Lab 8: RISC-V with Exceptions

Due: 11:59:59pm, Friday November 27, 2015

1 Introduction

In this lab you will add exceptions to a one-cycle RISC-V processor. With the support of exception, we will be able to do the following two things:

1. Implement `printInt, printChar, printStr` functions as system calls.
2. Emulate the unsupported multiply instruction (`mul`) in software exception handler.

We are using a one-cycle processor so you can focus on how exceptions work without including the complexities due to pipelining.

You have been given all the required programs for testing your processor. You only need to add hardware support to run exceptions. The following sections cover what has been changed in the processor and what you need to do.

2 CSRs

The `mkCsrFile` module in `src/includes/CsrFile.bsv` has been extended with new CSRs required for implementing exceptions.

Below are the summary of new CSRs in the `mkCsrFile` module. All these CSRs can be accessed by instructions such as `csrr, csrw, csrrw`. Therefore software can manipulate these CSRs.

`mstatus` The low 12 bits of this register store a 4-element stack of privilege/user mode (PRV) and interrupt enable (IE) bits. Each stack element is 3-bit wide. For example, `mstatus[2:0]` is stack element at the top of the stack, and contains the current PRV and IE bits. Specifically, `mstatus[0]` is the IE bit. If it is equal to 1, then interrupt is enabled. `mstatus[2:1]` contains the PRV bits. If the processor is in user mode, it should be set to `2'b00`. If the processor is in machine (privileged) mode, it should be set to `2'b11`. Other stack elements (i.e. `mstatus[5:3], ..., mstatus[11:9]`) have the same construction.

When an exception is taken, the stack will be pushed. Namely `mstauts[11:0]` is left-shifted by 3 bits, and the new PRV and IE bits (e.g. machine mode and interrupt disabled) are stored into `mstatus[2:0]`.

When we return from an exception using the `eret` instruction, the stack is popped. Namely `mstauts[11:0]` is right-shifted by 3 bits, and `mstatus[11:9]` is assigned to (user mode, interrupt enabled).

`mcause` It stores the cause of the most recent exception, and is updated when an exception happens. `ProcTypes.bsv` contains two cause values for the exceptions that we will implement in this lab:

- `excepUnsupport`: the cause value for unsupported instruction exception.
- `excepUserECall`: the cause value for system call exception.

`mepc` It stores the PC of the instruction that causes the exception, and is updated when an exception happens.

`mscratch` It stores the pointer to a "safe" data section that can be used to store all general purpose register (GPR) values when exception happens. This register is completely manipulated by software in this lab.

`mtvec` This is a read-only register, and it stores the start address of the exception handler program. The processor should set PC to `mtvec` when an exception happens.

The `mkCsrFile` module also incorporates additional interface methods, which are self-explained.

3 Decode Logic

The decoding logic has also been extended to support exceptions. The decoding of the following three new instructions are summarized below:

`eret` This instruction is used to return from exception handling. It is decoded to a new `iType` of `ERet` and everything else invalid and not taken.

`ecall` **(or** `scall`**)** This instruction is the system call instruction. It is decoded to a new `iType` of `ECall` and everything else invalid and not taken.

`csrrw rd, csr, rs1` This instruction writes the value of `csr` into `rd`, and writes the value of `rs1` into `csr`. Namely it performs `rd <= csr; csr <= rs1`. Both `rd` and `rs1` are GPRs, while `csr` is a CSR. This instruction replaces the `csrw` instruction we have used before, because `csrw` is just a special case of `csrrw`. This instruction is decoded to a new `iType` of `Csrrw`.

Since `csrrw` will write two registers, the `ExecInst` type in `ProcTypes.bsv` incorporates a new field "`Data csrData`", which contains the data to be written into `csr`.

The `eret` and `csrrw` instructions are allowed to happen in machine (privileged) mode. To detect the illegal use of such instructions in user mode, the `decode` function in `Decode.bsv` takes a second argument "`Bool inUserMode`". This argument should be set to `True` if the processor is in user mode. If the decode function detects the illegal use of `eret` and `csrrw` instructions in user mode, the `iType` of the instruction will be set to a new value `NoPermission`, and the processor will report error later.

4 Processor

We have provided most part of the processor code in `ExcepProc.bsv`, and you only need to fill out four places marked with the "`TODO`" comments:

1. Second argument for `decode` function.
2. Handle unsupported instruction exception: set `mepc` and `mcause`, push the new PRV and IE bits into the stack of `mstatus`, and chage PC to `mtvec`. You may want to use the `startExcep` method of `mkCsrFile`.
3. Handle system call exception: similar to handling unsupported instruction exception.
4. Handle `eret` instruction: pop the stack of `mstatus` and change PC to `mepc`. You may want to use the `eret` method of `mkCsrFile`.

5 Test Programs

The test programs can be grouped into three classes.

5.1 Old Programs

The first two classes are the assembly tests and benchmark programs we have used before. These programs will run in machine mode and will not trigger any exception. They can be compiled by going to `programs/assembly` and `programs/benchmarks` folders and running `make` command.

5.2 New Programs

The third class of programs are dealing with exceptions. These programs start in machine mode but go down to user mode immediately. All print functions are implemented as system calls, and the unsupported multiply instruction (`mul`) can be emulated in the software exception handler. The source codes for these programs also reside under the `programs/benchmarks` folder, but they are linked to libraries in the

`programs/benchmarks/excep_common` folder (instead of `programs/benchmarks/common`). To compile these programs, you can use the following commands:

```
$ cd programs/benchmarks
$ make -f Makefile.excep
```

The compilation results will appear in the `programs/build/excep` folder.

These programs not only include the original benchmarks we have seen before, but also include two new programs:

- `mul_inst`: This is an alternative version of the original `multiply` benchmark, which directly uses the `mul` instruction.
- `permission`: This program executes `csrrw` instruction in user mode, and should FAIL!

6 Implementing Exceptions

Exercise 1 (40 Points): Implement exceptions as described above on the processor in `ExcepProc.bsv`. You can build the processor using the `build -v excep` command under the `scemi/sim` folder. We have provided the following scripts to run the test programs in simulation:

- `run_asm.sh`: run assembly tests in machine mode without exception.
- `run_bmarks.sh`: run benchmarks in machine mode without exception.
- `run_excep.sh`: run benchmarks in user mode with exceptions.
- `run_permit.sh`: run the `permission` program in user mode.

Your processor should pass all the tests in the first three scripts (`run_asm.sh, run_bmarks.sh, run_excep.sh`), but should report error and terminate on the last script (`run_permit.sh`). Note that after you see the error message outputted by `bsim_dut` when running `run_permit.sh`, the software testbench `tb` is still running, so you need to hit `ctrl-c` to terminate it.

Discussion Question 1 (10 Points): In the spirit of the upcoming Thanksgiving holiday, list some reasons you are thankful you only have to do this lab on a one cycle processor. To get you started: what new hazards would exceptions introduce if you were working on a pipelined implementation?