从零开始的RISC-V模拟器开发

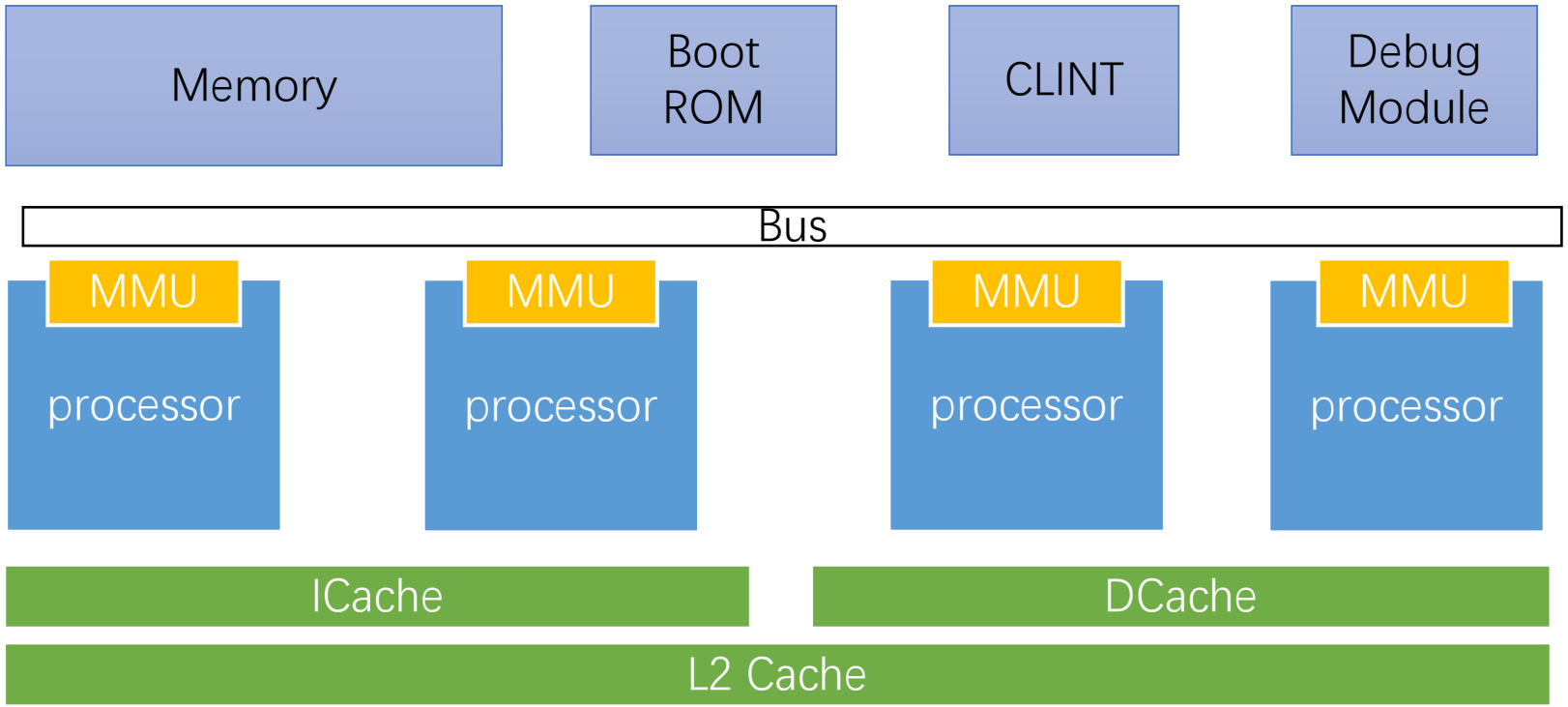
第1讲 Spike篇之CPU模拟

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

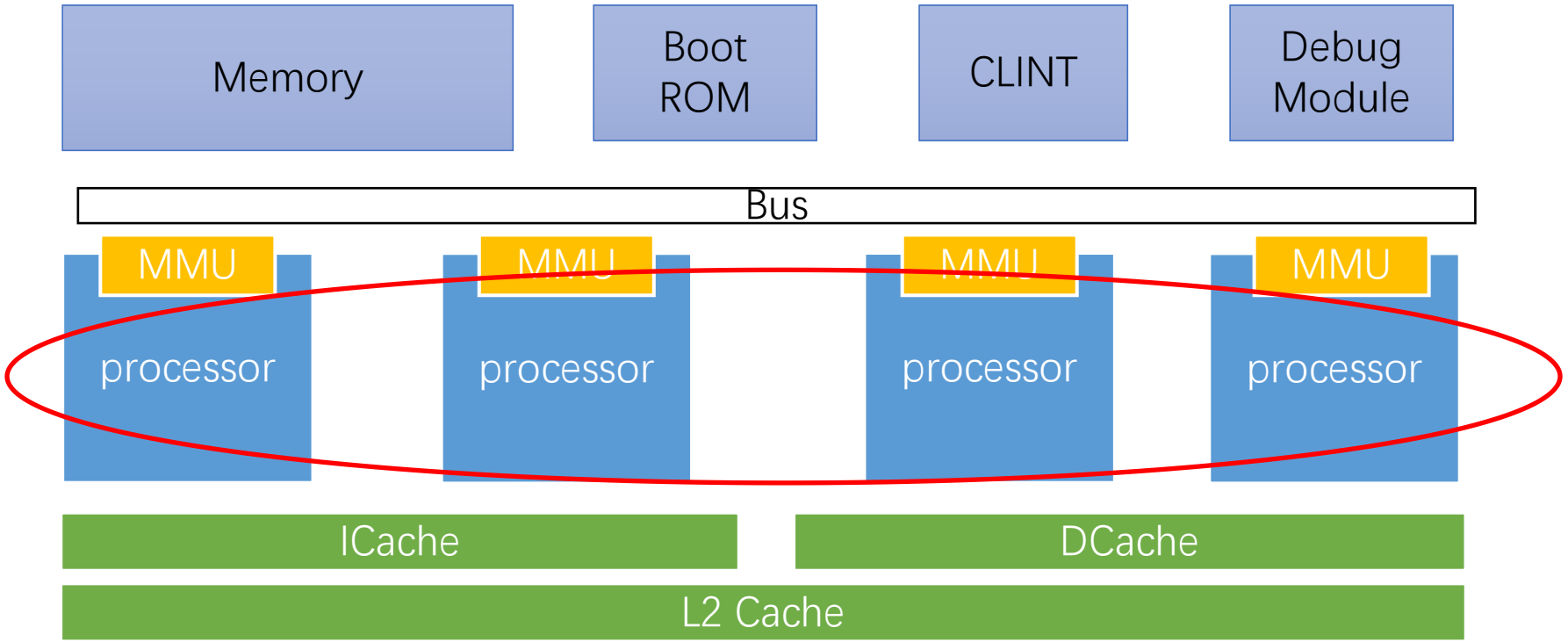
回顾

Spike是针对RISCV的轻量级指令集模拟器



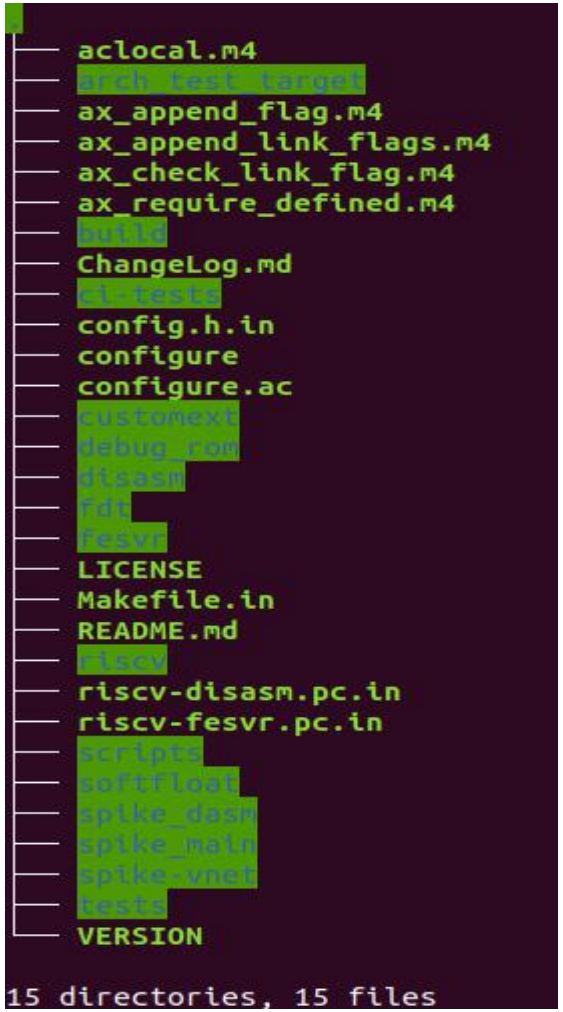
本节课内容

Spike是针对RISCV的轻量级指令集模拟器



指令集模拟

Spike代码结构



主要功能代码包括：

- riscv：包含主要的模拟功能支持代码
- disasm：包含反汇编支持代码
- fesvr：包含frontend server支持代码
- spike_dasm/spike_main：包含Spike相关工具main代码
- softfloat：包含浮点库支持代码
- fdt：包含flat device tree操作相关代码

Spike启动流程


解析各参数
初始化SIM
初始化processor
解析支持的指令集
解析支持的权限模式
解析Vector长度
注册支持的基础指令
初始化各设备
运行SIM

Spike启动流程—指令集相关参数

spike_main/spike.cc： main

--isa=<name> RISC-V ISA string [default RV64IMAFDC]

```
parser.option(0, "isa", 1, [&](const char* s){isa = s;});
```

```
for (size_t i = 0; i < nprocs; i++) {
  int hart_id = hartids.empty() ? i : hartids[i];
  procs[i] = new processor_t(isa, priv, varch, this, hart_id, halted,
                             log_file.get());
}
```

Spike启动流程—指令集相关参数

spike_main/spike.cc： main

--extension=<name> Specify RoCC Extension

```
parser.option(0, "extension", 1, [&](const char*
s){extensions.push_back(find_extension(s));});
```

```
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 e : extensions)
    s.get_core(i)->register_extension(e());
}
```

Spike启动流程—基础指令集相关参数

spike_main/spike.cc： main

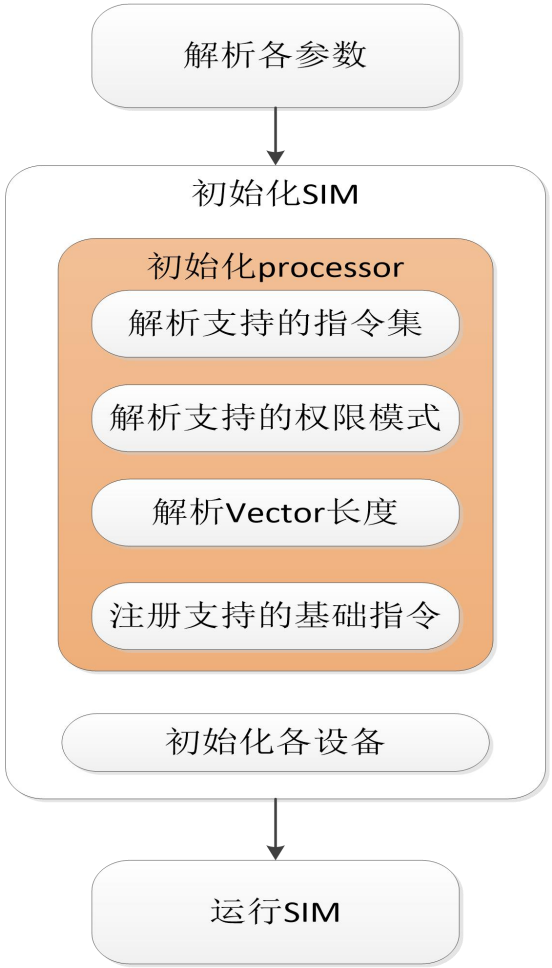
--extlib=<name> Shared library to load

```
parser.option(0, "extlib", 1, [&](const char *s){
  void *lib = dlopen(s, RTLD_NOW | RTLD_GLOBAL);
  if (lib == NULL) {
    fprintf(stderr, "Unable to load extlib '%s': %s\n", s, dlerror());
    exit(-1);
  }
});
```

riscv/extension.h

```
#define REGISTER_EXTENSION(name, constructor) \
  class register_##name { \
    public: register_##name() { register_extension(#name, constructor); } \
  }; static register_##name dummy_##name;
```

Spike启动流程—初始化processor



riscv/processor.cc

```
processor_t::processor_t(const char* isa, const char* priv, const char* varch,
                         simif_t* sim, uint32_t id, bool halt_on_reset,
                         FILE* log_file)
  : debug(false), halt_request(HR_NONE), sim(sim), id(id), xlen(0),
  histogram_enabled(false), log_commits_enabled(false),
  log_file(log_file), halt_on_reset(halt_on_reset),
  extension_table(256, false), impl_table(256, false), last_pc(1), executions(1)
{
  VU.p = this;

  parse_isa_string(isa);
  parse_priv_string(priv);
  parse_varch_string(varch);

  register_base_instructions();
  mmu = new mmu_t(sim, this);

  disassembler = new disassembler_t(max_xlen);
  for (auto e : custom_extensions)
    for (auto disasm_insn : e.second->get_disasms())
      disassembler->add_insn(disasm_insn);
 ...
}
```

Spike启动流程—解析指令集前缀

riscv/procesor.cc:

void processor_t::parse_isa_string(const char* str)

```
max_xlen = 64;
max_isa = reg_t(2) << 62;

if (strncmp(p, "rv32", 4) == 0)
  max_xlen = 32, max_isa = reg_t(1) << 30, p += 4;
else if (strncmp(p, "rv64", 4) == 0)
  p += 4;
else if (strncmp(p, "rv", 2) == 0)
  p += 2;

if (!*p) {
  p = "imafdc";
} else if (*p == 'g') { // treat "G" as "IMAFD"
  tmp = std::string("imafd") + (p+1);
  p = &tmp[0];
}
isa_string = "rv" + std::to_string(max_xlen) + p;
```

Spike启动流程—解析支持的标准指令集

riscv/procesor.cc:

void processor_t::parse_isa_string(const char* str)

```
while (*p) {
  if (islower(*p)) {
    max_isa |= 1L << (*p - 'a');
    extension_table[toupper(*p)] = true;

    if (strchr(all_subsets, *p)) {
      p++;
    } else if (*p == 'x') {
      ...
    } else {
      sprintf(error_msg, "unsupported extension '%c'", *p);
      bad_isa_string(str, error_msg);
    }
  } else if (*p == '_') {
      ...
  } else {
    sprintf(error_msg, "can't parse '%c(%d)'", *p, *p);
    bad_isa_string(str, error_msg);
  }
}
```

```
const char* all_subsets =
"imafdqch"
#ifdef __SIZEOF_INT128__
  "v"
#endif
  "";
```

Spike启动流程—解析支持的外部扩展指令集

riscv/procesor.cc:

void processor_t::parse_isa_string(const char* str)

```
while (*p) {
  if (islower(*p)) {
      ...
    if (strchr(all_subsets, *p)) {
      ...
    } else if (*p == 'x') {
     const char* ext = p + 1, *end = ext;
     while (islower(*end) || *end == '_')
       end++;

     auto ext_str = std::string(ext, end - ext);
     if (ext_str != "dummy")
       register_extension(find_extension(ext_str.c_str())());
     p = end;
    } else {
      ...
    }
  } else if (*p == '_') ...
}
```

扩展表示：xnice_demo

外部指令集查找和注册

riscv/extensions.cc

```
void register_extension(const char* name,
std::function<extension_t*()> f)
{
  extensions()[name] = f;
}
```

```
static std::map<std::string,
std::function<extension_t*()>>& extensions()
{
  static std::map<std::string,
std::function<extension_t*()>> v;
  return v;
}
```

```
std::function<extension_t*()> find_extension(const char* name)
{
  if (!extensions().count(name)) {
    // try to find extension xyz by loading libxyz.so
    std::string libname = std::string("lib") + name + ".so";
    std::string libdefault = "libcustomext.so";
    bool is_default = false;
    auto dlh = dlopen(libname.c_str(), RTLD_LAZY);
    if (!dlh) {
      dlh = dlopen(libdefault.c_str(), RTLD_LAZY);
      if (!dlh) {
        fprintf(stderr, "couldn't find shared library either '%s' or '%s')\n",
              libname.c_str(), libdefault.c_str());
        exit(-1);
      }

      is_default = true;
    }

    if (!extensions().count(name)) {
      fprintf(stderr, "couldn't find extension '%s' in shared library '%s'\n",
            name, is_default ? libdefault.c_str() : libname.c_str());
      exit(-1);
    }
  }

  return extensions()[name];
}
```

指令集内部注册

```
void processor_t::register_extension(extension_t* x)
{
  for (auto insn : x->get_instructions())
    register_insn(insn);
  build_opcode_map();

  if (disassembler)
    for (auto disasm_insn : x->get_disasms())
      disassembler->add_insn(disasm_insn);

  if (!custom_extensions.insert(std::make_pair(x->name(), x)).second) {
    fprintf(stderr, "extensions must have unique names (got two named
\"%s\"!)\n", x->name());
    abort();
  }

  x->set_processor(this);
}
```

```
class processor_t :  public abstract_device_t
{
 ...
  std::unordered_map<std::string, extension_t*> custom_extensions;
  disassembler_t* disassembler;
  ...
  std::vector<bool> extension_table;
   ...
   std::vector<insn_desc_t*> instructions;
   ...
}
```

Spike启动流程—解析支持的其它扩展指令集

riscv/procesor.cc:

void processor_t::parse_isa_string(const char* str)

```
while (*p) {
  if (islower(*p)) {
    ...
  } else if (*p == '_') {
    const char* ext = p + 1, *end = ext;
    if (*ext == 'x') {
      p++;
      continue;
    }
    while (islower(*end))
      end++;
    auto ext_str = std::string(ext, end - ext);
    if (ext_str == "zfh") {
      extension_table[EXT_ZFH] = true;
    } else {
      sprintf(error_msg, "unsupported extension '%s'", ext_str.c_str());
      bad_isa_string(str, error_msg);
    }
    p = end;
  } else ...
}
```

扩展表示：_zfh, _xnice_demo

Nuclei NICE指令集

Nuclei™ Instruction Co-unit Extension

“Nuclei Core Series supports configurable NICE (Nuclei Instruction Co-unit Extension) to support extensive customization and specialization. NICE allows customers to create user-defined instructions, enabling the integrations of custom hardware co-units that improve domain-specific performance while reducing power consumption.”

--《Nuclei™ N200 Nuclei Instruction Co-Unit Extension》

Nuclei NICE指令集—指令编码

Table 2-1 RISC-V base opcode map, inst[1:0]=11

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

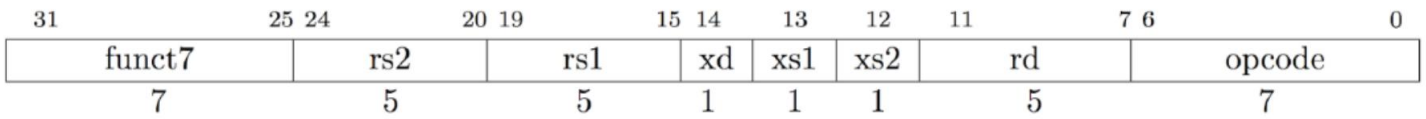


Figure 2-1 NICE instruction format

指令集扩展方法—内部指令集扩展方法

- 内部指令集支持数据结构：riscv/processor.h

```
std::vector<bool> extension_table;
```

```
typedef enum {
  // 65('A') ~ 90('Z') is reserved for standard isa in misa
  EXT_ZFH   = 0,
} isa_extension_t;
```

```
  const char* all_subsets =
"imafdqch"
#ifdef __SIZEOF_INT128__
    "v"
#endif
    "";
```

指令集扩展方法—内部指令集扩展方法举例

- 内部指令集支持数据结构：riscv/processor.h

```
std::vector<bool> extension_table;
```

```
typedef enum {
  // 65('A') ~ 90('Z') is reserved for standard isa in misa
  EXT_ZFH   = 0,
} isa_extension_t;
```

```
typedef enum {
  // 65('A') ~ 90('Z') is reserved for standard isa in misa
  EXT_ZFH   = 0,
  EXT_ZNICE,
} isa_extension_t;
```

```
void processor_t::parse_priv_string(const char* str)
```

```
      if (ext_str == "zfh") {
        extension_table[EXT_ZFH] = true;
      } else  if (ext_str == "znice") {
        extension_table[EXT_ZNICE] = true;
      } else {
        sprintf(error_msg, "unsupported extension '%s'",
ext_str.c_str());
        bad_isa_string(str, error_msg);
      }
```

指令集扩展方法—内部指令集扩展方法举例

- 内部指令集支持数据结构：riscv/processor.h

```
std::vector<bool> extension_table;
```

```
--isa=rv64gc_znice
```

```
typedef enum {
  // 65('A') ~ 90('Z') is reserved for standard isa in misa
  EXT_ZFH   = 0,
} isa_extension_t;
```

```
typedef enum {
  // 65('A') ~ 90('Z') is reserved for standard isa in misa
  EXT_ZFH   = 0,
  EXT_ZNICE,
} isa_extension_t;
```

```
void processor_t::parse_priv_string(const char* str)
```

```
      if (ext_str == "zfh") {
        extension_table[EXT_ZFH] = true;
      } else  if (ext_str == "znice") {
        extension_table[EXT_ZNICE] = true;
      } else {
        sprintf(error_msg, "unsupported extension '%s'",
ext_str.c_str());
        bad_isa_string(str, error_msg);
      }
```

指令集扩展方法—外部指令集扩展方法

riscv/extension.h

```
class extension_t
{
 public:
  virtual std::vector<insn_desc_t> get_instructions() = 0;
  virtual std::vector<disasm_insn_t*> get_disasms() = 0;
  virtual const char* name() = 0;
  virtual void reset() {};
  virtual void set_debug(bool value) {};
  virtual ~extension_t();

  void set_processor(processor_t* _p) { p = _p; }
 protected:
  processor_t* p;

  void illegal_instruction();
  void raise_interrupt();
  void clear_interrupt();
};
```

主要实现的功能：

- **name()**：返回扩展指令集的名称，用于查找该指令集
- **get_instructions()**：返回扩展指令集所支持的所有指令描述信息
- **get_disasms()**：返回扩展指令集所支持的所有指令的汇编描述信息

指令集扩展方法—外部指令集扩展举例

nicext/
nice_demo.cc

```
class nice_demo_t : public extension_t
{
 public:
  const char* name() { return "nice_demo"; }

  nice_demo_t() {}

  std::vector<insn_desc_t> get_instructions() {
    std::vector<insn_desc_t> insns;
    insns.push_back((insn_desc_t){0x0400207b, 0xFE00707F, custom_sbuf, custom_sbuf});
    insns.push_back((insn_desc_t){0x0800207b, 0xFE00707F, custom_wsetup, custom_wsetup});
    insns.push_back((insn_desc_t){0x0C00607b, 0xFE00707F, custom_rowsum, custom_rowsum});
    return insns;
  }

  std::vector<disasm_insn_t*> get_disasms() {
    std::vector<disasm_insn_t*> insns;
    insns.push_back(new disasm_insn_t("custom_sbuf", 0x0400207b, 0xFE00707F, {&xrs1}));
    insns.push_back(new disasm_insn_t("custom_wsetup", 0x0800207b, 0xFE00707F, {&xrs1}));
    insns.push_back(new disasm_insn_t("custom_rowsum", 0x0C00607b, 0xFE00707F, {&xrd, &xrs1}));
    return insns;
  }
};
```

指令集扩展方法—外部指令集扩展举例

```
class nice_demo_t : public extension_t
{
 public:
  const char* name() { return "nice_demo"; }

  nice_demo_t() {}

  std::vector<insn_desc_t> get_instructions() {
    std::vector<insn_desc_t> insns;
    insns.push_back((insn_desc_t){0x0400207b, 0xFE00707F, custom_sbuf, custom_sbuf});
    insns.push_back((insn_desc_t){0x0800207b, 0xFE00707F, custom_wsetup, custom_wsetup});
    insns.push_back((insn_desc_t){0x0C00607b, 0xFE00707F, custom_rowsum, custom_rowsum});
    return insns;
  }

  std::vector<disasm_insn_t*> get_disasms() {
    std::vector<disasm_insn_t*> insns;
    insns.push_back(new disasm_insn_t("custom_sbuf", 0x0400207b, 0xFE00707F, {&xrs1}));
    insns.push_back(new disasm_insn_t("custom_wsetup", 0x0800207b, 0xFE00707F, {&xrs1}));
    insns.push_back(new disasm_insn_t("custom_rowsum", 0x0C00607b, 0xFE00707F, {&xrd,
&xrs1}));
    return insns;
  }
};
```

```
--extension=nice_demo --extlib=libnicext.so
```

指令集扩展方法—外部指令集扩展举例

```
class nice_demo_t : public extension_t
{
 public:
  const char* name() { return "nice_demo"; }

  nice_demo_t() {}

  std::vector<insn_desc_t> get_instructions() {
    std::vector<insn_desc_t> insns;
    insns.push_back((insn_desc_t){0x0400207b, 0xFE00707F, custom_sbuf, custom_sbuf});
    insns.push_back((insn_desc_t){0x0800207b, 0xFE00707F, custom_wsetup, custom_wsetup});
    insns.push_back((insn_desc_t){0x0C00607b, 0xFE00707F, custom_rowsum, custom_rowsum});
    return insns;
  }

  std::vector<disasm_insn_t*> get_disasms() {
    std::vector<disasm_insn_t*> insns;
    insns.push_back(new disasm_insn_t("custom_sbuf", 0x0400207b, 0xFE00707F, {&xrs1}));
    insns.push_back(new disasm_insn_t("custom_wsetup", 0x0800207b, 0xFE00707F, {&xrs1}));
    insns.push_back(new disasm_insn_t("custom_rowsum", 0x0C00607b, 0xFE00707F, {&xrd,
&xrs1}));
    return insns;
  }
};
```

```
--isa=rv64gc_xnice_demo --extlib=libnicext.so
```

```
REGISTER_EXTENSION(nice_demo, []() { return new nice_demo_t; })
```

指令集扩展方法—ROCC扩展

riscv/rocc.h

```
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();
};
```

```
struct rocc_insn_t
{
  unsigned opcode : 7;
  unsigned rd : 5;
  unsigned xs2 : 1;
  unsigned xs1 : 1;
  unsigned xd : 1;
  unsigned rs1 : 5;
  unsigned rs2 : 5;
  unsigned funct : 7;
};

union rocc_insn_union_t
{
  rocc_insn_t r;
  insn_t i;
};
```

指令集扩展方法—ROCC扩展

```
#define customX(n) \
  static reg_t c##n(processor_t* p, insn_t insn, reg_t pc) \
  { \
    rocc_t* rocc = static_cast<rocc_t*>(p->get_extension()); \
    rocc_insn_union_t u; \
    u.i = insn; \
    reg_t xs1 = u.r.xs1 ? RS1 : -1; \
    reg_t xs2 = u.r.xs2 ? RS2 : -1; \
    reg_t xd = rocc->custom##n(u.r, xs1, xs2); \
    if (u.r.xd) \
      WRITE_RD(xd); \
    return pc+4; \
  } \
  \
  reg_t rocc_t::custom##n(rocc_insn_t insn, reg_t xs1, reg_t xs2) \
  { \
    illegal_instruction(); \
    return 0; \
  }
customX(0)
customX(1)
customX(2)
customX(3)
```

riscv/rocc.cc

```
std::vector<insn_desc_t> rocc_t::get_instructions()
{
  std::vector<insn_desc_t> insns;
  insns.push_back((insn_desc_t){0x0b, 0x7f, &::illegal_instruction, c0});
  insns.push_back((insn_desc_t){0x2b, 0x7f, &::illegal_instruction, c1});
  insns.push_back((insn_desc_t){0x5b, 0x7f, &::illegal_instruction, c2});
  insns.push_back((insn_desc_t){0x7b, 0x7f, &::illegal_instruction, c3});
  return insns;
}

std::vector<disasm_insn_t*> rocc_t::get_disasms()
{
  std::vector<disasm_insn_t*> insns;
  return insns;
}
```

指令集扩展方法—ROCC扩展举例

nicext/
nice_rocc_demo.cc

```
class nice_rocc_demo_t : public rocc_t
{
 public:
  const char* name() { return "nice_rocc_demo"; }

  reg_t custom3(rocc_insn_t insn, reg_t xs1, reg_t xs2)
  {
    ...
    switch (insn.funct)
    {
     ...   //function code
     default:
        illegal_instruction();
        break;
    }

   ...
  }

  nice_rocc_demo_t() { ... }
 private:
    ...
};
```

指令集扩展方法—ROCC扩展举例

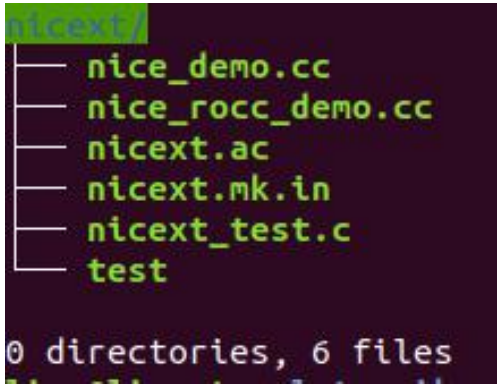
```
class nice_rocc_demo_t : public rocc_t
{
 public:
  const char* name() { return "nice_rocc_demo"; }

  reg_t custom3(rocc_insn_t insn, reg_t xs1, reg_t xs2)
  {
    ...
    switch (insn.funct)
    {
     ...   //function code
     default:
        illegal_instruction();
        break;
    }

   ...
  }

  nice_rocc_demo_t() { ... }
 private:
    ...
};
```

--extension=nice_rocc_demo --extlib=libnicext.so

--isa=rv64gc_xnice_rocc_demo --extlib=libnicext.so

REGISTER_EXTENSION(nice_rocc_demo, []() { return new nice_rocc_demo_t; })

指令集扩展方法—外部扩展编译




```
nicext_subproject_deps = \
        spike_main \
        riscv \
        disasm \
        softfloat \

nicext_srcs = \
        nice_demo.cc \
        nice_rocc_demo.cc \

nicext_CFLAGS = -fPIC

nicext_install_shared_lib = yes
```

参考customext扩展的支持方法

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

谢谢各位

欢迎提问、讨论、交流合作