OS and Compiler Support for RISC-V

Zhenyang Dai   Wei Zhang

December 16, 2018

Department of Computer Science and Technology, Tsinghua University

Table of contents

1. Introduction to RISC-V
2. OS Support for RISC-V
3. Compiler Support for RISC-V
4. Summary

Introduction to RISC-V

The Free and Open RISC Instruction Set Architecture

Figure 1: RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration.

General Registers

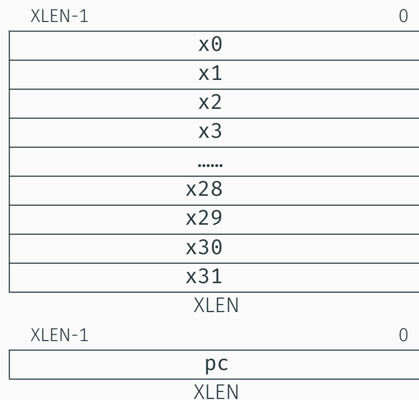

Figure 2: RISC-V base unprivileged integer register state.

RISC-V Privilege Levels

Level	Encoding	Name	Abbreviation
0	`00`	User/Application	U
1	`01`	Supervisor	S
2	`10`	~~Hypervisor~~	H
3	`11`	Machine	M

Table 1: RISC-V privilege levels.

Control and Status Registers (CSRs)

- `sstatus`: CPU state.
- `stvec`: trap vector base address.
- `sscratch`: hold kernel stack pointer when executing user code.
- `satp`: address translation mode.

RISC-V Privileged Software Stack

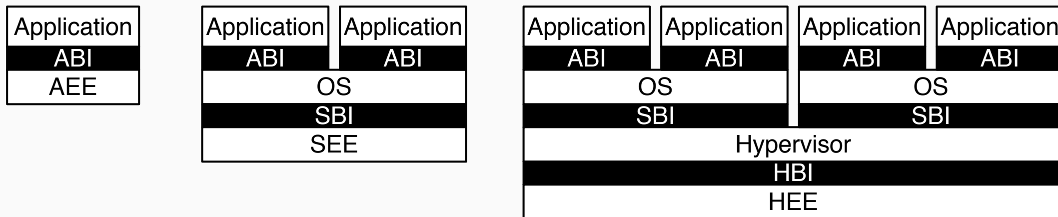

Figure 3: Different implementation stacks supporting various forms of privileged execution.

- ABI: Application Binary Interface
- AEE: Application Execution Environment
- SBI: Supervisor Binary Interface
- SEE: Supervisor Execution Environment

OS Support for RISC-V

WARNING

The talk is based on experiments (ucore_os_lab) on RISC-V at the course “Operating Systems” in Tsinghua University.

1. Suggestions given here may not be best practices.
2. Information may be outdated or even incorrect.
3. Figures and tables mostly come from riscv.org.
4. If you know little about RISC-V, riscvbook by Prof. Patterson and Dr. Waterman is a good starting point.

RISC-V OS Current Status

According to RISC-V Software Status, there are 6 operating systems (kernels) supporting RISC-V as of Dec. 2018.

- Linux Kernel
- seL4
- Apache Mynewt
- Real Time Operating System (RTOS)
  - RTEMS
  - FreeRTOS
  - Zephyr

Overview of Supporting RISC-V

- Booting
- Interrupt Handling
- Memory Management

Boot Loader

- Berkeley Boot Loader (BBL) is an official implementation of SEE for tethered RISC-V systems.
- BBL is actually designed for hosting the RISC-V Linux port.
- Upon booting, BBL will initialize the system and transfer control to the kernel.

Trap Handling

- Is there something like “interrupt vector table” in RISC-V? How to associate interrupt handlers with interrupt requests?
- What information has to be saved before processing interrupt requests? More specifically, what is the structure of the trap frame?
- Are there reserved registers like `$k0` and `$k1` in MIPS?

Trap Vector

Figure 4: Supervisor trap vector base address register (`stvec`).

Value	Name	Description
0	Direct	All exceptions set **pc** to BASE.
1	Vectored	Asynchronous interrupts set **pc** to BASE+4×cause.
≥2	—	Reserved

Table 2: Encoding of `stvec` MODE field.

Trap Frame Structure

All general purpose registers needs saving.

```
struct pushregs {
    uintptr_t zero;  // Hard-wired zero
    uintptr_t ra;    // Return address
    uintptr_t sp;    // Stack pointer
    uintptr_t gp;    // Global pointer
    uintptr_t tp;    // Thread pointer
    /* many other registers */
    uintptr_t t3;    // Temporary
    uintptr_t t4;    // Temporary
    uintptr_t t5;    // Temporary
    uintptr_t t6;    // Temporary
};
```

Trap Frame Structure (cont'd)

Besides general purpose registers, other control registers are necessary for interrupt handling.

```
struct trapframe {
    // general purpose registers
    struct pushregs gpr;
    // current operating state of CPU
    uintptr_t status;
    // addr of inst that encountered the exception
    uintptr_t epc;
    // exception-specific information
    uintptr_t tval;
    // event that caused the trap
    uintptr_t cause;
};
```

Setting Up Trap Entry

```
1      .align 2 # align on 4 bytes
2      .globl __alltraps
3  __alltraps:
4      SAVE_ALL # save registers to form a trapframe
5
6      move a0, sp
7      jal trap # void trap(struct trapframe *tf);
8
9      .globl __trapret
10 __trapret:
11     RESTORE_ALL # restore registers
12     sret # return from supervisor call
```

`void __alltraps()` can be set as the trap entry.

Setting Up Trap Entry (cont'd)

Writing the address of function `__alltraps()` to `stvec` to set the trap vector base address.

```
void init_interrupt() {
    extern void __alltraps(void);
    // Set the trap vector base address
    write_csr(stvec, &__alltraps);
}
```

Problems

1. Where am I from?
   - **User-mode** $\longrightarrow$ **Supervisor-mode**: Syscall
   - **Supervisor-mode** $\longrightarrow$ **Supervisor-mode**: Trap during trap handling
2. If I am from user-mode, how can I switch to kernel stack?
   - In x86, we have task state segment (TSS) and CPU switches to kernel stack automatically.

Supervisor Scratch Register

At the beginning of a trap handler, `sscratch` is swapped with a user register to provide an initial working register.

Figure 5: Supervisor Scratch Register.

Dedicated Usage of `sscratch`

1. Clear `sscratch` when booting and entering S-mode.

   ```
   csrw sscratch, x0
   ```

2. Store kernel stack pointer `sp` into `sscratch` whenever leaving S-mode

   ```
   csrw sscratch, sp
   ```

Invariants

1. When operating in S-mode, `sscratch` always contains 0.
2. When operating in U-mode, `sscratch` contains “kernel stack”.

When Interrupt Fires

1. Swap `sp` and `sscratch` at the beginning of trap handler.

   ```
   # atomic swap sscratch and sp
   csrrw sp, sscratch, sp
   ```

2. ```
   If from U-mode:
       sp == "kernel stack"
       sscratch == "user stack"
   ElseIf from S-mode:
       sp == 0
       sscratch == "kernel stack"
   ```

3. We know where we come from by checking `sp`.

   ```
   beqz sp, .Ltrap_from_supervisior_mode
   ```
   ```

SAVE_ALL

```
1         csrrw sp, sscratch, sp
2         bnez sp, _save_context
3     _restore_kernel_sp:
4         csrr sp, sscratch
5     _save_context:
6         addi sp, sp, -36 * REGBYTES
7         # save registers
8         sw x1, 1 * REGBYTES(sp)
9         sw x3, 3 * REGBYTES(sp)
10        ......
11        sw x31, 31 * REGBYTES(sp)
12        # Set sscratch to 0, so that if a recursive
13        # exception occurs, we know it came from the kernel
14        csrrw s0, sscratch, x0
15        sw s0, 2 * REGBYTES(sp)
```

RESTORE_ALL

```
1        lw s1, 32 * REGBYTES(sp) # s1 = sstatus
2        andi s0, s1, SSTATUS_SPP # back to U-mode?
3        bnez s0, _restore_context
4    _save_kernel_sp:
5        addi s0, sp, 36 * REGBYTES # Save kernel stack
6        csrw sscratch, s0
7    _restore_context:
8        # restore registers
9        lw x1, 1 * REGBYTES(sp)
10       lw x3, 3 * REGBYTES(sp)
11       ......
12       lw x31, 31 * REGBYTES(sp)
13       # restore sp lastly
14       LOAD x2, 2*REGBYTES(sp)
```

RISC-V Virtual-Memory Systems

- Sv32: Page-Based 32-bit Virtual-Memory System
  - Two level page table
  - 34-bit physical address
- Sv39: Page-Based 39-bit Virtual-Memory System
  - Three level page table
- Sv48: Page-Based 48-bit Virtual-Memory System
  - Four level page table

Sv32

Figure 6: Sv32 virtual address.

Figure 7: Sv32 physical address.

Figure 8: Sv32 page table entry.

Address Translation and Protection Register

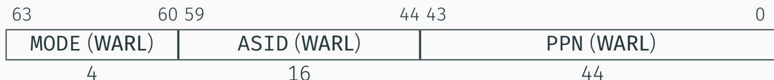

Figure 9: RV64 Supervisor address translation and protection register `satp`.

Value	Name	Description
0	Bare	No translation or protection.
1–7	—	Reserved
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	Sv57	Reserved for page-based 57-bit virtual addressing.
11	Sv64	Reserved for page-based 64-bit virtual addressing.
12–15	—	Reserved

Table 3: Encoding of `satp` MODE field for RV64.

Compiler Support for RISC-V

RISC-V Compiler Current Status

Let's limit our scope to C compilers.

`gcc`	`clang/llvm`[1]
Full support for RISC-V 32 / 64.	RISC-V 32: llvm 7.0+ basic support;
	RISC-V 64: no upstream support.
Compiles Linux kernel under RISC-V 32 / 64	RISC-V 32: in development;
	RISC-V 64: can't even compile HelloWorld

Table 4: C compilers for RISC-V 64

[1] Subject was the 7.0 release

LLVM for RISC-V

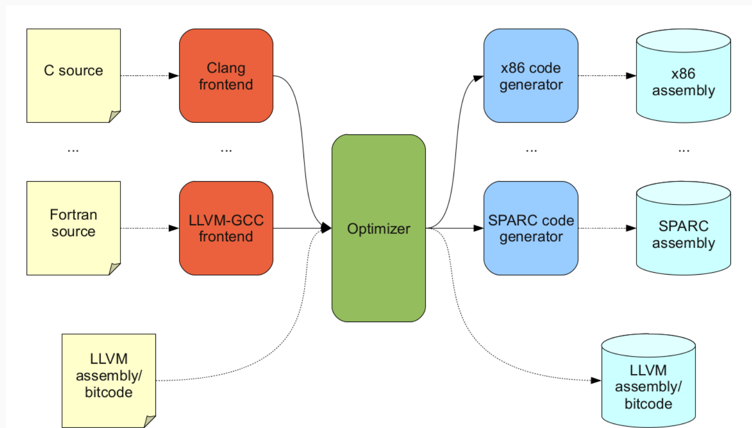

Figure 10: Basic LLVM architecture. From “Design and Implementation of a TriCore Backend for the LLVM Compiler Framework”, Christoph Erhardt

General status

- Still work in progrss.
- RISC-V 32: near completion.
- RISC-V 64: a series of external patches authored by Alex Bradbury (asb)

RISC-V 64 external patchset[2]

- MC subsystem (assembly parsing and object generation) for RV64IMAFD.
- Codegen subsystem (translating intermediate representation into target code): only preliminary support for RV64I.

[2] https://github.com/lowRISC/riscv-llvm/

asb's Patchset

- A total of 84 patches.
- After applying, the RISCV backend is less than 4500 loc.

How asb did it

- Registration for the RISC-V backend.
- MC subsystem (assembler) of RV32I.
- CodeGen subsystem (compiler) for RV32I.
- MC subsystem for RV32MAFD, then RV64IMAFD.
- CodeGen for RV32M, RV64I ...
- Details including float point support etc.

LLVM RISC-V: Registration

	Purpose	Files / Changes
1	Tell LLVM that we have a new archicture RISC-V.	`Triple.{h,cpp}`
2	Add necessary ELF definitions `e_machine`, `e_flags`, relocations ...	`ELF.{h,cpp}`, `ELFObject.h`
3	Describe target characteristics for the RISC-V backend	Define a target machine class, create TableGen files describing the register set and some simplest instructions, call registration routines.

```
class RISCVReg<bits<5> Enc, string n, list<string> alt = []>
    : Register<n> {
  let HWEncoding{4-0} = Enc;
  let AltNames = alt;
}

def X0  : RISCVReg<0, "x0", ["zero"]>, DwarfRegNum<[0]>;
// ...
def X31 : RISCVReg<31,"x31", ["t6"]>, DwarfRegNum<[31]>;

def XLenVT : ValueTypeByHwMode<[RV32, RV64, DefaultMode],
                               [i32,  i64,  i32]>;
def GPR : RegisterClass< "RISCV", [XLenVT], 32,
    (add (sequence "X%u", 0, 31))> { /* ... */ }
```

LLVM RISC-V: Registration

After this step, RISC-V targets can be seen with

- the compiler `llc -version`
- the assembler `llvm-mc -version`.

But the compiler / assembler is not ready. If you ever try to compile any code, it complains “MCAsmInfo not initialized.”.

LLVM RISC-V: MC Subsystem

MC stands for machine code. The MC subsystem in LLVM deals with assembly, disassembly, object code handling etc.

	Purpose	Files / Changes
1	Add `MCTargetDesc` components describing the assembly	Define an `MCAsmInfo`
2	Support instruction encoding and assembly printing	Subclass `MCAsmBackend`, `MCCodeEmitter` and `MCInstPrinter`; add a target ELF object writer
3	Add an assembly parser	Subclass `MCAsmParser` and add necessary `MCParsedAsmOperand`

After this step, the assembler is already working. Of course, it supports only those simplest instructions we added in the registration phase.

```
$ llvm-mc -arch=riscv32 -filetype=asm|obj input.s
```

will work fine if `input.s` contains only those simplest instructions.

```
$ llc -march=riscv32 -filetype=asm|obj input.ll
```

won't work because our backend doesn't yet know how to convert LLVM IR into RISC-V code.

RISC-V LLVM: CodeGen Subsystem

CodeGen is short for code generation. During CodeGen, target-independent LLVM IR code is translated into target-specific code (assembly or object file).

	Purpose	Files / Changes
1	Describe the whole set of RV32I instructions, add more operands (e.g. `uimm20`)	`RISCVInstrInfo.td` and `RISCVInstrFormats.td`
2	Instruction selection patterns	`RISCVInstrInfo.td`, and `SelectionDAGISel`
3	Handling function calling including frame lowering, args / return lowering	Define calling conventions, implement frame lowering, and subclass `TargetLowering`

RISC-V LLVM: CodeGen subsystem

```
// Define the ADDI instruction.
class ALU_ri<bits<3> funct3, string opcodestr>
    : RVInstI<funct3, OPC_OP_IMM,
              (outs GPR:$rd), (ins GPR:$rs1, imm12:$imm12),
              opcodestr, "$rd, $rs1, $imm12">;
def ADDI : ALU_ri<0b000, "addi">;

// Instruction selection pattern
def : Pat<(add GPR:$rs1, simm:$imm12),
          (ADDI GPR:$rs1, simm12:$imm12)>

// Constant materializing pattern
def : Pat<(simm32:$imm), (ADDI (LUI (HI20 imm:$imm)),
                               (LO12Sext imm:$imm))>;
```

RISC-V LLVM: CodeGen Subsystem

Finally after this step, LLVM IR could be translated to RISC-V code.

```
$ llc -march=riscv32 -filetype=asm|obj input.ll
```

produces the desired assembly / object file.

RISC-V LLVM

Still, some support is not ready...

- Fence instructions in RV64I
- Named CSR access
- Fix position independent code generation.
- CodeGen for RV64ACFD

Summary

Summary

OS and compiler support needs further improvement for promoting RISC-V.

- We need more operating systems supporting RISC-V.
  - Porting real-time operating systems is a better choice so far.
- We need LLVM for RISC-V.

Questions?

Backup Slide

- “J” Standard Extension for Dynamically Translated Languages
  - Additional ISA support for dynamic checks and garbage collection
- “T” Standard Extension for Transactional Memory
  - There is still much debate on the best way to support atomic operations involving multiple addresses
- “P” Standard Extension for Packed-SIMD Instructions
- “V” Standard Extension for Vector Operations
  - Cray-style vector operations with dedicated registers
- “N” Standard Extension for User-Level Interrupts