COMP 3221

Microprocessors and Embedded Systems

Lecture 40: Review of Exception

http://www.cse.unsw.edu.au/~cs3221

November, 2003

Saeid Nooshabadi

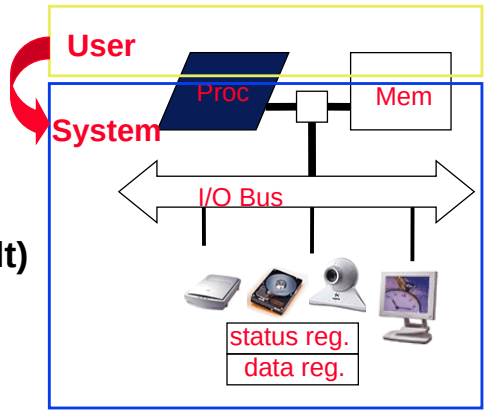
Saeid@unsw.edu.au

Outline

- Instruction Set Support for Exception
- Prioritized Exceptions
- Re-entrant Exception Routine

Crossing the System Boundary

- System loads user program into memory and 'gives' it use of the processor
- Switch back
  - SWI
    - request service
    - I/O
  - TRAP (page Fault)
  - Interrupt



Reasons for Exceptions/Interrupts

- Hardware errors: memory parity error
- External I/O Event
  - High Priority and Low Priority I/O events
- Illegal instruction
- Virtual memory
  - Write protection violation
  - Page fault - page is on disk
  - Invalid Address - outside of address range
- Arithmetic overflow
  - Floating Point Exceptions
- Software Interrupts (invoke Op Sys routine)

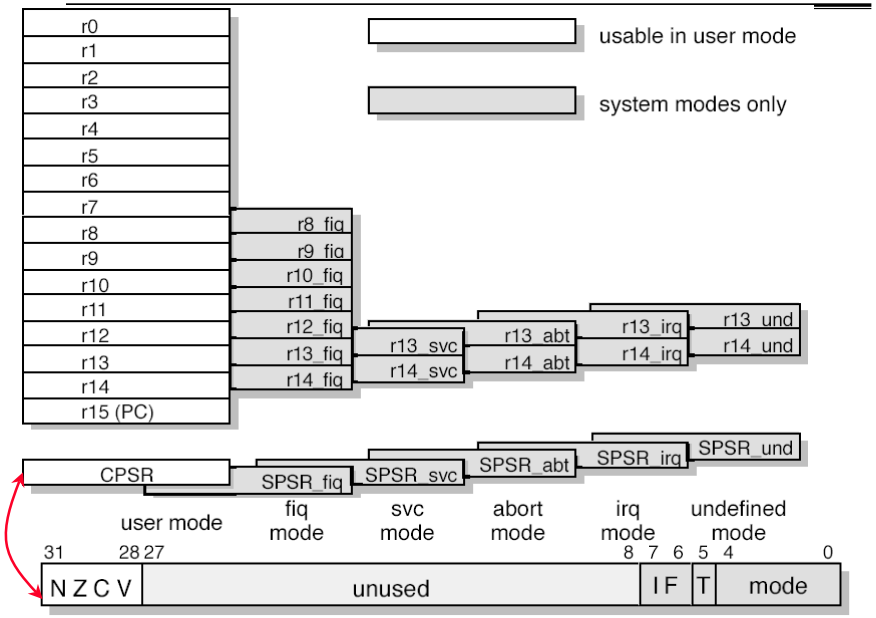
SWI

- How does user invoke the OS?
  - `swi` instruction: invoke the OS code (Go to 0x00000008, change to privileged mode)
  - By software convention,number `xxx` in `swi xxx` has system service requested: OS performs request

Software/Hardware Resources for Exceptions

- Registers to use in exception routine
  - All modes of operation have registers `lr_mode` and `sp_mode` in their bank for use; don't have to be saved
  - Registers `r8 – r12` are also available in `fiq` bank.
- Enable/Disable Interrupt & Fast Interrupt Bits
- Privileged/User mode to protect when can Disable Interrupts
- PC address of instruction causing the exception or the return address saved in `lr_mode`.
- Exception priority in case of multiple simultaneous exceptions
- Jump to Exception vectors at 0x00000008 - 0x0000001C
- Mode bits in CPSR to show the cause of exception
- Priority levels hardware/software to take multiple interrupts
- Possible register showing cause of interrupts

Support for ARM Modes of Operations

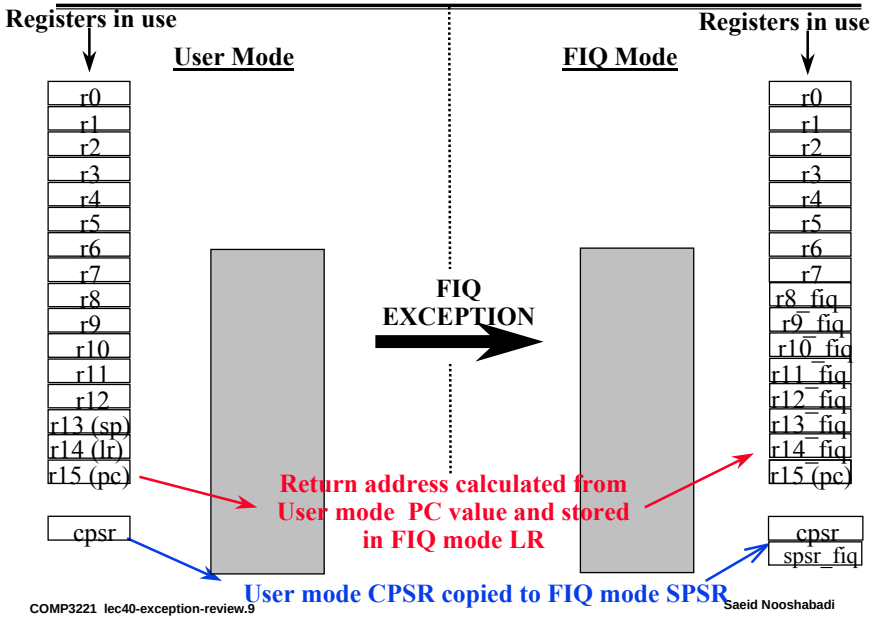


CPSR Encoding for Operating Modes & Interrupts

mode	Mode of Operation
T	ARM vs Thumb State (We only use ARM in COMP3221
F	FIQ Fast interrupts Disable bit
I	IRQ Normal interrupt Disable bit
NZCV	Condition Flags

Changing mode bits is only possible in privileged modes

Switching between Modes (User to FIQ Mode)



Exception Handling and the Vector Table

- When an exception occurs, the ARM core:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Interrupt disable flags if appropriate.
  - Maps in appropriate banked registers
  - Stores the "*return address*" in LR_<mode>
  - Sets PC to vector address

0x00000000	Reset
0x00000004	Undefined Instr
0x00000008	SWI
0x0000000C	Prefetch Abort
0x00000010	Data Abort
0x00000014	Reserved
0x00000018	IRQ
0x0000001C	FIQ

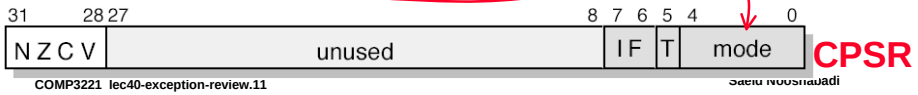
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode> via `movs pc, lr` or `subs pc, lr, #4`

ARM Modes of Operations

10000	User	Normal user code
10001	FIQ	Processing fast interrupts
10010	IRQ	Processing standard interrupts
10011	SVC	Processing software interrupts (SWIs)
10111	Abort	Processing memory faults
11011	Undef	Handling undefined instruction traps
11111	System	Running privileged operating system tasks

Non-Privileged Mode

Privileged Modes



Future of Wireless Embedded Devices

- The future will see smart technologies that will allow devices to be connected to any network that is available, while automatically collecting the information users want. Technically, the new wireless is also predicted to be smarter in size and power as traditional chips have been power hungry. This change will see tiny, inexpensive, embedded networks being integrated into almost any manufactured object. These embedded networks could also be tied to sensors that could monitor everything from environmental conditions to peak-hour traffic.
- Interestingly, speech recognition is predicted to boom as people prefer to talk rather than type commands or text on small key pads that are crammed into mobile phones and personal digital assistants

Gerry Purdy - wireless analyst

Silicon Valley 4.0 Conference

http://www.siliconvalley4.com/

Multiple Exceptions/Interrupts

- Problem: What if multiple exceptions and interrupts come simultaneously
- Options:
  - drop any conflicting interrupts/ exceptions: unrealistic, they may be important
  - simultaneously handle multiple interrupts exceptions: unrealistic, may not be able to synchronize them (such as with multiple I/O interrupts)
  - Handle them one by one in order of priority: sounds good

ARM Prioritized Exceptions

Exception Type	Priority (1=High, 6=Low)
Reset	1
Data Abort	2
Fast Interrupt (FIQ)	3
Interrupt (IRQ)	4
Prefetch Abort	5
Software Interrupt (SWI)	6
Undefined Instruction	6

Taking New Exception While Handling Exception

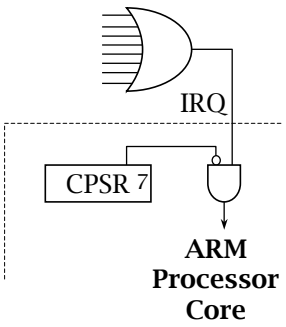
- Problem: What if we're handling an Data Abort exception and need to make an `SWI` call?
  - Each exception has its own version of registers `lr_mode` and `sp_mode` in their bank; so simply switch the supervisor mode
  - Other registers need saving on the mode stack.
- One `SWI` invoking another `SWI`?
  - Ok, provided `SWI` routine save `lr_swi` and `sp_swi` on the `SWI` stack on entry to routine as well.

Taking New Interrupt While Handling Exception

- Problem: What if we're handling an Data Abort exception and an I/O interrupt (printer ready, for example) comes in?
  - It is ignored since all exceptions disable IRQ (I bit = 1)
  - We can take interrupts by re-enabling them setting IRQ bit (I bit = 0); re-entrant interrupts
  - FIQ interrupt can be disabled (F bit = 1) only by FIQ interrupt.

So Many Devices One FIQ/IRQ

- Two interrupt request signals FIQ and IRQ never enough for all of the I/O devices
- Need a mechanism to attach multiple devices to the same IRQ pin.
- Solution: Use Interrupt Controller to attach multiple devices to the same IRQ pin.



  - Interrupt Controller controls how multiple peripherals can interrupt the ARM processor. Essentially, the interrupt controller acts as a large AND-OR gate
  - In the event of an IRQ a combination of hardware and software techniques are required to detect the sources of interrupt and provide a system of priority and queuing.

Nested Interrupt Support

- If going to support nested interrupts from multiple sources by re-enabling IRQ (I bit = 0), what must be saved/ restored on entry/exit of nested interrupt?
  - Save/restore all things associated with current interrupt: interrupt PC in `lr_irq`, `lr_sp`, `spsr`,
  - Any registers used beyond `lr_irq` and `sp_irq`
- Problem: How many levels of recursion can we allow in interrupts
  - i.e. how deep the stack can grow?
- Solution: Prioritization and Priority levels

Prioritized Interrupts (#1/2)

- Interrupt Controller support to simplify software:
  - Set priority levels for interrupts
  - Process cannot be preempted by interrupt at same or lower "level"
  - When an interrupt is handled, take the highest priority interrupt
    - The handler may need to save the state of the preempted program
  - Return to interrupted code as soon as no more interrupts at a higher level

Prioritized Interrupts (#2/2)

- To implement, we need an IRQ Stack:
  - portion of address space allocated for stack of "IRQ Frames"
  - each frame represents one interrupt: contains enough info to restart handling the preempted interrupt if necessary.
- In addition we need to keep the priority levels information
  - it is kept in a First-In Last-Out (FILO) stack in the Interrupt Controller (IC) hardware
  - current priority value is pushed onto a FILO stack and the current priority is updated to the higher priority.
  - on return the current interrupt level is updated with the last stored interrupt level from the stack.
- The priority levels information can be kept in IRQ stack if Interrupt controller does not support it.

Modified Interrupt Handler (#1/2)

- Problem: When an interrupt comes in while handling another interrupt, `lr_irq` and `spsr_irq` get overwritten *immediately* by hardware. Lost `lr_irq` means loss of user program.
- Solution: Modify interrupt handler. When first interrupt comes in:
  - disable interrupts (Done by hardware)
  - save `lr_irq, sp_irq` and `spsr_irq`, and any registers it may use on `IRQ` Stack
  - mask out the lower or same priority interrupts (Done via Interrupt controller hardware, when supports it)
  - re-enable interrupts
  - continue handling current interrupt
  - at the end disable the interrupts, unmask the lower or same priority interrupts, restore `lr_irq, sp_irq,` and `spsr_irq`, and any registers it has used from IRQ Stack and return to user code.

Modified Interrupt Handler (#2/2)

- When next (or any later) of higher priority interrupts comes in:
  - interrupt the first onedisable interrupts (Done by hardware)
  - save `lr_irq, sp_irq` and `spsr_irq`, and any registers it may use on `IRQ` Stack
  - mask out the lower or same priority interrupts (Done via Interrupt controller hardware, when supports it)
  - re-enable interrupts
  - continue handling current interrupt
  - at the end disable the interrupts, unmask the same priority interrupts, restore `lr_irq, sp_irq,` and `spsr_irq`, and any registers it has used from IRQ Stack and return to lower priority interrupt.

Re-entrant Interrupt Routine Review?

- How allow interrupt of interrupts and safely save registers?
- Stack?
  - Resources consumed by each interrupt, so cannot tolerate arbitrary deep nesting of exceptions/interrupts
- With priority level system only interrupted by higher priority interrupt, so cannot be recursive
- Only need one save area ("interrupt frame") per priority level

Supporting Multiple Interrupts in Software

- Exception/Interrupt behavior determined by combination of hardware mechanisms and operating system strategies
- same hardware with different OS acts differently
- A popular software model for multiple interrupts/exceptions, often used in Unix OS, is to set priority levels
  - This is an OS concept, not a HW concept
  - HW just needs mechanisms to support it

Things to Remember

- Privileged Modes vs. User Mode: OS can provide security and fairness
- swi: provides a way for a programmer to avoid having to know details of each I/O device
- To be acceptable, interrupt handler must:
  - service all interrupts (no drops)
  - service by priority
  - make all users believe that no interrupt has occurred