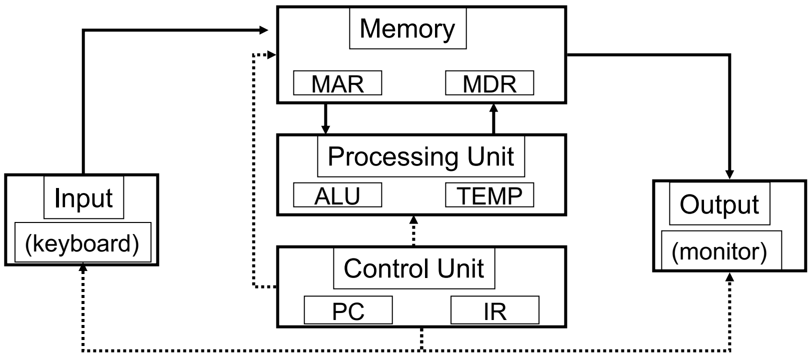
LC3 Assembly Programming: Input/Output & TRAP Instructions

The von Neumann Model



- **Memory: holds both data and instructions**
- **Processing Unit: carries out the instructions**
- **Control Unit: sequences and interprets instructions**
- **Input: external information into the memory**
- **Output: produces results for the user**

I/O: Connecting to Outside World

- So far, we've learned how to:
  - compute with values in registers
  - load data from memory to registers
  - store data from registers to memory
- But where does data in memory come from?
  - wide variety of Input devices
- And how does data get out of the system so that humans can use it?
  - Wide variety of output devices

3

3

I/O: Connecting to the Outside World

- Types of I/O devices 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 - 1 Gb/s
- We stick to keyboard and display
  - Cover basic concepts of I/O processing
  - Similar solutions used in real processors

4

4

Abstracting the Interaction with I/O Devices

- What do we need to know about I/O devices ?

- Only two aspects:
  - Are they ready to process CPU's request?
  - Where to send the data to be processed by I/O device ?

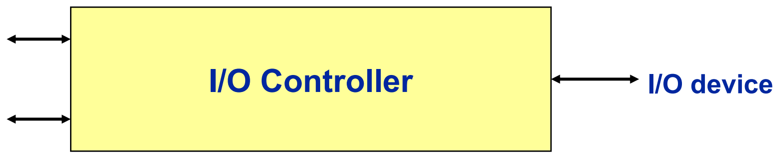
5

5

I/O Devices and Controllers

Most I/O devices are not purely digital themselves …
- Electro-mechanical: e.g., keyboard, mouse, disk, motor
- Analog/digital: e.g., network interface, monitor, speaker, mic

… all have digital interfaces presented by I/O Controller
- CPU (digital) talks to controller
- Not super-interested in controller/device internals for now..

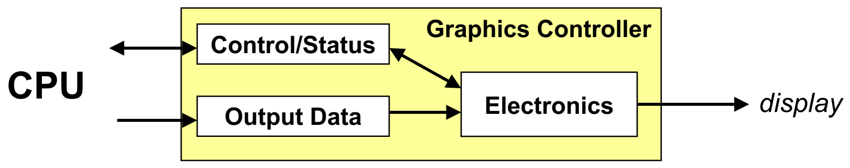


6

6

I/O Controller Interface: Abstraction

- I/O Controller interface presented as device registers
  - Control/status: may be one register or two
  - Data: may be more than one of these
- For input:
  - CPU checks status register if input is available
  - Reads input from data register (or waits if no input



- For output:
  - CPU checks status register to see if it can write (device free)
  - Writes output to data register
- Device electronics performs actual operation
  - pixels to screen, bits to/from disk, characters from keyboard

7

7

Programming Interface

- How are device registers identified?
  - Memory-mapped vs. special instructions
- How is timing of transfer managed?
  - Asynchronous vs. synchronous
- Who controls transfer?
  - CPU (polling) vs. device (interrupts)

8

8

Transfer Timing

- I/O events generally happen much slower than CPU cycles.

- Synchronous
  - data supplied at a fixed, predictable rate
  - CPU reads/writes every X cycles
- Asynchronous
  - data rate less predictable
  - CPU must ***synchronize*** with device,
    so that it doesn't miss data or write too quickly
    - How: some protocol is needed

9

9

Synchronous or Asynch.

- TV and Remote ?
- Mail delivery person and you ?
- Mouse and PC ?

- TV and Remote: synchronous
  - TV samples at specific intervals to see if key on remote has been pressed
- Mail delivery: asynchronous
  - Use mailbox as synchronization mechanism
- Mouse and PC: synchronous
  - PC samples mouse at specific intervals

10

10

How are Device Register Reads/Writes Performed?

Two options (aren't there always?)

I/O instructions

- Designate opcode(s) for I/O
- Register and operation encoded in instruction

Memory-mapped I/O

- Assign a memory address to each device register
- Use conventional loads and stores
- Hardware intercepts loads/stores to these address
- No actual memory access performed
- LC3 (and most other platforms) uses memory mapped I/O

11

11

Memory-Mapped vs. I/O Instructions

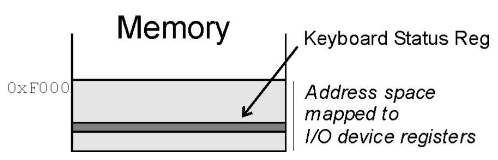
▪Instructions

- 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



12

12

LC-3

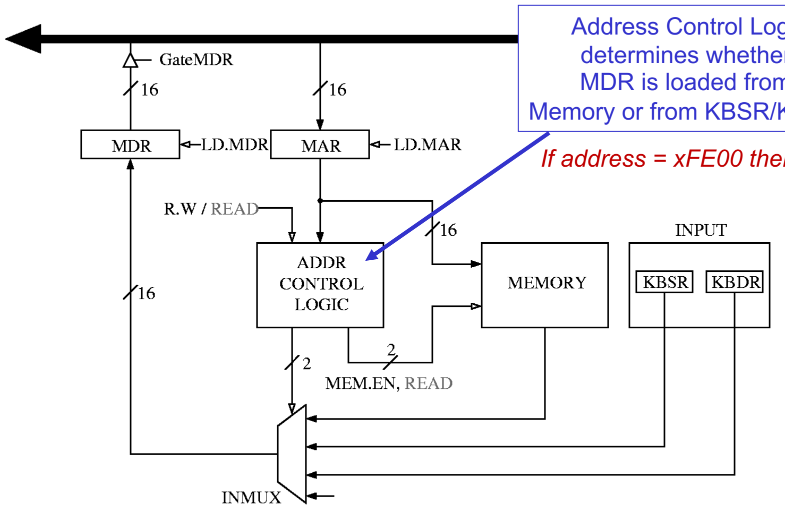
- Memory-mapped I/O (Table A.3)

Location	I/O Register	Function
xFE00	Keyboard Status Reg (KBSR)	Bit [15] is one when keyboard has received a new character.
xFE02	Keyboard Data Reg (KBDR)	Bits [7:0] contain the last character typed on keyboard.
xFE04	Display Status Register (DSR)	Bit [15] is one when device ready to display another char on screen.
xFE06	Display Data Register (DDR)	Character written to bits [7:0] will be displayed on screen.

13

13

Simple Implementation: Memory-Mapped Input



Address Control Logic determines whether MDR is loaded from Memory or from KBSR/KBDR.

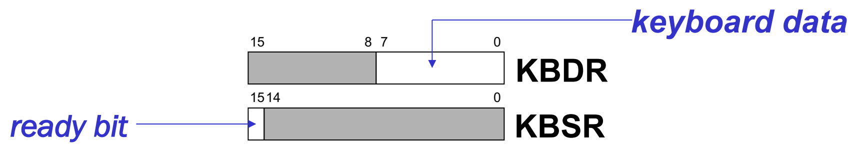
If address = xFE00 then KBSR

Note the abstraction of the Input device: Status Register (KBSR) and Data Register (KBDR)

14

Input from Keyboard

- When a character is typed:
  - It is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
  - the “ready bit” (KBSR[15]) is set to one
  - keyboard is disabled -- any typed characters will be ignored

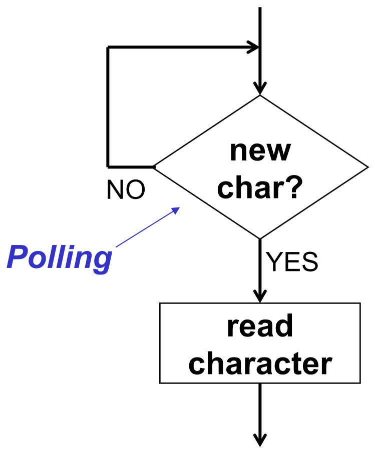


- When KBDR is read: the keyboard HW
  - KBSR[15] is set to zero
  - keyboard is enabled

15

15

Basic Input Routine

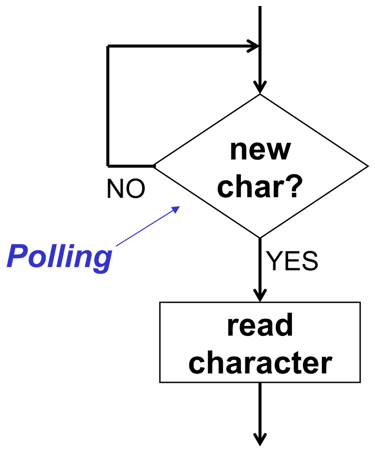


- Check if Keyboard ready
- Keep checking….
  How …if KBSR bit 15=1
- Once ready, it reads from keyboard into register R0

16

16

Basic Input Routine



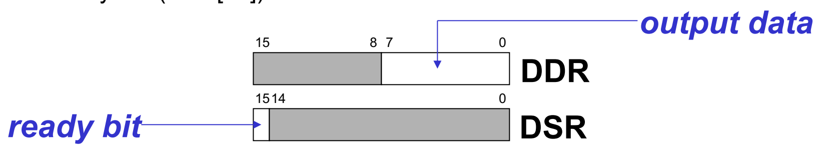
```
POLL     LDI  R0, KBSRPtr
         BRzp POLL
         LDI  R0, KBDRPtr

         ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

17

17

Output to Monitor

▪When Monitor is ready to display another character:
- the "ready bit" (DSR[15]) is set to one



▪When data is written to Display Data Register:
- DSR[15] is set to zero
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored (while DSR[15] is zero)

18

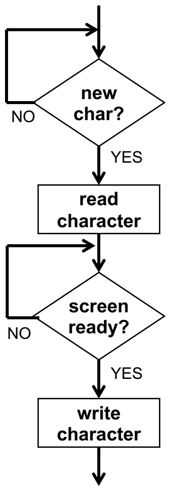
18

Keyboard Echo Routine

▪Usually, input character is also printed to screen.

- User gets feedback on character typed
  and knows its ok to type the next character.

```
POLL1    LDI  R0, KBSRPtr
         BRzp POLL1
         LDI  R0, KBDRPtr
POLL2    LDI  R1, DSRPtr
         BRzp POLL2
         STI  R0, DDRPtr

         ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
DSRPtr  .FILL xFE04
DDRPtr  .FILL xFE06
```



19

19

Some Questions

▪What is the danger of not testing the DSR before writing data to the screen?

▪What is the danger of not testing the KBSR before reading data from the keyboard?

What if the Monitor were a synchronous device, e.g., we know that it will be ready 1 microsecond after character is written.

- Can we avoid polling? How?
- What are advantages and disadvantages?

20

20

Who writes the I/O code ?
Trap Routines/ Service calls

- Not a good idea to let programmers write their code to do I/O?
- Send the request to the “system”
  - OS will service the request and return control back to user program
  - Eg: Printf

21

21

System Calls

- Certain operations require specialized knowledge and protection:
  - specific knowledge of I/O device registers and the sequence of operations needed to use them
  - I/O resources shared among multiple users/programs; a mistake could affect lots of other users!

- Not every programmer knows (or wants to know) this level of detail

- Provide *service routines* or *system calls* (part of operating system) to safely and conveniently perform low-level, privileged operations

22

22

System Call - steps

- 1. User program invokes system call.
- 2. Operating system code performs operation.
- 3. Returns control to user program.

23

23

Role of the OS

24

24

System Calls..how

how do I get my code to "ask" the OS for I/O?

- Call a special subroutine, called a "TRAP"
  - Also called syscall or callgate
- We don't simply use a Branch or Function call:
  - not "secure" enough
  - User can't set privilege bit themselves
    - great temptation to give themselves powers they shouldn't have

25

25

In the Real World

- System call/TRAP Specifics:
  - User can call to a restricted set of function addresses
  - Can upgrade privilege only through these channels
- This system call mechanism is commonly used in actual systems
  - As an example the BIOS (Basic Input Output System) on many Intel PCs provided precisely this functionality to allow programs to access basic input and output devices, keyboards, displays, timers etc.
- More modern systems use EFI (Extensible Firmware Interface) which is a more sophisticated version of the same thing.

26

26

LC-3 TRAP Mechanism

- *1. A set of service routines.*
  - part of operating system -- routines start at arbitrary addresses (convention is that system code is below x3000)
  - up to 256 routines
- *2. Table of starting addresses.*
  - stored at x0000 through x00FF in memory
  - called System Control Block in some architectures
- *3. TRAP instruction.*
  - used by program to transfer control to operating system
  - 8-bit trap vector names one of the 256 service routines
- *4. A linkage back to the user program.*
  - want execution to resume immediately after the TRAP instruction

27

27

LC3 TRAP Routines and their Assembler Names

vector	*symbol*	*routine*
`x20`	`GETC`	read a single character (no echo)
`x21`	`OUT`	output a character to the monitor
`x22`	`PUTS`	write a string to the console
`x23`	`IN`	print prompt to console, read and echo character from keyboard
`x25`	`HALT`	halt the program

28

28

TRAP Instruction

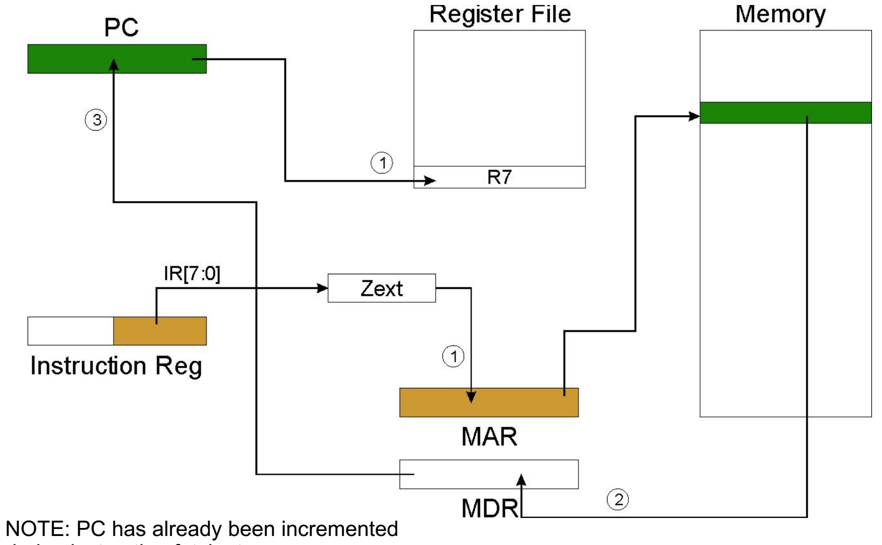
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

- Trap vector
  - identifies which system call to invoke
  - 8-bit index into table of service routine addresses
    - in LC-3, this table is stored in memory at 0x0000 – 0x00FF
    - 8-bit trap vector is zero-extended into 16-bit memory address
- Where to go
  - lookup starting address from table; place in PC
  - Load contents at trap vector address into the PC!
- How to get back
  - save address of next instruction (current PC) in R7
  - Last instruction in TRAP program sets PC equal to R7

29

29

TRAP



NOTE: PC has already been incremented
during instruction fetch stage.

30

RET (JMP R7)

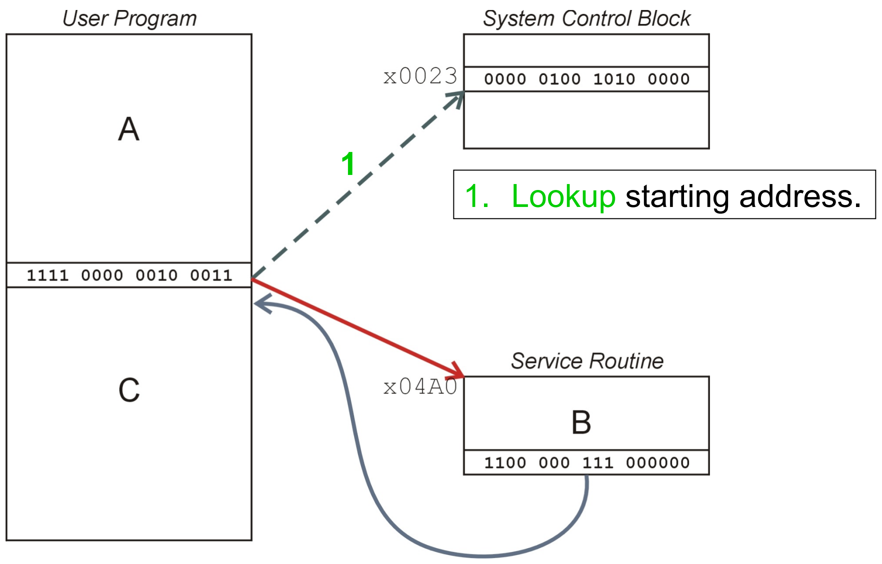
- How do we transfer control back to instruction following the TRAP?

- We saved old PC in R7.
  - JMP R7 gets us back to the user program at the right spot.
  - LC-3 assembly language lets us use RET (return) in place of “JMP R7”.

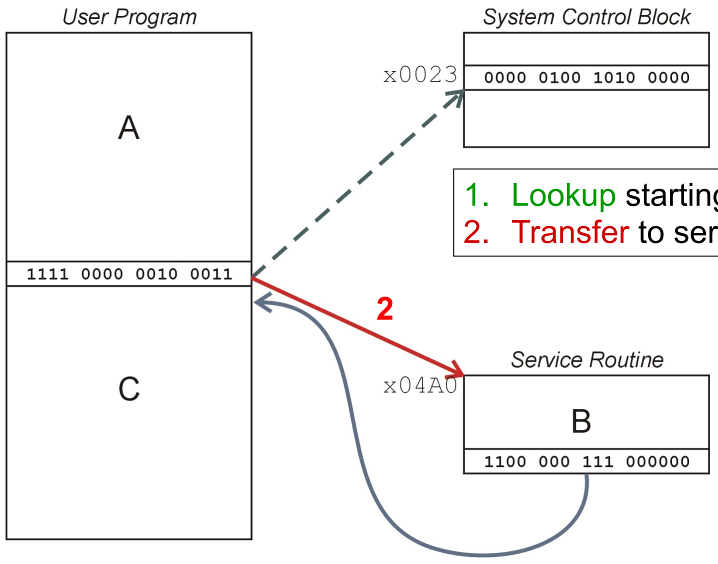
- Must make sure that service routine does not change R7, or we won’t know where to return.

31

TRAP Mechanism Operation



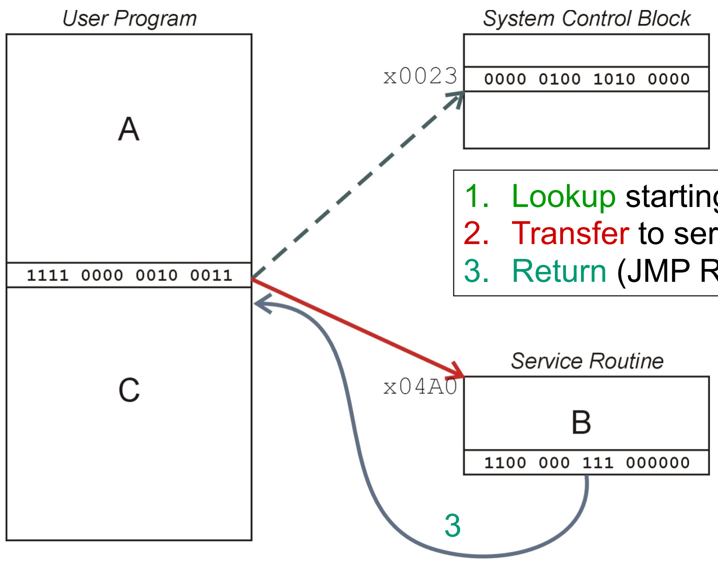
TRAP Mechanism Operation



1. Lookup starting address.
2. Transfer to service routine.

33

TRAP Mechanism Operation



1. Lookup starting address.
2. Transfer to service routine.
3. Return (JMP R7).

34

Example: Using the TRAP Instruction

```
▪           .ORIG x3000
▪           …
            …           ;  user code
            TRAP   x23       ;  input character into R0
            ADD   R1, R2, R0 ;  use R0
            …                ;  user code
            ADD   R0, R0, R3 ;  load output data into R0
            TRAP   x21       ;  Output to monitor...
            …                ;  ... User program...

EXIT        TRAP   x25       ;  halt
            .END
```

35

35

Saving and Restoring Registers

▪Can a service routine (TRAP) call another service routine (TRAP) ?

- Is there anything special the calling routine must do ?

▪Must save the value of a register if:

- Its value will be destroyed by service routine, and
- We will need to use the value after that action.

▪Who saves?

- caller of service routine?
  - knows what it needs later, but may not know what gets altered by called routine
- called service routine?
  - knows what it alters, but does not know what will be needed later by calling routine

36

36

How do actual I/O interactions take place – Protocols ?

Two schemes for interacting with I/O devices

What we have seen so far is polling

- "Are we there yet? Are we there yet? Are we there yet?"
- CPU keeps checking status register in a loop
- Very inefficient, multi-tasking CPU has better things to do

Alternative scheme is called interrupts

- "Wake me when we get there."
- Device sends special signal to CPU when status changes
- CPU stops current program, saves its state
- CPU "handles interrupt": checks status, moves data
- CPU resumes stopped program, as if nothing happened!!!!!!!!!

37

37

How interrupts work…

- CPU wants data from some I/O device…sends request to I/O device
- CPU continues executing user program (and does not stay idle waiting for I/O device to respond)
  - In user mode
- When device is ready with its data, it sends an Interrupt to the CPU
- CPU preempts the user program (suspends it), and processes the interrupt to fetch the data
  - In superuser (privileged) mode
- Once the interrupt is processed, it resumes execution of the user program

38

38

Protecting System space

- System calls go to specific locations in memory
  - We don't want users overwriting these
  - Write protect these locations
  - Halt a program that tries to enter unauthorized space/memory

39

39

Operating Systems (OSes)

First job of an OS:

- Handle I/O …2nd job of OS …
- OSes virtualize the hardware for user applications

In real systems, only the operating system (OS) does I/O

- "User" programs *ask* OS to perform I/O on their behalf
- Three reasons for this setup:

1) Abstraction/Standardization

- I/O device interfaces are nasty, and there are many of them
- Think of disk interfaces: S-ATA, iSCSI, IDE
- User programs shouldn't have to deal with these interfaces
  - In fact, even OS doesn't have to deal with most of them
  - Most are buried in "device drivers"

40

40

Operating Systems (OSes)

- 2) Raise the level of abstraction
  - Wrap nasty physical interfaces with nice logical ones
    - Wrap disk layout in file system interface
- 3) Enforce isolation (usually with help from hardware)
  - Each user program thinks it has the hardware to itself
    - User programs unaware of other programs or (mostly) OS
  - Makes programs much easier to write
  - Makes the whole system more stable and secure
    - A can't mess with B if it doesn't even know B exists

41

41

Implementing an OS: Privilege

OS isolates user programs from each other and itself

- Requires restricted access to certain parts of hardware to do this
- Restricted access should be enforced by hardware
- Acquisition of restricted access should be possible, but restricted

Restricted access mechanism is called privilege

- Hardware supports two privilege levels

"Supervisor" or "privileged" mode

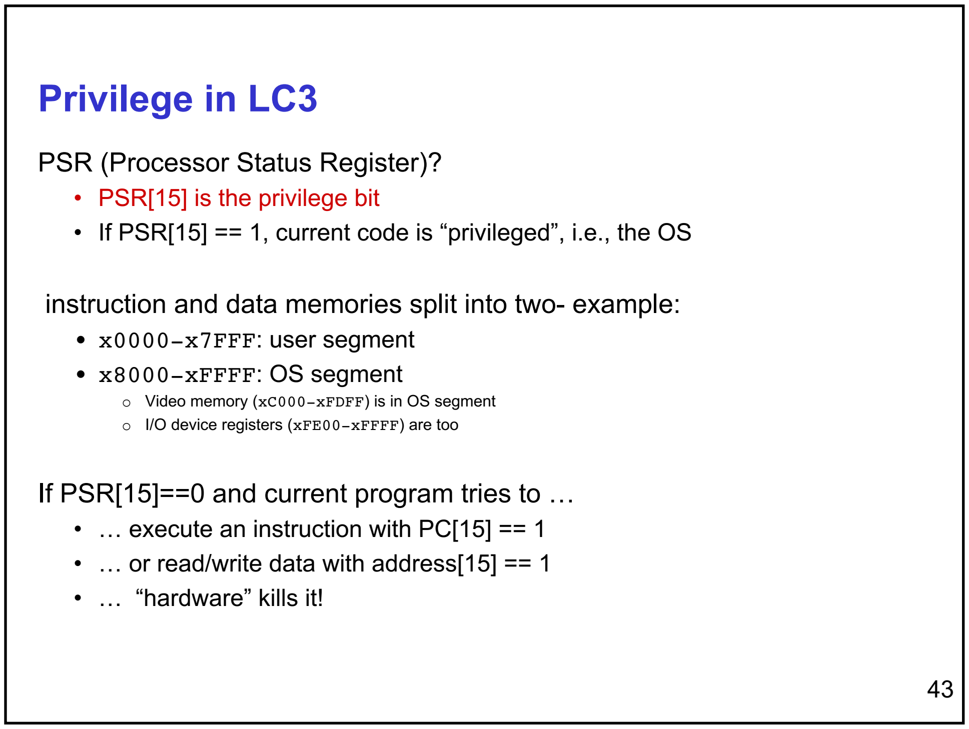
- Processor can execute any code, read/write any data

"User" or "unprivileged" mode

- Processor may not execute some code, read/write some memory
  - E.g., cannot read/write video memory or device registers

42

42



Privilege in LC3

PSR (Processor Status Register)?

- PSR[15] is the privilege bit
- If PSR[15] == 1, current code is “privileged”, i.e., the OS

instruction and data memories split into two- example:

- `x0000-x7FFF`: user segment
- `x8000-xFFFF`: OS segment
  - Video memory (`xC000-xFDFF`) is in OS segment
  - I/O device registers (`xFE00-xFFFF`) are too

If PSR[15]==0 and current program tries to …

- … execute an instruction with PC[15] == 1
- … or read/write data with address[15] == 1
- … “hardware” kills it!

43

43