中国科学技术大学
University of Science and Technology of China

计算系统概论A
Introduction to Computing Systems
（ CS1002A.03 ）

# Chapter 9-3
# Interrupt-Driven I/O

陈俊仕
cjuns@ustc.edu.cn
2023 Fall

计算机科学与技术学院
School of Computer Science and Technology

# Outline

中国科学技术大学
University of Science and Technology of China


1 Review

2 Interrupt-Driven I/O

3 Input/Output

# What is Interrupt-Driven I/O?

```
Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
Program A is executing instruction n+3
Program A is executing instruction n+4
   ......
   ......
   ......
```

# What is Interrupt-Driven I/O?

```
Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
Interrupt!!!
Program A is executing instruction n+3
Program A is executing instruction n+4
   ......
   ......
   ......
```

# What is Interrupt-Driven I/O?

```
Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
1: Interrupt signal is detected
1: Program A is put into suspended animation
1: PC is loaded with the starting address of Program B
2: Program B starts satisfying I/O device's needs
2: Program B continues satisfying I/O device's needs
2: Program B continues satisfying I/O device's needs
2: Program B finishes satisfying I/O device's needs
3: Program A is brought back to life
Program A is executing instruction n+3
Program A is executing instruction n+4
   ......
   ......
   ......
```

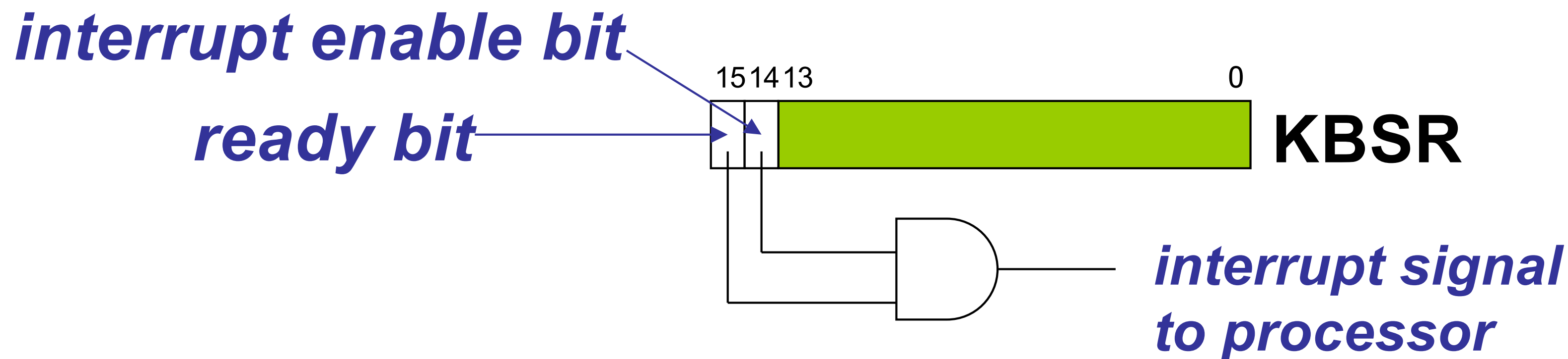
# When and Why to Use Interrupts

- **When timing of external event is uncertain**
  - Example: incoming packet from network
  - TRAP vs. Interrupt, 12.5s vs. 0.00001s @ 100 char read from KB
- **When device operation takes a long time**
  - Example: start a disk transfer,
    disk interrupts when transfer is finished
  - processor can do something else in the meantime
- **When event is rare but critical**
  - Example: building on fire -- save and shut down!

# How to implement an interrupt

- **To implement an interrupt mechanism, we need:**
  - **A way for the I/O device to signal the CPU that an interesting event has occurred.**
  - **A way for the CPU to test whether the interrupt signal is set.**

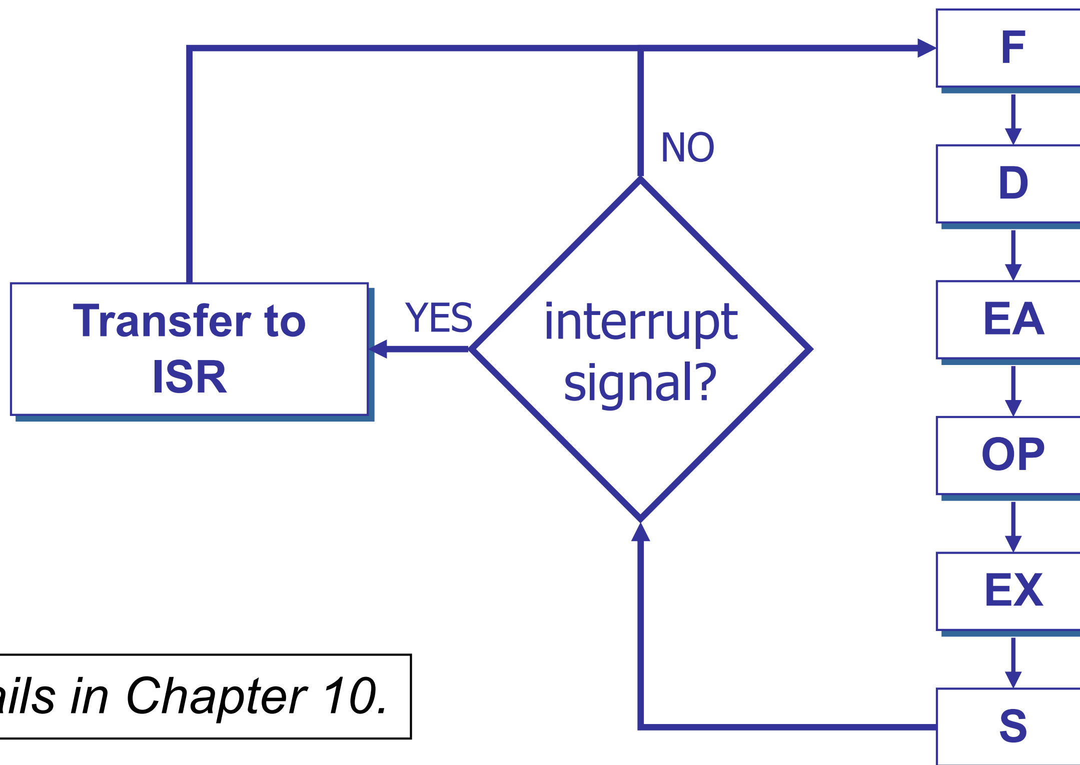
- **Generating Signal**
  - **Software sets "interrupt enable(IE)" bit in device register.**
  - **When ready bit is set and IE bit is set, interrupt is signaled.**



# Interrupt-Driven I/O

## Testing for Interrupt Signal

- CPU looks at signal between STORE and FETCH phases
- If not set, continues with next instruction
- If set, transfers control to Interrupt Service Routine(ISR)




*More details in Chapter 10.*

# Interrupt-Driven I/O

- Timing of I/O controlled by *device*
  - 1. Report: Tells processor when something interesting happens
    - Example: when character is entered on keyboard
    - Example: when monitor is ready for next character
    - Example: when block has been transferred from disk to memory
  - 2. Processing: Processor *interrupts* its normal instruction processing and executes a Interrupt Service Routine(ISR) (like a TRAP)
    - 1. Figure out what device is causing the interrupt
    - 2. Execute routine to deal with event
    - 3. Resume execution
- No need for processor to poll device
  - Can perform other useful work

*Interrupt is an **unscripted subroutine call,** triggered by an external event*

# How is Interrupt Signaled?

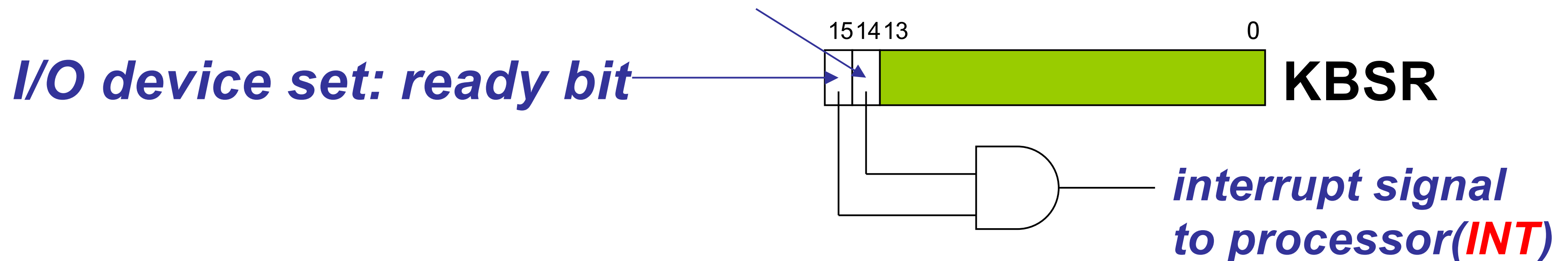
- **External interrupt signal by Devices: INT**
  - **Device sets INT=1 when it wants to cause an interrupt**
- **Interrupt vector: INTV**
  - **8-bit signal for device to identify itself**
  - **Also used as entry into Interrupt Vector Table, which gives starting address of *Interrupt Service Routine (ISR)***
    - **Just like Trap Vector Table and Trap Service Routine**
    - **TVT: x0000 to x00FF**
    - **IVT: x0100 to x01FF**

# Interrupt Signal to CPU

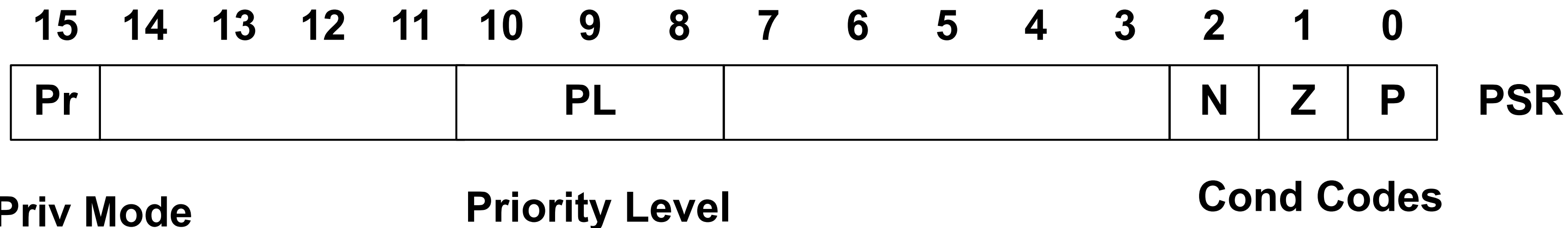
- **Several things must be true for an I/O device to actually interrupt the processor (ALL the three elements are present)**
  - **The I/O device must want service(*ready bit = 1*)**
  - **The device must have the right to request the service(*interrupt enable bit = 1*).**
  - **The device request must be more urgent than what the processor is currently doing.**

# Interrupt Signal to CPU

- **At the device side(*I/O device set*)**
  - Control register has "Interrupt Enable" bit
  - Must be set for interrupt to be generated
- **At the processor(*CPU set*)**
  - Sometimes have "Interrupt Mask" register (LC-3 does not)
  - When set, processor ignores INT signal
  - Why? - Example: may not want to be interrupted while in ISR



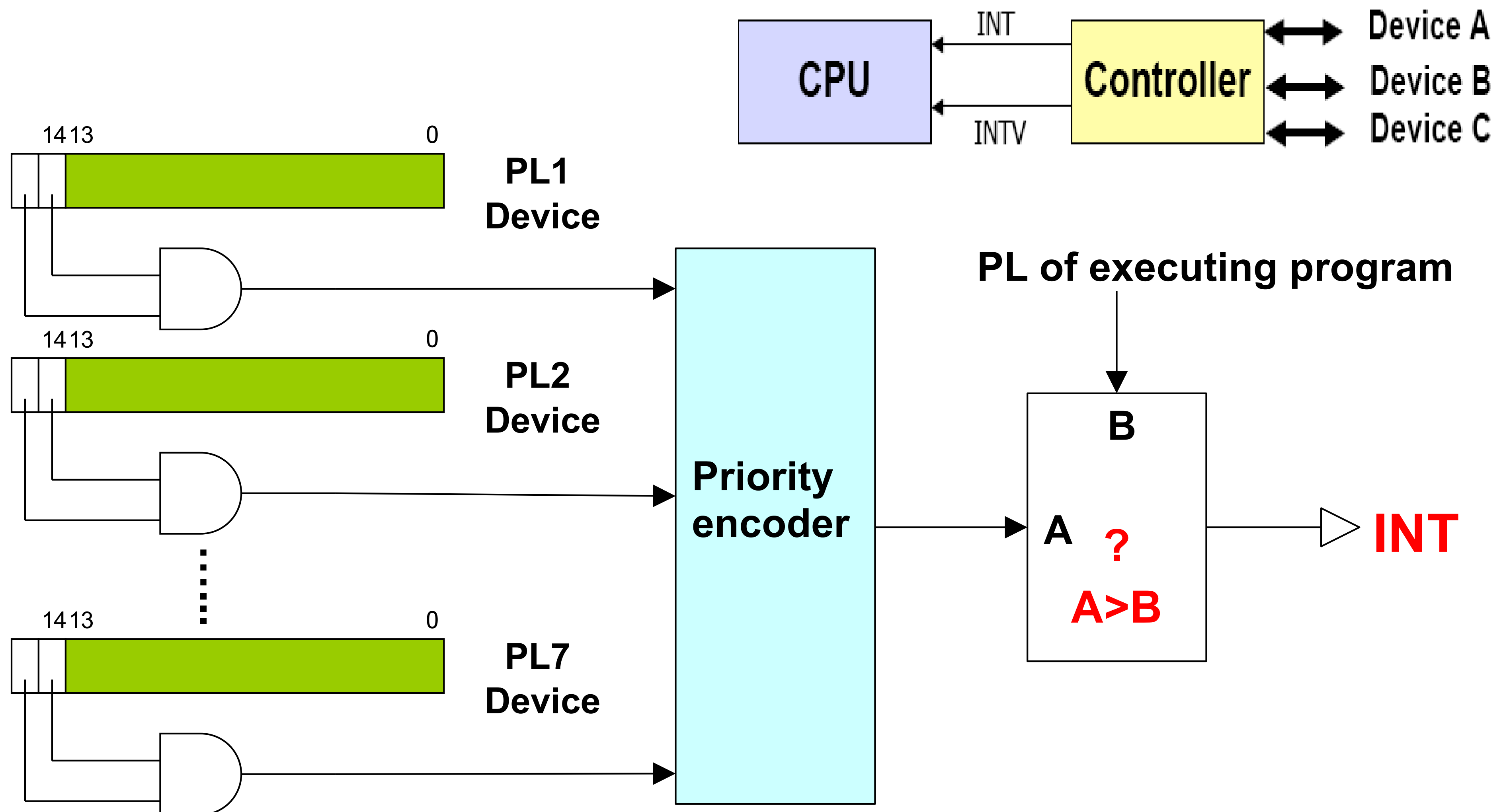
# The Processor State Register(PSR)




- **PSR[15]**
  - 0, privileged (supervisor) mode
  - 1, unprivileged(user)mode
- **The Priority Level of a program being executed**
  - PL[10:8]
  - 0(最低)~7(最高)

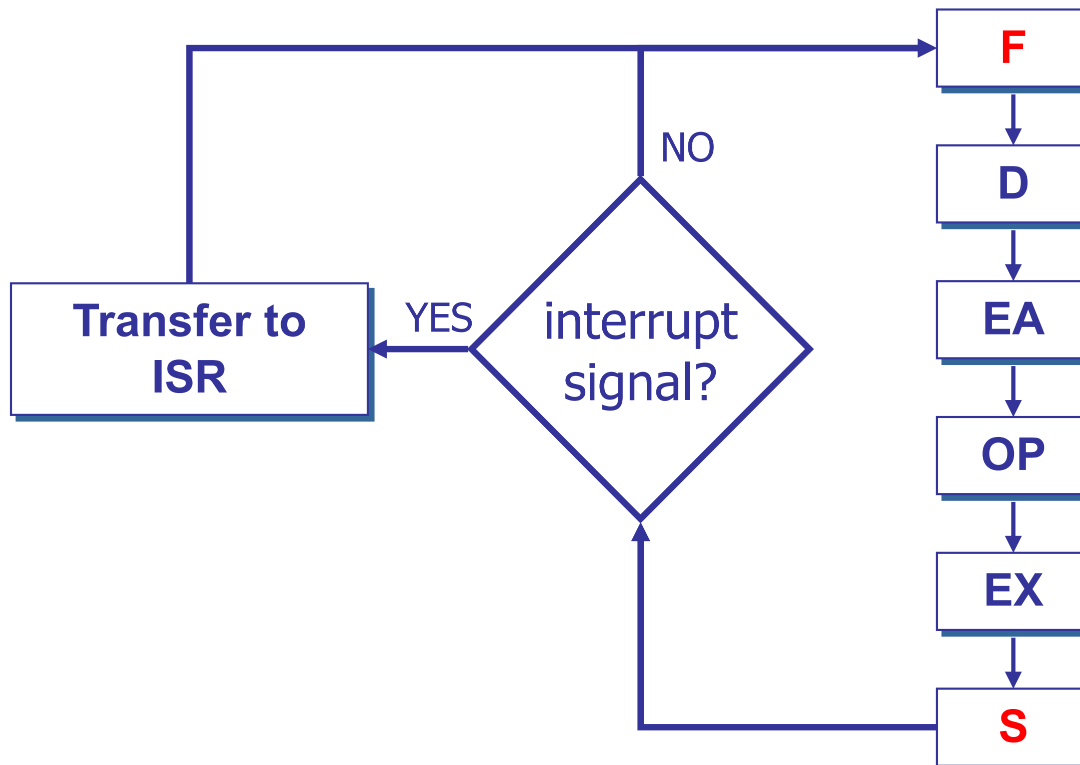
# Generation of the Interrupt Signal to CPU

- What if more than one device wants to interrupt?
  - External logic controls which one gets to drive signals



# Testing for Interrupt Signal

- CPU looks at signal between STORE and FETCH phases
- If not set, continues with next instruction
- If set, transfers control to interrupt service routine



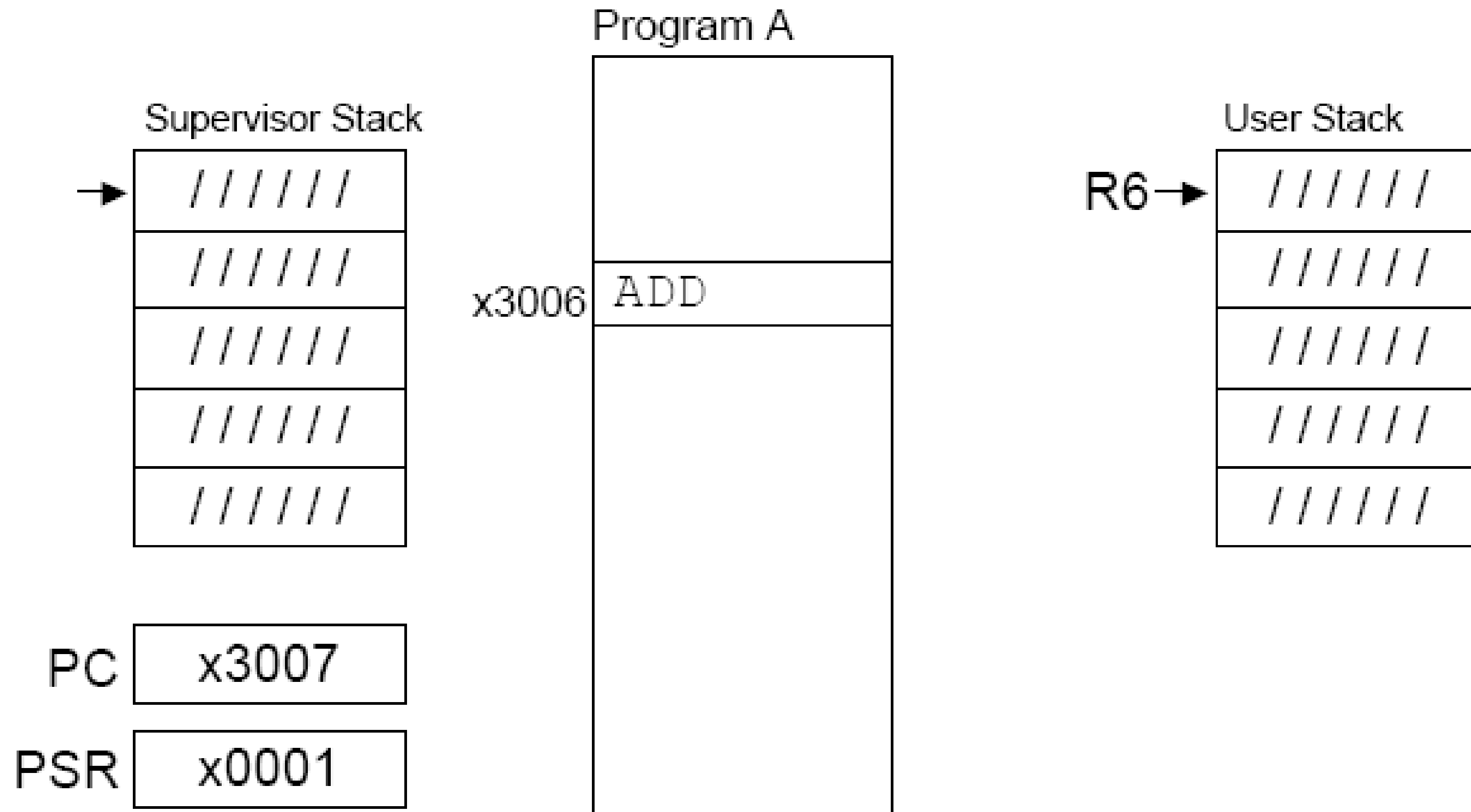
# How Does Processor Handle It?

- **Examines INT signal just before starting FETCH phase**
  - If INT=1, don’t fetch next instruction
  - Instead
    - Save program state (PC, PSR (privilege and CCs)) on stack
    - Update PSR (set privilege bit)
    - Index INTV into IVT to get start address of ISR (put in PC)
- **After service routine**
  - RTI instruction restores PSR and PC from stack
  - Need a different return instruction, because
    - RET gets PC from R7 and doesn’t update PSR
- **Processor only checks between STORE and FETCH phases -- Why?**

# Supervisor Mode and the Stack

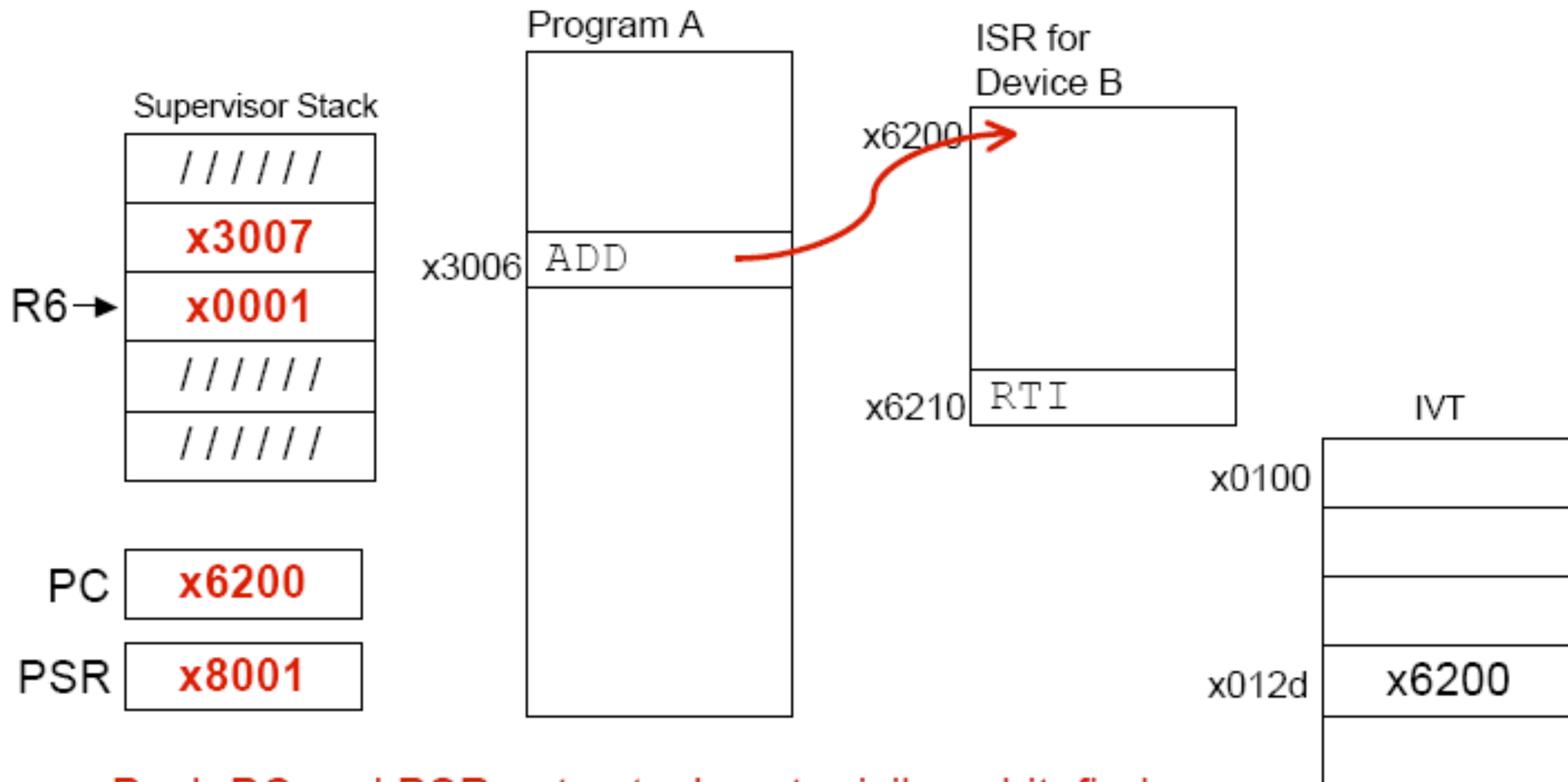
- **Problem**
  - PC and PSR shouldn't be saved on user stack
  - What if R6 is uninitialized?
  - What if user has set R6 to refer to OS memory?
  - User could see OS data (when trap returns)
- **Solution**
  - Create two versions of R6 (stack pointer) in register file
    - One is *user stack pointer* (what we've been using all along)
    - The other is *supervisor stack pointer*
  - Extra register file logic selects the appropriate register based on privilege bit in current PSR
  - Bottom line: OS code always uses its own stack

# An Example (1)

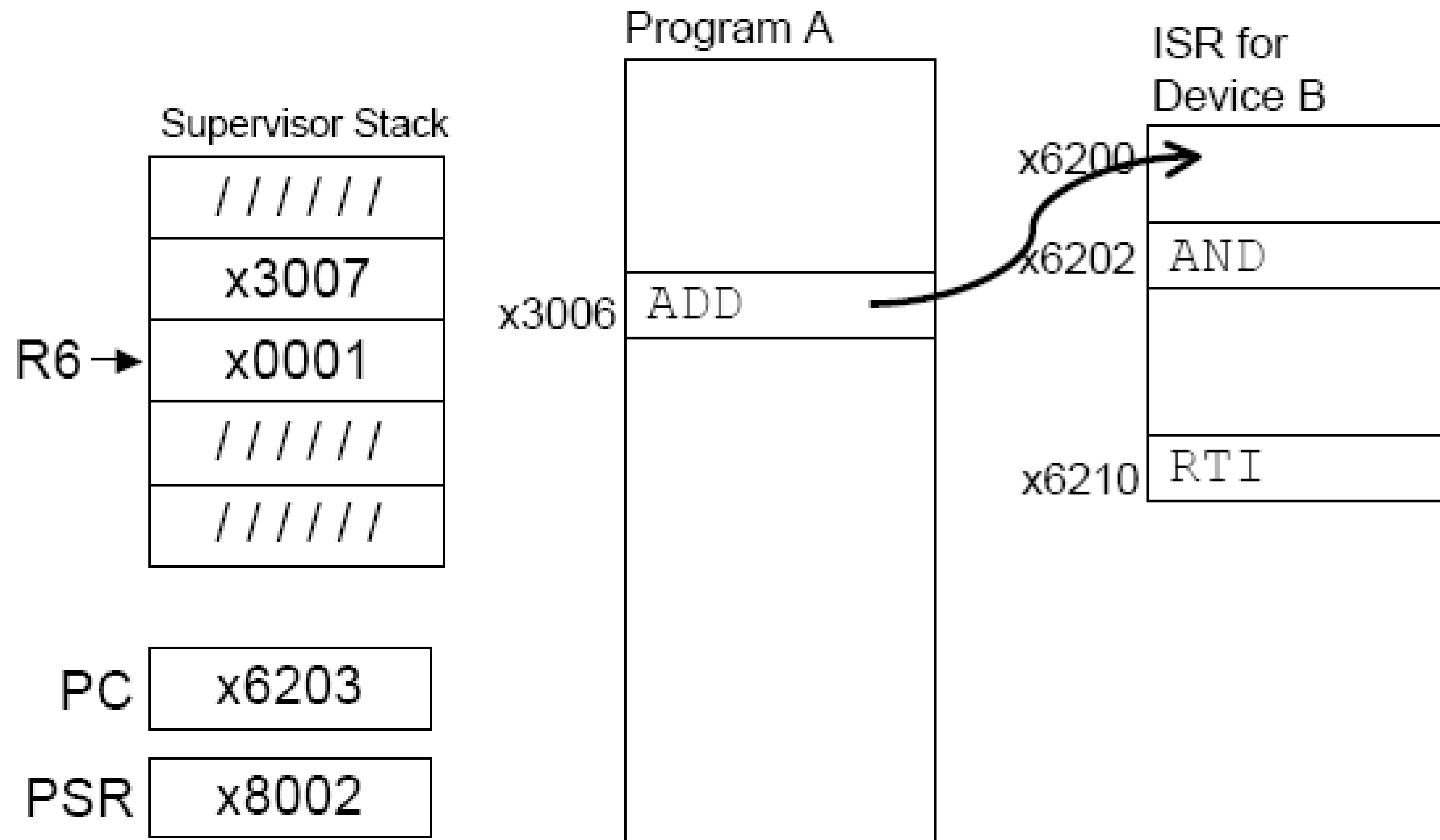



Executing ADD at location x3006 when Device B interrupts (INTV = x1d)

# An Example (2)

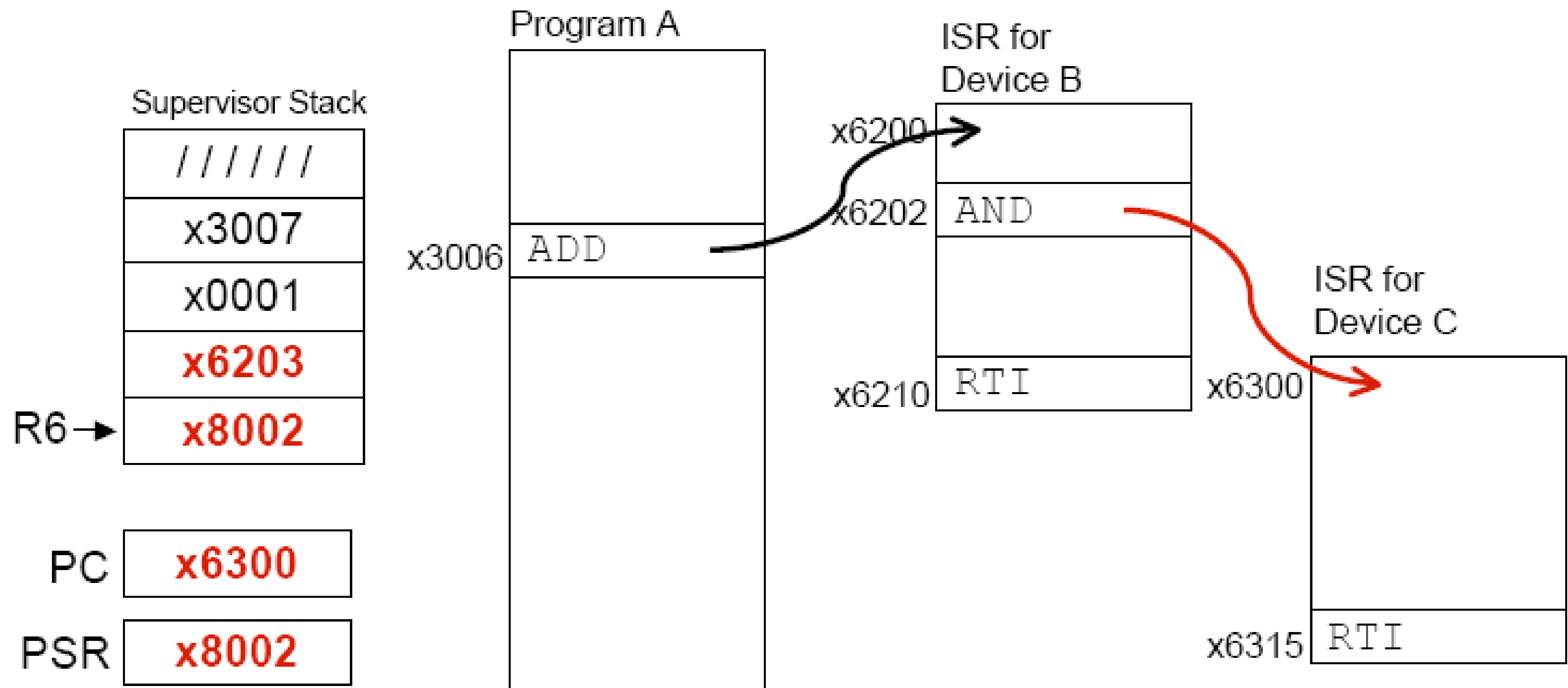


# An Example (3)



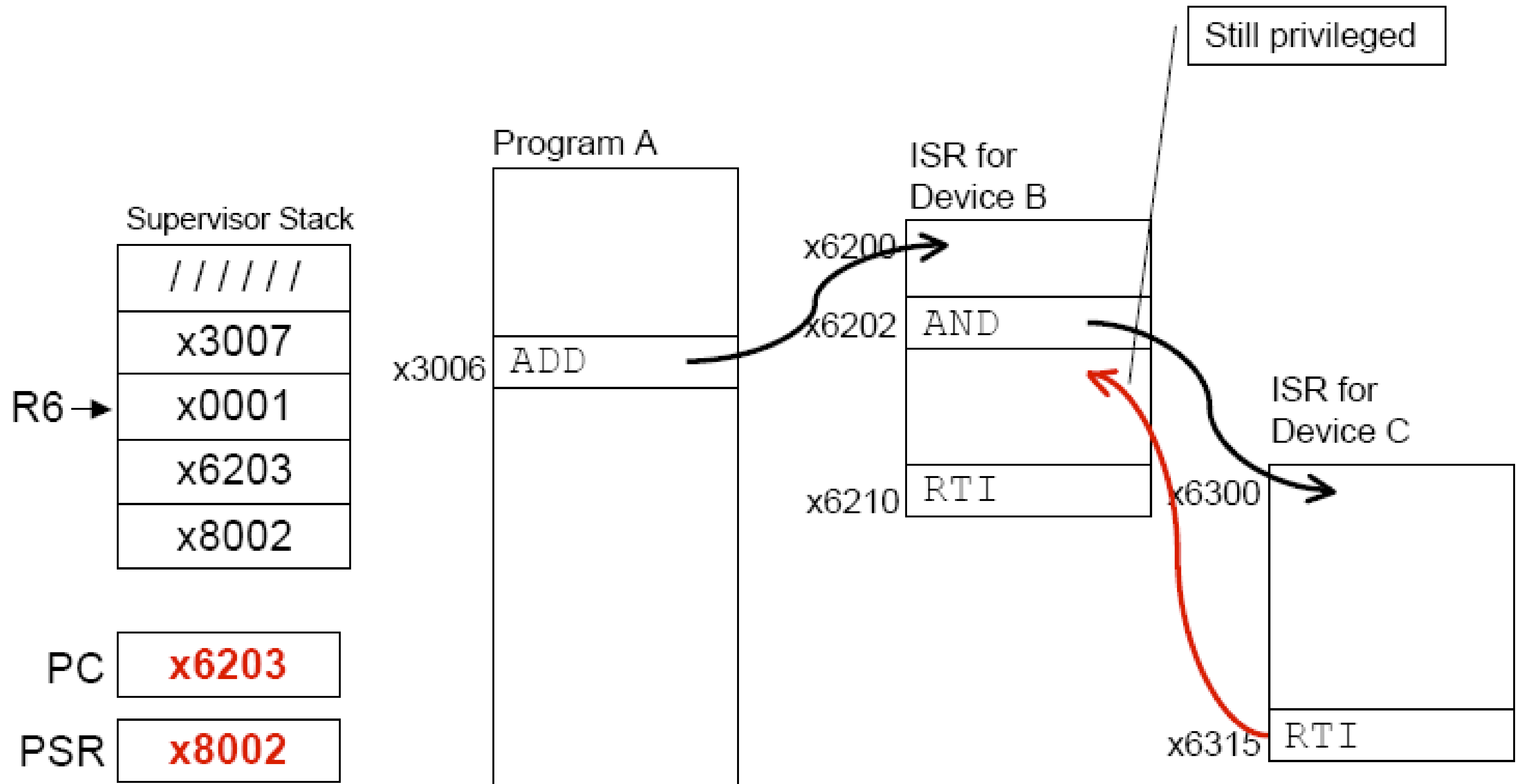

Executing AND at x6202 when Device C interrupts

# An Example (4)



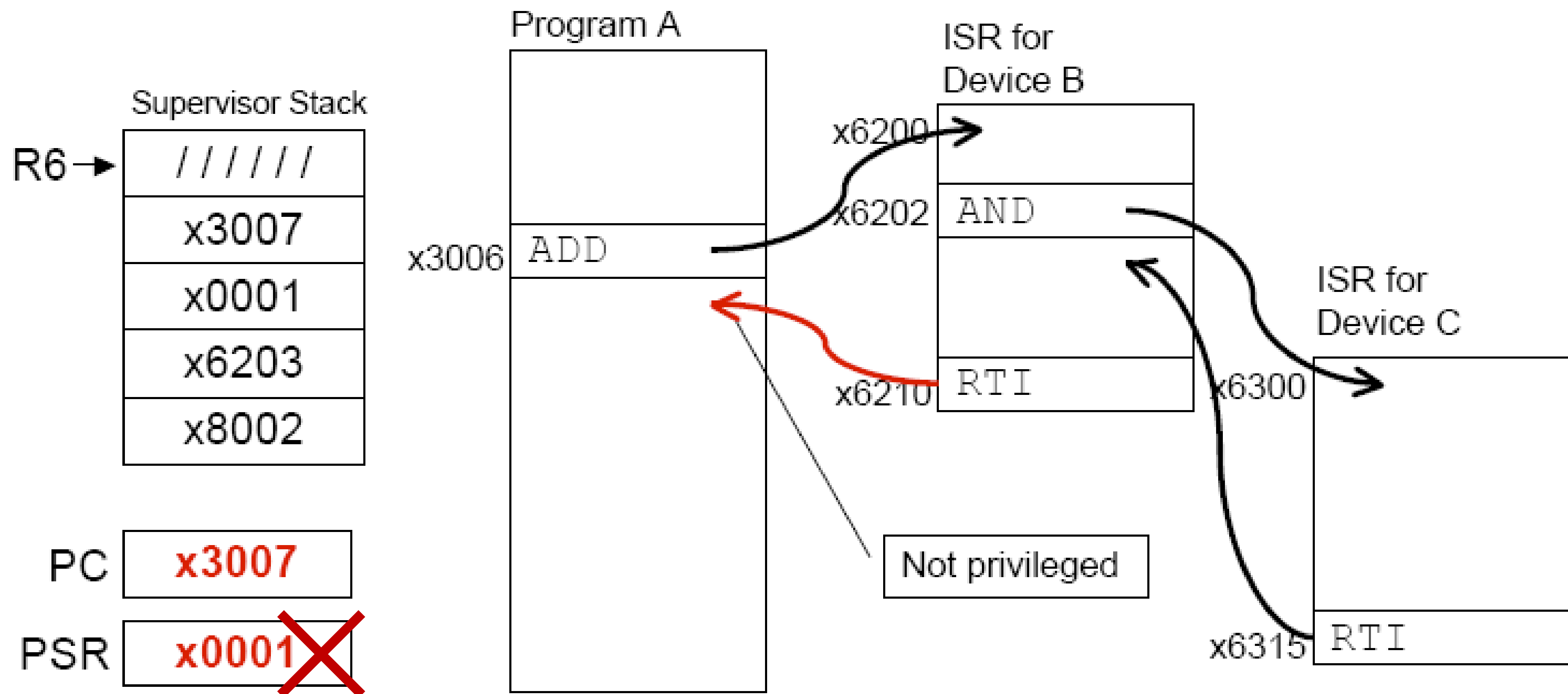

Push PC and PSR onto stack, set privilege bit, then transfer to Device C service routine (at x6300)

# An Example (5)




Execute RTI at x6315; pop PSR and PC from stack

# An Example (6)




Execute RTI at x6210; pop PSR and PC from stack; continue Program A as if nothing happened!