Computer Systems Architecture
http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 12: Interrupts, Exceptions and I/O
Memory-Mapped Input and Output

- How does the CPU communicate with external devices?
  - Within SPIM: console and keyboard
  - Hard disk, sound card, mouse &c.
- Some MIPS addresses are actually *device registers*
  - Certain memory ranges are *mapped* to external devices
  - Writing *sends* data to external device
  - Can *poll* device registers to check for events
  - Later we learn to use *interrupts*…
- Other processors use a separate I/O address space
  - e.g. x86 has special instructions for I/O ports
  - Most sane processors use MMIO
Console I/O in SPIM

- SPIM emulates MMIO
- Must select ‘Mapped I/O’ in PCSpim settings
- Receiver – keyboard
- Transmitter – display
- Can still use syscalls
- Appendix A.8, pp 38–40
**Keyboard Input**

- Receiver control (at `FFFF0000_{16}`) consists of 2 bits
  - Bit 0 is the ‘ready’ bit – unread input from keyboard
  - Ignore the ‘interrupt enable’ bit for now
- Receiver data (at `FFFF0004_{16}`) contains last pressed key
  - Reading data register resets ‘ready’ bit

```assembly
li $t0, 0xffff0000
rd_poll:
    lw $v0, 0($t0)
    andi $v0, $v0, 0x01
    beq $v0, $zero, rd_poll
    lw $v0, 4($t0)
    # last key code in $v0
```

- Read from control register
- AND off the ‘ready’ bit
- If zero, no keypress
  - Keep waiting
- Else read data register
Character Display

- Transmitter control (at FFFF0008_{16}) consists of 2 bits
  - Bit 0 is the ‘ready’ bit – display accepting new character
  - Ignore the ‘interrupt enable’ bit for now
- Transmitter data (at FFFF000C_{16}) takes ASCII character
  - Writing data register resets ‘ready’ bit

```assembly
# character in $a0
li  $t0, 0xfffff0008
wr_poll:
    lw  $v0, 0($t0)
   andi $v0, $v0, 0x01
    beq  $v0, $zero, wr_poll
    sw  $a0, 4($t0)
    # Read from control register
    # AND off the ‘ready’ bit
    # If zero, not ready
    #   Keep waiting
    # Else send to data register
```
Exceptions

- Exceptions are caused by program execution:
  - Integer arithmetic overflow – result outside $[-2^{31}, +2^{31})$
  - Division by zero
  - Illegal instruction – machine code doesn’t make sense
  - Address error – access to unaligned address
  - Bus error – access to non-existent address
  - syscall instruction – when we need an OS request
  - brk (breakpoint) instruction – used by debuggers

- Control hands over to operating system code
  - Different mechanism to Java exceptions

- Can be thought of as a *synchronous software interrupt*...
Interrupts arise from external devices:

- Disk read complete (via DMA, say)
- Memory error (but only with EEC RAM)
- User pressed a key on console
- Console ready to display another character
- Hardware timer expired

Can be thought of as *asynchronous hardware exception*
- Asynchronous because it can occur any time

Terminology varies on different architectures...
Examples of Exceptions

# divide by zero -- breakpoint
div $t0, $t0, $zero

# arithmetic overflow
li $t1, 0x7fffffff
addi $t1, $t1, 1

# non-existent memory address -- bus error
sw $t2, 124($zero)

# non-aligned address -- address error
sw $t2, 125($zero)

# invalid instruction
.word 0xdeadbeef
Exception Handlers

- Hardware saves current PC in a special register
- CPU jumps to an exception handler, which
  - Saves the current processor state
  - Take appropriate internal action for an exception, or
  - Deal with the external source of the interrupt
  - Restores the previous processor state
  - Resumes user program execution, if possible
- Exception handler resides in kernel memory, inside the OS
  - Kernel has extra privileges and protection over user code
  - However this is not simulated in SPIM...
MIPS Coprocessor 0 Registers

- MIPS uses coprocessor 0 for exception/interrupt handling
- SPIM simulates the following coprocessor 0 registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadVAddr</td>
<td>$8</td>
<td>offending memory reference</td>
</tr>
<tr>
<td>Count</td>
<td>$9</td>
<td>current timer; incremented every 10ms</td>
</tr>
<tr>
<td>Compare</td>
<td>$11</td>
<td>interrupt when Count $\equiv$ Compare</td>
</tr>
<tr>
<td>Status</td>
<td>$12</td>
<td>controls which interrupts are enabled</td>
</tr>
<tr>
<td>Cause</td>
<td>$13</td>
<td>exception type, and pending interrupts</td>
</tr>
<tr>
<td>EPC</td>
<td>$14</td>
<td>PC where exception/interrupt occurred</td>
</tr>
</tbody>
</table>

- Count and Compare implements a hardware timer
- Could implement a pre-emptive multitasking microkernel, or program threads – exercise for the keen reader.
Interrupt / Exception Status

**Interrupt mask** which of the 8 interrupts are allowed

**User mode** not simulated by SPIM; always 1

**Exception level** automatically set during an exception; prevents handler itself being interrupted

**Interrupt enable** global interrupt enable (or disable)

- Eight interrupt bits: 6 hardware, 2 software levels
  - Console receiver: HW level 1; transmitter: HW level 0
  - Timer: HW level 5
Interrupt / Exception Cause

- **Pending interrupts** is a bitfield of the 8 interrupts
- **Exception code** is a 5-bit integer field

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>Hardware interrupt pending</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>Address error on load (or instruction fetch)</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td>Address error on 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 (but not on SPIM!)</td>
</tr>
<tr>
<td>9</td>
<td>Bp</td>
<td>Breakpoint (usually used by debuggers)</td>
</tr>
<tr>
<td>12</td>
<td>Ov</td>
<td>Arithmetic overflow</td>
</tr>
</tbody>
</table>
Coproprocessor 0 Interface

\begin{itemize}
  \item mfc0 \( dst, esrc \) – move from coprocessor 0
    \begin{itemize}
      \item \( dst := esrc \)
    \end{itemize}
  \item mtc0 \( dst, esrc \) – move to coprocessor 0
    \begin{itemize}
      \item \( esrc := dst \)
    \end{itemize}
\end{itemize}

Example: Allow all hardware interrupts

\begin{verbatim}
mfc0 $t0, $12
ori $t0, $t0, 0xff01
mtc0 $t0, $12
\end{verbatim}

Example: Set a timer interrupt in 420ms

\begin{verbatim}
li $t0, 42
mtc0 $t0, $11
mtc0 $zero, $9
\end{verbatim}
Writing Your Own Interrupt Handler

- Handler always at address 80000180\textsubscript{16} in \textit{kernel memory}
  - Use the \texttt{.ktext 0x80000180} and \texttt{.kdata} directives
- Must save and later restore \textit{all} registers used
  - Including $at – use \texttt{.set noat} to supress SPIM’s errors
  - Except $k0 and $k1 may be used freely
  - Should not use stack – may point to invalid memory
  - Can temporarily spill registers to \texttt{.kdata}
- Return control to the user program with \texttt{eret}
  - Jumps to EPC, and resets \textit{Exception level} in Status
- Example: 112-timer.asm on course website
Polling I/O

- Previously we described memory-mapped I/O
  - **Polling** device registers for activity
  - e.g. *receiver control* changes when user presses a key
  - See `112-echo.asm` on course website for a full example
  - Ensure ‘Mapped I/O’ activated in PCSpim settings

- Certain memory ‘locations’ are actually device registers
  - Reading (or polling) receives, and writing sends data
  - Each access transfers a 32-bit word to/from the device

- Transferring large blocks of data from a device
  - Wait until ready, reads a word, write to memory. Repeat
  - Data flow: Device → Processor → Memory
  - Not computationally hard, yet very processor intensive
Interrupts for I/O and DMA

- Instead, can use interrupts – an asynchronous approach
- Needn’t constantly poll for multiple types of events
- Hardware will raise interrupt on an external event
- Or even combine both: 112-async.asm
- Use Direct Memory Access for larger blocks of data
  - DMA controller coordinates device ↔ memory
  - Data flow: Device → Memory
  - Processor available for other jobs in the meantime
Reading Material

- Exception Handlers
  - H&P Appendix §A.7 and §A.8
  - H&P §5.6 is pretty useless...

- I/O, Interrupts and DMA
  - H&P §8.1 and §8.5 are detailed and informative
    (But probably too much information)