Interrupts

Stopping program flow to execute a special piece of code that handles an event

- Definition and classification
- PowerPC interrupt structure
- Precise exceptions
- Handling multiple interrupts
  Nesting, prioritization

Reference: Chapter 6, Programming Environment Section 6.3.3, User’s Manual

I/O Data Transfer

Two key questions to determine how data is transferred to/from a non-trivial I/O device:

1. How does the CPU know when data is available?
   - (a) Polling
   - (b) Interrupts

2. How is data transferred into and out of the device?
   - (a) Programmed I/O
   - (b) Direct Memory Access (DMA)

Interrupts

Interrupt (a.k.a. exception or trap): An event that causes the CPU to stop executing the current program and begin executing a special piece of code called an **interrupt handler** or **interrupt service routine** (ISR). Typically, the ISR does some work and then resumes the interrupted program.

Interrupts are really glorified procedure calls, except that they:

- **can occur between any two instructions**
- are transparent to the running program (usually)
- are not explicitly requested by the program (typically)
- call a procedure at an address determined by the type of interrupt, not the program

Two basic types of interrupts (1/2)

- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
  - Names:
    - Trap, exception, **synchronous interrupt**
    - “unschedulable”
Two basic types of interrupts (2/2)

- Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error
- Names:
  - interrupt, external interrupt, *asynchronous interrupt*
  - “schedulable”

How it works

- Something tells the processor core there is an interrupt
- Core transfers control to code that needs to be executed
- Said code “returns” to old program
- Much harder then it looks.
  - Why?

… is in the details

- How do you figure out where to branch to?
- How to you ensure that you can get back to where you started?
- Don’t we have a pipeline? What about partially executed instructions?
- What if we get an interrupt while we are processing our interrupt?
- What if we are in a “critical section?”

Where

- If you know what caused the interrupt then you want to jump to the code that handles that interrupt.
  - If you number the possible interrupt cases, and an interrupt comes in, you can just branch to a location, using that number as an offset (this is a branch table)
  - If you don’t have the number, you need to *poll* all possible sources of the interrupt to see who caused it.
    - Then you branch to the right code
Get back to where you once belonged

• Need to store the return address somewhere.
  – Stack might be a scary place.
    • That would involve a load/store and might cause an interrupt!
  – So a dedicated register seems like a good choice
    • But that might cause problems later…

Snazzy architectures

• A modern processor has many (often 50+) instructions in-flight at once.
  – What do we do with them?
• Drain the pipeline?
  – What if one of them causes an exception?
• Punt all that work
  – Slows us down
• What if the instruction that caused the exception was executed before some other instruction?
  – What if that other instruction caused an interrupt?

Nested interrupts

• If we get one interrupt while handling another what to do?
  – Just handle it
    • But what about that dedicated register?
    • What if I’m doing something that can’t be stopped?
  – Ignore it
    • But what if it is important?
  – Prioritize
    • Take those interrupts you care about. Ignore the rest
    • Still have dedicated register problems.

Critical section

• We probably need to ignore some interrupts but take others.
  – Probably should be sure our code can’t cause an exception.
  – Use same prioritization as before.
## Power PC

- Power PC is a great teaching tool for this because it is so messed-up in how it handles things
  - It supports almost every option you could ever want.
- Names
  - Synchronous interrupts
  - Asynchronous interrupts

## PowerPC Interrupt Structure

**Machine Status Register (MSR)**
- Defines the state of the processor (Table 6-5): Power management, endian mode, external interrupt enable, machine check enable, privilege level, recoverable exceptions, etc.
- When an exception occurs, MSR bits are altered as determined by the exception.
- Support for interrupts
  - Two special-purpose registers to store program counter and MSR: Save/Restore Registers 0 and 1 (SRR0 and SRR1)
  - One instruction: return from interrupt (rfi)

### MSR

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-12</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>POW</td>
<td>power management</td>
</tr>
<tr>
<td>14</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ILE</td>
<td>exception little-endian</td>
</tr>
<tr>
<td>16</td>
<td>EE</td>
<td>external interrupt enable (0: delay recognition; 1: enabled)</td>
</tr>
<tr>
<td>17</td>
<td>PR</td>
<td>privilege level (0: user and supervisor-level; 1: user only)</td>
</tr>
<tr>
<td>18</td>
<td>FP</td>
<td>floating point available</td>
</tr>
<tr>
<td>19</td>
<td>ME</td>
<td>machine check enable</td>
</tr>
<tr>
<td>20</td>
<td>FE0</td>
<td>floating point exception mode 0</td>
</tr>
<tr>
<td>21</td>
<td>SE</td>
<td>single-step trace enable</td>
</tr>
<tr>
<td>22</td>
<td>BE</td>
<td>branch trace enable</td>
</tr>
<tr>
<td>23</td>
<td>FE1</td>
<td>floating point exception mode 1</td>
</tr>
<tr>
<td>24</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>IP</td>
<td>exception prefix (000 or FFF)</td>
</tr>
<tr>
<td>26</td>
<td>IR</td>
<td>instruction address translation</td>
</tr>
<tr>
<td>27</td>
<td>DR</td>
<td>data address translation</td>
</tr>
<tr>
<td>28-29</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>RI</td>
<td>recoverable exception</td>
</tr>
<tr>
<td>31</td>
<td>LE</td>
<td>little-endian enable (0: big-endian; 1: little-endian)</td>
</tr>
</tbody>
</table>

## PowerPC: Interrupt Process

- Basic interrupt process
  - Stop executing current program (stop fetching new instructions)
  - Save program counter of next instruction in SRR0
  - Save processor mode bits from MSR in SRR1
  - Change some of the processor mode bits in MSR
  - Branch to address determined by type of interrupt
  - The last instruction in an ISR will be an rfi which will
  - Restore the processor mode bits from SRR1
  - Branch to the address in SRR0
Example: External Interrupt

An external interrupt is signaled to the processor through the assertion of the external interrupt signal.

Execution continues at offset 0x00500 from base physical location determined by MSR[IP] bit.

SRR0: Set to the effective address that processor would have attempted to execute next if no interrupt condition were present.

SRR1: 1-4 cleared
10-15 cleared
16-23, 25-27, 30-31 loaded with equivalent bits from MSR

MSR:
POW = 0
FP = 0
BE = 0
DR = 0
ILE = same
ME = same
FE1 = 0
RI = 0
EE = 0
FE0 = 0
IP = same
LE = ILE
PR = 0
SE = 0
IR = 0

MPC823 Interrupts (Table 7-1)

<table>
<thead>
<tr>
<th>Type</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>System reset</td>
<td>Implementation dependent.</td>
</tr>
<tr>
<td>Machine check</td>
<td>Bus-parity errors, access invalid physical address.</td>
</tr>
<tr>
<td>DSI</td>
<td>Cannot perform data memory access due to page faults, protection violations etc.</td>
</tr>
<tr>
<td>ISI</td>
<td>Cannot fetch next instruction due to page faults, protection violations etc.</td>
</tr>
<tr>
<td>External interrupt</td>
<td>Assertion of external interrupt signal.</td>
</tr>
<tr>
<td>Alignment</td>
<td>Cannot perform memory access due to alignment problems, endian problems, etc. Implementation dependent.</td>
</tr>
<tr>
<td>Program</td>
<td>Floating point operations, illegal instruction, privilege violation etc.</td>
</tr>
<tr>
<td>Floating point unavailable</td>
<td></td>
</tr>
<tr>
<td>Decrementer</td>
<td>MSB of decrementer changes from 0 to 1.</td>
</tr>
<tr>
<td>System call</td>
<td></td>
</tr>
<tr>
<td>Trace</td>
<td>Complete instruction w/o exception or context change.</td>
</tr>
</tbody>
</table>

Precise Exceptions

- Concurrent execution: To maximize performance, instructions are processed concurrently, independent of the sequence in program.
- Hardware ensures that program semantics are preserved.
- Difficult requirement to assure when interrupt occurs after instructions following “faulting” instruction have started or completed.
- Precise exception model: Automatically back machine up to the instruction that caused interrupt.
- MPC823 uses a history buffer to support precise exceptions.

Interrupt Ordering

- Synchronous (i.e. instruction-related) exceptions are detected at any stage during instruction execution.
- The earliest exception found in the processing of an instruction precludes detection of further exceptions and is eventually handled.
- If more than one instruction in the pipeline causes an exception, only the first is taken. Any remaining synchronous exceptions are ignored.
- More than one asynchronous interrupt causes may be present at any time, in which case only the highest priority interrupt is taken.
## PowerPC Exception Priorities

<table>
<thead>
<tr>
<th>Class</th>
<th>Priority</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-maskable, async</td>
<td>1</td>
<td>System reset</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Machine check</td>
</tr>
<tr>
<td>Synchronous precise</td>
<td>3</td>
<td>Instruction dependent</td>
</tr>
<tr>
<td>Synchronous imprecise</td>
<td>4</td>
<td>Floating point</td>
</tr>
<tr>
<td>Maskable, async</td>
<td>5</td>
<td>External interrupt</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Decrementer</td>
</tr>
</tbody>
</table>

If MSR[EE]=0, delayed until bit is set

## Nesting Interrupts

- Nested interrupt: An interrupt that happens during the execution of an ISR.
  - Multiple interrupting devices with long ISRs
  - Virtual memory support for ISRs
  - Debugging ISR code

- What must ISRs do to support nested interrupts?

## Disabling Interrupts

- Sometimes interrupts must be avoided.
  - Time-critical instruction sequences (real-time applications)
  - Prologue sequence of ISRs
  - Changes in data structures shared with ISRs

- Synchronous interrupts: Not much choice. Can only be avoided by not executing instructions that might cause them such as illegal instructions or loads or stores to non-existent addresses.

- Asynchronous interrupts from external devices: Can be disabled (masked) using the external interrupt enable (EE) mode bit of MSR.
  - If 0, external interrupt signal is ignored.
  - If 1, external interrupt signal causes interrupt when asserted.
  - EE is automatically set to 0 in the beginning of every interrupt.

- Some interrupts are not maskable: Reset

## Managing Interrupts from Multiple Devices

- When multiple devices can interrupt the CPU, the system must determine which device to service

- Related issues
  - Identification: Which device caused the current interrupt?
  - Prioritization: If more than one device is simultaneously interrupting, which one is handled first?

- Three standard approaches
  - Non-vectored
  - Vectored
  - Autovectored

http://www.chipcenter.com/eexpert/dgilbert/dgilbert005.html
# Non-Vectored Interrupts

- Simplest hardware, least flexible.
- 6802, PowerPC, MIPS,…
- Single interrupt input to CPU.
- CPU always branches to same ISR.
- ISR polls each device to see which may have caused interrupt.
- Prioritization?

# Vectored Interrupts

- Used in 8080, 80x86, Z80
- As with non-vectored interrupts, single CPU interrupt input.
- Interrupt handler (implemented in hardware) enables a branch to occur to a different address for each specific interrupt.
- CPU performs special *interrupt acknowledge* bus cycle to obtain interrupt vector number directly from device.
- Typically requires that devices be designed to work with specific CPU.
- Prioritization typically via daisy chain (look up in 370 material).

# Auto-Vectored Interrupts

- Used in 68000, SPARC.
- Can be built on top of vectored or non-vectored interrupts.
- Multiple CPU interrupt inputs, one for each priority level.
- Interrupt vector is supplied automatically based on highest-priority asserted input.
- CPU *interrupt priority level* controls which inputs are recognized.
  - For example, if IPL=3, levels 3, 4, 5, … are disabled.
  - On interrupt, CPU automatically raises IPL to match level of request being serviced.
- Intel 8259A interrupt controller builds autovectoring on top of 80x86.
- MPC823 provides on-chip interrupt controller for pseudo-autovectored interrupts.

# MPC823 Interrupt Controller

- Part of on-chip System Interface Unit (SIU), not part of core. (Section 12.3 of data book)
- Eight external interrupt pins, each with its own dedicated interrupt priority level: IRQ0’ (highest priority) through IRQ7’ (lowest priority).
- Eight internal interrupt priorities, Level 0 through 7, generated by on-chip devices.
- Sixteen interleaved priorities: IRQ0’ (non-maskable), Level 0, IRQ1’, Level 1,…
- Assertion of any of these 16 interrupts can potentially assert the PowerPC external interrupt signal.
Programming the Interrupt Controller (1/2)

- Interrupt vector register (SIVEC)
  - 8-bit code gives the unmasked interrupt source of the highest priority level (0-15).
  - Read-only memory-mapped register gives interrupt code (= source number * 4) that may be read as byte, half-word, or word.
  - Used to index interrupt table.

<table>
<thead>
<tr>
<th>Base</th>
<th>Base + 4</th>
<th>Base + 8</th>
<th>Base + 16</th>
<th>Base + 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>b ISR1</td>
<td>b ISR2</td>
<td>b ISR3</td>
<td>b ISR4</td>
<td>b ISR5</td>
</tr>
</tbody>
</table>

Programming the Interrupt Controller (2/2)

- Interrupt mask register (SIMASK)
  - Read/Write memory-mapped register.
  - Bits enable/disable each interrupt.

- Interrupt pending register (SIPEND)
  - Read/Write memory-mapped register.
  - Bits indicate which interrupts are pending.

- Interrupt edge/level register (SIEL)
  - Read/Write memory-mapped register.
  - Bits indicate level-sensitive/edge-triggered interrupt detection and exit from low-power mode.

Priority of SUI Interrupt Sources

<table>
<thead>
<tr>
<th>number</th>
<th>priority level</th>
<th>source</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>IRQ’0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>Level 0</td>
<td>IRQ’1</td>
<td>00000100</td>
</tr>
<tr>
<td>2</td>
<td>Level 1</td>
<td>IRQ’2</td>
<td>00010010</td>
</tr>
<tr>
<td>3</td>
<td>Level 2</td>
<td>IRQ’3</td>
<td>00011000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>14</td>
<td>IRQ’7</td>
<td>Level 7</td>
<td>00111000</td>
</tr>
<tr>
<td>15</td>
<td>lowest</td>
<td>Level 8</td>
<td>00111100</td>
</tr>
<tr>
<td>16—31</td>
<td>reserved</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
PPC: Interrupts in detail

Flip-Flops \( \rightarrow \) mux \( \rightarrow \) SIPEND \( \rightarrow \) SIVEC \( \rightarrow \) Core

8 \( \rightarrow \) IRQ#[0:7] \( \rightarrow \) 8 \( \rightarrow \) 16 \( \rightarrow \) 16 \( \rightarrow \) Core

8 \( \rightarrow \) LVL[0:7] \( \rightarrow \) SIPEND \( \rightarrow \) SIVEC

8 \( \rightarrow \) SIEL