Lab Sheet 8 for **CS F342 Computer Architecture**

Semester 1 – 2019-20

Goals for the Lab:

- To know exception mechanism in MIPS
- Be able to write a simple exception handler for MIPS machine

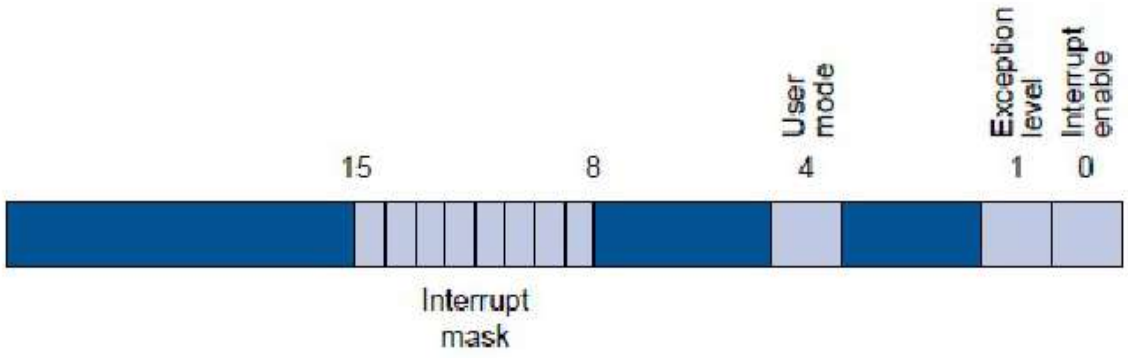
Background:

The relevant registers for the exception handling, in coprocessor 0

Register Name	Usage
BadVAddr	(Bad Virtual Address) Memory address where exception occurred
Status	Interrupt mask, enable bits, and status when exception occurred
Cause	Type of exception and pending interrupt bits
EPC	Address of instruction that caused exception

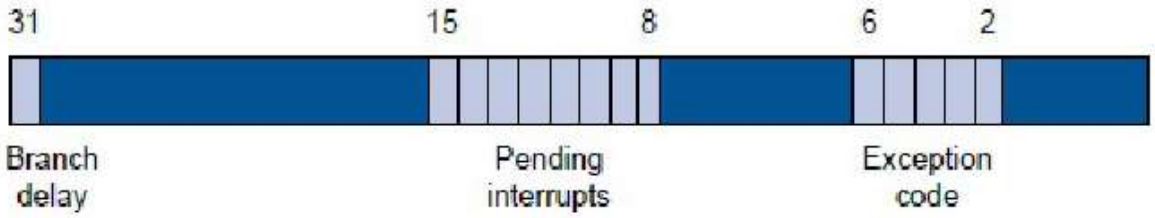
Question: Why do we need EPC? Why can't we use RA?

The Status Register:



- The user mode bit is 0 if the processor is running in kernel mode and 1 if it is running in user mode.
- The exception level bit is normally 0, but is set to 1 after an exception occurs. When this bit is 1, interrupts are disabled and the EPC is not updated if another exception occurs. This bit prevents an exception handler from being disturbed by an interrupt or exception, but it should be reset when the handler finishes.
- If the interrupt enable bit is 1, interrupts are allowed. If it is 0, they are disabled.

The Cause Register:



- The branch delay bit is 1 if the last exception occurred is an instruction executed in the delay slot of a branch.
- The interrupt pending bits become 1 when an interrupt is raised at a given hardware or software level.
- The exception code register describes the cause of an exception through the following codes:

Code	Name	Description
0	INT	Interrupt
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	`syscall` instruction executed
9	BKPT	`break` instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

a. Codes from 1 to 3 are reserved for virtual memory, (TLB exceptions), 11 is used to indicate that a particular coprocessor is missing, and codes above 12 are used for floating point exceptions or are reserved.

Exercise 1:

1. Download the file containing code for causing exceptions at https://sourceforge.net/p/spimsimulator/code/HEAD/tree/Tests/tt.core.s
2. Load it in QtSpim, ignore any compilation warnings, Click on Run
3. Whenever exception pop-up appears, observe the values of registers used for exception handling and console message (also understand the corresponding piece of code used for causing that exception), then click 'OK'.

Exercise 2: Set branch delay bit as '1' by causing an exception in the delay slot of branch.

Note: Simulator => Settings => MIPS tab => check 'Enable delayed Branches' then run the code

Exercise 3: Study the code below to understand exception handling mechanism in MIPS (Non-evaluative)

Exceptions and interrupts cause a MIPS processor to jump to a piece of code, at address 80000180hex(in the kernel, not user address space), called an exception handler. This code examines the exception's cause and jumps to an appropriate point in the operating system which then responds to it accordingly.

The code in the example below is a simple exception handler, which invokes a routine to print a message at each exception (but not interrupts). This code is similar to the exception handler (exceptions.s) used by the SPIM simulator.

```
.ktext 0x80000180
#The exception handler cannot store the old values from $at,$a0,$a1) registers on the stack, as would an
#ordinary routine, because the cause of the exception might have been a memory reference that used a
#bad value (such as 0) in the stack pointer.

    mov $k1, $at        # Save $at register. $k1,$k2 are exception handler register

    sw $a0, save0       # Handler is not re-entrant(i.e interrupts are disabled) and can't use
    sw $a1, save1       # stack to save $a0, $a1
                        # Don't need to save $k0/$k1

#the exception handler uses the value from the Cause register to test if the exception was caused by an
#interrupt (see the preceding table). If so, the exception is ignored. If the exception was not an interrupt, #the
#handler calls print_excp to print a message.


    mfc0 $k0, $13       # Move Cause into $k0, $13 is CAUSE register
    srl $a0, $k0, 2     # Extract ExcCode field(bits 2-6 in cause register)
```

```
        andi $a0, $a0, 0xf
        bgtz $a0, done          # Branch if ExcCode is int (0)
        mov $a0, $k0            # Move Cause into $a0
        mfc0 $a1, $14           # Move EPC into $a1
        jal print_excp          # Print exception error message

done: mfc0 $k0, $14     # Bump EPC
        addiu $k0, $k0, 4       # Do not re-execute  faulting instruction
        mtc0 $k0, $14           # EPC
        mtc0 $0, $13            # Clear Cause register
        mfc0 $k0, $12           # Fix Status register
        andi $k0, 0xfffd    # Clear EXL bit which allows subsequent exceptions to change the EPC register


        ori $k0, 0x1            # Enable interrupts
        mtc0 $k0, $12
        lw $a0, save0           # Restore registers
        lw $a1, save1
        mov $at, $k1
        eret                # exception return, Return to EPC
        .kdata
save0:  .word 0
save1:  .word 0
```

Exercise 4: Identify the cause of exception from the following values of CAUSE register

1. `0x24`
2. `0x30`
3. `0x1c`

References:

- http://www.cs.iit.edu/~virgil/cs470/Labs/Lab7.pdf
- Appendix A from Patterson and Hennessey, Computer Organization and Design: The Hardware/Software Interface