

# ECE290 Fall 2012

## Lecture 24

Dr. Zbigniew Kalbarczyk

# Outline

- LC-3 Interrupt Driven I/O
- Interrupt and exception handling

# I/O: Connecting To Outside World

- So far we have learned how to
  - Compute with values in registers
  - Load data from memory to registers
  - Store data from register to memory
- Where does data in memory come from?
  - Input
- How does data get of the system for humans to use?
  - Output

# I/O Transfer Control

- Polling

- CPU keeps checking status register until new data arrive or device ready for next data
- “Are we there yet?”, “Are we there yet?”, ...

- Interrupts

- Device sends a special signal to CPU when new data arrive or device is ready for next data
- CPU can be performing other tasks instead of polling device
- “wake me when we get there”

# Interrupt-Driven I/O

- External devices can:
  - Force currently executing program to stop
  - Have the processor satisfy the device's needs
  - Resume the stopped program if nothing had happened
- Why Interrupt-Driven I/O?
  - Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation
  - e.g., process previous inputs while collecting current inputs

# Interrupts: Functional Requirements

- **What capabilities do we need?**
  - Stop the running program on any instruction
  - Vector to some other piece of code
  - Resume right where we left off

# Interrupts vs. Exceptions

- **Interrupts** are asynchronous and due to some outside influences beyond the currently running program
  - Examples: I/O, timer interrupt, etc.
- **Exceptions** are synchronous and caused by the currently running program
  - Examples: illegal instructions, protection violation, etc.

Next, let us focus on interrupts

# General Interrupt Handling

- At the software level, *handling an interrupt is like calling a subroutine*, only that the software state at the time of the call is less structured
  - there is no user control over when interrupts occur in comparison with subroutine calls in program
- **Save PC** so system knows where to return when interrupt service is done
- **Save all registers** so that they can be used, and restore them when interrupt service is done
- **Save the condition codes (NZP)** because they are set and tested in different instructions



# LC3 Interrupt Handling

- Not all interrupts are created equal (PL0-PL7)
  - LC3 maintains an interrupt priority (PSR[10:8])
  - Devices wanting to interrupt have a 3-bit priority
- **When interrupt happens**
  - Device asserts the interrupt request signal (INT) and presents an 8-bit interrupt vector (INTV)
    - INTV is used to construct a memory address that contains the location of the interrupt handler in a jump table
  - Process interrupts in supervisory mode (PSR[15]=0)
    - Processing in supervisory mode uses a different stack pointer (Supervisory Stack Pointer SSP) than in user mode

# LC3 Interrupt Handling (cont.)

- **When interrupt happens (cont.)**

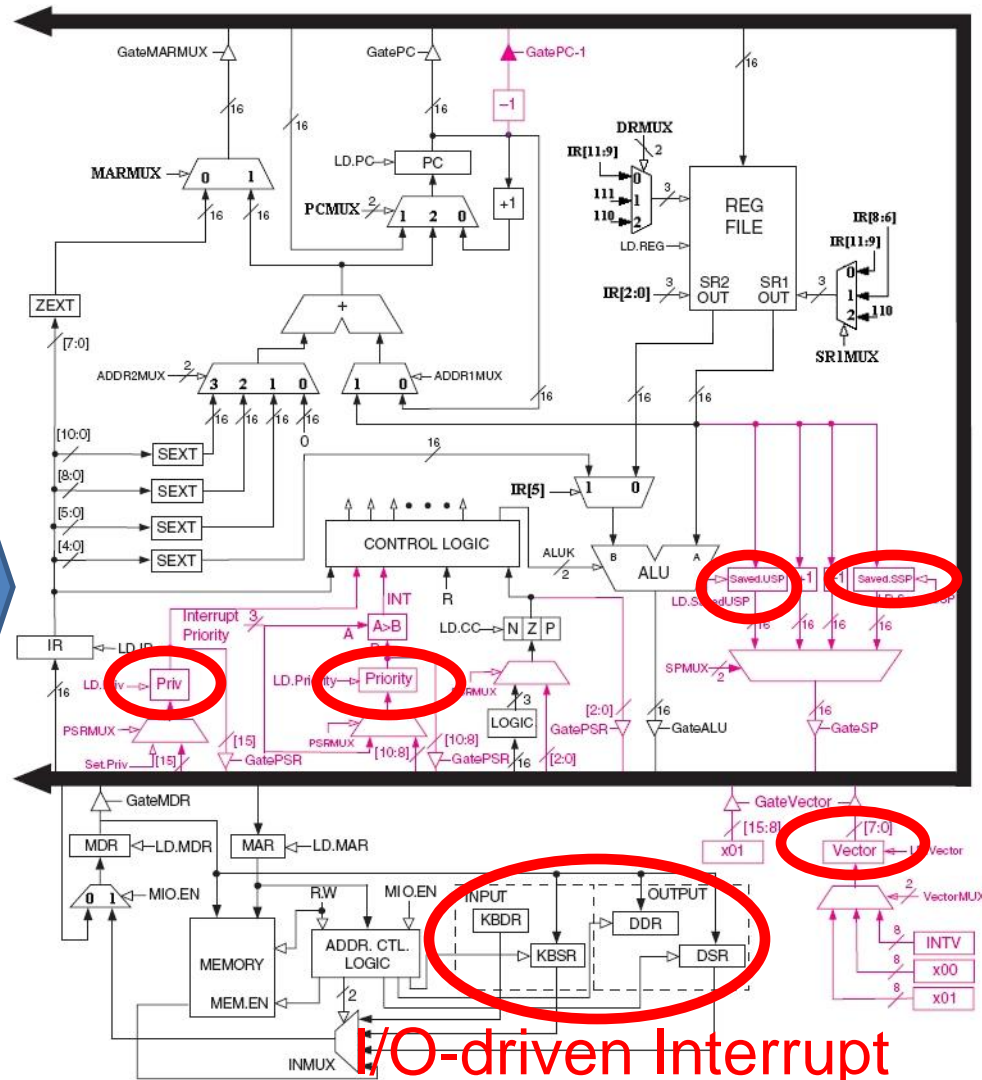
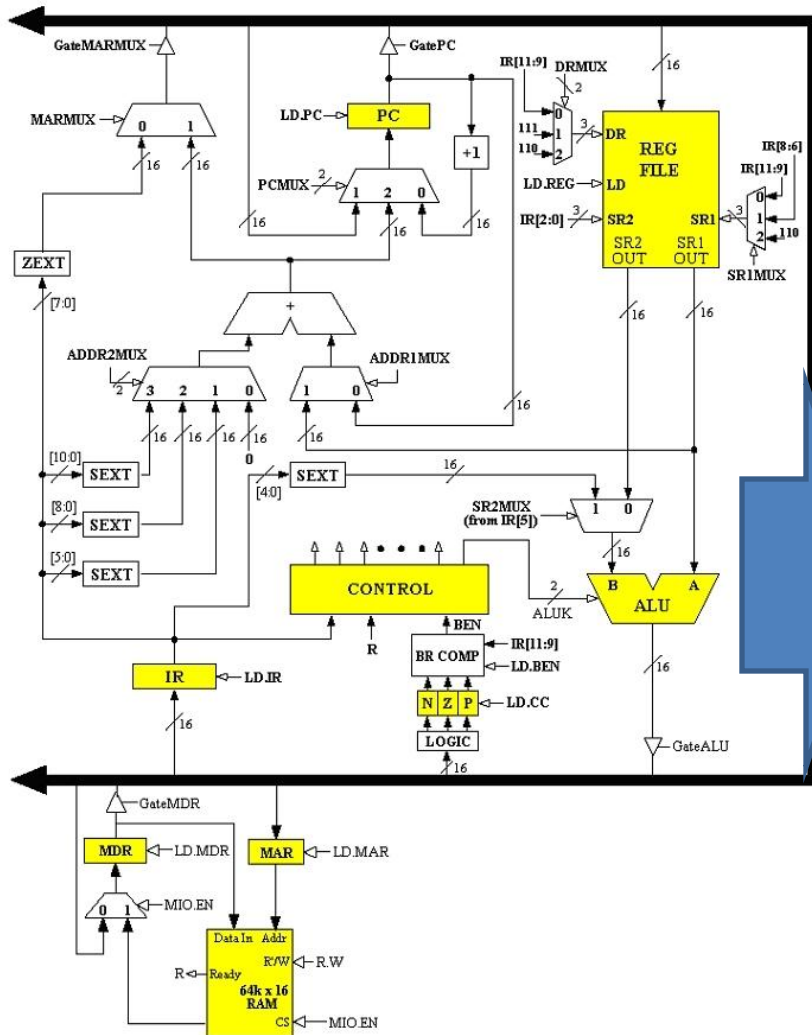
- Information saved onto supervisory stack before interrupts are processed
  - USP (User Stack Pointer)
  - PSR[15] (supervisory mode), PSR[10:8] (current priority mode), PSR[2:0] (condition code NZP)
  - PC - 1 (decrement PC because PC points to instruction past the one subverted)

# Where Are Interrupt Registers and Control Signals ?

## We need

- Interrupt vector register (INTV)
- Priority register
- Processor status register (PSR)
- Memory for pointers (user stack, supervisor stack)
- Temporary storage for PC and PSR
- Circuits to generate and handle interrupt signals

# Adding Interrupt Handling to CPU



# LC3 Interrupt Table

- Each device is associated with an 8-bit vector to index an interrupt vector table
- Interrupt vector table is in memory
  - Between x0100 and x01FF
  - Each contains beginning address of service routine for handling interrupt
- **Exception** service routines (x0100-x017F)
  - Handle exception events that prevent program from executing correctly
- **Interrupt** service routines (x0180-x01FF)
  - Handle service events external to running program

# I/O Interrupt Handling

- Only interrupt from keyboard in LC3
  - Priority level PL4 (out of 8 levels)
  - 8-bit interrupt vector (INTV=x80 located at x0180)
- Assumptions of the I/O interrupt
  - A program is running at priority level less than 4
  - Interrupt Enable is set (MIO.EN=1) for Keyboard Status Register (KBSR) when key is pressed

# Procedure for I/O Interrupt Handling

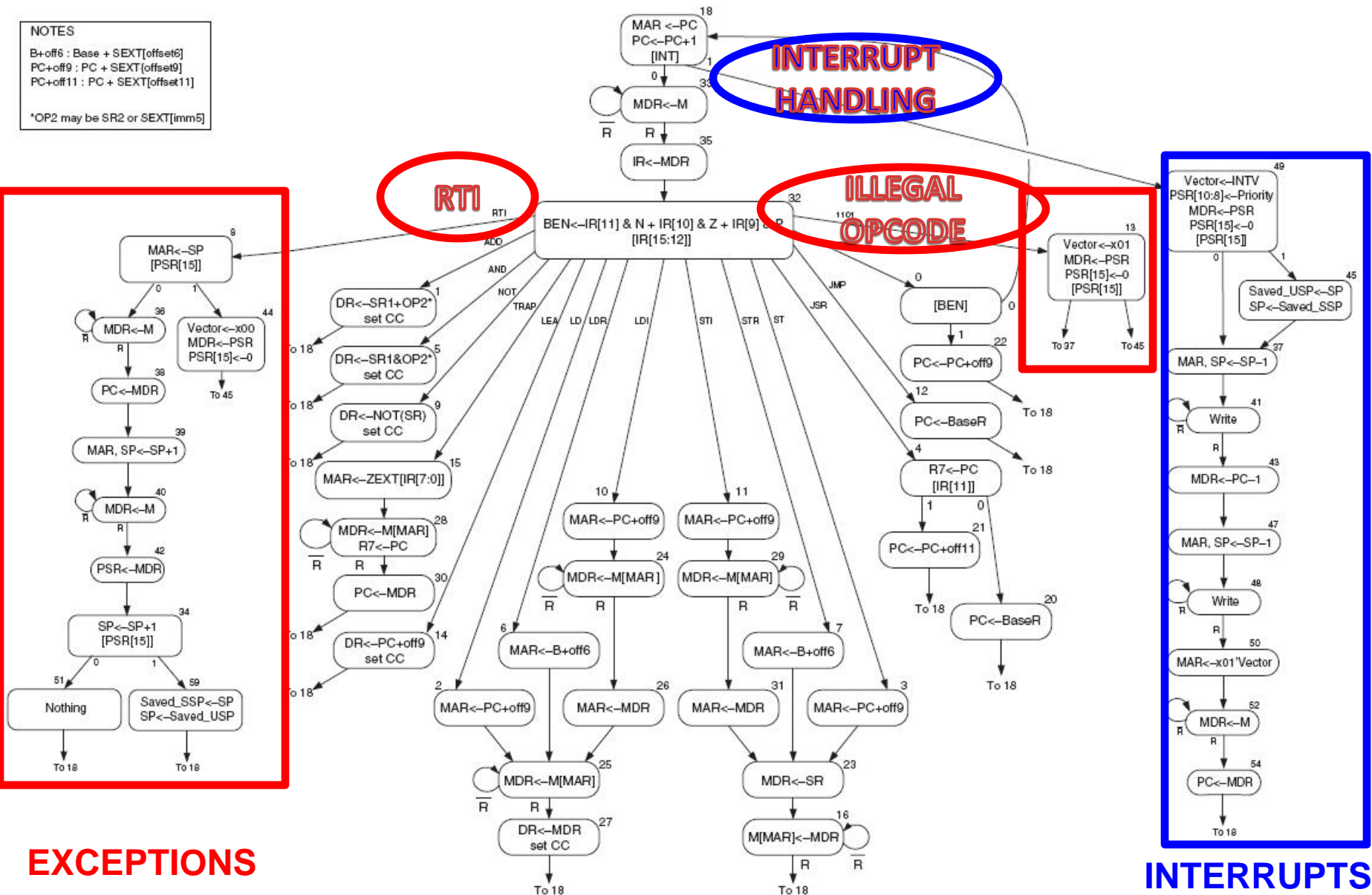
- **If** program is running at priority < PL4 **AND** MIO.EN=1 **AND** someone strikes a key on a keyboard **then**
  - Set Supervisory mode (PSR[15]=0)
  - Set Priority to PL4 (PSR[10:8] = 100)
  - $R6 \leftarrow$  Supervisory Stack Pointer (SSP)
  - Push Processor Status Register (PSR) and PC of interrupted program to Supervisor Stack
  - Expand 8-bit interrupt vector (x80) from keyboard to x0180 (address to interrupt table)
  - Load PC with address at x0180

# LC3 State Machine: Interrupt Support

## NOTES

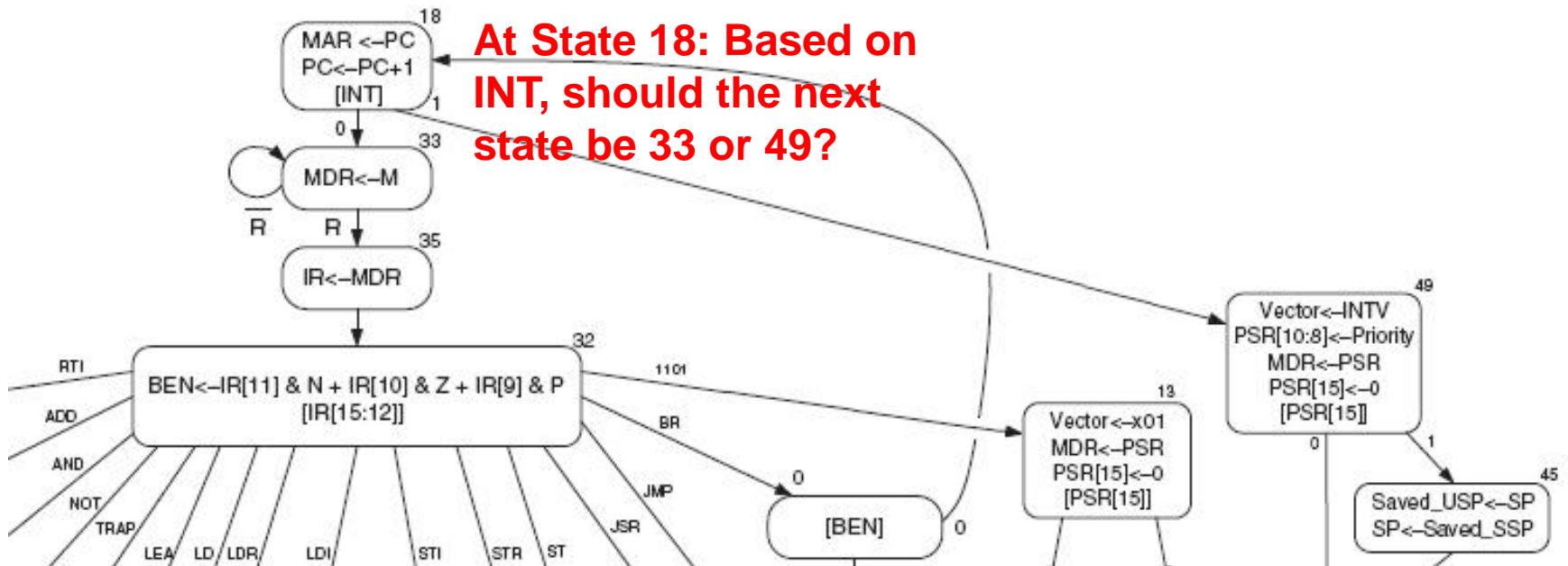
B+off6 : Base + SEXT[offset6]  
PC+off9 : PC + SEXT[offset9]  
PC+off11 : PC + SEXT[offset11]

\*OP2 may be SR2 or SEXT[imm5]



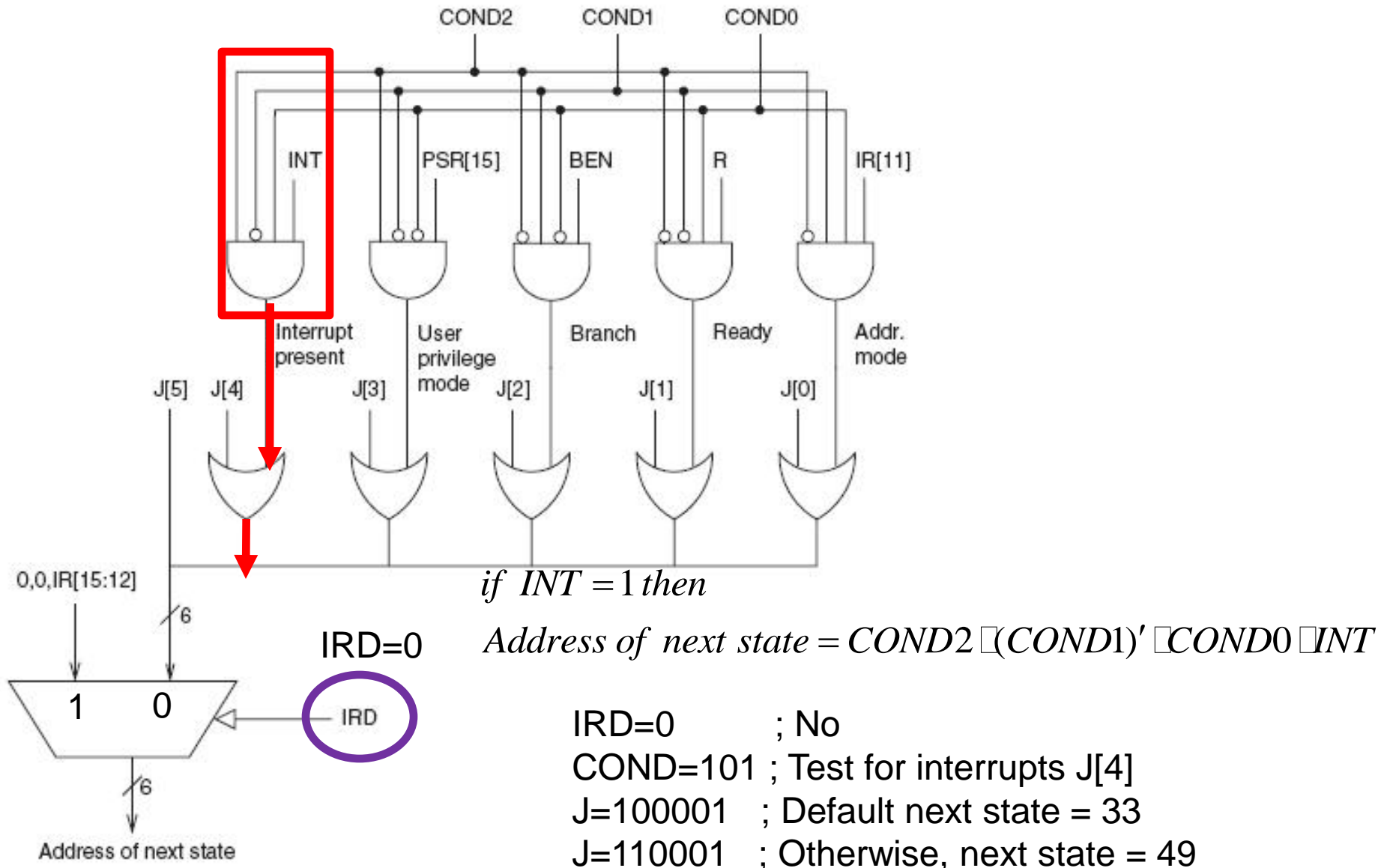


# Checking for Interrupts



- Best to check for interrupts before a new instruction is executed
- State 18 is the only state in which the processor checks for interrupts (before 'begin fetch' phase)

# State 18: Micro-sequencer Control




# Interrupt Micro-Instruction

In State 18, Check for INT

- If INT=0 (no interrupt) go to State 33
  - Next state (NS) = 100001 (33)
- If INT=1 go to State 49 (110001)

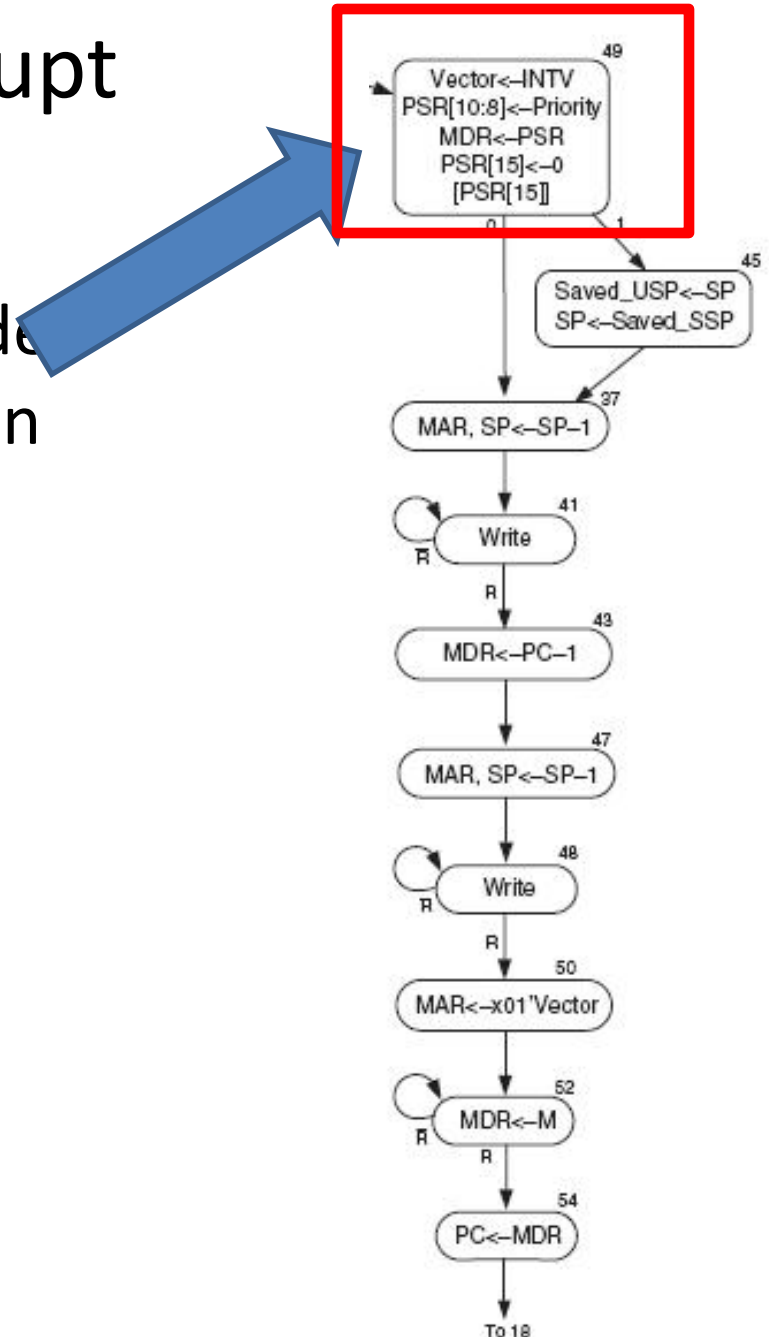
**Control Address 49**

48	47	46	45	44	43	42	41	40	39	38			0
0	1	0	1	1	1	0	0	0	1				
IRD	COND					J	CONTROL SIGNALS						
If INT=1 then NS = 110001 (49)													

Next, what to do in State 49?

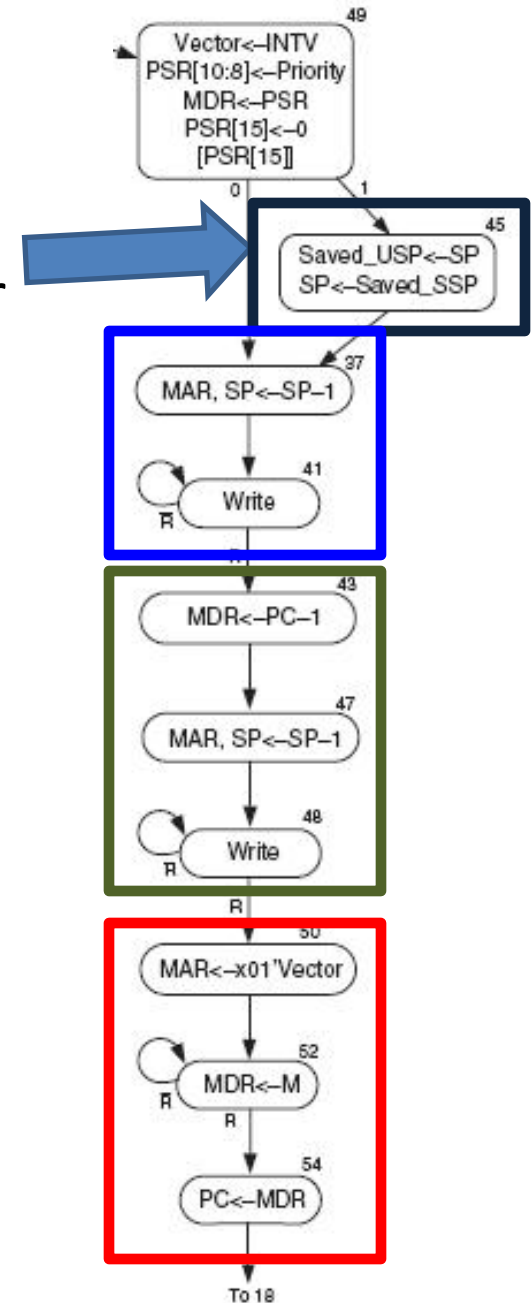
# Processing an Interrupt

- Load PSR (with privilege mode, priority level, and condition code of interrupt program) to MDR, in preparation for pushing into Supervisory Stack
- Record Priority Level and INTV provided by interrupting device



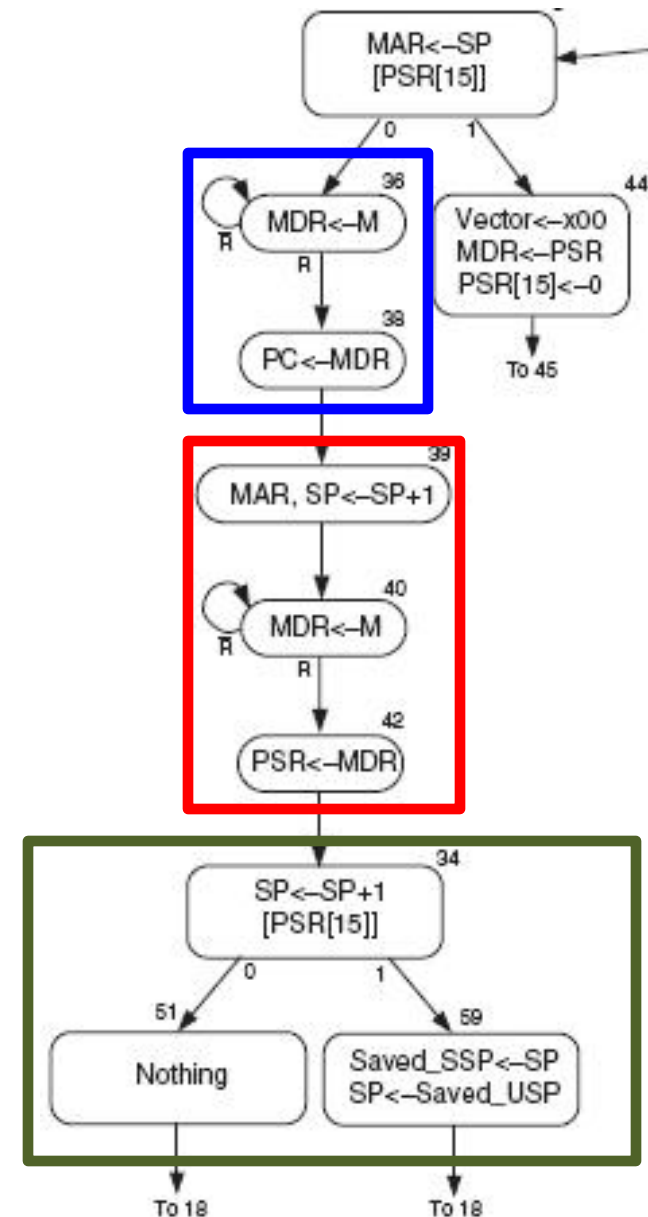
# Processing an Interrupt

- 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

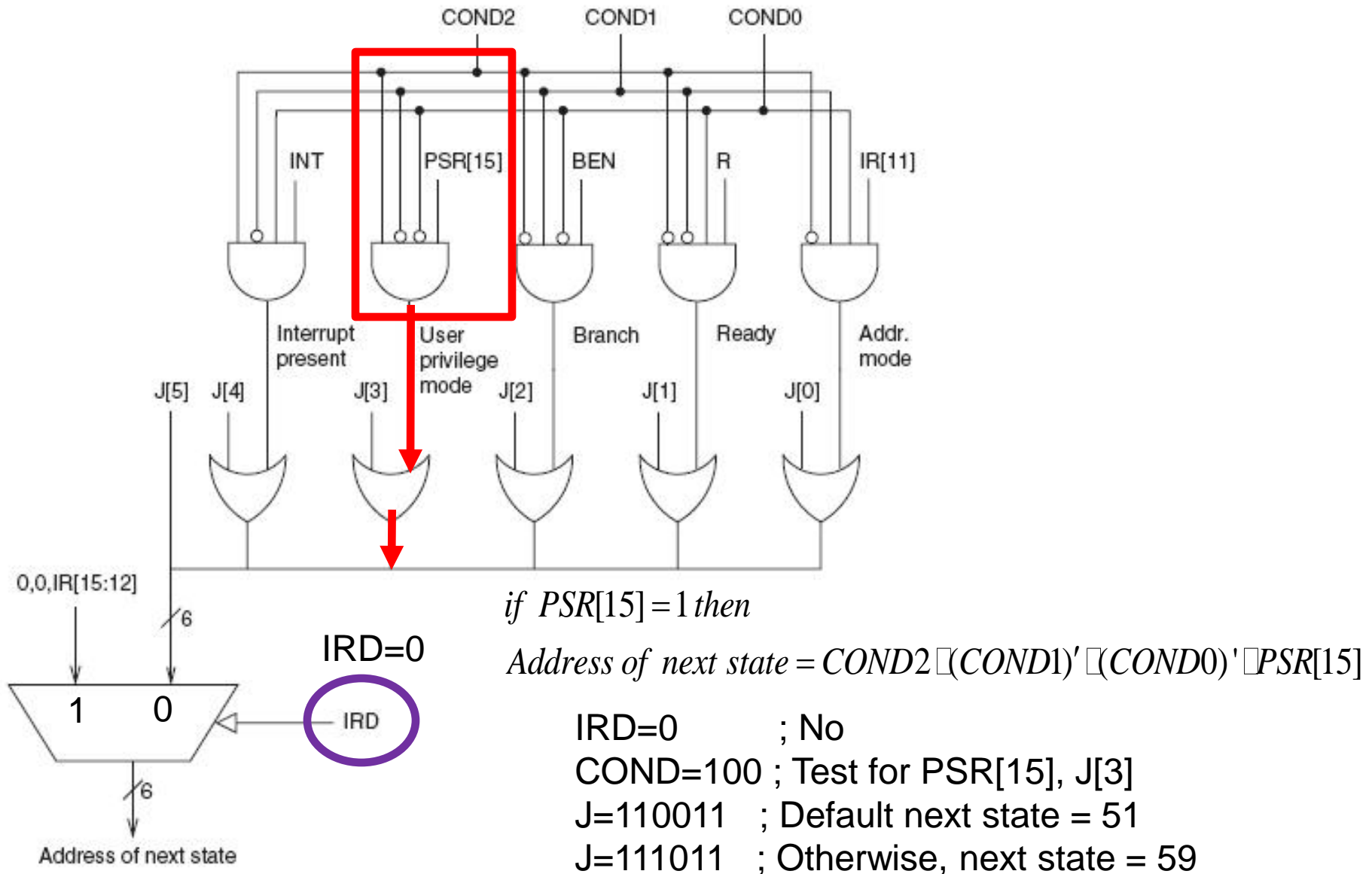


# Return from Interrupt (RTI)

- Restore PSR and PC
- If PSR[15] == 0 then RTI continues
  - Restore PC first
  - Restore PSR next
  - Micro-sequencer control
    - IRD=0; No
    - COND= ???; Test PSR[15], J[3]
    - Default 51/else 59
    - State 59: Restore SP to Saved\_USP if returning to user mode

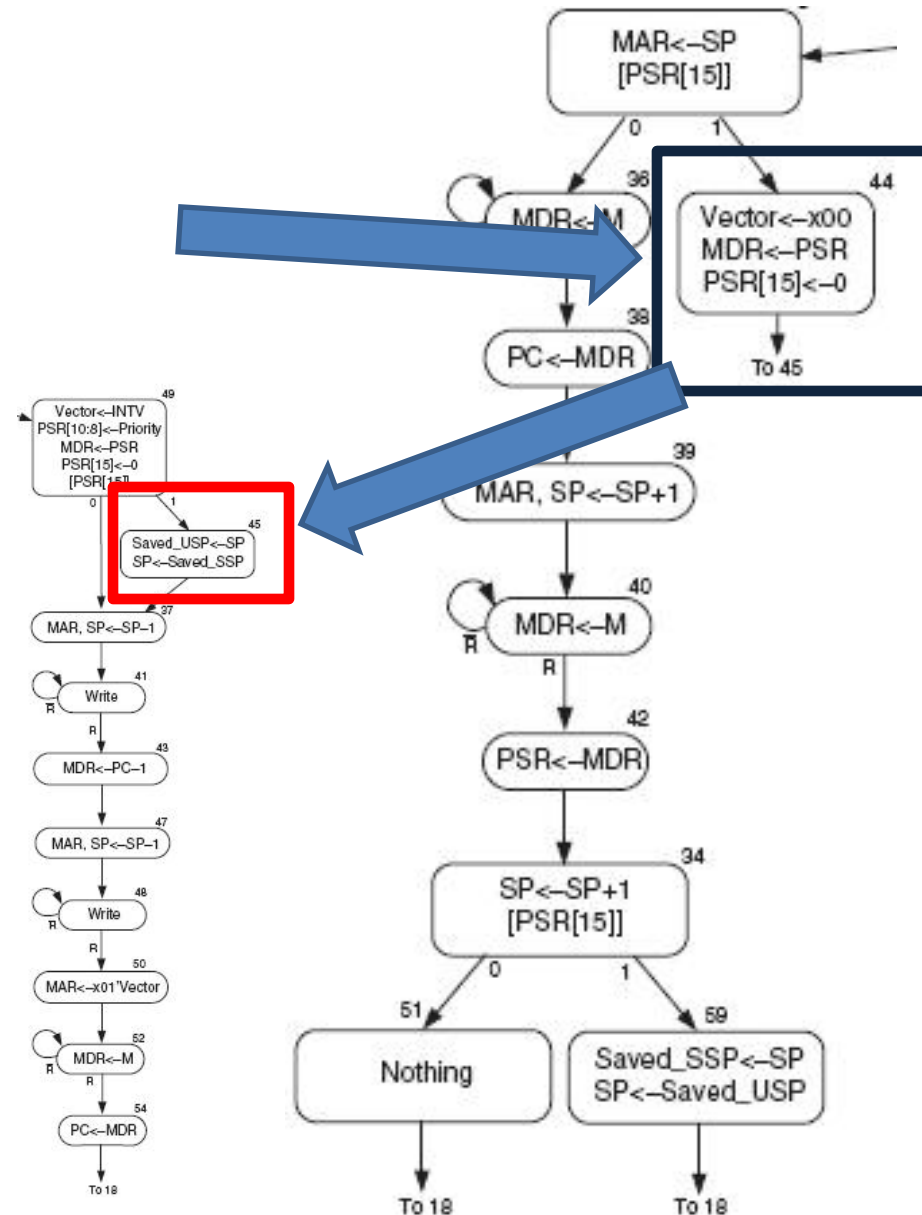


# State 34: Micro-sequencer Control



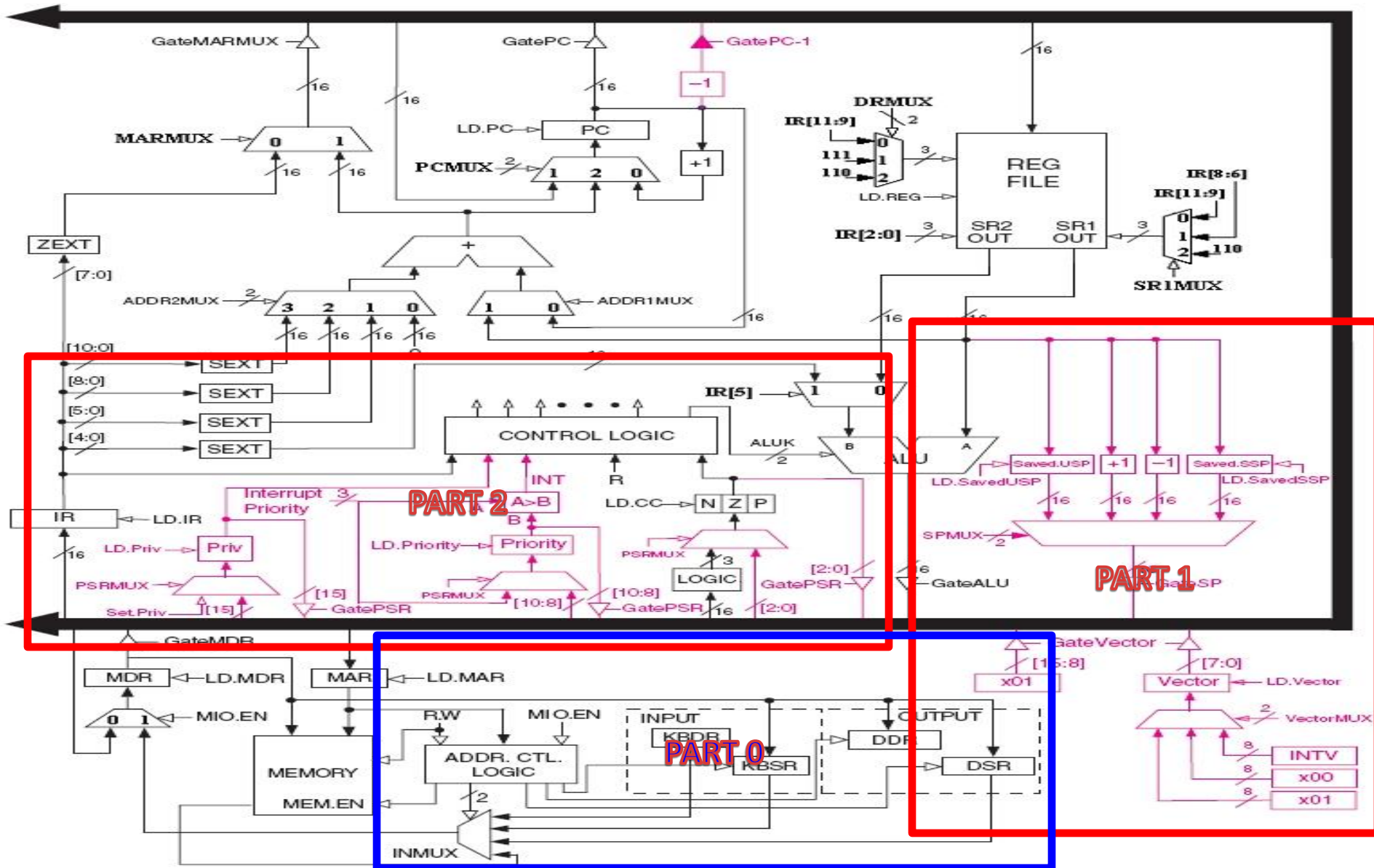
# Return from Interrupt (RTI) – cont.

- 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 (see previous slides)





# LC3 Data Path for Supporting Interrupts



# LC-3: Micro-Architecture

