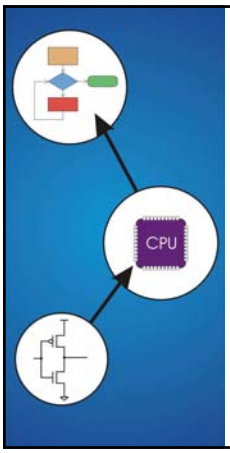
Introduction to Computer Engineering

ECE/CS 252, Fall 2008
Prof. Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin – Madison



Chapter 8 & 9.1
I/O and Traps

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?

And how does data get out of the system so that humans can use it?

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

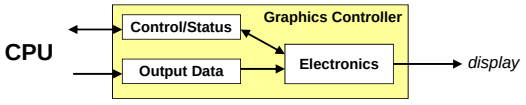
I/O Controller

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

Programming Interface

How are device registers identified?

- Memory-mapped vs. special instructions

How is timing of transfer managed?

- Asynchronous vs. synchronous

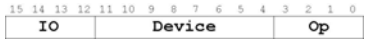
Who controls transfer?

- CPU (polling) vs. device (interrupts)

Memory-Mapped vs. I/O Instructions

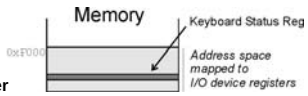
Instructions

- designate opcode(s) for I/O
- register and operation encoded in instruction



Memory-mapped

- assign a memory address to each device register
- use data movement instructions (LD/ST) for control and data transfer



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

Transfer Control

Who determines when the next data transfer occurs?

Polling

- CPU keeps checking status register until ***new data*** arrives OR ***device ready*** for next data
- "Are we there yet? Are we there yet? Are we there yet?"

Interrupts

- Device sends a special signal to CPU when ***new data*** arrives OR ***device ready*** for next data
- CPU can be performing other tasks instead of polling device.
- "Wake me when we get there."

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.

Asynchronous devices

- synchronized through status registers

Polling and Interrupts

- the details of interrupts will be discussed in Chapter 10

Input from Keyboard

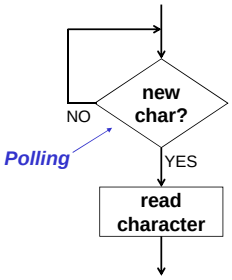
When a character is typed:

- its ASCII code 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:

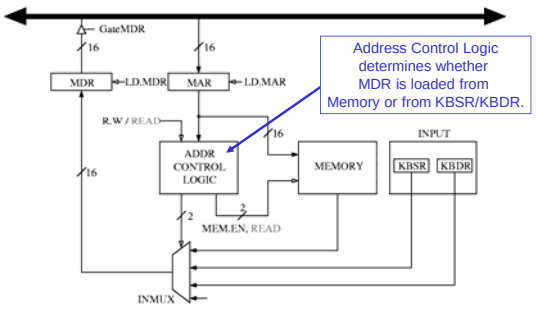
- KBSR[15] is set to zero
- keyboard is enabled

Basic Input Routine




```
POLL    LDI  R0, KBSRPtr
        BRzp POLL
        LDI  R0, KBDRPtr
        ...
KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

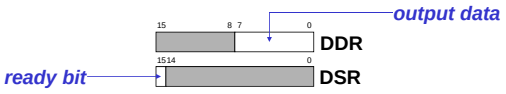
Simple Implementation: Memory-Mapped Input



Output to Monitor

When Monitor is ready to display another character:

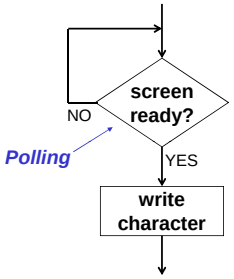
- **the "ready bit" (DSR[15]) is set to one**



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)**

Basic Output Routine

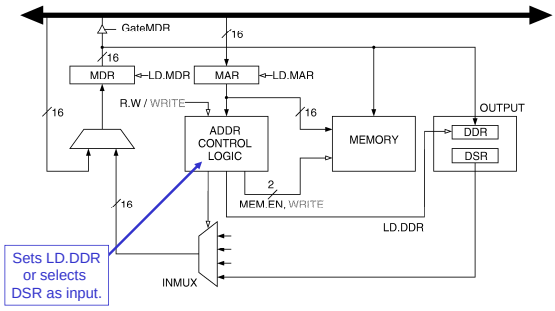



```
POLL    LDI  R1, DSRPtr
        BRzp POLL
        STI  R0, DDRPtr

        ...

DSRPtr .FILL xFE04
DDRPtr .FILL xFE06
```


Simple Implementation: Memory-Mapped Output



Keyboard Echo Routine

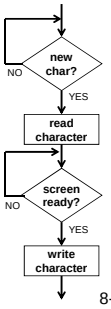
Usually, input character is also printed to screen.

- **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
```



Interrupt-Driven I/O

External device can:

(1) Force currently executing program to stop;

(2) Have the processor satisfy the device's needs; and

(3) Resume the stopped program as if nothing happened.

Why?

- **Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.**
- **Example: Process previous input while collecting current input. (See Example 8.1 in text.)**

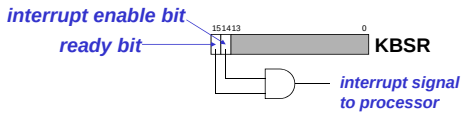
Interrupt-Driven I/O

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 and whether its priority is higher than the current program.**

Generating Signal

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



Priority

Every instruction executes at a stated level of urgency.

LC-3: 8 priority levels (PL0-PL7)

- **Example:**
  - **Payroll program runs at PL0.**
  - **Nuclear power plant control program runs at PL6.**
- **It's OK for PL6 device to interrupt PL0 program, but not the other way around.**

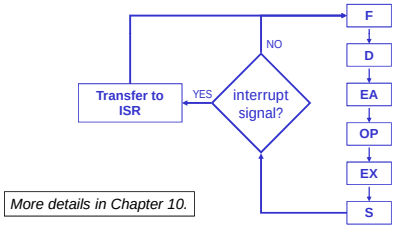
Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.

Testing for Interrupt Signal

CPU looks at signal between STORE and FETCH phases.

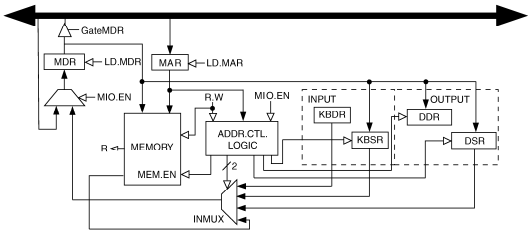
If not set, continues with next instruction.

If set, transfers control to interrupt service routine.



More details in Chapter 10.

Full Implementation of LC-3 Memory-Mapped I/O



Because of interrupt enable bits, status registers (KBSR/DSR) must be written, as well as read.

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

System Call

1. User program invokes system call.

2. Operating system code performs operation.

3. Returns control to user program.

In LC-3, this is done through the *TRAP mechanism*.

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

TRAP Instruction

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

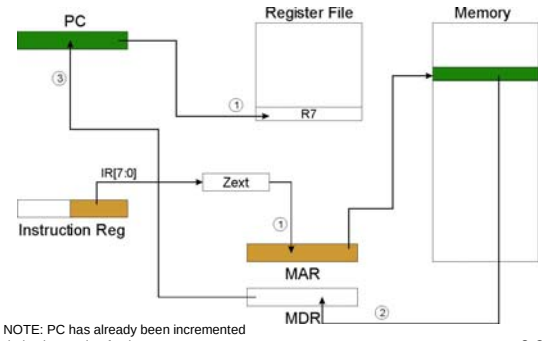
Where to go

- lookup starting address from table; place in PC

How to get back

- save address of next instruction (current PC) in R7

TRAP



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

RET (JMP R7)

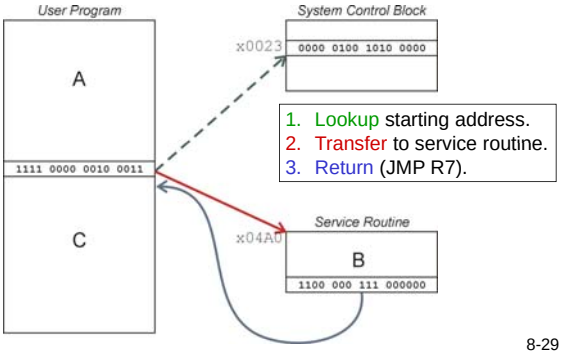
How do we transfer control back to instruction following the TRAP?

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.

TRAP Mechanism Operation



Example: Using the TRAP Instruction

```
        .ORIG x3000
        LD   R2, TERM    ; Load negative ASCII '7'
        LD   R3, ASCII   ; Load ASCII difference
AGAIN   TRAP  x23        ; input character
        ADD  R1, R2, R0  ; Test for terminate
        BRz  EXIT        ; Exit if done
        ADD  R0, R0, R3  ; Change to lowercase
        TRAP  x21        ; Output to monitor...
        BRnzp  AGAIN     ; ... again and again...
TERM    .FILL xFFC9      ; -'7'
ASCII   .FILL x0020      ; lowercase bit
EXIT    TRAP  x25        ; halt
        .END
```

Example: Output Service Routine

```
            .ORIG x0430       ; syscall address
            ST    R7, SaveR7  ; save R7 & R1
            ST    R1, SaveR1
; ----- Write character
TryWrite    LDI   R1, CRTSR   ; get status
            BRzp  TryWrite    ; look for bit 15 on
WriteIt     STI   R0, CRTDR   ; write char
; ----- Return from TRAP
Return      LD    R1, SaveR1  ; restore R1 & R7
            LD    R7, SaveR7
            RET               ; back to user
CRTSR       .FILL xF3FC
CRTDR       .FILL xF3FF
SaveR1      .FILL 0
SaveR7      .FILL 0
            .END
```

stored in table, location x21

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

Saving and Restoring Registers

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**

Example

```
            LEA  R3, Binary
            LD   R6, ASCII   ; char->digit template
            LD   R7, COUNT   ; initialize to 10
AGAIN       TRAP x23         ; Get char
            ADD  R0, R0, R6  ; convert to number
            STR  R0, R3, #0  ; store number
            ADD  R3, R3, #1  ; incr pointer
            ADD  R7, R7, -1  ; decr counter
            BRp  AGAIN       ; more?
            BRnzp NEXT
ASCII       .FILL xFFD0
COUNT       .FILL #10
Binary      .BLKW #10
```

What's wrong with this routine? What happens to R7?

Saving and Restoring Registers

Called routine -- *"callee-save"*

- **Before start, save any registers that will be altered (unless altered value is desired by calling program!)**
- **Before return, restore those same registers**

Calling routine -- *"caller-save"*

- **Save registers destroyed by own instructions or by called routines (if known), if values needed later**
  - ➢ **save R7 before TRAP**
  - ➢ **save R0 before TRAP x23 (input character)**
- **Or avoid using those registers altogether**

Values are saved by storing them in memory.

Summary

Chapter 8: Input/output

- **Behavior and data rate of I/O device**
- **Asynchronous vs. synchronous**
- **Polled vs. interrupt-driven**
- **Programmed vs. memory-mapped**
- **Control registers, data registers**

Chapter 9: Traps and System Calls

- **Hide details of I/O device interaction**
- **TRAP/RET instructions**
- **Caller- vs callee-saved registers**