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

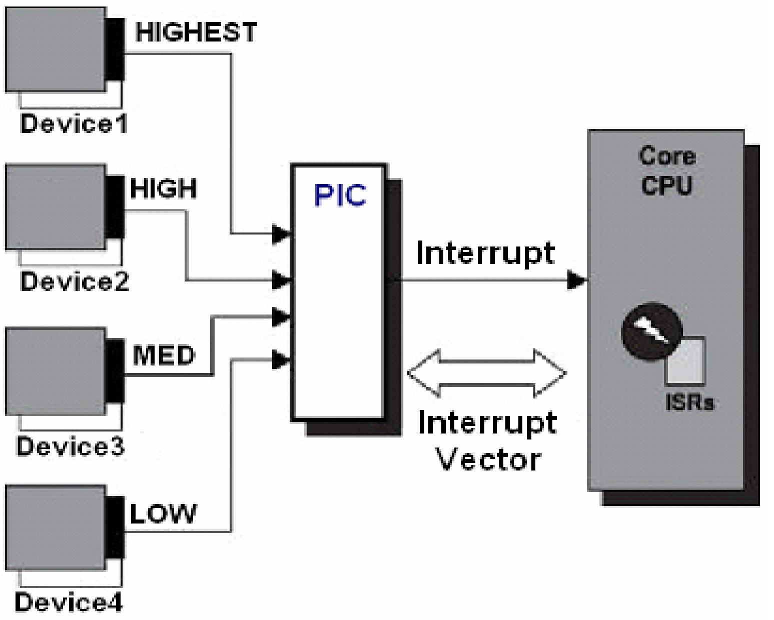
- **Interrupts (Asynchronous)**
  - **Triggered by electrical signals generated by hardware circuits.**
    - **Maskable interrupts:** masked or unmaskes states
      - All IRQs issued by I/O devices are maskable.
    - **Nonmaskable interrupts (NMI):**
      - For critical events.
- **Exceptions (Synchronous)**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

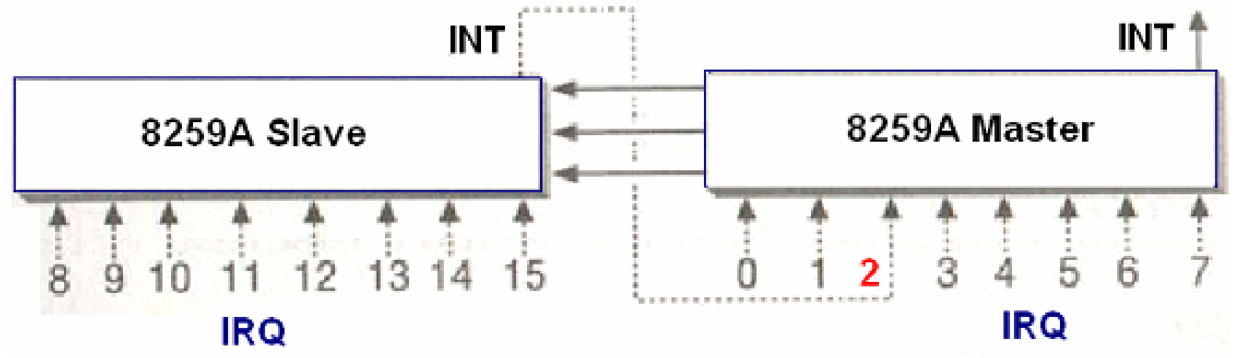
Introduction- Interrupt Controller

- **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.**
  - 3. Goes back to step1.
- **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

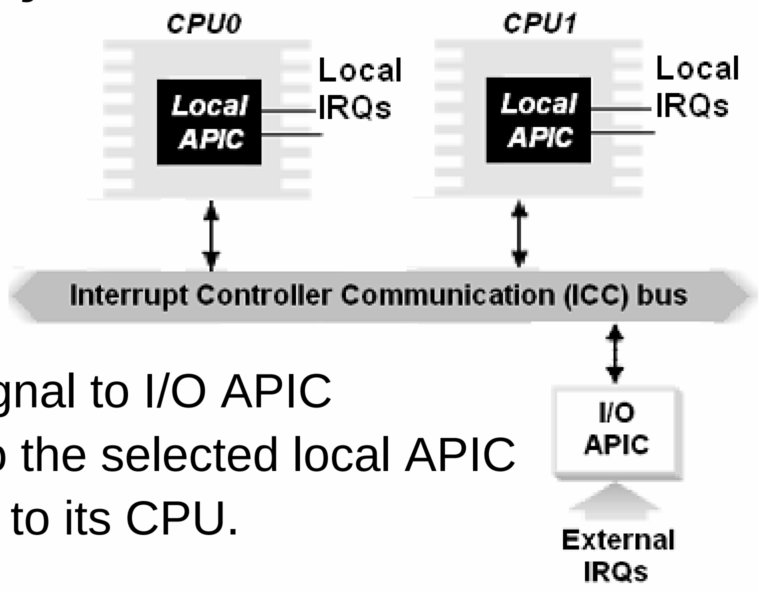
- **PIC (Programmable Interrupt Controller)**
  - Designed for uni-processor
  - Interrupt priority, vectors are programmable
- **Cascading two PICs**
  - Extending the number of IRQs from 8 to 15.





Introduction- APIC

- **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**
    - 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 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

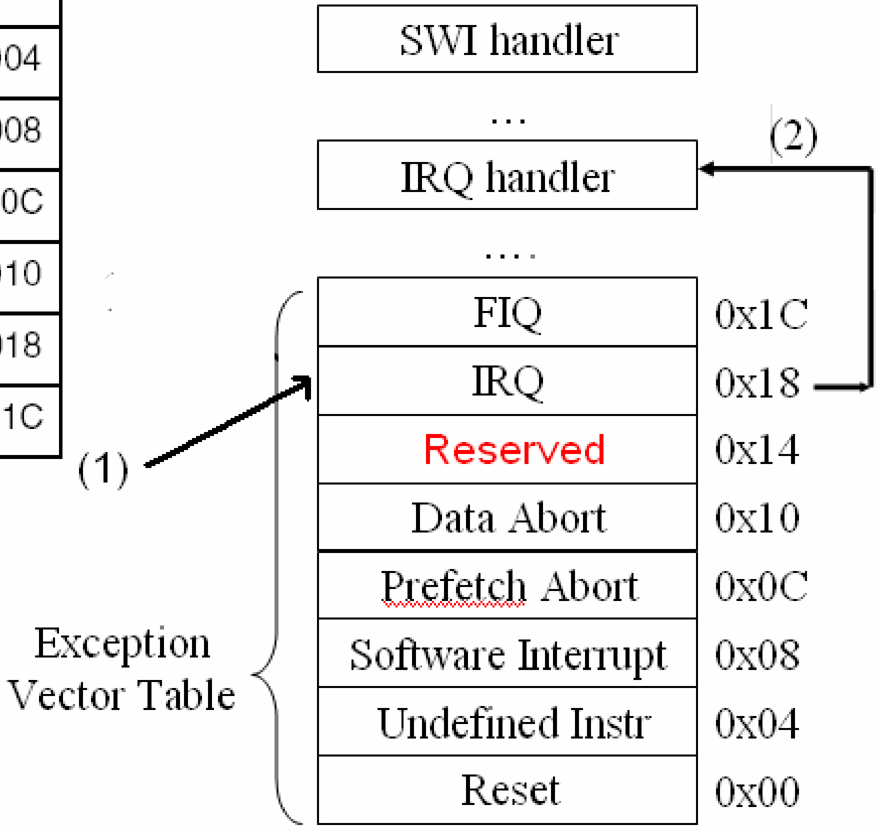
- **Introduction**
  - Interrupt controller, PIC and APIC
- **ARM Exception handling**
  - **Entering an exception**
  - **Leaving an exception**
  - **Exception Handler**
  - **Exception flow summary**

Entering an Exception (1/5)

Exception Type	Mode	Address
Reset	Supervisor	0x00000000
Undefined Instructions	Undefined	0x00000004
Software Interrupts (SWI)	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast interrupt)	FIQ	0x0000001C

Priority

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



Target Memory

Entering an Exception (2/5)

- **How the processor responses 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 = 0x0000004

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)

Prefetch Abort	**R14_abt = PC + 4
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10111 //Abort Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] unchanged // FIQ flag
	CPSR[7] = 1 // Disable IRQ
	PC = 0x000000C**

Data Abort	**R14_abt = PC + 8
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10111 //Abort Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] unchanged // FIQ flag
	CPSR[7] = 1 // Disable IRQ
	PC = 0x00000010**

Interrupt Request	**R14_abt = PC+4
	SPSR_abt = CPSR
	CPSR[4:0] = 0b10010 // IRQ Mode
	CPSR[5] = 0 // ARM state
	CPSR[6] unchanged // FIQ flag
	CPSR[7] = 1 // Disable IRQ
	PC = 0x0000018**

Entering an Exception (5/5)

Fast Interrupt Request	**R14_abt = PC + 4** **SPSR_abt = CPSR** **CPSR[4:0] = 0b10001 // FIQ Mode** **CPSR[5] = 0 // ARM state** **CPSR[6] = 1 // Disable FIQ** **CPSR[7] = 1 // Disable IRQ** **PC = 0x0000001C**

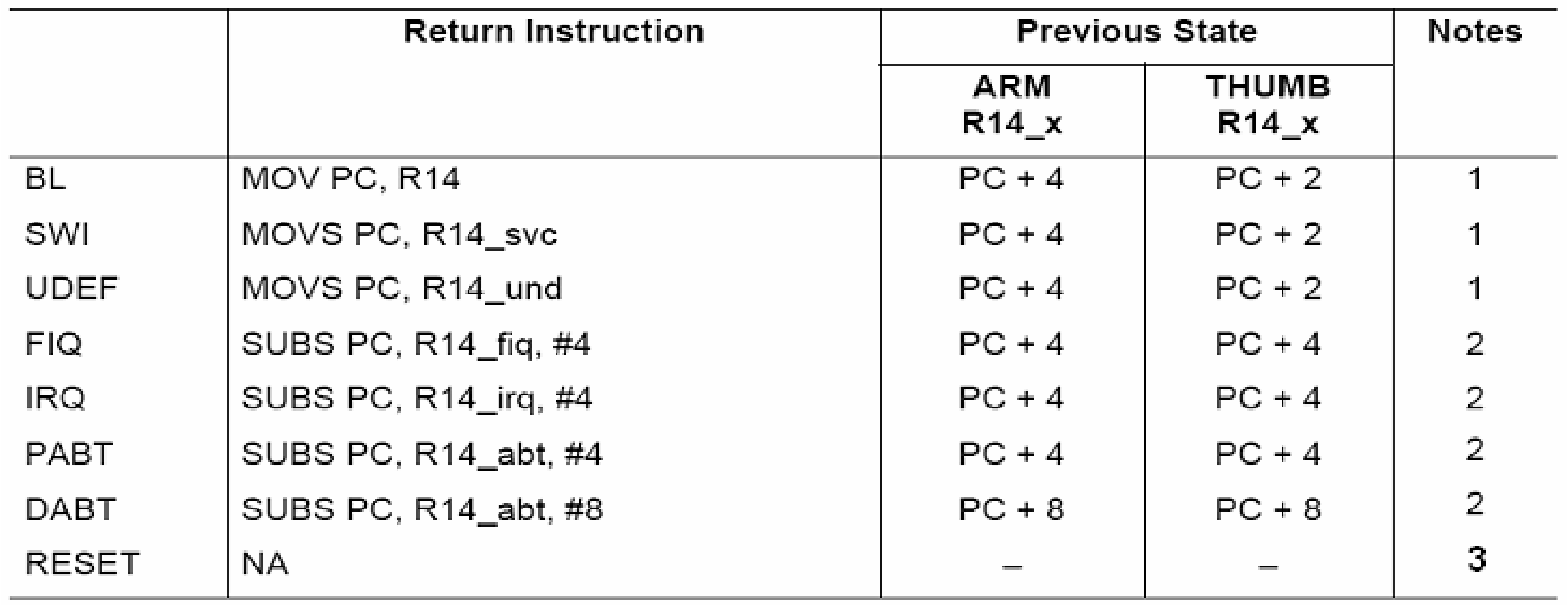
Leaving an Exception (1/2)

- 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**S** pc, lr_mode
  - **FIQ and IRQ**
    - generated by *unexpected* interrupt, so return to the *interrupted* instr.
    - ***Processor***: lr_mode = pc + 4 ⇨ ***Handler***: SUB**S** 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**S** pc, lr_abt, #4
  - **DAbort**
    - Same as PAbort to return to *interrupted* instr.
    - ***Processor***: lr_abt = PC + 8 ⇨ ***Handler***: SUB**S** 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



	Return Instruction	Previous State		Notes
		ARM R14_x	THUMB R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	1
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	1
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	1
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	2
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	2
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	2
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	2
RESET	NA	–	–	3

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**

```
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
```

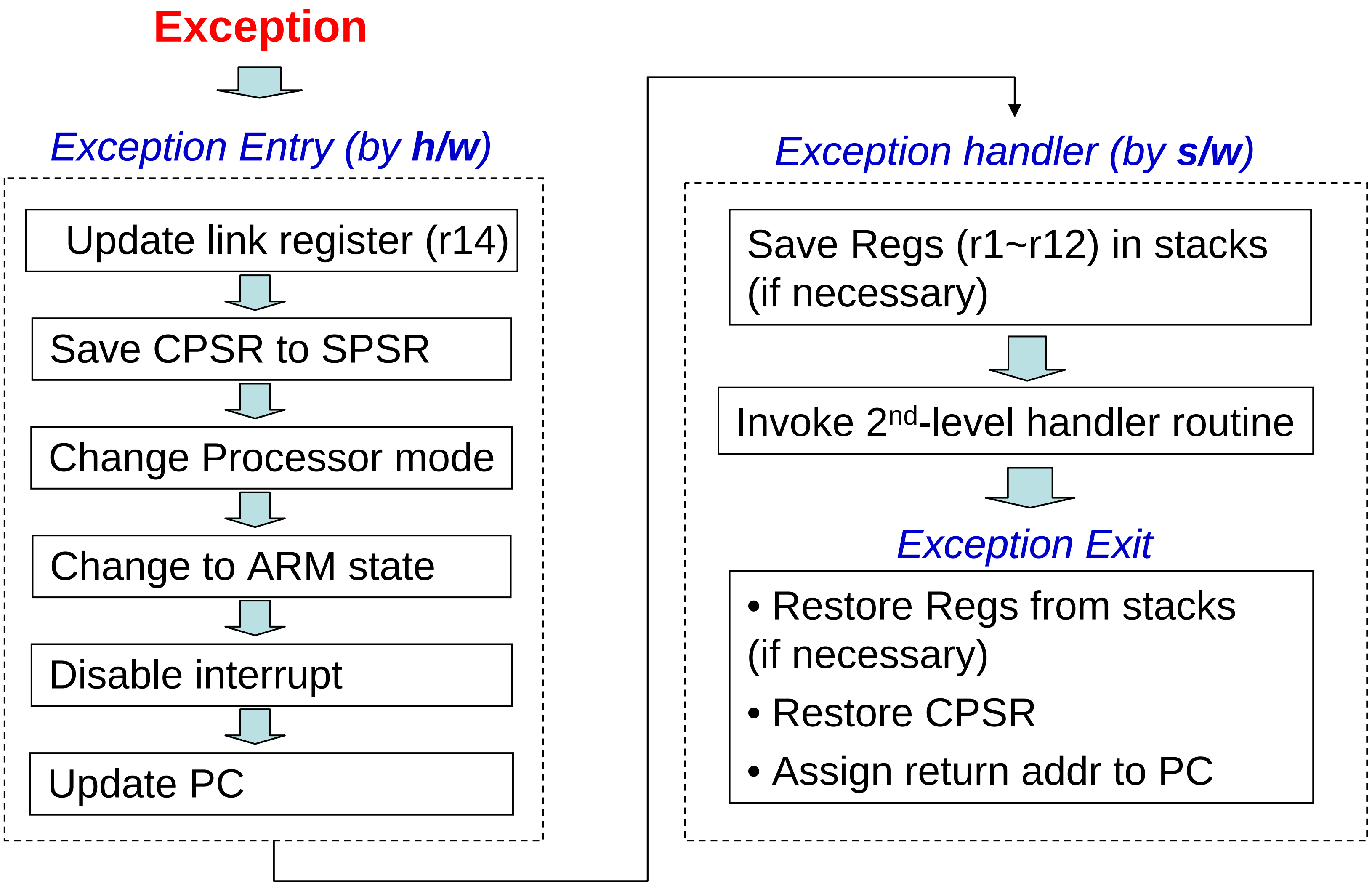
- **How to handle FIQ more faster ?**
  - Only need to save r0-r7 because FIQ mode has banked r8-r12.
  - FIQ is the last entry in vector table, so can do it right inside top-handler
    - No need the time to branch to 2nd-level handler
  - Lock FIQ handler and the vector table into cache for speedup.

```
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
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 2nd-level handler routine
Exception Exit
• Restore Regs from stacks (if necessary)
• Restore CPSR
• Assign return addr to PC

Reference

- “Understanding the Linux Kernel (3rd Edition),” Daniel P.Bovert & Macro Cesati, O’Reilly, ISBN0-596-00565

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