18-447 Lecture 11: Interrupt and Exception

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – understand the simplicity of interrupt mechanisms in HW and appreciate its powerful uses by SW
  – first peek outside of the “user-level” abstraction

• Notices
  – Lab 2, status check this week, due next week
  – HW 3, due Mon 2/28
  – Midterm 1, Wed 3/2, covers up to Lec 10

• Readings
  – P&H Ch 4
Interrupt Control Transfer

- **Basic Part:** an “unplanned” fn call to a “third-party” routine; and later return control back to point of interruption
  - must be 100% transparent
  - not enough to impose all callee-save convention (*return address??*)

- **Tricky Part:** interrupted thread cannot anticipate/prepare for this control transfer

- **Puzzling Part:** why is there a hidden routine running invisibly?
Use #1: Interrupts

• How to handle rare events with unpredictable arrival time and must be acted upon quickly?
  E.g., keystroke, in-bound network, disk I/O

• **Option 1:** write every program with periodic calls to a service routine (i.e., polling)
  – polling frequency affects worst-case response time
  – expensive for rare events needing fast response
  
  What if a programmer does it wrong or forgets?

• **Option 2:** normal programs blissfully unaware
  – event triggers an interrupt on-demand
  – forcefully and transparently transfer control to the service routine and back
Use #2: Exceptions

• How to handle rare exceptional conditions in a program itself, e.g., arithmetic overflow, divide-by-0, page fault, TLB miss, etc.)

• **Option 1:** write program with explicit checks at every potential site
  - do you want to check overflow after every ADD?
  - after every instruction and data memory access?
  What if a programmer does it wrong or forgets?

• **Option 2:** write program for common case
  - detect exceptional conditions in HW
  - transparently transfer control to an exception handler that knows how to fix things up
Use #3: Multitasking Preemption

• Many programs time-multiplex a processor

• **Option 1:** write programs to voluntarily give up the processor after running for a while

  What if a programmer does it wrong or forgets?

• **Option 2:** normal programs blissfully unaware
  – a timer interrupts process A
  – handler returns to an earlier interrupted process B
  – a timer interrupts process B
  – handler returns to process A
  – Neither A nor B aware anything funny happened!!

Really just a clever use of #1
Terminology: Interrupt vs Exception

- **Interrupt** is the more general concept
- **Synchronous** interrupt (a.k.a “exception”)
  - exceptional conditions tied to a particular instruction
  - a faulting instruction cannot be finished
  - must be handled immediately
- **Asynchronous** interrupt (a.k.a. “interrupt”)
  - events not tied to instruction execution
  - some flexibility on when to handle it
  - cannot postpone forever

- **System Call**
  - an instruction to trigger exception on purpose
  
  *If intentional, why not just called the handler with JAL?*
Use #4: Privileged Systems
User-Level Abstraction:

- **Protected:** a “user-level” process thinks it is alone
  - private set of user-level architectural states
  - cannot see or manipulate (directly) state outside of abstraction

- **Virtualized:** UNIX user process sees a file system
  - corresponds to storage and non-storage devices
  - all devices look like files; accessed through a common set of interface paradigms

- OS+HW support and enforce this abstraction
  - enforce protection boundaries
  - bridge between abstract and physical

OS must live beyond user-level abstractions and be more “powerful”
Privilege Levels

- A level is a set of architectural state and instructions to manipulate them
- A more privileged level is a superset (usually) of the less privileged level
  - lowest level has basic compute state and insts
  - higher level has state and insts to control virtualization and protection of lower levels
  - only highest-level sees “bare-metal” hardware

user level

kernel level

“hypervisor” level for virtualizing multiple OSs
Interrupt and Privilege Change

- Combine privilege level change with interrupt/exception transfer
  - switch to next higher privilege level on interrupt
  - privilege level restored on return from interrupt
- Interrupt control transfer is only gateway to privileged mode
  - lower-level code can never escape into privileged mode
  - lower-level code don’t even need to know there is a privileged mode
MIPS Interrupt Architecture
MIPS Interrupt Architecture

• On interrupt transfer, HW saves interrupted address (to return to) to a special **EPC** register
  – can’t just leave in PC: overwritten immediately
  – can’t use “x1”: need to preserve user value

• In general, HW saves state that cannot be handler callee-saved/restored (very few such things)

• For example, GPR can be preserved by interrupt handler using callee-saved convention
  – MIPS convention reserves r26 and r27 for the interrupt handler to allow for very short handlers
  – more elaborate handler can save to memory
MIPS Interrupt Architecture

- Privileged system control registers; loaded automatically on interrupt transfer events
  - **EPC** (CR14): exception program counter, which
    instruction location to go back to
  - **Cause** (CR 13): what caused the interrupt

- **Status** (CR 12): enable/disable interrupts, set
  privilege modes

- Accessed by “move from/to co-processor” instructions

Figures from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Where to go on an interrupt?

• **Option 1:** control transfers to default handler
  – default handler examines CR12 & CR13 to select specialized handler

• **Option 2:** vectored interrupt
  – separate specialized handler addresses registered with hardware
  – hardware transfer control directly to appropriate handler to reduce interrupt processing time

Note: handler in address region/space protected from user so user can’t just branch to it
unprivileged user also can’t imitate handler code
## Examples of Causes in MIPS

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Cause of exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>interrupt (hardware)</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>address error exception (load or instruction fetch)</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td>address error exception (store)</td>
</tr>
<tr>
<td>6</td>
<td>IBE</td>
<td>bus error on instruction fetch</td>
</tr>
<tr>
<td>7</td>
<td>DBE</td>
<td>bus error on data load or store</td>
</tr>
<tr>
<td>8</td>
<td>Sys</td>
<td>syscall exception</td>
</tr>
<tr>
<td>9</td>
<td>Bp</td>
<td>breakpoint exception</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
<td>reserved instruction exception</td>
</tr>
<tr>
<td>11</td>
<td>CpU</td>
<td>coprocessor unimplemented</td>
</tr>
<tr>
<td>12</td>
<td>Ov</td>
<td>arithmetic overflow exception</td>
</tr>
<tr>
<td>13</td>
<td>Tr</td>
<td>trap</td>
</tr>
<tr>
<td>15</td>
<td>FPE</td>
<td>floating point</td>
</tr>
</tbody>
</table>
Handler Examples

• On asynchronous interrupt, device-specific handler invoked to service the device

• On exception, kernel handler either
  – correct faulting condition and continue the program (e.g., emulate missing FP functionality, update virtual memory management), or
  – “signal” back to user process if a user-level handler function is registered, or
  – kill the process if exception cannot be corrected

• “System call” is a special kind of fxn call from user process to kernel-level service routines (e.g., open, close, read, write, seek on “files”)
Returning from Interrupt

• Adjust EPC depending on situation
  – return to faulting EPC to retry the instruction
  – return to faulting EPC+4 to skip (e.g., if emulated)
  – return to somewhere entirely different . . . .

• Undo what happened on the way in
  – handler restores callee-saved state
  – HW to undo the rest

• MIPS32 uses ERET to ***atomically***
  – restore HW-saved processor states
  – restore privilege level
  – jump to address in EPC
An Extremely Short Handler

_handler_shortest:
# no prologue needed

... short handler body ... # can use only r26 and r27;
# must get the job done before
# anything else happens

# epilogue
eret # restore privilege and jump to EPC
A Short Handler

```assembly
reserved:
   .space 4
_handler_short:
   # prologue
   la r26, _reserved:  # point to reserved space
   sw r8, 0x0(r26)    # back-up r8 for use in body

   ... short handler body ...  # can use r26, r27, and r8
   # must get the job done before
   # anything else happens

   # epilogue
   la r26, _reserved:  # point to reserved space
   lw r8, 0x0(r26)    # restore r8
   eret               # restore privilege and jump to EPC
```

What happens to EPC if exception or interrupt in handler?
Nesting Interrupts

• On an interrupt transfer, further asynchronous interrupts are disabled (by HW)
  – if not, another interrupt would overwrite EPC/Cause/Status
  – similarly, handler must not generate exceptions itself until prepared

• For long-running handlers, interrupt must be re-enabled
  – handler examines or save EPC/Cause/Status to memory before re-enabling interrupt
  – once re-enabled, handler cannot rely on EPC/Cause/Status register contents anymore
Interrupt Priority

- Interrupt sources assigned to priority levels
  - higher-priority means more time critical
  - if multiple interrupts triggered, should handle highest-priority interrupt first

- Different priority interrupts can be selectively disabled by interrupt mask in Status

- When servicing an interrupt, re-enables only higher-priority interrupts
  - ensure higher-priority interrupts not delayed
  - re-enabling same/lower-priority interrupts could cause infinite loop
Nestable Handler (not re-entrant)

_handler_nest:
    # prologue
    la r27, _reserved
    mfc0 r26, epc
    sw r26, 0x0(r27)
    sw r8, 0x4(r27)
    addi r26, r0, 0x405
    mtc0 r26, status

    ... interruptible
    longer handler body ...

    addi r26, r0, 0x404
    mtc0 r26, status
    la r27, _reserved
    ld r8, 0x4(r27)
    ld r26, 0x0(r27)
    mtc0 r26, epc
    eret

    # point to reserved
    # get EPC contents
    # backup EPC value
    # backup r8 for use later
    # set interrupt enable bit
    # write into status reg

    # could free-up more registers
    # if needed (cannot use r26 and r27)

    # clear interrupt enable bit
    # write into status reg
    # point to reserved
    # restore r8
    # get saved EPC value
    # restore EPC contents
    # restore privilege and jump to EPC

r26 can change spontaneously while interrupt enabled
Implementing Interrupt in a Pipeline
Even with overlapped execution, interrupt must appear (to the handler) to have taken place in between two instructions

- older instructions finished completely
- younger instructions as if never happened
### “Flushing” a Pipeline

<table>
<thead>
<tr>
<th></th>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
<th>t₄</th>
<th>t₅</th>
<th>t₆</th>
<th>t₇</th>
<th>t₈</th>
<th>t₉</th>
<th>t₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>bub</td>
<td>bub</td>
<td>Iₕ</td>
<td>Iₕ+1</td>
<td>Iₕ+2</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>bub</td>
<td>bub</td>
<td>Iₕ</td>
<td>Iₕ+1</td>
<td>Iₕ+2</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td>I₄</td>
<td>bub</td>
<td>bub</td>
<td>Iₕ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>I₀</td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td></td>
<td>bub</td>
<td>bub</td>
<td>Iₕ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Kill faulting and younger inst; drain older inst**
- **Don’t start handler until faulting inst. is oldest**
- **Better yet, don’t start handler until pipeline is empty**

Better to be safe than to be fast
Exception Sources in Different Stages

- **IF:** I-mem address/protection fault
- **ID:**
  - illegal opcode
  - trap to SW emulation of unimplemented instructions
  - syscall instruction (a SW requested exception)
- **EX:** invalid results: overflow, divide by zero, etc.
- **MEM:** D-mem address/protection fault
- **WB:** nothing can stop an instruction now...

Okay to associate async interrupts (I/O) with any instruction/stage we like
Pipeline Flush for Exceptions

Where is “the current instruction”?

Figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]