RISC-V ULI: Rocket Chip support

Gwyneth Chen

October 2022

1 Introduction

The aim of this project is ostensibly to enable user-level interrupts on the open-source RISC-V Rocket Chip from Berkeley, but the documentation here and in the referenced code are also intended to assist others with understanding and navigating some of the codebase. I embarked on this project not only ignorant of the Chipyard system, but also completely new to RISC-V, Scala, Verilog, waveform debugging, and indeed hardware design and implementation in general. Thus, while I have only a limited view of the system as a whole, I am hopeful that the information included here may be comprehensible and useful to a relatively inexpert audience.

1.1 User-level interrupts

Often, interrupts from an I/O device are used to notify a particular user process about the completion of a request or the arrival of a message from a peer process. General-purpose machines do not provide a mechanism for I/O to directly deliver the notification. As such, the kernel is responsible for acknowledging the interrupt and delivering a notification to the appropriate user process. The act of taking the interrupt, determining the proper destination process, and delivering the notification can be costly in both the number of cycles spent and in the cache overhead incurred.

When the true destination of the interrupt is a user-level process, I/O hardware that supports user-level access could easily be extended to compute the destination process for an interrupt being generated. This hardware could exploit knowledge of the destination process to drastically reduce interrupt overhead if the processor were extended to allow interrupts to be delivered directly to the user process.

User-level interrupts (ULI) allow a hardware interrupt to be handled directly by a user process. In comparison to standard Unix-style signals, ULIs eliminate significant overhead from saving and restoring context in the kernel. This can potentially come at the expense of reduced functionality, such as not having access to floating-point operations if these registers are not preserved, but allows

such context management to be tailored to what is needed for handling each particular interrupt.

ULIs have been proposed as an alternative method of installing device drivers from software, increasing the convenience of adding new devices and isolating impact if the device driver crashes or contains errors. Direct delegation of interrupts to user handlers may also facilitate high-performance I/O operations. Server-client systems with high throughput can also benefit from the use of ULIs. Since interrupt handling in these systems is often limited to small operations like enqueuing and dequeuing messages, execution time is dominated by unnecessary context-switching overhead, which provides a prime opportunity for speedup using ULIs. ULIs are similarly being considered to facilitate fast inter-process communication via active message-passing. In addition to reducing execution time, ULIs may also help reduce buffering overheads needed in some message-passing designs.

2 Enabling ULIs on Rocket Chip

As it stands, user-level interrupts are still only a proposed extension and are therefore not supported by Rocket Chip. Unless otherwise specified, all code discussed in the following section exists in the file `src/main/scala/rocket/CSR.scala`.

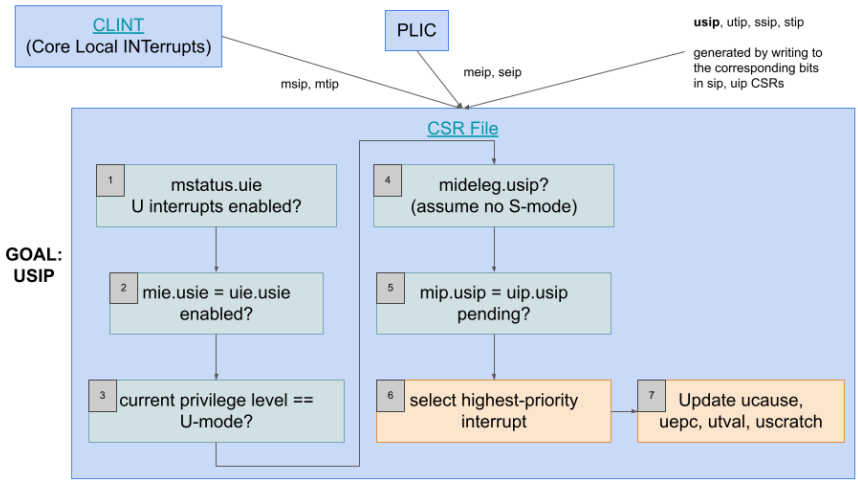

Figure 1: Target USIP processing on Rocket Chip

Figure 1 illustrates the rough pathway that we wish a pending user software interrupt (USIP) to travel, defined analogously to the current handling for machine and supervisor-level interrupts. The USIP is first created by writing the corresponding interrupt-pending bit in the UIP register, which is a view of the MIP register with only the user interrupt bits visible in U-mode. In the CSR

file (defined as object CSRFile), a number of checks (1-5 in the diagram) are performed to generate the mask of interrupts that are both pending and viable. For a USIP to be handled in U-mode, we want these checks include (1) whether user-level interrupts are enabled at all, (2) whether user software interrupts are specifically enabled, (3) whether the current privilege level is U-mode, (4) whether USIPs have been delegated to U-mode, and of course, (5) whether a USIP is actually pending. Once the mask of possible interrupts has been created, the highest-priority one is then selected, and the corresponding CSR states are updated. Note that for check (4), we are currently only checking `mideleg`, which indicates which interrupts are delegated from M-mode. In a system where supervisor mode (S-mode) also exists, we would need to check an additional `sideleg` register; however, this register is not yet defined in Rocket Chip, so we are assuming no S-mode exists in order to simplify this first step.[1]

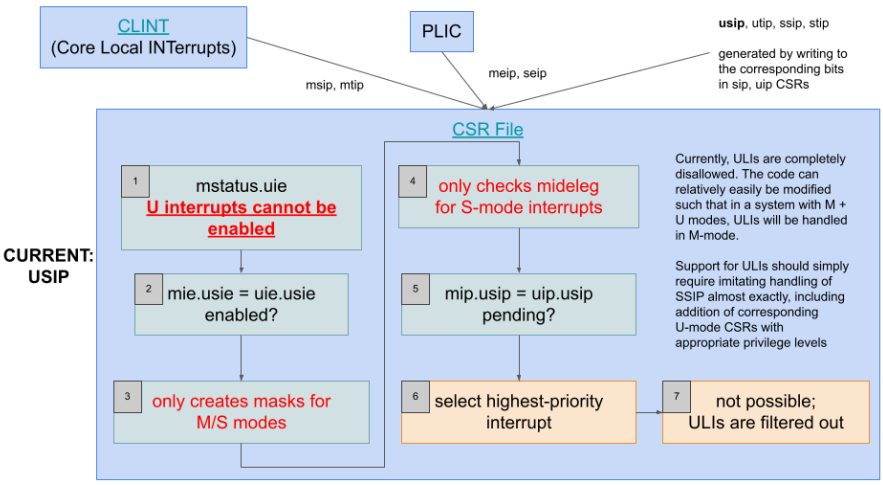

Figure 2: Current USIP processing on Rocket Chip

In contrast, Figure 2 illustrates the current situation; i.e., those parts that have been modified in the workflow.

1. Since user interrupts are not yet supported, `mstatus.uie` is in fact always set to 0. We instead wish to change this so that it is initialized to 1 when U-mode exists in the system.

   Note that changing only this step should be sufficient to allow USIPs to be enabled for handling in higher privilege modes, i.e. handled in M-mode if one is triggered while running in U-mode.

[1] It was nontrivial to disable S-mode, so the full cascading pathway with both mideleg and sideleg being used to delegate the USIP was implemented in the linked code.

3. We need to modify the existing interrupt masking to consider handling user-level interrupts when we are running in U-mode.

4. `mideleg` should be checked at the bit corresponding to USIP

7. The U-mode CSRs for interrupt handling must be added to the CSR file analogously to their S- and M-mode counterparts (uepc, utvec, etc.)

Note that although the RISC-V specification explicitly states that interrupts are distinct from, not a subset of, exceptions, the `exception` flag in CSR.scala IS raised when interrupts are triggered.

3 Navigating the Chipyard codebase

Working with Rocket Chip involves two repositories, rocket-chip and rocket-tools. Both repositories utilize Git submodules, notably riscv-gnu-toolchain and riscv-tests under riscv-tools. All modifications are committed to the `uli` branch of each repository, forked under the user-level-interrupts organization. All modifications to preexisting files should be demarcated by `// start ULI` and `// end ULI` comments indicating which lines were added or edited.

3.1 Adding a new CSR

Most code and logic relating to CSRs resides in the rocket-chip repository in `src/main/scala/rocket/CSR.scala`. For a simple readable/writable CSR, the basic behavior is defined in the `csr_wen` clause, which simply indicates a read/write operation involving a CSR. One can simply case on the specific CSR being selected, and set the value according the incoming `wdata`.

However, adding a new CSR or instruction also requires modifying a few other files to support translating between the mnemonic name and the actual binary encoding. First, a binary encoding for the CSR must be selected. Note that bits 8-9 denote the minimum access privilege level where 3 indicates (M)achine mode, 1 indicated (S)upervisor mode, and 0 indicates (U)ser mode.

In the rocket-tools repository under riscv-gnu-toolchain, the CSR encoding must be added to riscv-opc.h. After rebuilding the riscv-gnu-toolchain project, this will enable the CSR mnemonic to be used in assembly tests.

Under the rocket-chip repository, the CSR encoding must also be added to the source code in Instructions.scala. Note that both files say they were generated from parse-opcodes, as were several files sprinkled throughout the codebase. However, the files consist of different subsets of each other, so I found it less disruptive to modify them directly.

3.2 Tests

3.2.1 Running tests

The original description for running tests can be found in the rocket-chip README.

3.3 Adding a new instruction

New instructions, like CSRs, must have a bitpattern defined in the same locations in riscv-gnu-toolchain and Instructions.scala. Instructions must further have a decoding defined in IDecode.scala. If a new decode module is created, it must also be added to the `decode_table` in RocketCore.scala.

3.3.1 Adding a new test

Test source code is located in the rocket-tools repository under the `riscv-tests` directory. Since this project's modifications involve modifying the ISA, including privileged CSRs and instructions, test(s) for ULIs are added to `riscv-tests/isa/rv64mi`. Adding a new test involves:

1. Writing the test file.
   (a) Each test has the following structure:

```
#include ''riscv_test.h''
#include ''test_macros.h''

RVTEST_RV##X
RVTEST_CODE_BEGIN
  // test code
RVTEST_CODE_END

  .data
RVTEST_DATA_BEGIN
    TEST_DATA
RVTEST_DATA_END
```

   In the above code, `RVTEST_RV##X` is a standin for a choice of macro where the number is either 32 or 64, depending on the target hardware, and `X` can be one of M, S, U depending on which mode/privilege level the test needs.

   (b) Useful macros:

   - `TEST_PASSFAIL`
     Including this macro enables indicating passing and failing conditions by jumping to the `pass` and `fail` labels respectively; e.g.

     ```
     bne test, target, fail
     ```

```
j pass
```

   (c) Note that if a test fails or loops indefinitely, it may not produce or may delete the output file during cleanup. When actively developing a feature or test, it may be helpful to allow the tests to "pass" and set flags

2. Adding the test name to the `Makefrag` file in the same directory
3. Adding the test to the Rocket test suite
4. Re-building the riscv-tests project

   (a) This is one of the lines in `build.sh` in the rocket-tools repository. It is strongly recommended to re-build individual projects as necessary to save time.

3.4 Debugging

I found that viewing the .vcd waveforms was the most effective way for me to debug. I used gtkwave as a simple, easily-installed solution. Wires (values) in the CSR file module can be found in the hierarchy under TOP → TestHarness → ldut → tile_prci_domain_tile → core → csr.

Any new wires, muxes, etc. should automatically be compiled into wires visible in the hierarchy. However, if they do not show up, note that any values that never change (i.e. are always 0 or some other constant) may be optimized away.