

University of Illinois at Urbana-Champaign  
Dept. of Electrical and Computer Engineering

# ECE 220: Computer Systems & Programming

---

## Interrupts and exceptions

# Memory-mapped I/O

---

In **memory-mapped I/O**, interaction with the I/O devices is controlled by our program

- Our program polls ready bits of I/O registers to see if the I/O devices are ready for interaction
- This leads to inefficiencies since our program effectively stalls until an I/O operation is complete

# Interrupt-driven I/O

---

In **interrupt-driven I/O**, interaction with the I/O device is controlled by the I/O device itself

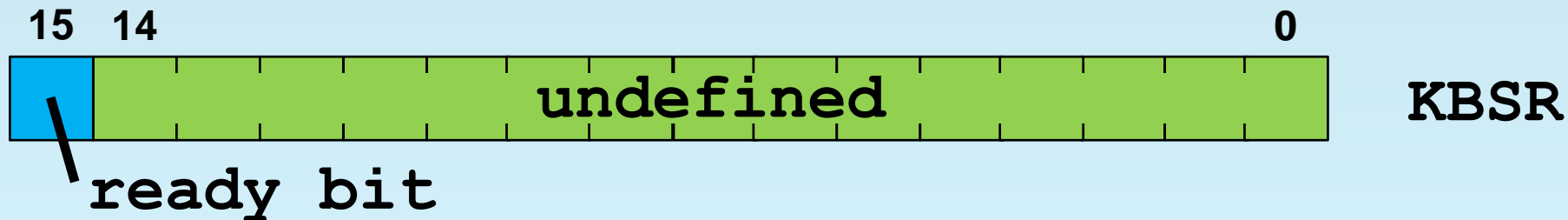
- An I/O device generates an **interrupt signal** to indicate that I/O device is ready with new I/O operation
- In response to this interrupt, the currently executed program stops its execution and the control is passed to some subroutine designed to handle the interrupt
- Once the subroutine processes the interrupt, the control is passed back to the program that was previously executed

---

Several things must be true for an I/O device to actually interrupt the processor

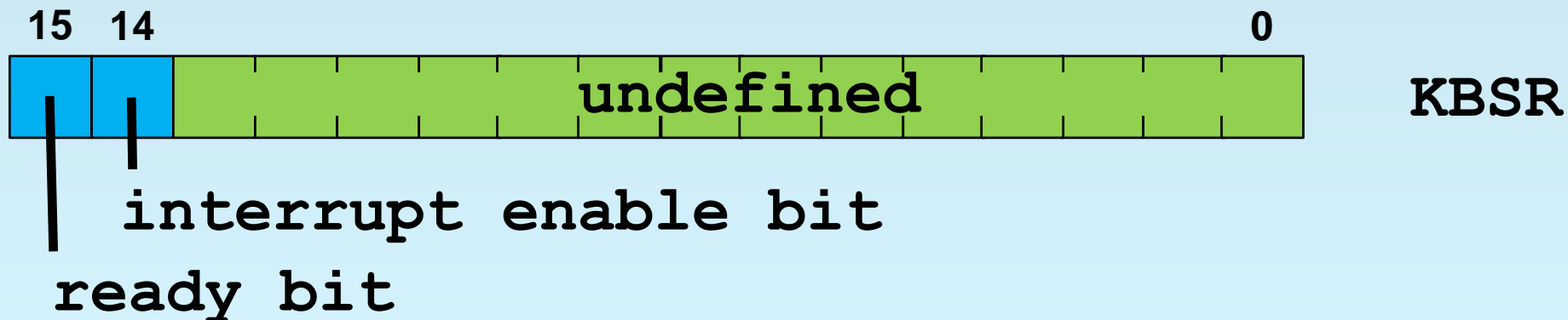
1. The device must want to request service.

This is indicated by **ready bit (KBSR[15] and DSR[15])**. If these bits are set, there is a new I/O request ready to be served



2. The device must have the right to request service.

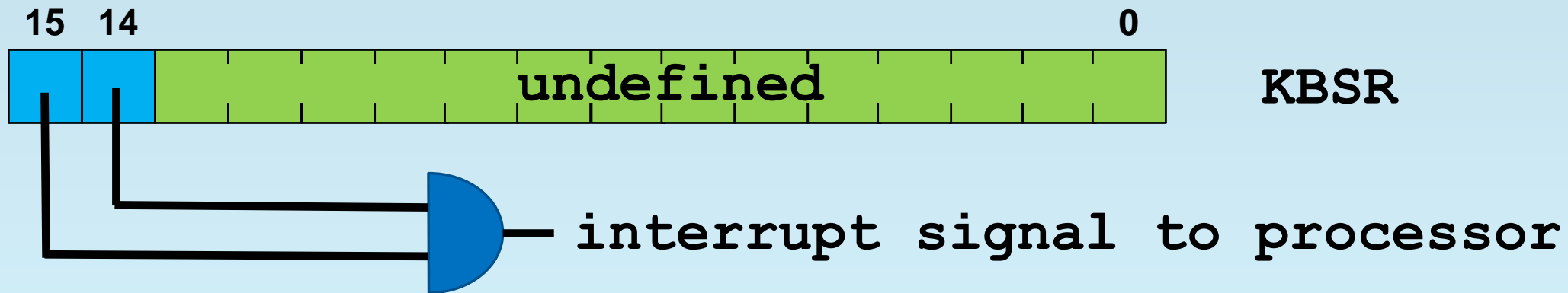
This is indicated by an **interrupt enable bit** (KBSR[14] and DSR[14]). If such bit is set by the processor, the processor wants to give the I/O device the right to request the interrupt service



# Interrupt Signal (INT)

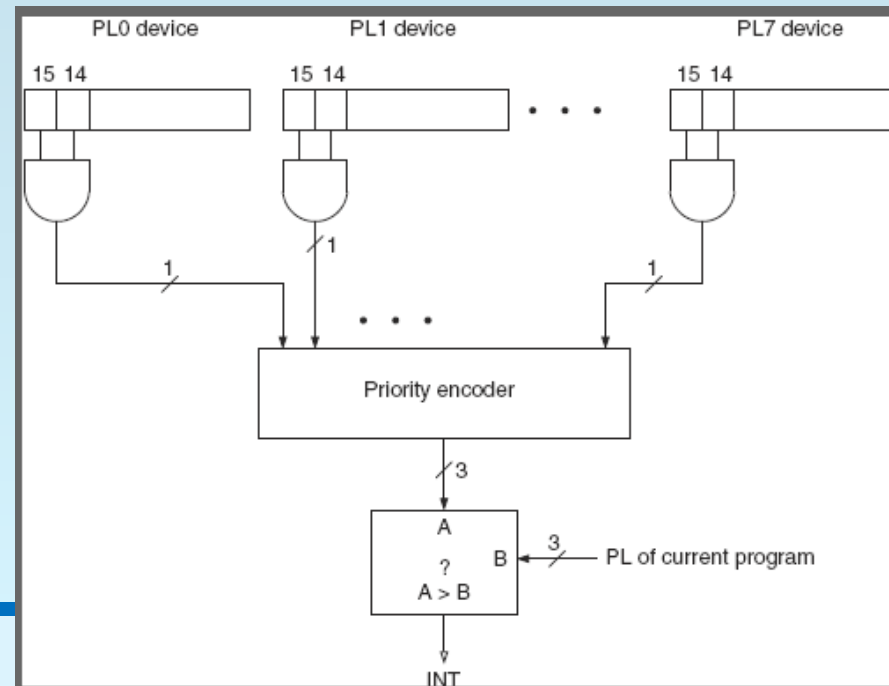
---

Ready bit and interrupt enable bit together are used to generate an interrupt

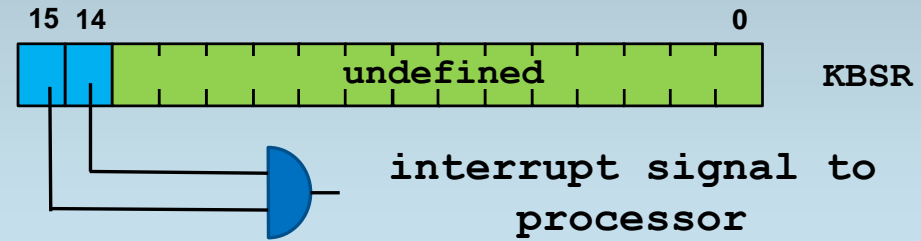
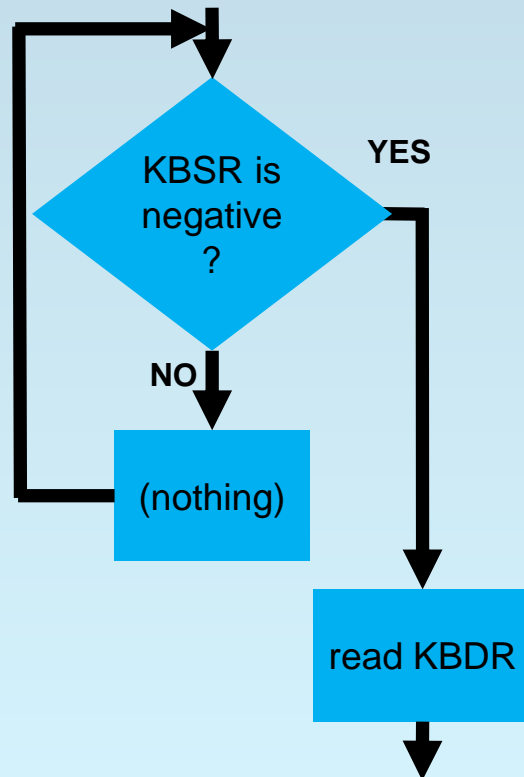
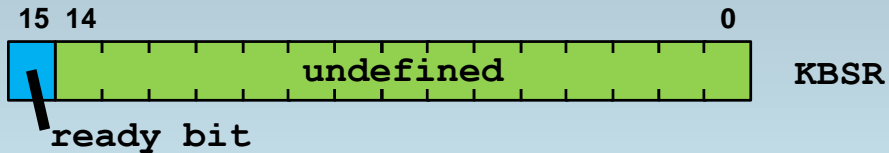


3. This request must be more urgent than the processor's current task.

A program is executed with some specified priority level, LC-3 has 8 such priority levels PL0..PL7.



# Polling vs Interrupt-driven I/O



$IE = 0$

- I/O device will NOT be able to interrupt
- Have to use polling

$IE = 1$

- Interrupt-driven I/O enabled
- Interrupt request generated as soon as Ready bit sets (e.g., a key is typed)



# Flow of Interrupt-driven I/O

---

## Stage 1: **Initiate** the interrupt

- 1.1 Stop the running program on any instruction
- 1.2 Save the state of the running program
- 1.3 Generate address of the interrupt servicing subroutine

## Stage 2: **Service** the interrupt

- 2.1 Transfer control to the interrupt subroutine
- 2.2 Execute the interrupt subroutine

## Stage 3: **Return** from the interrupt

- 3.1 Resume right where we left off

# Stage 1: Initiating the Interrupt

---

An I/O device generates an **interrupt signal (INT)** to indicate that I/O device is ready with a new I/O operation (e.g., a new character has been entered on the keyboard)

I/O device presents an 8-bit **interrupt vector (INTV)** which is used construct a memory address that contains the location of the interrupt handler in a **interrupt table**

# Interrupt Priority

---

For an interrupt to be served, the request must be more urgent than the processor's current task

LC-3 priority levels are PL0-PL7

- Higher is more urgent, e.g., keyboard is PL4

LC-3 maintains an **interrupt priority** in **PSR[10:8]**

Devices wanting to interrupt also have a 3-bit priority

The interrupt will be served only when program is running at priority  $< \text{PL4}$

# LC-3 Interrupt Table

---

Each device is associated with an 8-bit vector **INTV** to index an **interrupt vector table**

**Interrupt vector table** is in memory between **x0100** and **x01FF**

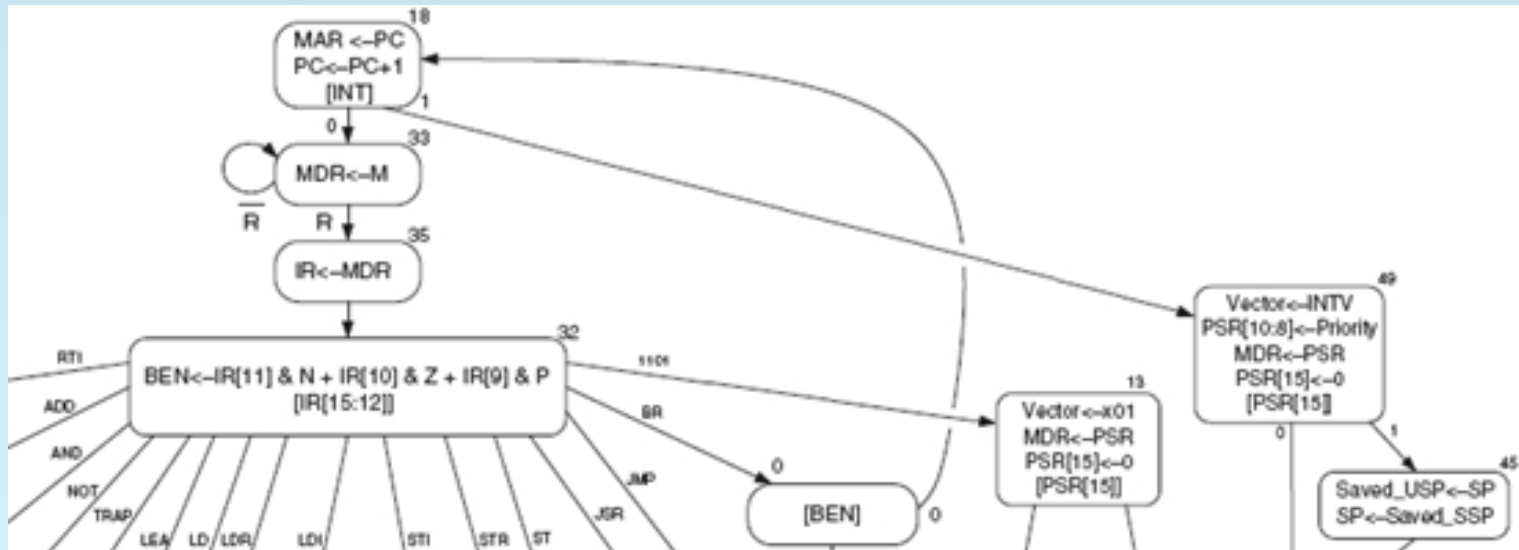
Each record in the **interrupt vector table** contains beginning address of service routine for handling interrupt

- Exception service routines (x0100-x017F)
- Interrupt service routines (x0180-x01FF)

# 1.1 Stopping the Execution of the Program

State 18 in LC-3 FSM is the only state in which the processor checks for interrupts

- If **INT=0** (no interrupt) go to State 33
- If **INT=1** go to State 49 (110001)

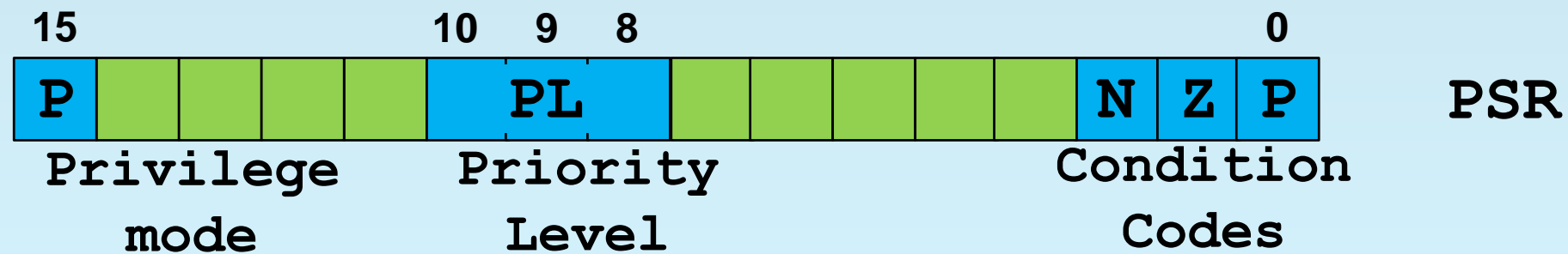


## 1.2 Saving the State - What

**PC** so that we can return to execute the next instruction after the interrupt has been served

**NZP** condition codes in case they are needed by a BR instruction later on

**Processor Status Register (PSR)**



## 1.2 Saving the State - Where

---

**Supervisor Stack** - a special region of memory used as the stack for serving interrupts

**Supervisor Stack Pointer (SSP)**

- **Saved.SSP**: Internal register to store SSP

**User Stack** - a stack accessed by user programs

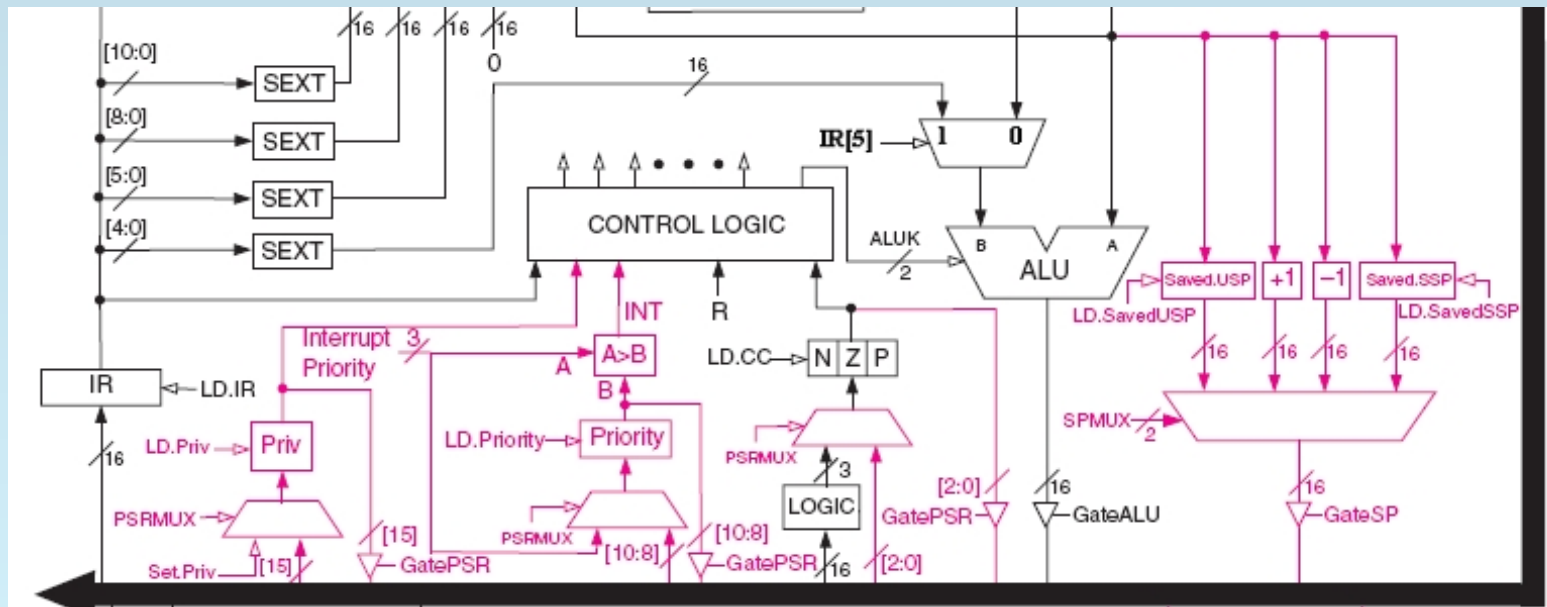
**User Stack Pointer (USP)**

- **Saved.USP**: Internal register to store USP

Access both stacks using R6 as the stack pointer.

When switching from **User mode** to **Supervisor mode**, save R6 to **Saved.USP**

# LC-3 Hardware to Support Interrupts





## 1.3 Generating ISR address

---

Set MAR to  $x01vv$ , where  $vv$  is 8-bit **interrupt vector (INVT)** from interrupting device

- e.g., for keyboard  $INTV=x80$ ,  $MAR \leftarrow x0180$

Load from memory:  $MDR \leftarrow MEM[x01vv]$

Set PC to MDR

# LC-3 FSM for Handling an Interrupt

Load PSR to MDR in preparation for pushing into Supervisory Stack

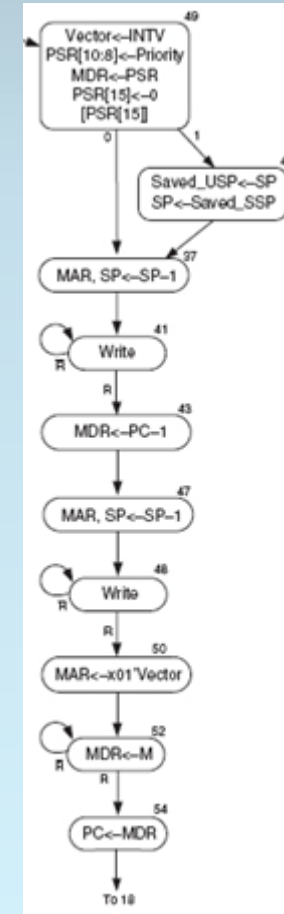
Record Priority Level and INTV provided by interrupting device

Test old PSR[15]

- If old PSR[15] = 1 then system was in User mode and hence save USP (R6) in Saved.USP, load R6 with Saved.SSP, go to state 37
- If old PSR[15] = 0 then system was in supervisory mode already

Save PSR, old PC to Supervisory Stack

Load PC with address of interrupt service routine



## Stage 2: Service the interrupt

---

PC contains the starting address of the ISR

The ISR will execute, and the requirements of the I/O device will be served

- For example, copy KBDR into some memory location
- Callee-save for general purpose registers

Only Input from the keyboard interrupt is implemented on LC-3

# Return from the Interrupt (RTI) Instruction

---

To return from **ISR**, we need special instruction, **RTI**

**RTI** is a privileged instruction

- Can only be executed in **Supervisor mode**
- If executed in **User mode**, causes an exception

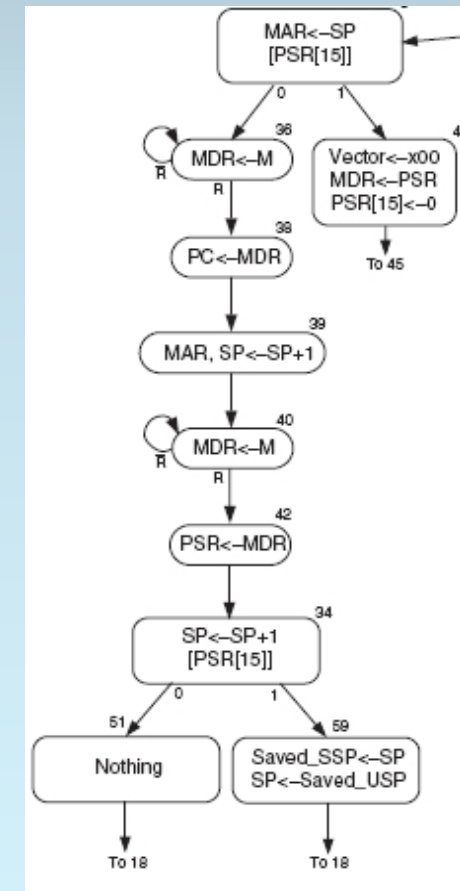
# Stage 3: Return from the interrupt (RTI)

If  $PSR[15] = 0$

- Restore PC and PSR
- Test old PSR[15]
  - If old  $PSR[15] = 1$  then the system returns to User mode and hence restore USP (R6) and store SSP
  - If old  $PSR[15] = 0$  then system continues to be in the Supervisory mode

If  $PSR[15] = 1$  (Privilege Mode Exception)

- Handle condition as an privileged mode violation
- Load Interrupt Vector with starting address of Privilege mode violation
- Go to State 45 to handle interrupt as if by INT



# Exception Handling

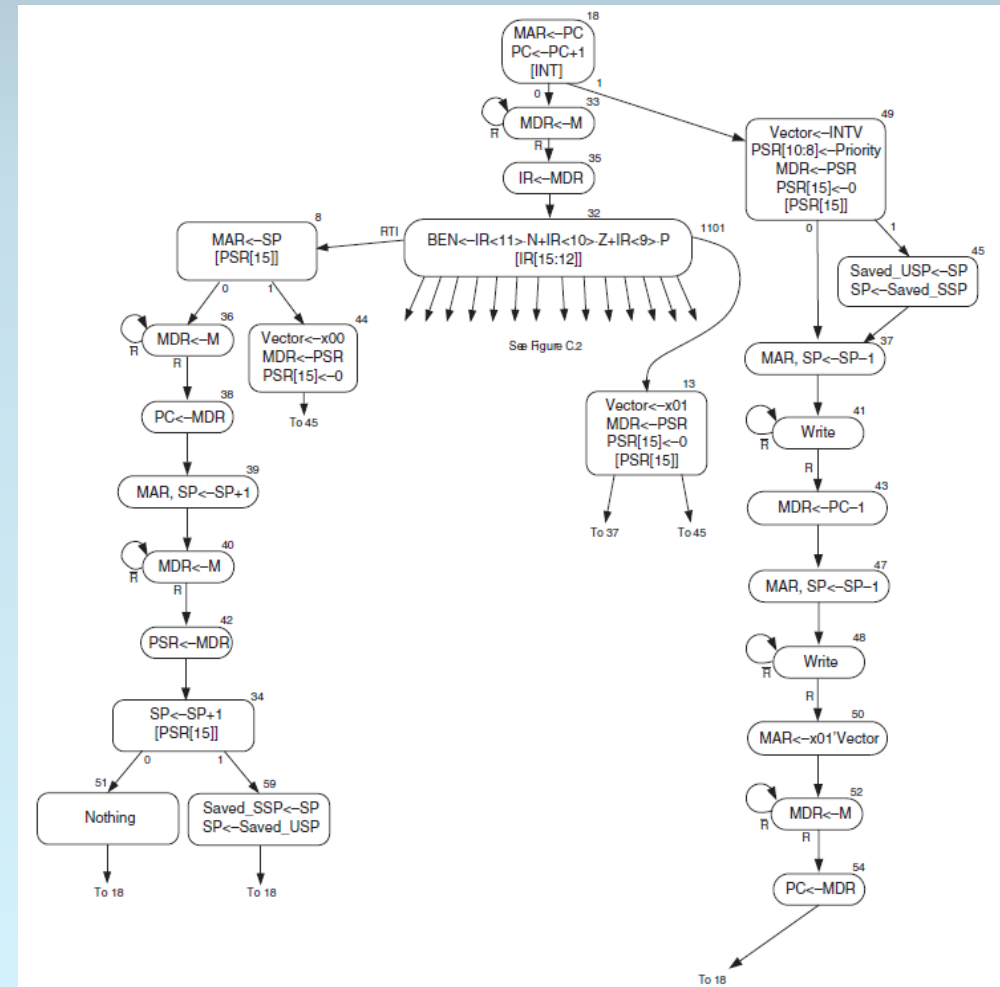
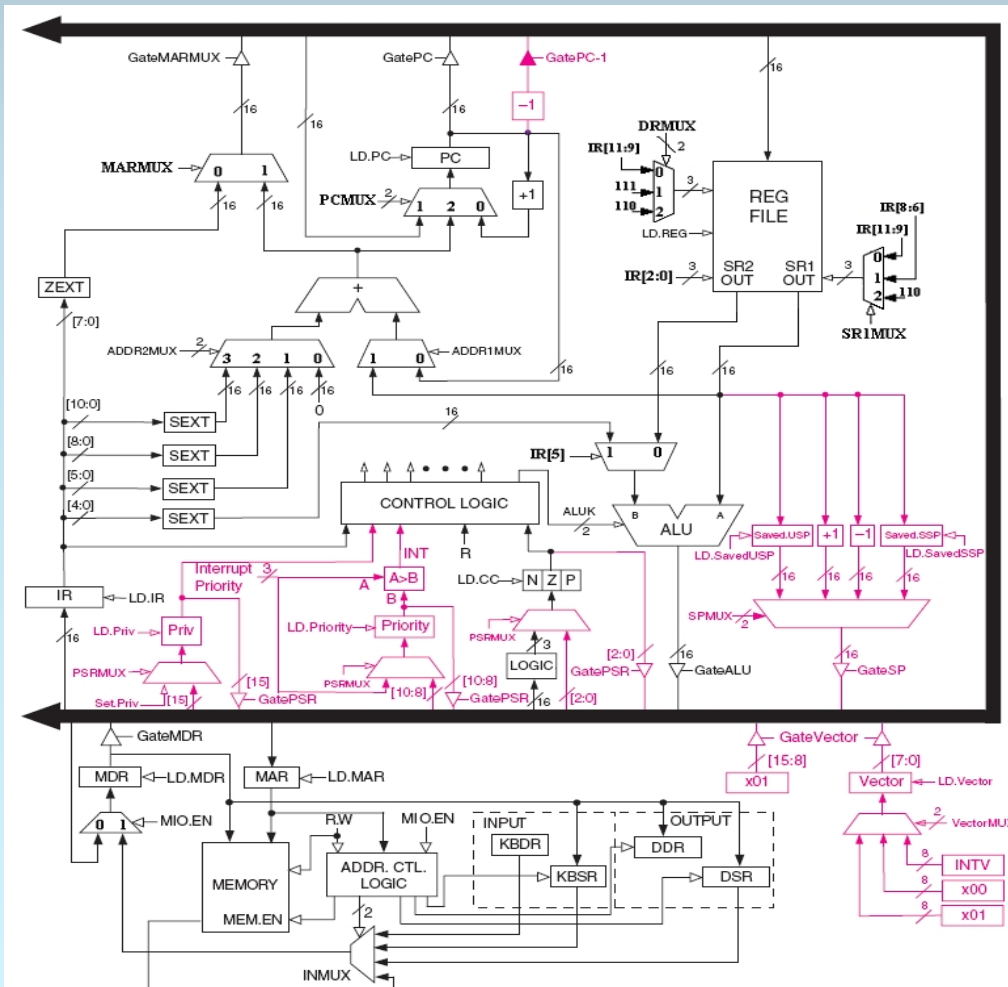
---

Only exceptions from

- Privilege mode violation
  - If processor encounter RTI when in User mode
- Illegal opcode
  - If  $IR[15:12] = 1101$  is true (unused opcode)

Exception handling is similar to interrupt handling

# Extended LC-3 Datapath and FSM



# Example

---

; run this with  
; <https://courses.grainger.illinois.edu/ece220/fa2019/lc3web/index.html>  
(based on <https://wchargin.github.io/lc3web>)

LEA R0, ISR\_KB

STI R0, KBINTV ; load ISR address to INTV

LD R3, EN\_IE

STI R3, KBSR ; enable IE bit of KBSR

; main code (just an infinite output loop)

AGAIN LD R0, NUM2 ; infinite loop printing '2'

OUT

BRnzp AGAIN

HALT



# Example - continued

---

; Interrupt Service Routine

ISR\_KB

ST R0,SaveR0 ;callee-save

ST R7,SaveR7 ;callee-save

LDI R0,KBDR ;read a charcter from keyboard and clear ready bit

HALT

LD R0,SaveR0

LD R7,SaveR7

RTI

EN\_IE .FILL x4000 ; enable IE 0100\_0000\_0000\_0000

NUM2 .FILL x0032 ; '2'

KBSR .FILL xFE00

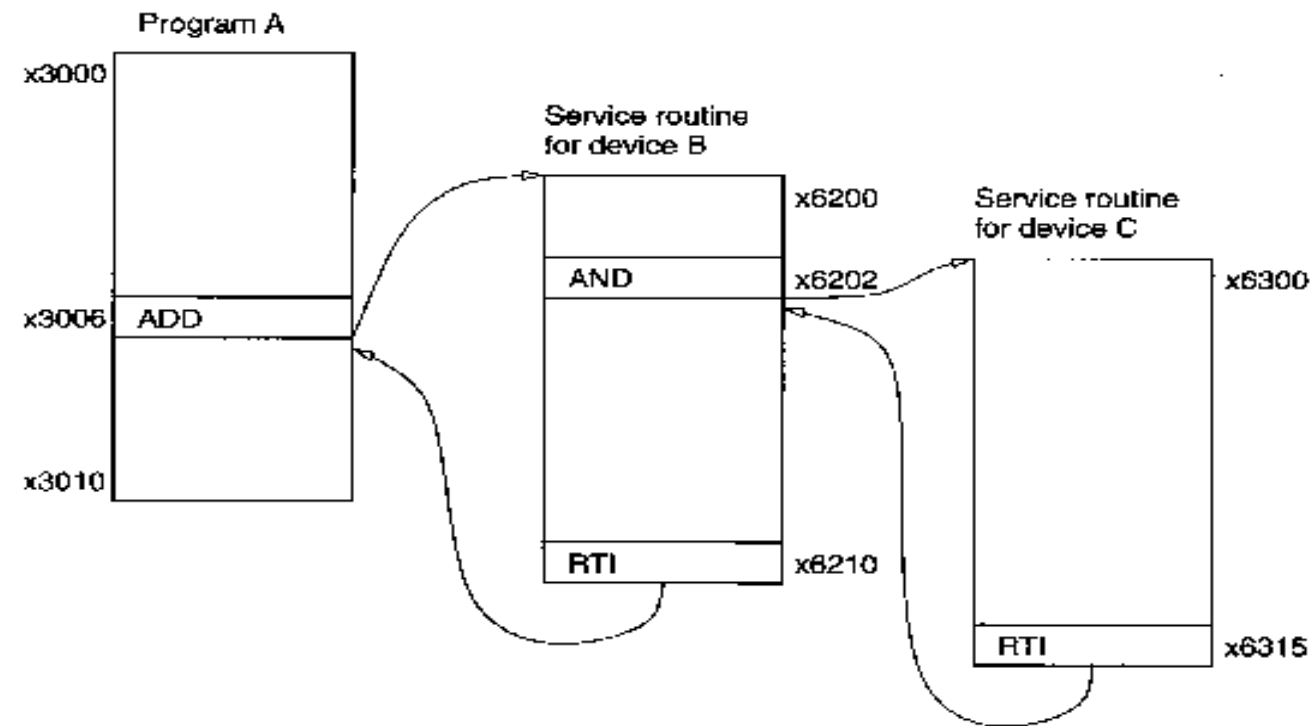
KBDR .FILL xFE02

KBINTV .FILL x0180 ; INT vector table address for keyboard

SaveR0 .BLKW #1

# Nested Interrupts

Example:



# Interrupt vector table

Addr	Data
x01F1	x6200
x01F2	x6300

## INTV

Device B = xF1  
Device C = xF2

PL: A<B<C

