Because learning changes everything.®

From Bits and Gates to C and Beyond

I/O

Chapter 9

© 2020 McGraw Hill. All rights reserved. Authorized only for instructor use in the classroom.
No reproduction or further distribution permitted without the prior written consent of McGraw Hill.

Details of Input and Output

Operating System Basics

- Privilege, Priority, and Memory Space

I/O Basics

- Devices, Device Registers, Memory-Mapped I/O, Interrupts

TRAP Instruction

- Invoking a system call

Interrupt-Driven I/O

- Hardware-triggered service routines

© McGraw Hill

Operating System

What is an Operating System (OS)?

Software that manages resources for the computing system.

- Resources: memory, I/O devices, use of CPU, ...
- Two primary goals:
  - Optimize access to resources
  - Make sure no software does harmful things to programs or data that it should not access
- Key concepts: **privilege** and **priority**

Examples: Windows, MacOS, Linux

© McGraw Hill

Privilege and Priority

Privilege = What is this software allowed to do / access?

- Special instructions (for example, HALT)
- Special memory address (for example, operating system code)
- Only privileged instructions should be able to access these things.
- **Supervisor** mode = privileged, **User** mode = non-privileged

Priority = Which software is more urgent to execute?

- User programs operate at lowest priority level (0).
- Some events may need immediate attention, require system software:
  - Input from the keyboard: priority 4
  - Power interruption: priority 6
  - If both happen, power interruption code runs because it has a higher priority.

© McGraw Hill

LC-3 Processor Status Register

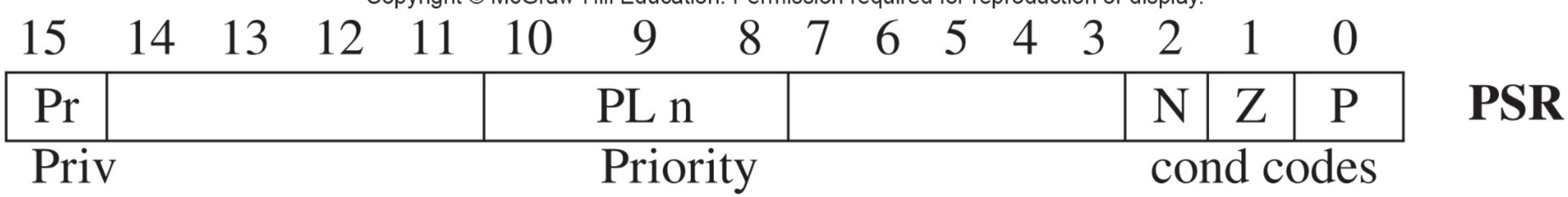
Processor (hardware) needs to know the privilege level and the priority of the instruction that is running.

LC-3 has two privilege levels: supervisor, user.

LC-3 has eight priority levels: user = 0, highest priority = 7.

Information is kept in the Processor Status Register (PSR), which also contains the condition code bits.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



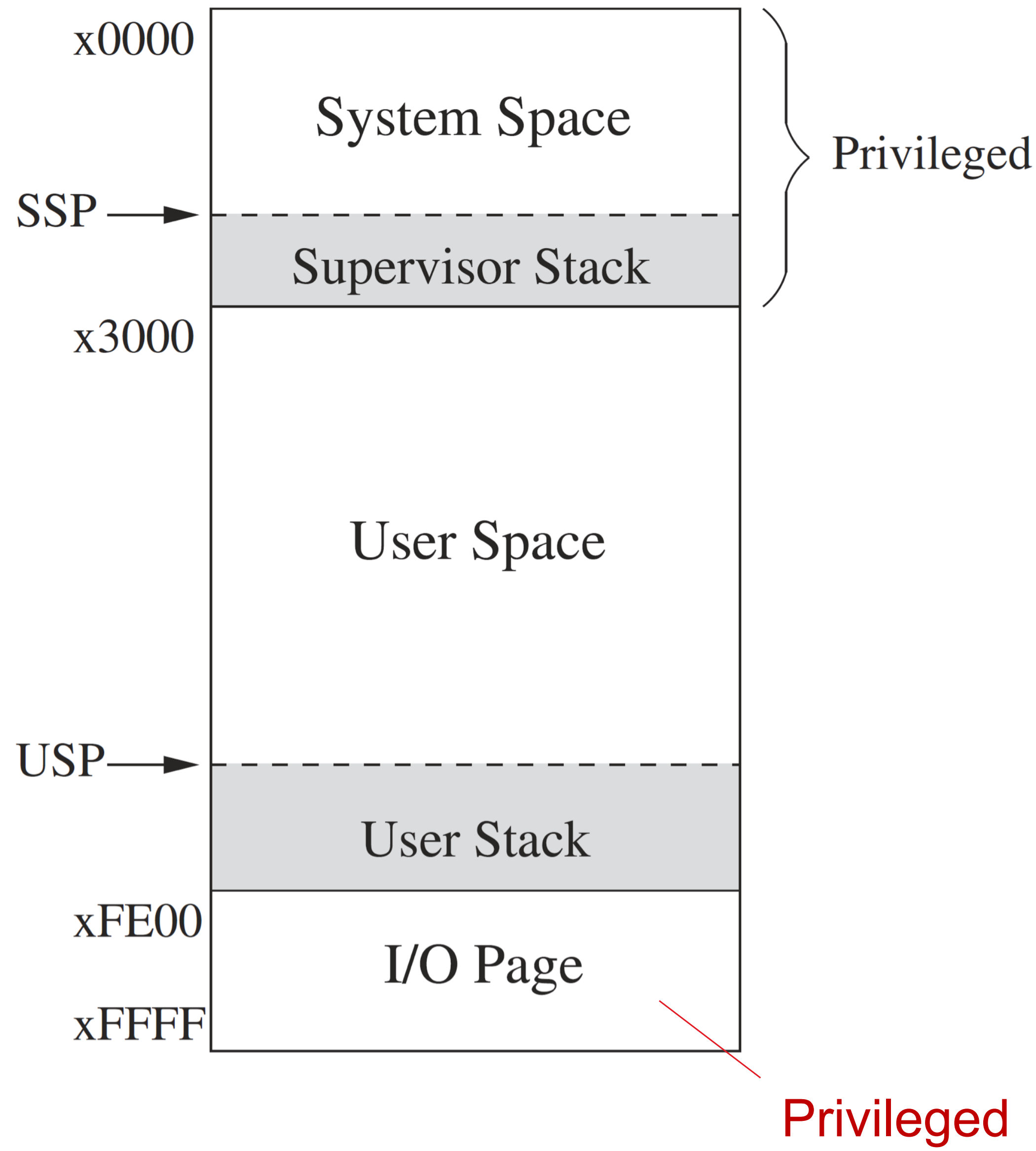
Access the text alternative for slide images.

LC-3 Privileged Memory

A portion of the memory address space is reserved for the operating system. This can only be accessed by instructions while running in supervisor mode.

While in supervisor mode, R6 refers to the top of the supervisor stack. In user mode, R6 refers to the top of the user stack. The values of the SSP and USP are saved in hardware and moved into/out of R6 when switching modes.

Addresses xFE00 to xFFFF are reserved for memory-mapped I/O, described later. This range of addresses is also privileged.



Access the text alternative for slide images.

© McGraw Hill

OS and I/O

The previous slides just give you an idea of the mechanisms provided by hardware to distinguish between OS (privileged) and user (non-privileged) software.

How does this relate to I/O?

We only want OS code to deal directly with I/O devices.

- Remove complexity and create abstraction for user code.
- Avoid dangerous behavior from programmers who may be incompetent or malicious.

We're now ready to discuss how to write software that interacts with input and output devices.

© McGraw Hill

I/O: Connecting to the Outside World

Types of I/O devices are generally characterized by:

- behavior: input, output, storage
  - input: keyboard, motion detector, network interface
  - output: monitor, printer, network interface
  - storage: disk, CD-ROM
- data rate: how fast can data be transferred?
  - keyboard: 100 bytes/sec
  - disk: 30 MB/s
  - network: 1 Mb/s to 1 Gb/s

© McGraw Hill

I/O Controller: Registers to interface with CPU

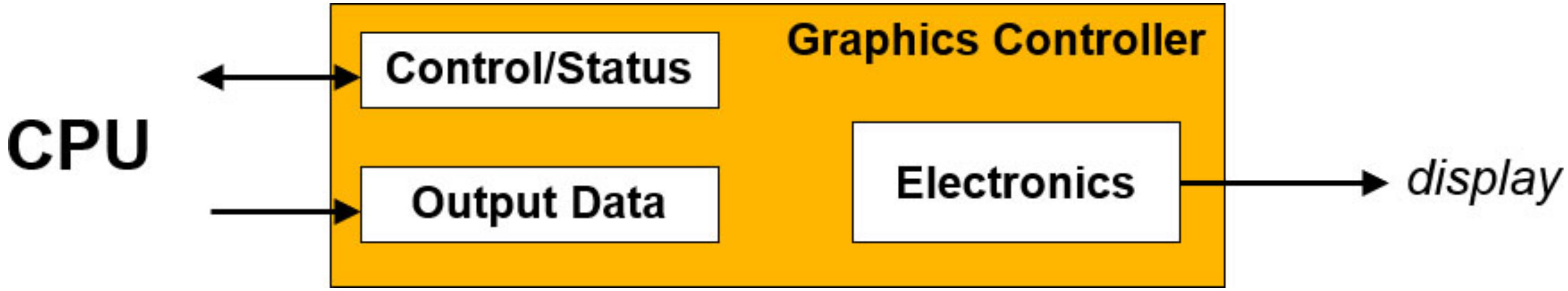
Control/Status Registers

CPU tells device what to do -- write to control register

CPU checks whether task is done -- read status register

Data Registers

CPU transfers data to/from device



Device electronics

performs actual operation

pixels to screen, bits to/from disk, characters from keyboard

Access the text alternative for slide images.

© McGraw Hill

I/O Programming Interface

How are device registers identified?

memory-mapped versus I/O instructions

How is timing of transfer managed?

synchronous versus asynchronous

Who initiates / controls the transfer?

CPU (polling) versus device (interrupts)

© McGraw Hill

Memory-Mapped versus I/O Instructions

Instructions

designate opcode(s) for I/O

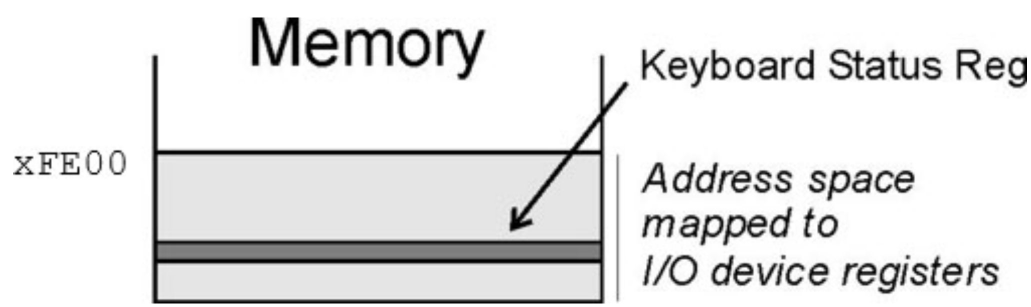
register and operation encoded in instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IO				Device								Op			

Memory-mapped

assign a memory address to each device register

use data movement instructions (LD/ST) for control and data transfer



Note: There is no memory here at all!
Hardware must recognize the address and direct the read/write to the proper I/O device register.

Access the text alternative for slide images.

© McGraw Hill

Data Transfer Timing

Synchronous

Device supplies (or accepts) data at fixed, predictable times

CPU reads/writes every X cycles

Asynchronous

Data rate and availability is not predictable

CPU must *synchronize* with device for each transfer, to make sure data is not missed.

© McGraw Hill

Data Transfer Control

Polling

CPU decides when it wants to perform I/O operation.

Reads device status register (over and over) until (a) device has new input data or (b) device is ready to accept new output data.

"Are we there yet? Are we there yet? Are we there yet? ..."

Interrupts

Device recognizes when an interesting change in status occurs -- for example, new input data is available, or device ready to accept new data.

Device sends a signal to the CPU.

CPU interrupts the current program to handle the I/O event, then resumes when the event handling is complete.

"Wake me up when we get there."

© McGraw Hill

I/O in the LC-3

The LC-3 provides the following mechanisms for I/O.

Memory-mapped Device Registers

Memory addresses xFE00 to xFFFF are reserved for device registers.
Must be implemented by the computer system hardware.
Access to these addresses is privileged.

Asynchronous Transfers

Though it's possible to interface with synchronous devices, we will only discuss asynchronous in this class.

Polling and Interrupts

Polling is performed by reading / writing device registers.

Interrupt mechanism will be described later in the chapter.

© McGraw Hill

Input from the Keyboard

The Keyboard device in the LC-3 uses two registers.

Keyboard Data Register (KBDR = xFE02)

When a key is typed, its ASCII code is placed in KBDR[7:0].
KBDR[15:8] is always zero.

Keyboard Status Register (KBSR = xFE00)

Bit 15 is the "ready" bit.
KBSR[15] = 1 means new data has been written to the KBDR.
KBSR[15] = 0 means no new data has been written.

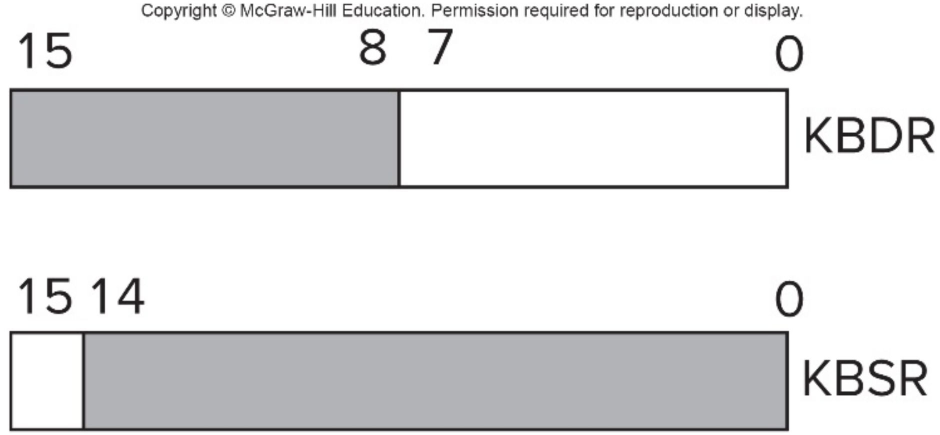
When a key is typed:

Device (hardware) puts ASCII code into KBDR, and sets KBSR[15] to 1.

When KBDR is read by CPU:

Device sets KBSR[15] to 0.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Basic Input Routine (Polling)

```
; Read from keyboard, put data in R0.
START  LDI R1, A    ; read KBSR
       BRzp START   ; loop until ready bit
set
       LDI R0, B    ; read KBDR
       BRnzp NEXT
A      .FILL xFE00    ; KBSR address
B      .FILL xFE02    ; KBDR address
```

First two lines are polling loop. Keep reading status register until the ready bit is set.

Why did we choose bit 15 as the ready bit? Value looks negative when read.

Note the use of LDI. Device register addresses are far away, so we must use a register or memory location to hold the address.

© McGraw Hill

Output to the Monitor

The Monitor device in the LC-3 uses two registers.

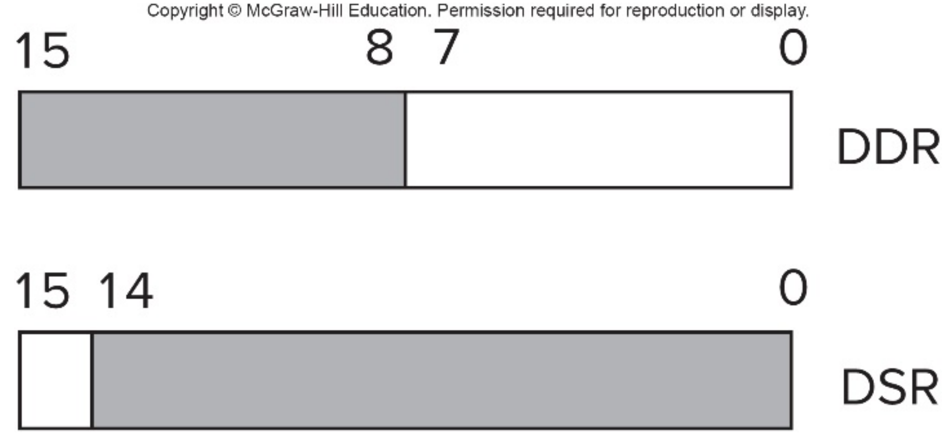
Display Data Register (DDR = xFE06)

An ASCII character written to this register will be displayed on the monitor.
Bits written to DDR[15:8] are ignored.

Display Status Register (DSR = xFE04)

Bit 15 is the "ready" bit.
DSR[15] = 1 means device is ready to accept the next character.
DSR[15] = 0 means the previous character has not been displayed.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Basic Output Routine (Polling)

```
; Display character in R0 to monitor.
START  LDI R1, A    ; read DSR
       BRzp START   ; loop until ready bit set
       STI R0, B    ; write DDR
       BRnzp NEXT
A      .FILL xFE04    ; DSR address
B      .FILL xFE06    ; DDR address
```

Keyboard Input with Echo

```
; Read from keyboard, put data in R0.
; Echo character to monitor.
START  LDI R1, A    ; wait for keyboard input
       BRzp START
       LDI R0, B    ; read KBDR
ECHO   LDI R1, C    ; wait for monitor ready
       BRzp ECHO
       STI R0, D    ; write DDR
       BRnzp NEXT
A      .FILL xFE00    ; KBSR address
B      .FILL xFE02    ; KBDR address
C      .FILL xFE04    ; DSR address
D      .FILL xFE06    ; DDR address
```

© McGraw Hill

Input with Prompt

How does user know it's time to enter a character from the keyboard?

This routine prints a prompt string and then waits for keyboard input.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

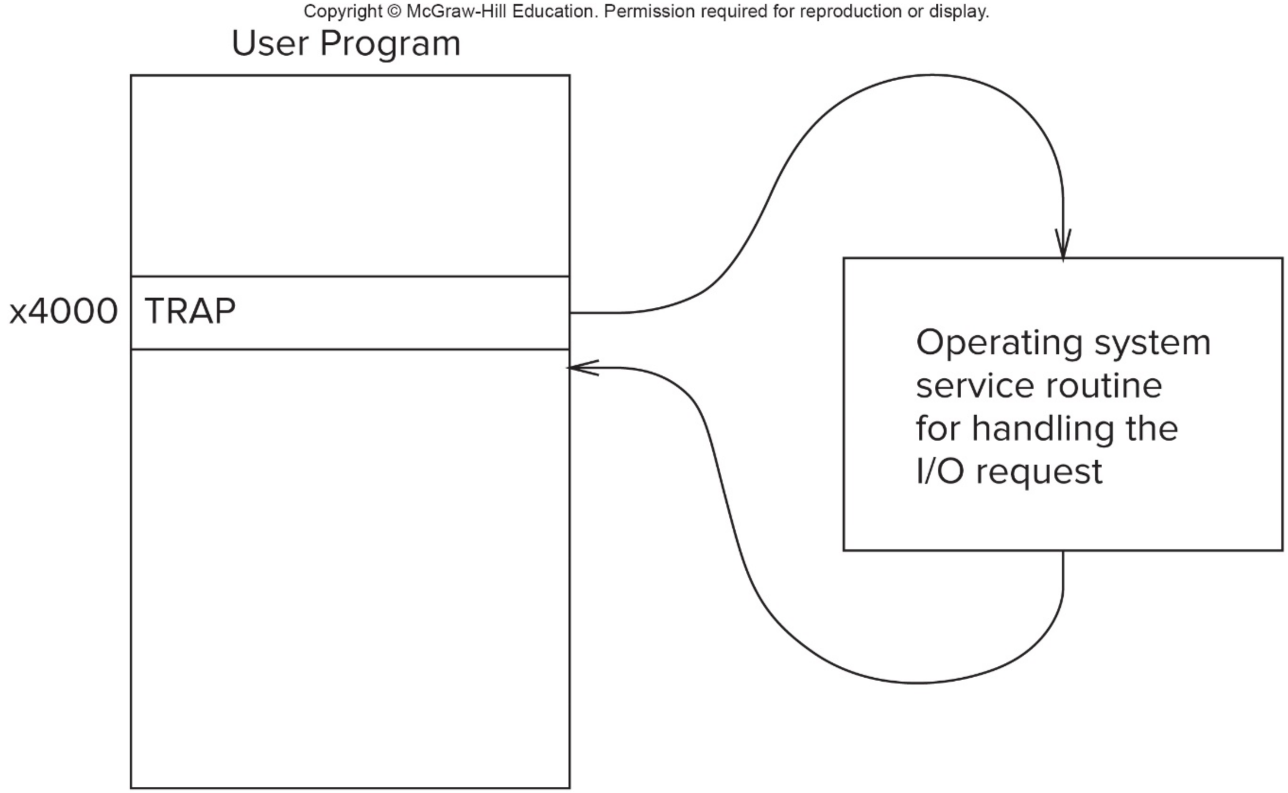
```
START   ST      R1,SaveR1    ; Save registers needed
        ST      R2,SaveR2    ; by this routine
        ST      R3,SaveR3
;
        LD      R2,Newline
L1      LDI     R3,DSR
        BRzp    L1           ; Loop until monitor is ready
        STI     R2,DDR       ; Move cursor to new clean line
;
        LEA     R1,Prompt    ; Starting address of prompt string
Loop    LDR     R0,R1,#0     ; Write the input prompt
        BRz     Input        ; End of prompt string
L2      LDI     R3,DSR
        BRzp    L2           ; Loop until monitor is ready
        STI     R0,DDR       ; Write next prompt character
        ADD     R1,R1,#1     ; Increment prompt pointer
        BRnzp   Loop         ; Get next prompt character
;
Input   LDI     R3,KBSR
        BRzp    Input        ; Poll until a character is typed
        LDI     R0,KBDR      ; Load input character into R0
L3      LDI     R3,DSR
        BRzp    L3           ; Loop until monitor is ready
        STI     R0,DDR       ; Echo input character
;
L4      LDI     R3,DSR
        BRzp    L4           ; Loop until monitor is ready
        STI     R2,DDR       ; Move cursor to new clean line
        LD      R1,SaveR1    ; Restore registers
        LD      R2,SaveR2    ; to original values
        LD      R3,SaveR3
        BRnzp   NEXT_TASK    ; Do the program's next task
;
SaveR1  .BLKW   1            ; Memory for registers saved
SaveR2  .BLKW   1
SaveR3  .BLKW   1
DSR     .FILL   xFE04
DDR     .FILL   xFE06
KBSR    .FILL   xFE00
KBDR    .FILL   xFE02
Newline .FILL   x000A        ; ASCII code for newline
Prompt  .STRINGZ ''Input a character>''
```

OS Service Routines

Instead of requiring a user program to know the details of device registers, we **abstract** the interaction with I/O by providing a service routine.

The service routine is invoked using a system call: TRAP.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Access the text alternative for slide images.

LC-3 System Call (Trap) Mechanism

1. A set of service routines (part of the OS).
2. A table of starting addresses for the service routines. This is called the Trap Vector Table.
3. The TRAP instruction, which transfers control to the service routine specified in the Trap Vector Table.
4. A linkage mechanism for returning control back to the user program.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

x0000 ⋮	⋮
x0020	x03E0
x0021	x0420
x0022	x0460
x0023	x04A0
x0024	x04E0
x0025	x0520
⋮ x00FF	⋮

The LC-3 Trap Vector Table has one entry per trap vector (x00 to xFF). It is stored at memory locations x0000 to x00FF.

Access the text alternative for slide images.

TRAP versus JSR

TRAP has a lot in common with JSR. Both want to:

a) Transfer control by putting an address into the PC.

b) Provide a way to get back to the calling routine.

JSR uses:

a) PC-Relative (JSR) or Base+offset (JSRR) addressing mode to specify the target address.

b) R7 to save return address.

TRAP uses:

a) Trap vector table to provide target address, indexed by trap vector.

b) Supervisor stack to save return address.

© McGraw Hill

TRAP Execution

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

1. Push **PSR** and **PC** onto **Supervisor Stack**.

   If TRAP is executed in user mode, hardware switches R6 to point to the Supervisor Stack before pushing.

   - Copy R6 to Saved_USP.
   - Copy Saved_SSP to R6.

2. Set **PSR[15]** to 0, indicating **supervisor mode**. This allows OS code (service routine) to execute privileged instructions and access privileged memory address.

3. **Trap vector** IR[7:0] is zero-extended to form the address of the specified entry in the **trap vector table**. This address is loaded, which provides the starting address of the service routine. The starting address is placed into the **PC**.

© McGraw Hill

RTI: Return from a Service Routine

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1. Pop **PC** from Supervisor Stack. This will resume the calling routine when this instruction completes.
2. Pop **PSR** from Supervisor Stack. This restores the priority and privilege level (and conditions codes) for the calling routine.
3. If PS[15] = 1, that means we have transitioned from supervisor mode to **user mode**, and we need to restore the User Stack Pointer.

    - Copy R6 to Saved_SSP.
    - Copy Saved_USP to R6.

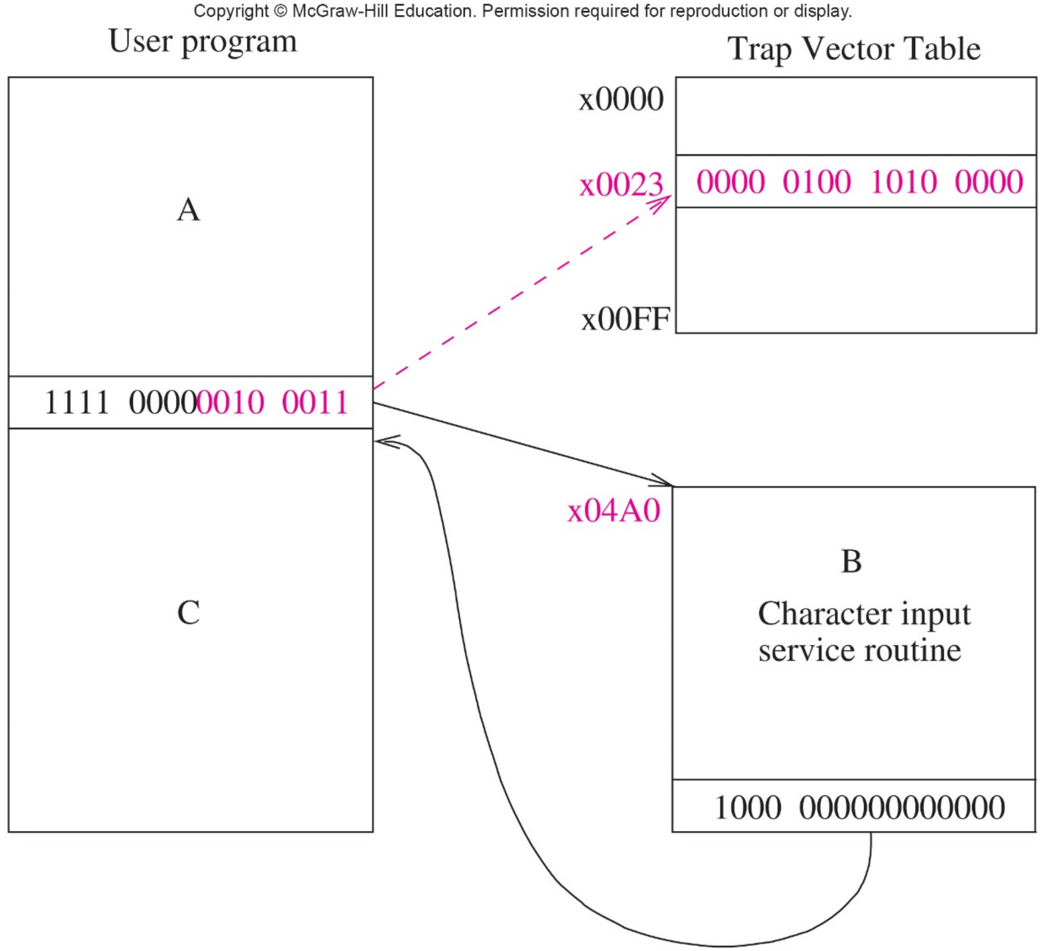
Note: If PS[15] = 0, that means the TRAP was called in supervisor mode, so the calling routine is still using the supervisor stack.

© McGraw Hill

Trap Mechanism Summary

The picture illustrates what happens when the user program executes TRAP x23.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Access the text alternative for slide images.

© McGraw Hill

Service Routine for Character Output

```
; Display character in R0 to monitor.
        .ORIG x0420 ; starting address of service routine
        ST   R1,SaveR1

Try     LDI R1,A    ; read DSR
        BRzp Try    ; loop until ready bit set
        STI R0,B    ; write DDR

        LD   R1,SaveR1
        RTI

A       .FILL xFE04    ; DSR address
B       .FILL xFE06    ; DDR address
SaveR1 .BLKW 1
        .END
```

© McGraw Hill

Service Routine for Halting the Machine

In Chapter 3, we noted that we could halt the execution of the processor by ANDing a zero with the clock signal.

LC-3 uses the bit 15 of the Master Control Register (MCR) to control the clock signal. The MCR is memory-mapped to xFFFE.

Therefore, to stop the clock, we want to set MCR[15] to zero.

However, we don't want to change anything else in that register,
so we first load the value, AND it with x7FFF, then write the result back.

In the following service routine, we also print a message to tell the user that the machine is about to halt.

Note that our service routine uses the TRAP instruction -- because the PC and PSR are pushed (and popped by RTI), we can safely nest calls to other service routines inside a service routine.

© McGraw Hill

HALT Service Routine 1

```
            .ORIG x0520
HALT        ST    R1,SaveR1      ; save registers
            ST    R0,SaveR0

            LD    R0,NewLine     ; print linefeed
            TRAP x21
            LEA   R0,Message     ; print message
            TRAP x22           ; (PUTS service routine)
            LD    R0,NewLine     ; another linefeed
            TRAP x21

            LDI   R1,MCR         ; get current MCR
            LD    R0,MASK        ; bit mask to clear [15]
            AND   R1,R1,R0
            STI   R1,MCR         ; clear bit, stop clock

            ; return -- how can this code return if clock is stopped?
            LD    R1,SaveR1
            LD    R0,SaveR0
            RTI

            ; data on next page
```

© McGraw Hill

HALT Service Routine 2

```
NewLine     .FILL x0A    ; linefeed character
MASK        .FILL x7FFF  ; bit mask to clear bit 15
MCR         .FILL xFFFE  ; address of machine control reg (MCR)
Message     .STRINGZ "Halting the machine."

SaveR1      .BLKW 1
SaveR0      .BLKW 1

            .END
```

© McGraw Hill

Assembler Pseudo-Ops for Service Routines

For the programmer's convenience and to provide a layer of abstraction, the LC-3 assembler recognizes special "opcodes" to represent common usages of the TRAP instruction.

Pseudo-Opcode	Actual Instruction	Description
`GETC`	**`TRAP x20`**	read a single character (no echo) into R0
`OUT`	**`TRAP x21`**	print character (R0) to monitor
`PUTS`	**`TRAP x22`**	print string to monitor; R0 contains a pointer to the string
`IN`	**`TRAP x23`**	print a prompt, read a single character with echo into R0
`HALT`	**`TRAP x25`**	halt the machine

© McGraw Hill

Interrupt-Driven I/O

I/O event may have nothing to do with the instructions that are being executed by the CPU. Allow the CPU instruction sequence to be **interrupted**, handle the I/O event, and then resume execution as if nothing had happened.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.

```
            .
            .
            .
  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 satisying 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
            .
            .
            .
```

Actual execution flow

```
            .
            .
            .
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
            .
            .
            .
```

Execution seen by Program A

Why Interrupts?

Processor can perform useful computation instead of "spinning" on ready bit.

Example:

Suppose program needs to (a) read a 100-character sequence from the keyboard and (b) process the input -- repeat 1000 times.

Assume user types 80 words/minute = 0.125 secs/character.
Assume processing takes 12.49999 seconds.
How long to execute entire program?

Without interrupts: $0.125 \times 100 - 12.5$ secs to read data. Most of that time is spent polling. Total time = 24.49999 seconds for each sequence = 7 hrs

With interrupts: Instead of polling, can process previous 100-char sequence and only spend a few instructions on each character as it's typed. By overlapping, time reduced to 12.5 seconds per sequence = 3.5 hrs.

© McGraw Hill

Interrupt Mechanism: Two Parts

1. A mechanism that enables an I/O device to interrupt the processor
   - I/O device **must want** service.
   - I/O device **must have the right** to request service.
   - Device request **must be more urgent** than current processor execution.

2. A mechanism that handles the interrupt request
   - Similar to a subroutine call, but unscripted -- not anticipated or requested by the program code.

© McGraw Hill

Part 1: Initiating the Interrupt Signal

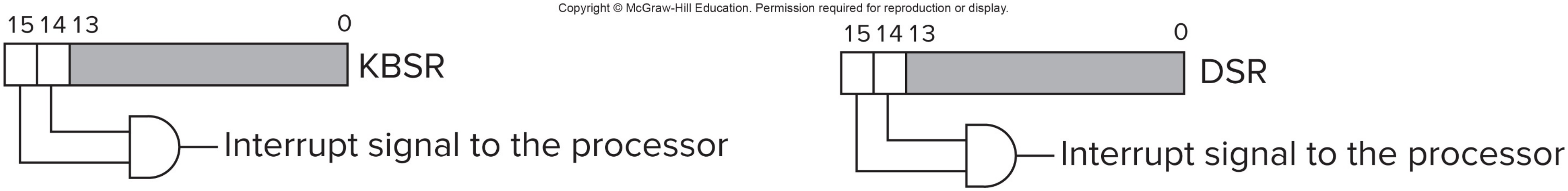
An interrupt signal may occur whenever some event occurs that causes the device to need service.

- Examples: Key is pressed on the keyboard. Character has been displayed on the monitor. Network packet has arrived. Temperature sensor has exceeded its threshold.

Interrupts must be **enabled** from a particular device.

- Typically controlled by a bit in the status/control register.
- For LC-3: Both KBSR and DSR use bit 14 to enable interrupts. When this bit it set and the ready bit is set, and interrupt signal is generated.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Access the text alternative for slide images.

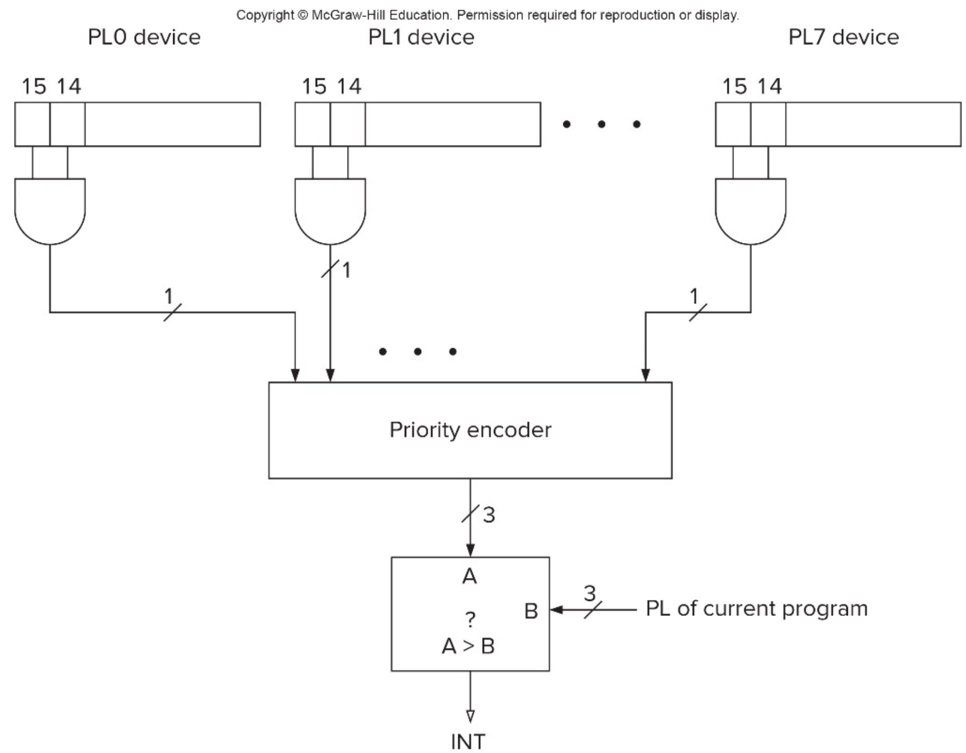
Interrupt Priority

Is this interrupt more urgent than the code currently running?

LC-3 priority is held in PSR[10:8].
Each device is assigned an interrupt priority level: 0 to 7.

Interrupt signal (INT) is handled by CPU only if the priority is higher than the current program.

If multiple devices want to interrupt, only highest priority signal is sent to the CPU.



Access the text alternative for slide images.

Part 2: Handling the Interrupt

INT signal is inspected before the FETCH phase of each instruction.

This means that an interrupt will not disrupt the execution of an instruction -- it only changes execution flow between instructions. The instruction is the fundamental unit of execution: once an instruction is fetched, it is guaranteed to complete execution.

If the INT signal is asserted:

1. Save state of the currently executing program.
2. Transfer control to the code that handles this particular interrupt.
3. Restore state of the program and fetch the next instruction.

© McGraw Hill

Saving Program State

Program state = PC, PSR, registers, memory used by program

Minimal state is **PC + PSR**. These will be **pushed** to the **Supervisor Stack**, as we do for the TRAP instruction. Interrupt handling code is responsible for saving and restoring registers as needed.

- As with TRAP, changing from user mode to supervisor mode will swap R6 from the user stack pointer (USP) to the supervisor stack pointer (SSP).
- NOTE: Condition codes are part of the PSR. This means that even if a program is interrupted right before a BR, it will make the same decision as if the interrupt didn't happen, because the condition codes will not change.

© McGraw Hill

Transferring Control to Interrupt Handler

How does the hardware know what to put in the PC?
Where is the code that handles this interrupt?

Along with the INT signal, the interrupting device provides an 8-bit interrupt vector. This is used to look up a starting address for the interrupt handling code, similar to TRAP.

LC-3 Interrupt Vector Table is in memory at x0100 to x01FF.

Initializing PSR:

- Conditions codes are set to 010. (No particular reason, but must be set to something...).
- PSR[15] = 0, supervisor mode.
- PSR[10:8] = priority level of interrupt signal.

© McGraw Hill

Returning from Interrupt: RTI

When we are ready to return from the interrupt handling routine, we are in the same situation as a TRAP service routine:

- PSR and PC of the "calling" routine are on the supervisor stack.
- Pop the stack, restore the USP (if going back to user mode).

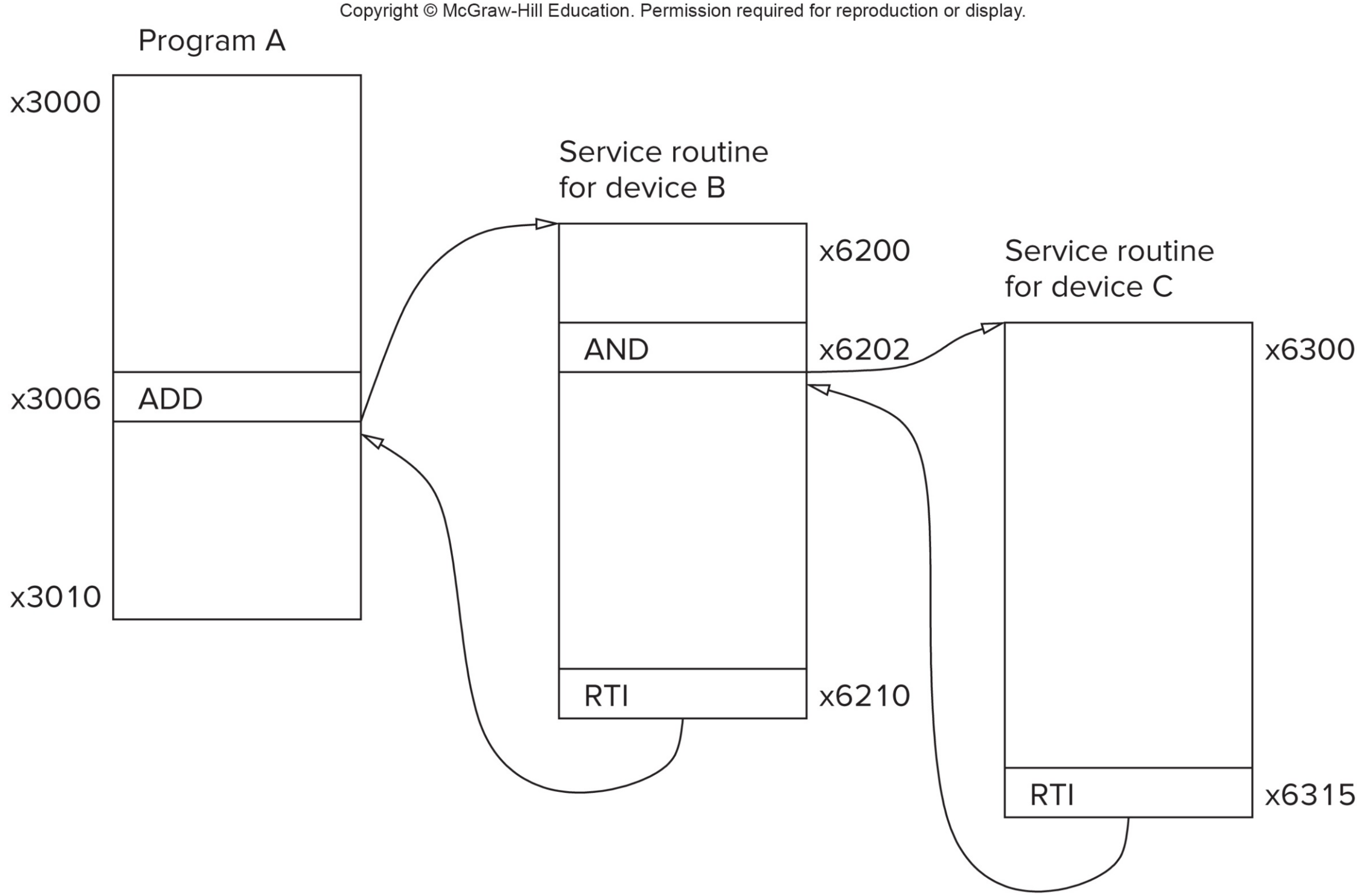
This exact sequence is performed by the **RTI** instruction, described earlier.

© McGraw Hill

Interrupt Example

Program A is interrupted by Device B. While interrupt service routine for B is running, it is interrupted by Device C, which has a higher priority.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Access the text alternative for slide images.

Nested Interrupts: Enabled by the Stack

The nesting of interrupts described on the previous slide is enabled by the user of the supervisor stack to save PC and PSR state.

(a) Program A is running.

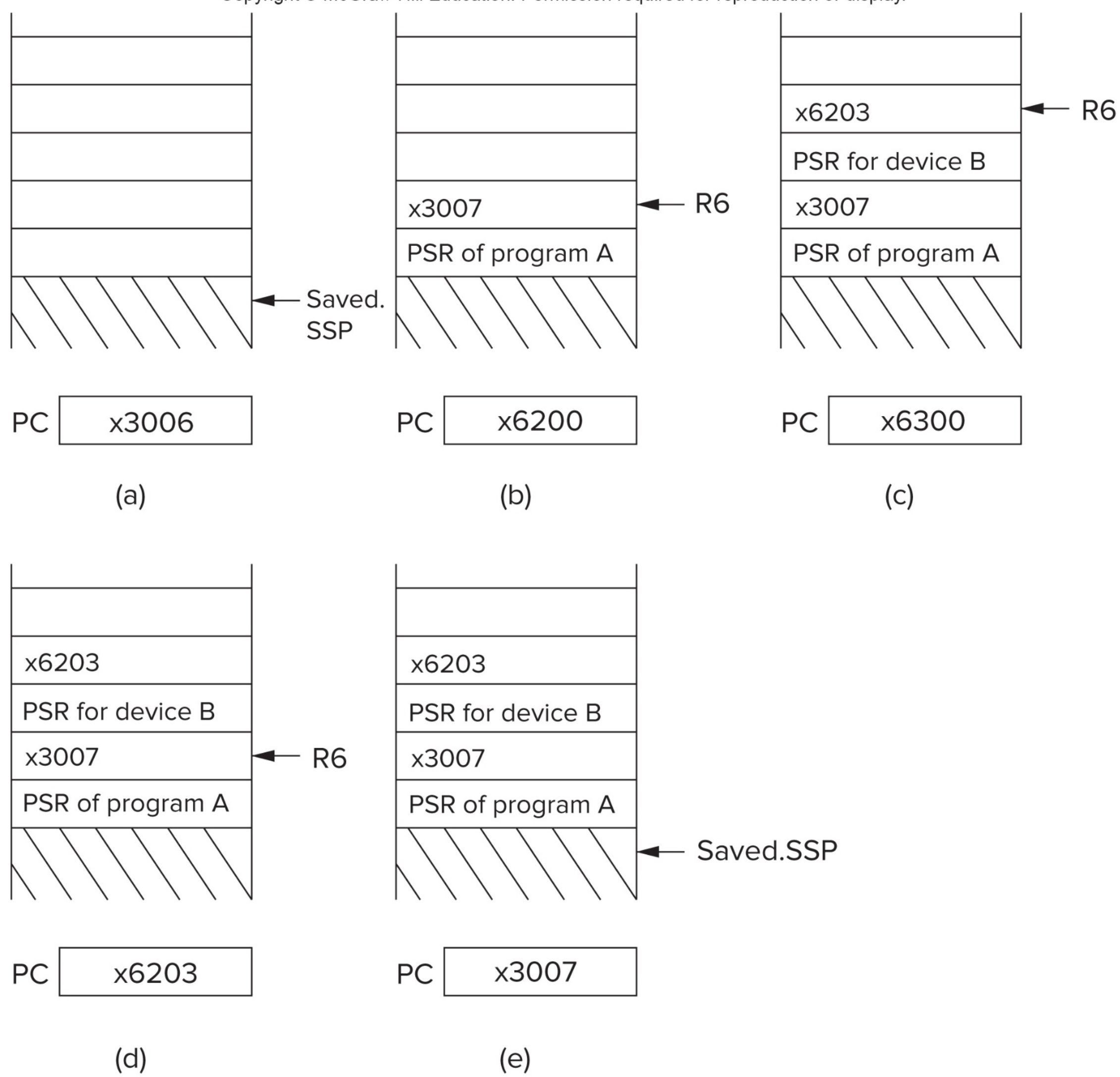
(b) Device B interrupts, saving A's PC and PSR. R6 is written to Saved_USP, and Saved_SSP is written to R6.

(c) Device C interrupts, saving B's PC and PSR.

(d) Device C handler returns, restoring B's PC and PSR and popping the stack.

(e) Device B handler returns, restoring A's PC and PSR and popping the stack. R6 is written to Saved_SSP, and Saved_USP is written to R6.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Access the text alternative for slide images.

Interrupts: Not Just for I/O

There are events that happen in a computer system that require attention, but are not (directly) related to I/O devices:

- Timer expires.
- Machine check -- component failure.
- Power failure.

These events can also generate INT signals, vectors, etc., and can be handled in the same way as the interrupts we have been discussing.

In addition, there may be abnormal events caused by the execution of instructions in the CPU, such as:

- Divide by zero.
- Illegal instruction.

These "internal interrupts" are often called ***exceptions***.

© McGraw Hill

Polling Revisited: What about Interrupts?

We usually think of a polling loop as "atomic" -- that is, if the ready bit is set and the branch is not taken, we "immediately" perform the related load/store operation to the device.

Example -- the output character loop:

```
POLL  LDI R1,DSR    ; check monitor
      BRzp POLL     ; keep checking if not read
      STI R0,DDR    ; ready -- store the character to device
```

But now we know that an interrupt can happen at any time, and significant time may elapse before returning control to the program.

What if an interrupt happens between the BRzp and the STI, and when the handler returns, the device is **no longer ready** to accept a character?

- In other words, the status that was returned by the LDI is no longer valid because something else occurred between the LDI and the STI.

Disabling Interrupts 1

To avoid the problem (and many others like it), we may want to disable interrupts during certain sequences of instructions.

Mechanism:

- Use bit 14 of the PSR as a global interrupt enable bit. It is set to 1 by default, so that interrupts are enabled. But we can ignore all interrupts by setting it to 0.
- We also need to memory-map the PSR. We'll use xFFFC.

© McGraw Hill

Disabling Interrupts 2

```
; To disable interrupts, write 0 to PSR[14].
; Use a mask to preserve all other bits.
       LDI R1,PSR    ; get current PSR bits
       LD  R2,INTMASK
       AND R2,R2,R1  ; R2 now has bit 14 clear
       STI R2,PSR    ; disable interrupts
       ...
       ...
       STI R1,PSR    ; enable when ready
       BRnzp NEXT
PSR    .FILL xFFFC   ; PSR address
INTMASK .FILL xBFFF
```

© McGraw Hill

Disabling Interrupts: DANGER

If we disable interrupts for long periods of time, we may miss very important events and prevent high-priority tasks from running.

Better: Code below shows a polling loop that reenables interrupts at the beginning of each loop -- this allows the important LDI - BR - STI sequence to be uninterrupted, but does not disable interrupts for the entire polling period. (See complete code in Figure 9.22.)

```
        LDI   R1, PSR
        LD    R2,INTMASK
        AND   R2,R1,R2    ; R1=original PSR, R2=PSR with interrupts disabled

POLL    STI   R1,PSR    ; enable interrupts (if they were enabled to begin)
        STI   R2,PSR    ; disable interrupts
        LDI   R3,DSR
        BRzp  POLL      ; Poll the DSR
        STI   R0,DDR   ; Store the character into the DDR
        STI   R1,PSR   ; Restore original PSR
```

Access the text alternative for slide images.

© McGraw Hill

Because learning changes everything.®

www.mheducation.com

© 2020 McGraw Hill. All rights reserved. Authorized only for instructor use in the classroom.
No reproduction or further distribution permitted without the prior written consent of McGraw Hill.