Spike 代码框架及具体实现分析

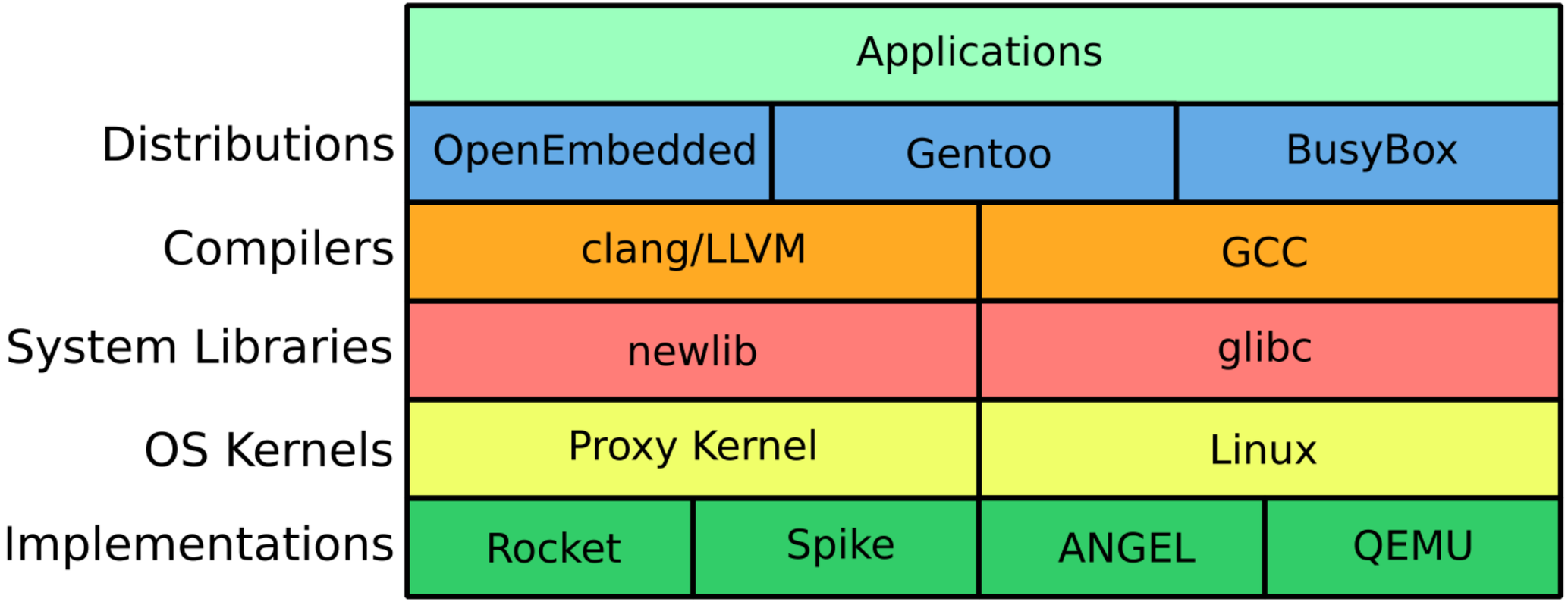
王萌 2020.06.17

目录

RISC-V指令集简介

- 一个开源的基于RISC(精简指令集计算机)的指令集架构
- 模块化，低功耗，架构简单
- 基本指令集：32位(RV32I,RV32E)，64位(RV64I)，128位(RV128I)
- 可选择的扩展指令集M，A，F，D，C等
- 一个完整的64位计算机一般需要RV64GC(RV64IMAFDC的简称)

RISC-V软件栈


Applications
Distributions
OpenEmbedded
Gentoo
BusyBox
Compilers
clang/LLVM
GCC
System Libraries
newlib
glibc
OS Kernels
Proxy Kernel
Linux
Implementations
Rocket
Spike
ANGEL
QEMU

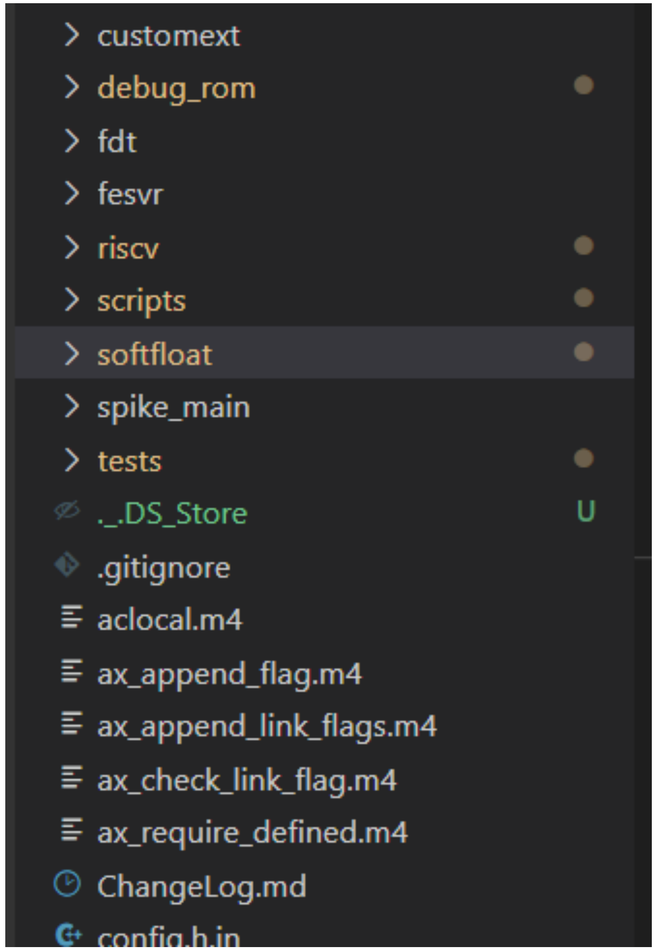
RISC-V模拟器分类

- Functional(QEMU)：采用二进制翻译，实际执行的是翻译之后的机器码，执行效率高
- Trace-accurate(Spike) :模拟实际代码执行过程中的软硬件行为，提供指令级别的仿真
- Cycle-accurate：提供硬件级别的仿真，可以针对特定的实现作周期级别的模拟

Type	Example	Performance
Functional	QEMU	10^8-10^9 条指令/秒
Trace-accurate	Spike	10^7-10^8 条指令/秒
Cycle-accurate	Rocket-chip	10^4-10^5 条指令/秒

Spike整体结构

- 目录结构如下
- fdt：为模拟器生成device tree
- fesvr：提供了target与主机host交互的接口（借助proxy kernel实现）
- softfloat库提供了浮点数的支持
- riscv实现了risc-v机器码的翻译和执行
- spike_main：程序的entry_point和interface

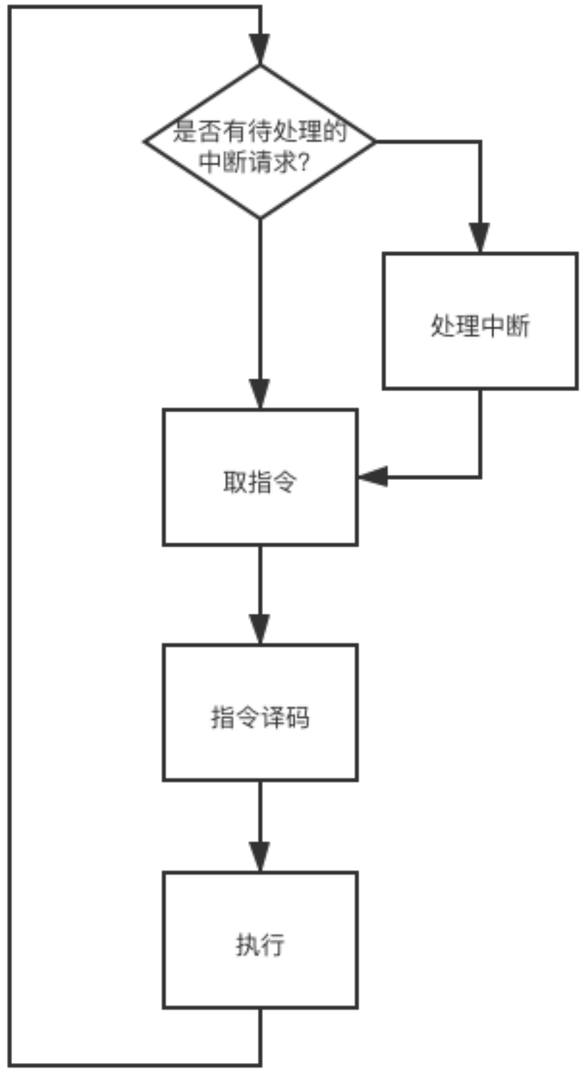


Spike模拟器执行binary的流程

- decode the instructions
- memory load and store
- maintain the registers
- debug mode——like a tiny gdb
- execute the instructions
- others

Spike CPU的指令执行过程

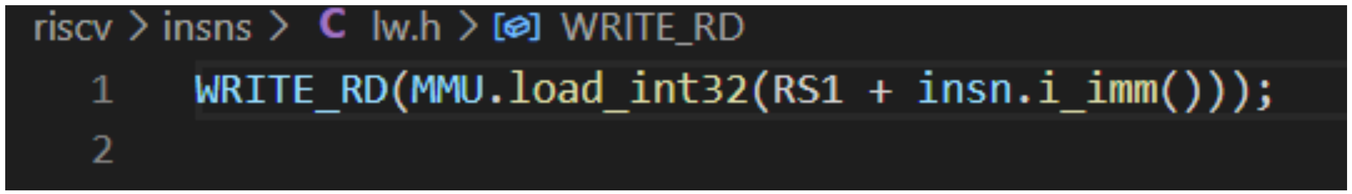
- Spike执行代码时，执行的实际上是C++ Function
- CPU从pc指针处读取指令，通过译码来确定接下来要执行的功能
- 指令的行为定义在riscv/insns下的 指令名称.h文件中



Spike CPU的指令执行过程

- 指令的编码定义在riscv/encoding.h中
- 定义一条指令需要定义如下内容：
- #define MATCH_[instruction name] 0x12341234
- #define MASK_[instruction name] 0xabcdabcd
- DECLARE_INSN(XXX, MATCH_XXX, MASK_XXX)
- alternative：通过riscv-opcodes生成encoding.h

Spike CPU的指令执行过程

- 例：riscv/insns/lw.h
- 

- 实际上模拟指令的行为相当于写对应的c++ function
- riscv/decode.h中定义了一系列操作存储器和寄存器的宏定义，以便于在指令中操作它们

Spike 对多核的支持

- 位于riscv/sim.cc中
- static const size_t INTERLEAVE = 5000;
- 各个核心会轮流执行，用单核模拟多核的行为

```
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();
    }
  }
}
```

Spike对CPU执行过程的优化

- 为了加快执行效率，Spike在cpu的execute阶段引入了Duff’s Device进行优化
- Duff’s Device是循环展开的一种实现，通过将switch语句嵌入到循环体中，从而减少分支次数，提高运行效率
- According to Andrew Waterman's recollection, this optimization
resulted in approximately a 2x performance increase.

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch(count % 8) {
    case 0: do {    *to = *from++;
    case 7:     *to = *from++;
    case 6:     *to = *from++;
    case 5:     *to = *from++;
    case 4:     *to = *from++;
    case 3:     *to = *from++;
    case 2:     *to = *from++;
    case 1:     *to = *from++;
        } while(--n > 0);
    }
}
```

Spike对CPU执行过程的优化

```
size_t idx = _mmu->icache_index(pc);

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; \
}

switch (idx) {
  // "icache.h" is generated by the gen_icache script
  #include "icache.h"
}
```

优点：

1.对于每个cache entry设置一个单独的case，可以提高分支预测时的准确度

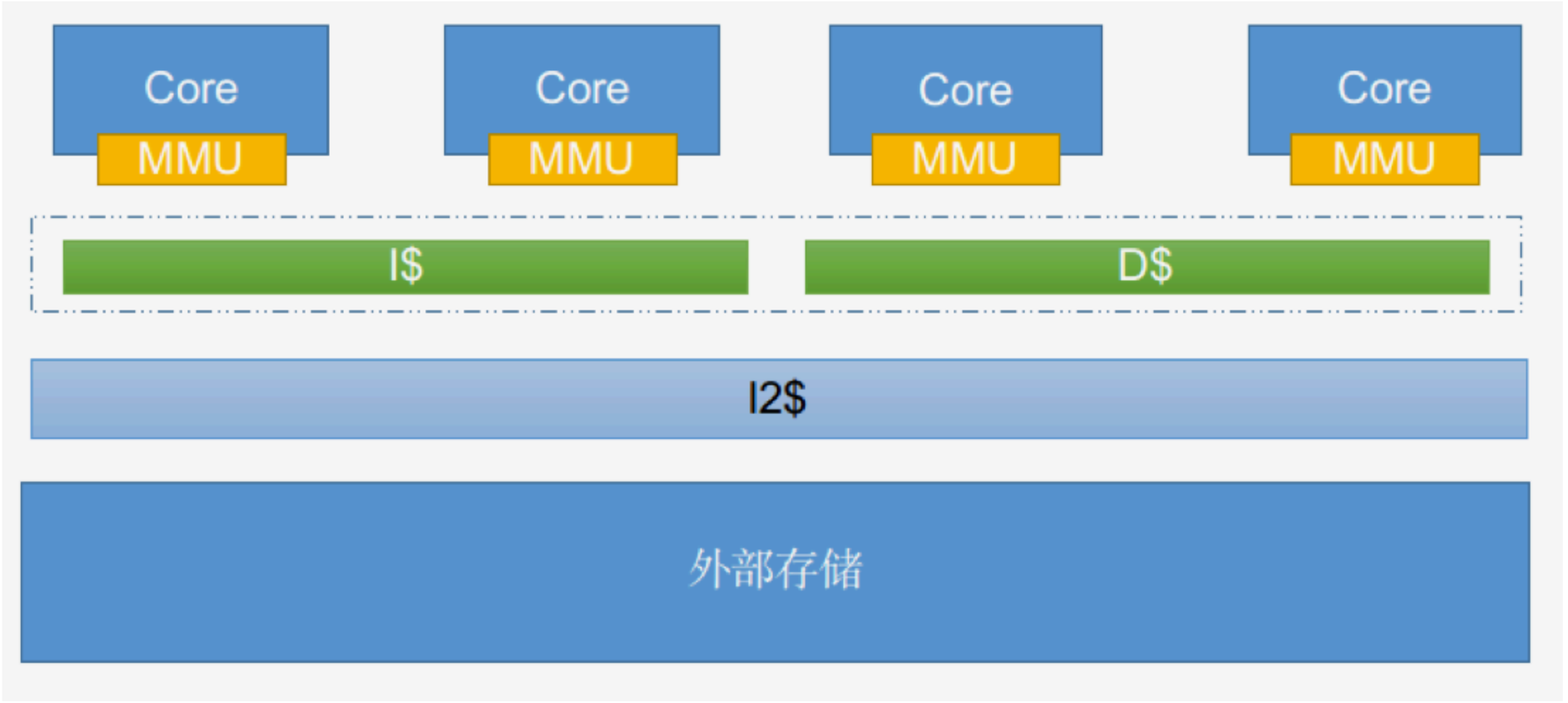
2.优化了顺序执行的效率（程序可以直接跳转到下一个case）

Spike中的存储器和cache模型

- 采用分页存储方式
- 每个cpu核心有一个MMU处理内存请求
- MMU中有TLB和ICACHE缓存页表项
- PAGE SIZE=4KB

Spike中的存储器和cache模型

- Spike的CPU采用二级缓存结构，所有核心共享同一组缓存



Spike的cache模型与真实CPU的区别

- Spike中的MMU并不负责虚拟地址（VA）到物理地址（PA）的翻译

```
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的cache模型与真实CPU的区别

- 所有核心共用同一组L1 cache
- 实际的CPU架构中，L1 cache通常是每个核心私有的
- 不存在cache一致性
- private L1 cache已经在开发中，未合并
- cache中存储的是地址的索引，实际上读写的仍然是物理内存
- 真实CPU中会将页面读取到缓存中

Spike的cache模型与真实CPU

- 读写操作并不在缓存的access操作中完成，而是在MMU中完成
- cache只是用来记录缓存是否命中和l1，l2命中率等指标
- 更像是一个tracer

```
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++;

  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提供的拓展接口

- RISC-V在设计时预留了4个opcode用于自定义的扩展指令：

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

- Spike中提供了使用这四个拓展的接口

```
class rocc_t : public extension_t
{
 public:
  virtual reg_t custom0(rocc_insn_t insn, reg_t xs1, reg_t xs2);
  virtual reg_t custom1(rocc_insn_t insn, reg_t xs1, reg_t xs2);
  virtual reg_t custom2(rocc_insn_t insn, reg_t xs1, reg_t xs2);
  virtual reg_t custom3(rocc_insn_t insn, reg_t xs1, reg_t xs2);
  std::vector<insn_desc_t> get_instructions();
  std::vector<disasm_insn_t*> get_disasms();
};
```

Spike提供的debugger接口

- 几个接口函数：
- processor_t *p = sim_t::get_core(0);
- 获取寄存器值：p->get_state()->XPR[0]；
- 获取pc：p->get_state()->pc;
- 获取mmu：mmu_t* mmu = p->get_mmu();
- 获取内存：reg_t val = mmu->load_uint64/32/16/8(0x00000000)
- 使用openOCD连接到gdb进行调试

Thanks for listening!