White Paper on adding Custom RISC-V Instructions to QEMU

Hugh O’Keeffe, VP Global Engineering, Ashling
v0.3, 22nd Feb 2021
Table of Contents

Introduction .................................................................................................................................................. 3
Requirements and Overview ...................................................................................................................... 4
Requirements ........................................................................................................................................... 4
Overview .................................................................................................................................................. 4
The RISC-V RV32I Custom Instruction ...................................................................................................... 4
Adding a Custom RISC-V Instruction to QEMU ......................................................................................... 5
  Installing MSYS2 including the required QEMU packages/sources and building the QEMU executable .......................................................... 5
  Modifying the QEMU source-code to support the new custom instruction ........................................ 8
  Translating New Target Instructions ..................................................................................................... 8
Debugging the Custom RISC-V Instruction using RiscFree™ ................................................................. 10
  Building a RISC-V application which uses the new custom instruction (using RiscFree™) ............. 10
  Debugging the new instruction within RiscFree™ .................................................................................. 11
Conclusion ................................................................................................................................................ 12
  More Information ................................................................................................................................. 12

List of Figures

Figure 1. RiscFree™ IDE and Debugger running QEMU ........................................................................ 3
Figure 2. Downloading latest MSYS repository updates .................................................................... 5
Figure 3. Configuring for QEMU Build ................................................................................................. 6
Figure 4. Building QEMU for RISCV32 .............................................................................................. 6
Figure 5. QEMU for RISCV32 Build complete ..................................................................................... 7
Figure 6. R-Type Encoding from Green Card ........................................................................................ 8
Figure 7. R-Type Encoding from Green Card ........................................................................................ 10
Figure 8. Debugging the new Custom Instruction ................................................................................... 11
Figure 9. After executing the new Custom Instruction ........................................................................... 11
QEMU (www.qemu.org) is a “Quick EMUlator” which provides software-based emulation of core architectures including RISC-V, Arm and many others. QEMU includes a built-in debugger interface allowing end-users to begin software development for their target architecture before hardware availability – the process generally referred to as simulating or using an Instruction Set Simulator (ISS).

QEMU supports all target core Instruction Set Architectures (ISAs) – for example, for RISC-V, the RV32I and RV64I ISAs are supported amongst others. QEMU support can also be extended to support any custom instructions, enhancements or additions end-users may make to the ISA for the purposes of optimising their chip design. Of course, having QEMU support for custom instructions provides a powerful mechanism for evaluating the effectiveness of these instructions before committing them to silicon via RTL changes.

This paper provides an overview of how a unique custom instruction can be added to the RISC-V version of QEMU and how to use and debug applications using that instruction in Ashling’s RiscFree™ RISC-V IDE and Debugger.

Figure 1. RiscFree™ IDE and Debugger running QEMU
Requirements and Overview

Requirements
Adding a Custom RISC-V instruction requires modifying the QEMU source-code and rebuilding the QEMU executable. This requires you have some software engineering expertise and familiarity with:

1. The RISC-V instruction-set and programming architectures.
2. The ‘C’ programming language and development tools.

Overview
A step-by-step guide is provided to show how to add a single RISC-V custom instruction to QEMU (for the RV32I ISA) on a 64-bit Windows™ host with the MSYS2 (https://www.msys2.org/) build environment. Steps include:

1. Installing MSYS2 including the required QEMU packages.
2. Installing the QEMU source-code (we will use v5.0.0).
3. Building QEMU before we make any changes (to ensure steps 1 and 2 above completed ok).
4. Modifying and extending the QEMU source-code to support the new custom instruction.
5. Rebuilding the QEMU executable with support for the new custom instruction.
6. Building and debugging a RISC-V application which uses the new custom instruction using Ashling’s RiscFree™ (we will use v1.2.8).

The RISC-V RV32I Custom Instruction
Our custom instruction will be an R-type/R-format RISC-V RV32I instruction which supports two register inputs and one register output. The instruction is a bit counter as follows:

`BITCNT  dest=t0, src1=t1, src2=t2`

After execution, register t0 will equal the total number of bits set in t1 and t2.

For example, assume:

A2 = 0x0000-3000 and A3 = 0xF000-000F

After execution of:

`BITCNT  A1, A2, A3`

A1 will be equal to 0x0000-000A (i.e. A2 has 2 bits set and A3 has 8 bits set giving a total of 10 (0x0A) bits set).

For more details on the RISC-V ISA and R-type instructions, see https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.
Adding a Custom RISC-V Instruction to QEMU

Installing MSYS2 including the required QEMU packages/sources and building the QEMU executable

1. Follow ALL the instructions in the following link to install the MSYS build environment up to and including the Download the QEMU source code step: https://wiki.qemu.org/Hosts/W32#Native_builds_with_MSBYS2

![Figure 2. Downloading latest MSYS repository updates](image)

2. Install the ninja build package:
   $ pacman -Syu ninja

3. Install (checkout) the v5.0.0 QEMU source code:
   $ cd QEMU
   $ git checkout v5.0.0
4. Configure for building as follows:
$$
./configure --cross-prefix=x86_64-w64-mingw32- --enable-gtk --enable-sdl --target-list=riscv32-softmmu
$$

5. Build as follows:
$$
$make riscv32-softmmu all
$$
6. After the build, the QEMU simulator executable will reside in:
qemu/riscv32-softmmu/qemu-system-riscv32.exe

7. The simulator executable needs to be copied to replace the existing simulator executable in the **RiscFree™** v128 installation.

```bash
    copy "C:\msys64\home\<USER>\qemu\riscv32-softmmu\qemu-system-riscv32.exe"
    "C:\Users\<USER>\AppData\Local\Ashling\RiscFree_IDEv128\qemu\qemu-system-riscv32.exe"
```

You may make a backup of the original **RiscFree™** version first as follows:

```bash
    copy "C:\Users\<USER>\AppData\Local\Ashling\RiscFree_IDEv128\qemu\qemu-system-riscv32.exe"
    "C:\Users\<USER>\AppData\Local\Ashling\RiscFree_IDEv128\qemu\qemu-system-riscv32.exe.bak"
```

8. Finally, copy the latest MSYS2 DLLs to the **RiscFree™** v128 installation directory (replacing the existing ones)

```bash
    copy "C:\msys64\mingw64\bin\*.dll" "C:\Users\<USER>\AppData\Local\Ashling\RiscFree_IDEv128\qemu\*.\"
```
Modifying the QEMU source-code to support the new custom instruction

In this section, we will outline the steps to add the custom instruction to the RV32I ISA in QEMU. As previously mentioned, our custom instruction will be an R-type/R-format RISC-V instruction which supports two register inputs and one register output as follows:

\[
\text{BITCNT} \quad \text{dest}\text{-t0}, \text{src1}\text{-t1}, \text{src2}\text{-t2}
\]

The green-card provides a good overview of the encoding of the RISC-V instructions and an arbitrary insertion point was selected for the new BITCNT instruction which does not overlap with any existing RV32I instructions.

![32-bit Instruction Formats](https://www.cl.cam.ac.uk/teaching/1516/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf)

BITCNT has the following fields:

- **OPCODE** = "0110011", **FUNCT3** = "111" and **FUNCT7** = "0100000".

When adding a new custom instruction it is best to try to find an existing instruction structured similarly to the new instruction. In our case, the AND instruction is a good fit which has fields as follows:

- **OPCODE** = "0110011", **FUNCT3** = "111" and **FUNCT7** = "0000000".

**Translating New Target Instructions**

New target instructions must be translated into QEMU operations which in turn are transferred into host operations by the provided ports. This process is known as the “decodetree flow” and is documented here: [https://qemu.readthedocs.io/en/latest/devel/decodetree.html](https://qemu.readthedocs.io/en/latest/devel/decodetree.html). Given that the existing AND instruction and the new BITCNT were similar, reviewing the code to understand how AND was implemented greatly helped in understanding the changes needed for the new BITCNT implementation.

Implementing support for a new target instruction requires the following steps:

1. Fill out an encoding specification for the custom instruction BITCNT as follows:

   ```
   file: target/riscv/insn32.decode
   BITCNT   0100000 ..... ..... 111 ..... 0110011 @r
   ```

   See the previous decodetree link above ([Formats description](https://qemu.readthedocs.io/en/latest/devel/decodetree.html)) for more information.
2. Provide a translator function for the new custom instruction which implements (emulates) the required BITCNT functionality in the QEMU instruction set (also known as a Tiny Code Generator or tcg).


The new BITCNT translator ‘C’ function is in:

```c
file: target/riscv/insn_trans/trans_rvi.inc.c

static bool trans_bitcnt(DisasContext *ctx, arg_bitcnt *a) {
    TCGLabel *loop_source1 = gen_new_label();
    TCGLabel *loop_source2 = gen_new_label();
    TCGv source1, source2, dstval, cntval;
    source1 = tcg_temp_local_new();
    source2 = tcg_temp_local_new();
    dstval = tcg_temp_local_new();
    cntval = tcg_temp_local_new();
    // Count all the bits set in rs1 and rs2 and put that number in rd
    gen_get_gpr(source1, a->rs1);
    gen_get_gpr(source2, a->rs2);
    tcg_gen_movi_tl(cntval, 0x0);
    /* Count the bits that are set in the first register */
    gen_set_label(loop_source1);
    tcg_gen_andi_tl(dstval, source1, 0x1);
    tcg_gen_shri_tl(source1, source1, 0x1);
    tcg_gen_add_tl(cntval, cntval, dstval);
    tcg_gen_brcondi_tl(TCG_COND_NE, source1, 0x0, loop_source1);
    /* Count the bits that are set in the second register */
    gen_set_label(loop_source2);
    tcg_gen_andi_tl(dstval, source2, 0x1);
    tcg_gen_shri_tl(source2, source2, 0x1);
    tcg_gen_add_tl(cntval, cntval, dstval);
    tcg_gen_brcondi_tl(TCG_COND_NE, source2, 0x0, loop_source2);
    /* Update the destination register with the bits total */
    gen_set_gpr(a->rd, cntval);
    tcg_temp_free(source1);
    tcg_temp_free(source2);
    tcg_temp_free(dstval);
    tcg_temp_free(cntval);
    return true;
}
```

With the above changes made in the two files, re-build and copy the simulator into the RiscFree™ directory as explained earlier. Operation of the new custom instruction can now be observed as outlined in the following section.
Adding Custom RISC-V Instructions to QEMU

www.ashling.com
Adding Custom RISC-V Instructions to QEMU
www.ashling.com

Debugging the new instruction within RiscFree™

1. Set a breakpoint at the NOP instruction and run to it. Notice how the registers: a0, a1, a2 and a3 have been initialised as expected.

2. Now run/step over the BITCNT instruction and notice how the a1 register is updated as expected (i.e. it shows a total of 10 (0x0A) bits set in a2 and a3).
Conclusion

This paper provided an overview of how a unique custom instruction for the RV32I ISA can be added to the RISC-V version of QEMU and how to use and debug applications using that instruction in Ashling’s RiscFree™ RISC-V IDE and Debugger.

The RISC-V ISA is designed to be extendable to support custom instruction enhancements or additions allowing end-users to implement specific optimisations for their RISC-V based design. Having QEMU support for custom instructions provides a powerful mechanism for evaluating the effectiveness of these instructions before committing them to silicon via RTL changes.

More Information

If you have any questions or comments, then please contact me at hugh.okeeffe@ashling.com. For more details on how Ashling can help with your customised toolchain requirements then contact me and/or see the SERVICES section of our website at www.ashling.com. For example:

https://www.ashling.com/services-compilers/ covers custom compilers, IDEs, simulators and debuggers.

https://www.ashling.com/services-taas/ explains our Tools-as-a-Service™ (TaaS™) engagement model.