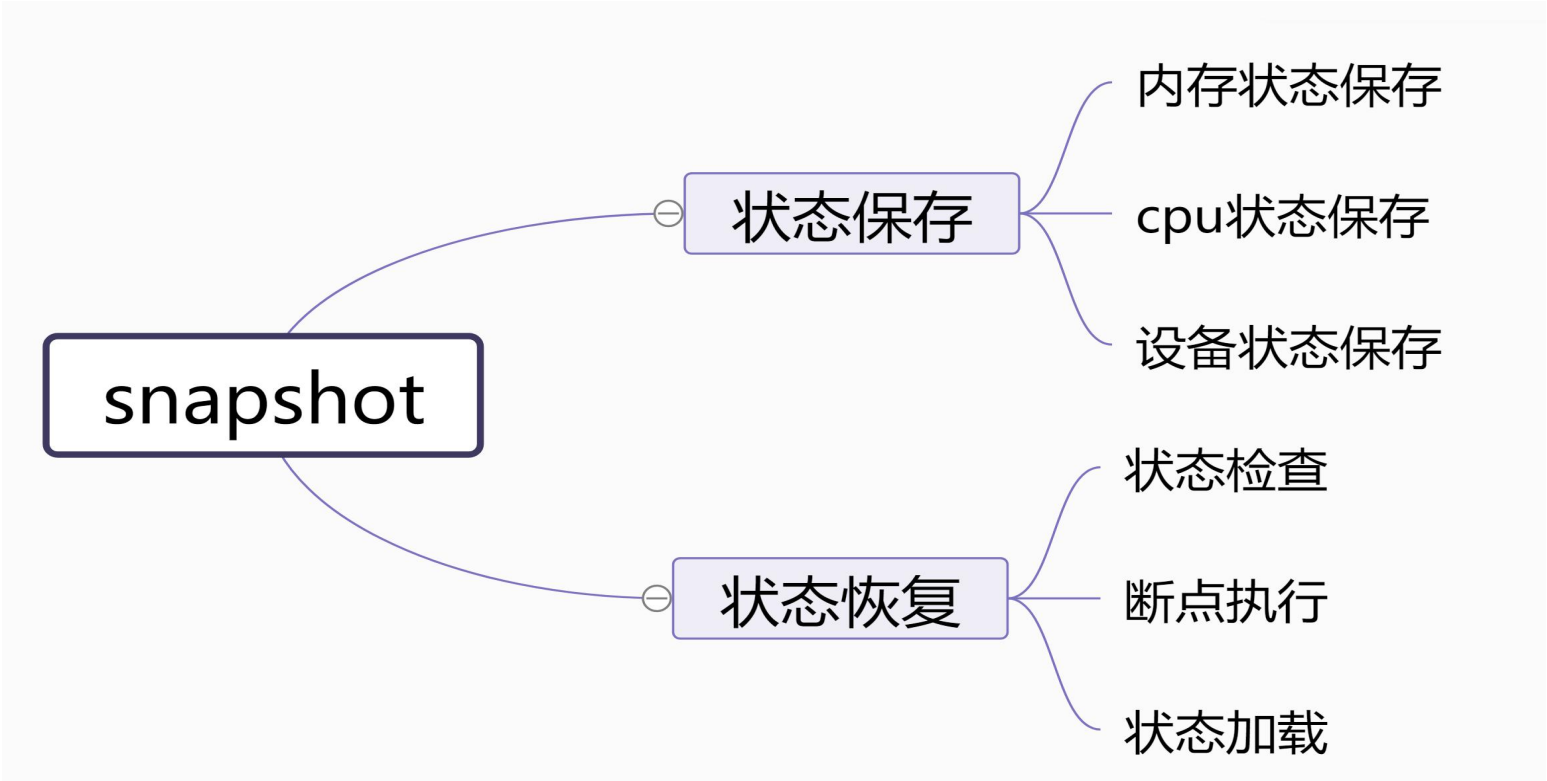
riscv/mmu.cc

```
void mmu_t::store_slow_path(reg_t addr, reg_t len, const uint8_t* bytes,
uint32_t xlate_flags)
{
 reg_t paddr = translate(addr, len, STORE, xlate_flags);

 if (!matched_trigger) {
  reg_t data = reg_from_bytes(len, bytes);
  matched_trigger = trigger_exception(OPERATION_STORE, addr, data);
  if (matched_trigger)
   throw *matched_trigger;
 }

 if (auto host_addr = sim->addr_to_mem(paddr)) {
  memcpy(host_addr, bytes, len);
  if (tracer.interested_in_range(paddr, paddr + PGSIZE, STORE))
   tracer.trace(paddr, len, STORE);
  else
   refill_tlb(addr, paddr, host_addr, STORE);
 } else if (!mmio_store(paddr, len, bytes)) {
  throw trap_store_access_fault((proc) ? proc->state.v : false, addr, 0, 0);
 }
}
```



# snapshot机制回顾

snapshot(快照)功能，可以用来保存系统运行到某一时刻的状态，提供某一时刻系统运行状态的完全拷贝，同时支持从该保存点继续运行的能力




github库：https://github.com/plctlab/plct-spike/tree/plct-snapshot

# snapshot机制—写内存保存

对写内存进行标记

```
void mmu_t::store_slow_path(reg_t addr, reg_t len, const uint8_t* bytes, uint32_t xlate_flags)
{
  reg_t paddr = translate(addr, len, STORE, xlate_flags);

  if (!matched_trigger) {
    reg_t data = reg_from_bytes(len, bytes);
    matched_trigger = trigger_exception(OPERATION_STORE, addr, data);
    if (matched_trigger)
      throw *matched_trigger;
  }

  if (auto host_addr = sim->addr_to_mem(paddr)) {
    (*sim->get_tags())[paddr >> PGSHIFT] = true;
    memcpy(host_addr, bytes, len);
    if (tracer.interested_in_range(paddr, paddr + PGSIZE, STORE))
      tracer.trace(paddr, len, STORE);
    else
      refill_tlb(addr, paddr, host_addr, STORE);
  } else if (!mmio_store(paddr, len, bytes)) {
    throw trap_store_access_fault(addr, 0, 0);
  }
}
```

# snapshot机制—写内存保存

## 只dump标记为写的内存

snapshot/
ramdump.cc

```
bool snapshot_t::ramdump(ofstream &out)
{
  for (auto tag: *tags) {
    out.write((char *)&tag.first, sizeof(tag.first)); // vaddr
    size_t size = 1 << PGSHIFT;
    char *addr = sim -> addr_to_mem(tag.first * size);
    if(addr == NULL)
      return false;
    out.write((char *)addr, size);
  }
  return true;
}
```

# 谢谢各位

欢迎提问、讨论、交流合作