Lab 8: SMIPS with Exceptions

Due: Friday November 21, 2014

1 Introduction

In this lab you will add exceptions to a one cycle SMIPS processor. We are using a one cycle processor so you can focus on how exceptions work without including the complexities due to pipelining.

The initial processor has been modified slightly from the previous lab that used the one cycle SMIPS processor. Hi and lo registers have been added along with supporting `mfhi`, `mflo`, `mthi`, and `mtlo` instructions. You have also been given all the required programs for testing your processor. You will need to add everything else required to run exceptions. The following sections cover what you will need to add to the processor.

2 Co-processor

The Co-processor module in `includes/Cop.bsv` needs to be extended to handle the new registers required for implementing exceptions. You will have to modify the module to include the additions below.

New registers:

- `Status` – Register 12 – Stores a 3 element stack of kernel/user mode and interrupt enable bits. The current kernel/user mode is stored in bit 0, and the current interrupt enable bit is stored in bit 1. Bits 2 and 3 are the previous kernel/user mode and interrupt enable, and bits 4 and 5 are the old kernel/user mode and interrupt enable. A kernel/user mode value of 0 corresponds to kernel mode, and a value of 1 corresponds to user mode.

  When an exception is taken, bits 4 and 5 get the previous value of bits 2 and 3; bits 2 and 3 get the previous value of bits 0 and 1; and bits 0 and 1 both get 0s.

  When you return from an exception, bits 0 and 1 get the previous value of bits 2 and 3; and bits 2 and 3 get the previous value of bits 4 and 5.

  This register can be written and read using `mtc0` and `mfc0` instructions. This register should be initialized to 0 when the processor starts.

- `Cause` – Register 13 – Stores the 5-bit cause of the most recent exception in bits 6 down to 2. The cause register is updated when an exception happens. This register can be read using the `mfc0` instruction. This register should be initialized to 0 when the processor starts.

- `EPC` – Register 14 – Stores the pc of instructions that cause exceptions. The EPC register is updated when an exception happens. This register can be written and read using `mtc0` and `mfc0` instructions.

New interface methods:

- `Action causeException(Addr current_pc, Bit#(5) cause)` – Called every time an exception happens. Sets EPC, cause, and status registers appropriately.

- `Action returnFromException` – Called every time `eret` is executed. Sets status register appropriately.

- `Addr getEPC` – Returns the current value of the EPC.

- `Bool isUserMode` – Returns true if the processor is currently in user mode according to the status register.

3 Decode Logic

You will have to add some decoding logic to support exceptions and SW emulated multiplication. All the decoding has been added for the `hi` and `lo` instructions, but you will need to add decoding for the instructions below:

- `eret` – opcode is `opRS`, rs value is `rsERET` – This instruction should decode to an iType of `ERet` and everything else invalid and not taken.
- `syscall` – opcode is `opFunc`, funct value is `fcSYSCALL` – This instruction should decode to an iType of `Syscall` and everything else invalid and not taken.
- `mult` and `multu` – opcode is `opFunc`, funct values are `fcMULT` and `fcMULTU` – This instruction should decode to an iType of `Unsupported` and everything else invalid and not taken.

There are some instructions that should only be allowed to happen in kernel mode. These instructions should decode as illegal instructions (iType = `Illegal`) when decoded in user mode. These instructions include `mfc0`, `mtc0`, and `eret` (they all have an opcode of `opRS`). You need to modify the arguments of the decode to pass in the current mode so you can decode these instructions properly depending on the current mode.

4 Processor

Your processor is going to need some top-level changes to support exceptions too. The following instructions need additional support:

- Unsupported instructions – Throw an RI (reserved instruction) exception with cause `excRI` – Redirect PC to exception handler at `excHandlerPC`.
- `syscall` – Throw a syscall exception with cause `excSys` – Redirect PC to exception handler at `excHandlerPC`.
- `eret` – Return from exception – Redirect PC to EPC.

In each of the above cases, you should not call `cop.wr` in order to avoid a potential conflict inside the module. This will also cause your instruction count to not get incremented for these instructions so your instruction count will no longer match your cycle count in the benchmarks.

5 Example Programs

There are new programs to fully test your implementation of exceptions on the SMIPS processor. The scripts to run these programs are described below:

- `run_assembly` – Runs all the `*.asm.vmh` programs in kernel mode. Uses `mtc0` and `mfc0` directly.
- `run_benchmarks` – Runs all the `*.bench.vmh` programs (except the `multiply_inst` benchmark) in kernel mode. Uses `mtc0` and `mfc0` directly.
- `run_ku_benchmarks` – Runs all of the `*.ku_bench.vmh` programs (except `multiply_inst` and `kfunc`). They start in kernel mode and then switch to user mode when jumping to the start of main. Uses `syscall` instructions instead of using `mtc0` and `mfc0` directly.
- `run_ku_kfunc` – Runs `kfunc.ku_bench.vmh`. This program tries to use a kernel function in user mode. It should exit after executing an illegal instruction.
- `run_mult_inst` – Runs `multiply_inst.bench.vmh` and `multiply_inst.ku_bench.vmh`. These programs run the same test as `multiply_funct` except the SMIPS `mult` instruction is used.

6 Implementing Exceptions

Exercise 1 (40 Points): Implement exceptions as described above on the processor in `ExcProc.bsv`. Test your implementation using the scripts described in the example programs section. All the example programs should pass except `kfunc.ku_bench.vmh` should quit due to an illegal instruction error since it tries to run a kernel mode instruction in user mode.

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?