EE 109 Unit 12 - Exceptions
What are Exceptions?

• Any event that causes a break in normal execution
  – Error Conditions
    • Invalid address, Arithmetic/FP overflow/error
  – Hardware Interrupts / Events
    • Handling a keyboard press, mouse moving, USB data transfer, etc.
    • We already know about these so we won't focus on these again
  – System Calls / Traps
    • User applications calling OS code

• General idea: When these occur, automatically call some function/subroutine to handle the issue, then resume normal processing
  – This is what we've done with interrupts: call an ISR routine
Error Condition Exceptions

- Here are some representative examples of error conditions the hardware automatically checks for and then generates an exception:
  - **Bus Error**
    - No memory or I/O device responds to a read or write
  - **Address Error**
    - Unaligned address used for half or word accesses
    - Access to Kernel memory space when in User mode
  - **Floating Point Exceptions**
    - Many possible exceptions encompassed by this
  - **Integer Overflow**
    - An instruction that causes 2’s complement overflow
  - **TRAP/ Syscalls Instructions**
    - SW definable error conditions
  - **Reserved Instruction**
    - User code attempting to perform kernel mode operations
System Calls / TRAP Exceptions

• Provide a controlled method for user mode applications to call kernel mode (OS) code

• Syscall’s and traps are very similar to subroutine calls but they switch into “kernel” mode when called
  – Kernel mode is a special mode of the processor for executing trusted (OS) code
  – Certain features/privileges are only allowed to code running in kernel mode

• User applications are designed to run in user mode
• OS and other system software should run in kernel mode
• User vs. kernel mode determined by some bit in some register
MIPS Kernel Mode Privileges

• Privileged instructions
  – User apps. shouldn’t be allowed to disable/enable interrupts, change memory mappings, etc.

• Privileged Memory or I/O access
  – Processor supports special areas of memory or I/O space that can only be accessed from kernel mode

• Separate stacks and register sets
  – MIPS processors can use “shadow” register sets (alternate GPR’s when in kernel mode).
Syscalls

• Provided a structured entry point to the OS
  – Really just a subroutine call that also switches into kernel mode
  – Often used to allow user apps. to request I/O or other services from the OS

• MIPS Syntax: syscall
  – Necessary arguments are defined by the OS and expected to be placed in certain registers
Exception Processing

• Now that you know what causes exceptions, what does the hardware do when an exception occurs?
• Save necessary state to be able to restart the process
  – Save PC of current/offending instruction
• Call an appropriate “handler” routine to deal with the error / interrupt / syscall
  – Handler identifies cause of exception and handles it
  – May need to save more state
• Restore state and return to offending application (or kill it if recovery is impossible)
Problem of Calling a Handler

• Processor will automatically call a routine known as an “exception handler” to handle the exception

• Problems with calling exception handlers
  – How does the HW ‘automatically’ call handler routine since we don’t know where we will be in our code when an exception occurs
  – Calling the exception handler can cause changes in state (registers) when we return to the running application

```
.text
MAIN:
    jr $ra
```

Many instructions could cause an error condition. Or a hardware event like a keyboard press could occur at any point in the code.
Solution for Calling a Handler

• Since we don’t know when an exception will occur there must be a preset location where an exception handler should be defined or some way of telling the processor in advance where our exception handlers will be located

• Method 1: Single hardwired address for master handler
  – Early MIPS architecture defines that the exception handler should be located at 0x8000_0180. Code there should then examine CAUSE register and then call appropriate handler routine

• Method 2: Vector tables
  – Table in memory holding start address of exception handlers (i.e. overflow exception handler pointer at 0x0004, FP exception handler pointer at 0x0008, etc.)
  – This is what gets filled in when you write ISR(ADC_vect) in your Arduino code
Handler Calling Methods

Method 1
- Kernel Space: 0xffffffff
- Exception Handler: 0x80000180
- User Space: 0x00000000

Method 2
- Kernel Space: 0xffffffff
- INT 2 Hand.: 0x80007000
- Addr. Err. Handler: 0x80000400
- INT 1 Hand.: 0x80000200
- User Space: 0x00000000

Addresses:
- 0x80000200
- 0x80000400
- 0x800007000
Problem of Returning

• When an exception occurs and we call a handler, where should we save the return address?

```
.text
F1:  addi $sp,$sp,-4
     sw   $ra,0($sp)
     jr    $ra
```

What if an exception occurs at this point in time? We’d want to call an exception handler and thus store a return address to come back to this location after processing the exception? Can we store that return address in $ra?

No!! We’d overwrite the $ra that hadn’t been saved to the stack yet.
EPC Register

• Exception PC holds the address of the offending instruction
  – EPC is the return address after exception is done
  – Since we don't know if the user has saved the $ra yet, we can't put our return address there...

• 'eret' instruction used to return from exception handler and back to execution point in original code (unless handling the error means having the OS kill the process)
  – 'eret' Operation: PC = EPC

\[
\text{EPC} = \begin{array}{c}
31 \\
\text{Exception PC}
\end{array}
\]

Address of instruction that generated the exception
Sample Exception Handler

- Main handler needs to examine cause register and call a more specific handler
- Handlers must end with 'eret' rather than 'jr $ra'

```
.text
L1:   li   $t0,0x100A1233
     lw   $s0,0($t0)
0x400: ---
```

EPC = PC = 0x400 (Save return address)

```
0x8000_0180:
  mfc0  $k0,C0_Status
  mfc0  $k1,C0_Cause
  srl   $t2,$k1,2
  andi  $t4,$k1,0x001f
  bne   $t4,0,E1
  j     INT_HAND
E1:  ...
E4:  bne  $t4,4,E2
     j    ADDR_HAND
...
ADDR_HAND:
  ...  
eret
```

Invalid Address Exception

Main handler can determine cause and
Problem of Changed State

- Since exceptions can occur at any time, the handler must save all registers it uses.
- Exception is $k0, $k1 (kernel registers) which the OS can assume are free to be used in a handler.

```
.text
L1:    add $t0,$t1,$s0
       slt $t2,$s2,$t3
       add $t4,$t5,$t7
       beq $t4,$t5,L2
       ----

L2:    ...

OV_HAND:
       ...
       srl $t2,$k1,2
       andi $t4,$0x001f
       ...
       eret
```

We don't know if the interrupted app. was using $t2, $t4, etc...We should save them on the stack first.

Handlers need to save/restore values to stack to avoid overwriting needed register values (e.g. $t2, $t4)
Problem of Changed State

• Since exceptions can occur at any time, the handler must save all registers it uses
• Exception is $k0, $k1 (kernel registers) which the OS can assume are free to be used in a handler

```
.text
L1:
    add $t0,$t1,$s0
    slt $t2,$s2,$t3
    add $t4,$t5,$t7
    beq $t4,$t5,L2
    ----
L2:
OV_HAND:
    addi $sp,$sp,-8
    sw $t2, 4($sp)
    sw $t4, 0($sp)
    srl $t2,$k1,2
    andi $t4,$r0x001f
    add $k0, $t4, $t2
    lw $t4, 0($sp)
    lw $t2, 4($sp)
    addi $sp, $sp, 8
    eret
```

Handlers need to save/restore values to stack to avoid overwriting needed register values (e.g. $t2, $t4)
Summary

• Exceptions are usually "asynchronous" events
  – They occur in the middle of normal instruction processing

• Exceptions should trigger some other subroutine to automatically be called and executed
  – Requires placing the subroutine/function code at some particular address or setting up a table with the addresses of the functions to handle each exception type

• At the assembly level, exceptions handler routines MUST save all the registers onto the stack before they overwrite them since they have no idea which registers were in use when the exception was triggered