QEMU Support for RISC-V

Current State and Future Releases

Peer Adelt, Bastian Koppelmann, Wolfgang Mueller, Christoph Scheytt
Heinz Nixdorf Institute
System and Circuit Technology
Paderborn, Germany
Overview

- Virtual prototyping for RISC-V systems
  - QEMU
  - Supported Machines and Peripherals

- Configurable QEMU RISC-V cores
  - DecodeTree for Instruction Set Specification
QEMU as Virtual Prototyping Platform

- fast and functional Instruction Set Simulator

- emulates many Instruction Sets:
  - Intel x86_32 / x86_64
  - ARM
  - Infineon TriCore™ (maintained by UPB)
  - RISC-V (maintained by UC Berkeley, UPB, SiFive & WD)
  - ...

- support for full system emulation for RISC-V FE310 & U540 Processors (SiFive)

Image sources:
- https://commons.wikimedia.org/wiki/File:Qemu_logo.svg

Available since QEMU v2.12.0
RISC-V board support in QEMU

SiFive HiFive 1:
- Freedom E310 Microcontroller
  - Single core (RV32IMAC)
  - Clock rate 320 MHz
  - 16 KB ICache
  - 16 KB Scratchpad Memory

SiFive HiFive Unleashed:
- Freedom U540 SoC
  - 4 application cores (RV64GC) with virtual memory support
  - 1 management core (RV64IMAC)
  - Clock rate 1.5 GHz
  - 2 MB L2 Cache
  - Memory Protection Unit
  - 8 GB DDR4 memory
Peripheral Overview
Current status of the supported peripherals in QEMU

<table>
<thead>
<tr>
<th></th>
<th>HiFive1</th>
<th>HiFive Unleashed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug Interface</td>
<td></td>
<td>Yes (GDB)</td>
</tr>
<tr>
<td>Platform Level Interrupt Controller</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Core Local Interruptor</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>UART</td>
<td></td>
<td>Yes (QEMU virtual console)</td>
</tr>
<tr>
<td>QSPI</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>PWM</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

- **UART** is implemented and connected to **virtual console** for debugging
- All other peripherals and control regs are modelled as RAM **without functionality**
Next QEMU Release: Configurable RISC-V core
Automatic RISC-V Instruction Decoder Generation

Motivation

Current QEMU release v3.1.0: **Hand-written** instruction decoder

- No separation of instruction decoder and translator: hard to maintain
- Time-consuming & error-prone  

**> 180 Instructions for RV64G (ISA Spec. v2.2)**

First efforts to separate instruction decoder and translator carried out by UPB in 2017

- In-house Python script to generate C-Code for instruction decoder
- In-house YAML-based formal ISA Specification
- QEMU developer mailing list: concept was discussed and considered
  - QEMU DecodeTree was developed in cooperation with other QEMU developers
Automatic RISC-V Instruction Decoder Generation

DecodeTree

QEMU Release v4.0: RISC-V Code Generator will be based on DecodeTree

- RISC-V Basic Block → QEMU Code Generator → Host PC Translated Block
  - Instruction Decoder
  - Instruction Translator

DecodeTree

- Script for automatic generation of instruction decoder from textual ISA specification
- Automatic operand extraction from instruction words
- TCG translator becomes strictly separated into *Instruction Decoder* and *Instruction Translator*
- Instruction Translator has to implemented manually
### DecodeTree

#### Overview

- **Instruction Set Specification**
  - For all 189 RV32/64GC Instructions

- **Instruction Decoder**

- **Instruction Translator**

- **Virtual RISC-V CPU Core**

- `decodetree.py`
RISC-V Integer Register-Register Operations Example
How to write an Instruction Set Specification

Excerpt from: Volume I: RISC-V User-Level ISA V2.2

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>0000000</td>
<td>src2</td>
<td>src1</td>
<td>ADD/SLT/SLTU</td>
<td>dest</td>
<td>OP</td>
</tr>
<tr>
<td>0000000</td>
<td>src2</td>
<td>src1</td>
<td>AND/OR/XOR</td>
<td>dest</td>
<td>OP</td>
</tr>
<tr>
<td>0000000</td>
<td>src2</td>
<td>src1</td>
<td>SLL/SRL</td>
<td>dest</td>
<td>OP</td>
</tr>
<tr>
<td>0100000</td>
<td>src2</td>
<td>src1</td>
<td>SUB/SRA</td>
<td>dest</td>
<td>OP</td>
</tr>
</tbody>
</table>

System and Circuit Technology
DecodeTree
How to write an Instruction Set Specification

Example: Integer Register-Register instructions from the RV32I Base Integer ISA

# Operand fields:
%rd    7:5
%rs1   15:5
%rs2   20:5

# Formats:
@r     .........

# Instructions:
add    00000000 ......... 000 ......... 0110011 @r
sub    01000000 ......... 000 ......... 0110011 @r
xor    00000000 ......... 100 ......... 0110011 @r
...

Operands
%rs2   %rs1   %rd

Opcode bits

Example: Integer Register-Register instructions from the RV32I Base Integer ISA
DecodeTree
The generated instruction decoder

static void extract_r(arg_0 *a, uint32_t insn){
    a->rd = extract32(insn, 7, 5);
    a->rs1 = extract32(insn, 15, 5);
    a->rs2 = extract32(insn, 20, 5);
}

static bool decode(DisasContext *ctx, uint32_t insn){ ... }

Automatic extraction of instruction operands as C Struct

# Formats:
@r ....... ....... ....... ....... ....... ....... $rs2 $rs1 $rd

# Instructions:
add 0000000 ....... ....... 000 ....... 0110011 @r
sub 0100000 ....... ....... 000 ....... 0110011 @r
xor 0000000 ....... ....... 100 ....... 0110011 @r
### DecodeTree

The generated instruction decoder

```c
static void extract_r(arg_0 *a, uint32_t insn){
    a->rd = extract32(insn, 7, 5);
    a->rs1 = extract32(insn, 15, 5);
    a->rs2 = extract32(insn, 20, 5);
}

static bool decode(DisasContext *ctx, uint32_t insn){
    union {
        arg_0 f_0;
    } u;
    extract_r(&u.f_0, insn);
    switch (insn & 0xfe00707f) {
        case 0x00000033:
            return trans_add(ctx, &u.f_0); /* 0000000 ....... ....... 000 ....... 0110011 */
        case 0x00004033:
            return trans_xor(ctx, &u.f_0); /* 0000000 ....... ....... 100 ....... 0110011 */
        case 0x40000033:
            return trans_sub(ctx, &u.f_0); /* 0100000 ....... ....... 000 ....... 0110011 */
    }
    return false;
}
```

Decoder calls user-provided instruction translation routines
DecodeTree
Last step: instruction translation has to be implemented manually

```
static bool trans_add(DisasContext *ctx, arg_add *a)
{
    TCGv t1 = gen_get_gpr_as_temp(a->rs1);
    TCGv t2 = gen_get_gpr_as_temp(a->rs2);
    tcg_gen_add_tl(t1, t1, t2);
    gen_set_gpr(a->rd, t1);
    ...
}

static bool trans_sub(DisasContext *ctx, arg_sub *a)
{
    TCGv t1 = gen_get_gpr_as_temp(a->rs1);
    TCGv t2 = gen_get_gpr_as_temp(a->rs2);
    tcg_gen_sub_tl(t1, t1, t2);
    gen_set_gpr(a->rd, t1);
    ...
}
```
Configurable RISC-V Core
How to create a RISC-V ISA Subset Configuration

Example:
Create a virtual CPU core that supports only ISA Extensions I,F,M,A

Necessary steps:
1. Copy DecodeTree ISA Specification file shipped with QEMU (contains entries for all 189 Instructions of RV64G)
2. Disable (comment out or delete) all instructions for ISA modules that should not be supported in your RISC-V implementation (here: ISA modules C and D)
3. Add your custom instructions, if applicable
4. Rebuild QEMU
DecodeTree for RISC-V

Summary

RISC-V Implementation completely rewritten and migrated to DecodeTree:

- Ships with the DecodeTree RISC-V ISA Specification document
- Implements 189 Instructions of all ISA Modules (Spec. V2.2):
  - I (Base Integer)
  - F (Single-Precision Float)
  - D (Double-Precision Float)
  - M (Multiplier)
  - A (Atomic)
  - C (Compressed)
- Separation of decoder logic and translation functionality
- Allows for easy modification and extension with your custom RISC-V features
Thank you for your attention

Questions? Don‘t hesitate to ask 😊