Lecture 3: Interrupt and Exception

- **Introduction**
  - Interrupt controller, PIC and APIC

- **ARM Exception handling**
  - Entering an exception
  - Leaving an exception
  - Exception Handler
  - Exception flow summary
Introduction

• Exceptions
  – An event alters the normal sequence of execution and force the processor to execute special instructions in a privileged state.
  – Two kinds of exceptions:
    • Synchronous (exceptions): Ex. Page fault, divided by zero.
    • Asynchronous (interrupts): Ex. Push the reset button.
  – An interrupt or an exception handler is not a process
    • It is a kernel control path.
Introduction

• **Interrupts (Asynchronous)**
  – **Triggered by electrical signals** generated by hardware circuits.
    • **Maskable interrupts**: masked or unmaskes states
      – All IRQs issued by I/O devices are maskable.
    • **Nonmaskable interrupts (NMI)**:
      – For critical events.

• **Exceptions (Synchronous)**
  – **Processor-detected exceptions**: detects an anomalous condition.
    • **Faults**: the program is allowed to restart with no loss of continuity after corrected, ex. Page Fault.
    • **Traps**: triggered when no need to re-execute the instruction that terminated, main use for debugging.
    • **Aborts**: For a serious error, the affected process is terminated.
  – **Programmed exceptions**: occur at the request of the programmer.
    • int, int3, into, and bound instructions.
    • For system call and notifying a debugger.
Introduction- Interrupt Controller

• Hardware device controller has an output line designated as an IRQ (Interrupt ReQuest).

• The IRQ lines are connected to the input pins of a hardware circuit called the Interrupt Controller.

• An Interrupt Controller performs the following actions:
  – 1. Monitors the IRQ lines, checking for raised signals.
  – 2. If a raised signal occurs on an IRQ line:
    • Converts the raised signal received into a corresponding vector.
    • Stores the vector in its I/O port.
    • Sends a raised signal to the processor INTR pin.
    • Wait until CPU acknowledges the interrupt signal, then clears INTR line.

• Interrupts to the Interrupt Controller are identified by a vector.
  – Non-maskable interrupts: the vectors are fixed.
  – Maskable interrupts: can be altered by programming Interrupt Controller.
Introduction- PIC

- **PIC (Programmable Interrupt Controller)**
  - Designed for uni-processor
  - Interrupt priority, vectors are programmable

- **Cascading two PICs**
  - Extending the number of IRQs from 8 to 15.
Advanced Programmable Interrupt Controller (APIC)

- Designed for a multi-processor system
  - Each CPU include a local-APIC
  - All the local APICs are connected to an external I/O APIC, giving raise to a multi-APIC system.

- External Interrupt flow
  - Hardware device raises an IRQ signal to I/O APIC
  - The I/O APIC delivers the signal to the selected local APIC
  - The local APIC issues an interrupt to its CPU.

- Interprocessor Interrupt flow
  - \( \text{CPU}_0 \) stores interrupt vector and the identifier of local APIC\(_1\) in the Interrupt Command Register (ICR) of its local APIC\(_0\).
  - Local APIC\(_0\) then send a message via ICC bus to local APIC\(_1\).
  - The local APIC\(_1\) in turn issues an interrupt to \( \text{CPU}_1 \).
Introduction- APIC

• **External interrupt distribution in multi-APIC system**
  - **Static distribution**
    • According to *Redirection Table* (which is programmable)
  - **Dynamic distribution**
    • According to “*lowest priority*” scheme - Sent to the local APIC of the processor that is executing process with the lowest priority.
    • Each local APIC has a programmable *Task Priority Register (TPR)*, updated by OS to compute the priority of currently running process.
  - **In Linux**
    • The booting CPU calls `setup_IO_APIC_irqs()` to initialize I/O APIC and set Redirection Table entries to allow all IRQs to be routed to each CPU based on “lowest priority” scheme.
    • All CPUs then call `setup_local_APIC()` to initialize their own local APIC, and give a fixed value to TPR.
    • Linux kernel never modifies this value after initialization, resulting in a *round-robin* distribution of external IRQs among all the CPUs.
Lecture 3: Interrupt and Exception

• Introduction
  – Interrupt controller, PIC and APIC

• ARM Exception handling
  – Entering an exception
  – Leaving an exception
  – Exception Handler
  – Exception flow summary
Entering an Exception (1/5)

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Mode</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Undefined Instructions</td>
<td>Undefined</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Software Interrupts (SWI)</td>
<td>Supervisor</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>0x00000010</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
<td>0x00000010</td>
</tr>
<tr>
<td>IRQ (Normal Interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
</tr>
<tr>
<td>FIQ (Fast interrupt)</td>
<td>FIQ</td>
<td>0x0000001C</td>
</tr>
</tbody>
</table>

Priority
1. Reset (highest)
2. Data abort
3. FIQ
4. IRQ
5. Prefect abort
6. SWI, undefined instr

Diagram:
- SWI handler
- IRQ handler
- Exception Vector Table
  - FIQ
  - IRQ
  - Reserved
  - Data Abort
  - Prefetch Abort
  - Software Interrupt
  - Undefined Instr
  - Reset
- Target Memory
  - 0x00
  - 0x04
  - 0x08
  - 0x10
  - 0x14
  - 0x18
  - 0x1C
entering an exception (2/5)

• how the processor responds to an exception?
  – save the address of the next instruction in the appropriate link register
  – save the cpsr into the appropriate spsr
  – force the cpsr mode bits to a value which depends on the exception
  – force to run in arm state
  – disable irq and fiq (if necessary) interrupt
  – force pc to begin executing at the relevant exception vector address

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0 /* execution in ARM state */
If <exception_mode> == Reset or FIQ then
  CPSR[6] = 1 /* disable FIQ interrupt */
/* else CPSR[6] is unchanged */
CPSR[7] = 1 /* disable normal interrupt */
PC = exception vector address
```
# Entering an Exception (3/5)

| Reset                              | R14_svc     = unexpected  
|                                    | SPSR_svc    = unexpected  
|                                    | CPSR[4:0]   = 0b10011     // Supervisor Mode 
|                                    | CPSR[5]     = 0           // ARM state 
|                                    | CPSR[6]     = 1           // Disable FIQ 
|                                    | CPSR[7]     = 1           // Disable IRQ 
|                                    | PC = 0x00000000          |
| Undefined Instructions             | R14_und     = PC + 4      
|                                    | SPSR_und     = CPSR       
|                                    | CPSR[4:0]   = 0b11011     // Undefined Mode 
|                                    | CPSR[5]     = 0           // ARM state 
|                                    | CPSR[6] unchanged = FIQ flag 
|                                    | CPSR[7]     = 1           // Disable IRQ 
|                                    | PC = 0x00000004          |
| Software Interrupt                 | R14_svc     = PC + 4      
|                                    | SPSR_svc     = CPSR       
|                                    | CPSR[4:0]   = 0b10011     // Supervisor Mode 
|                                    | CPSR[5]     = 0           // ARM state 
|                                    | CPSR[6] unchanged = FIQ flag 
|                                    | CPSR[7]     = 1           // Disable IRQ 
|                                    | PC = 0x00000008          |
### Entering an Exception (4/5)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prefetch Abort</strong></td>
<td>R14_abt = PC + 4</td>
<td>SPSR_abt = CPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10111 //Abort Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[5] = 0       // ARM state</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[6] unchanged // FIQ flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[7] = 1       // Disable IRQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC = 0x0000000C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Abort</strong></td>
<td>R14_abt = PC + 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPSR_abt = CPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10111 //Abort Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[5] = 0       // ARM state</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[6] unchanged // FIQ flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[7] = 1       // Disable IRQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC = 0x00000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Interrupt Request</strong></td>
<td>R14_abt = PC+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPSR_abt = CPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10110 // IRQ Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[5] = 0       // ARM state</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[6] unchanged // FIQ flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPSR[7] = 1       // Disable IRQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC = 0x00000018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Entering an Exception (5/5)

<table>
<thead>
<tr>
<th>Fast Interrupt Request</th>
<th>R14_abt = PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPSR_abt = CPSR</td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10001 // FIQ Mode</td>
</tr>
<tr>
<td></td>
<td>CPSR[5] = 0       // ARM state</td>
</tr>
<tr>
<td></td>
<td>CPSR[6] = 1       // <strong>Disable FIQ</strong></td>
</tr>
<tr>
<td></td>
<td>CPSR[7] = 1       // <strong>Disable IRQ</strong></td>
</tr>
<tr>
<td></td>
<td>PC = 0x0000001C</td>
</tr>
</tbody>
</table>
Leaving an Exception (1/2)

- The “return address” depends on the exception types.
  - **RESET**
    - No need to return. The handler should re-execute your bootup code.
  - **SWI and UDef**
    - generated by the instruction *itself*, so return to the *next* instr.
    - *Processor*: lr_mode = pc + 4  □   *Handler*: MOV pc, lr_mode
  - **FIQ and IRQ**
    - generated by *unexpected* interrupt, so return to the *interrupted* instr.
    - *Processor*: lr_mode = pc + 4  □   *Handler*: SUB pc, lr_mode, #4
  - **PAbort**
    - It may occur due to memory *fault* in MMU system, so return to retry the *interrupted* instr. again.
    - *Processor*: lr_abt = PC + 4  □   *Handler*: SUB pc, lr_abt, #4
  - **DAbort**
    - Same as PAbort to return to *interrupted* instr.
    - *Processor*: lr_abt = PC + 8  □   *Handler*: SUB pc, lr_abt, #8

Note: The S flag on MOV and SUB means to update CPSR as well *(why?)*
Leaving an Exception (2/2)

Summary

<table>
<thead>
<tr>
<th>Return Instruction</th>
<th>Previous State</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM R14_x</td>
<td>THUMB R14_x</td>
</tr>
<tr>
<td>BL</td>
<td>PC + 4</td>
<td>PC + 2</td>
</tr>
<tr>
<td>SWI</td>
<td>PC + 4</td>
<td>PC + 2</td>
</tr>
<tr>
<td>UDEF</td>
<td>PC + 4</td>
<td>PC + 2</td>
</tr>
<tr>
<td>FIQ</td>
<td>PC + 4</td>
<td>PC + 4</td>
</tr>
<tr>
<td>IRQ</td>
<td>PC + 4</td>
<td>PC + 4</td>
</tr>
<tr>
<td>PABT</td>
<td>PC + 4</td>
<td>PC + 4</td>
</tr>
<tr>
<td>DABT</td>
<td>PC + 8</td>
<td>PC + 8</td>
</tr>
<tr>
<td>RESET</td>
<td>NA</td>
<td>–</td>
</tr>
</tbody>
</table>

Notes
1. Return to the address *next to* the instruction that caused the exception.
2. Return to the address of *the instruction that caused the exception*.
3. No need to return from exception handler. System will restart.
Exception Handler

• **IRQ Handler as an Example**

```plaintext
IRQ_Handler: ; top-level handler
    STMFD sp!, {r0-r12,lr} ; Handler entry: to store Regs.
    BL ISR_IRQ ; Second-level handler
    SUB lr, lr, #4 ; Handler exit: to calculator return addr.
    LDMFD sp!, {r0-r12,pc}^ ; to restore Reg and return
```

• **How to handle FIQ more faster ?**
  
  – Only need to save r0-r7 because FIQ mode has banked r8-r12.
  – FIQ is the last entry in vector table, so can do it right inside top-handler
    - No need the time to branch to 2nd-level handler
  – Lock FIQ handler and the vector table into cache for speedup.

```plaintext
FIQ_Handler: ; top-level handler
    SUB lr, lr, #4 ; Handler entry: to calculator return addr.
    STMFD sp!, {r0-r7,lr} ; Handler entry: to store Regs.
    ; Handle FIQ event right here …
    LDMFD sp!, {r0-r7,pc}^ ; Handler exit: to restore Reg and return.
```
Exception Flow Summary

• **Entering an Exception** (by Processor/Hardware)
  – Update banked Link Register (r14)  *(to return to original program flow)*
  – Update banked SPSR  *(to preserve CPSR state before exception)*
  – Change to correct Processor mode  *(according to exception type)*
  – Force to run in ARM state  *(all the exception must run in ARM state)*
  – **Disable** interrupt (IRQ and FIQ (if necessary))  *(to prevent re-entrance)*
  – Force PC to have correct vector address  *(prepare for jump)*
  – **Jump** to corresponding Exception Handler registered by software

• **Inside Exception Handler** (by software)
  – Save registers in the stack  *(according to banked sp_)*
  – **Call** to the 2nd-level routine to do the handling  *(FIQ can do it here, why?)*

  – **Leaving the Exception**
    • Restore registers from the stack  *(according to backed sp_)*
    • Restore CPSR  *(to previous Processor Mode, State, FIQ/IRQ status)*
    • Give return address to PC  *(to return to previous program flow)*
Exception Flow Summary

Exception Entry (by h/w)
- Update link register (r14)
- Save CPSR to SPSR
- Change Processor mode
- Change to ARM state
- Disable interrupt
- Update PC

Exception handler (by s/w)
- Save Regs (r1~r12) in stacks (if necessary)
- Invoke 2^nd^-level handler routine

Exception Exit
- Restore Regs from stacks (if necessary)
- Restore CPSR
- Assign return addr to PC
Reference


• ARM7TDMI Technical Reference Manual
  – http://www.eecs.umich.edu/~tnm/power/ARM7TDMIvE.pdf