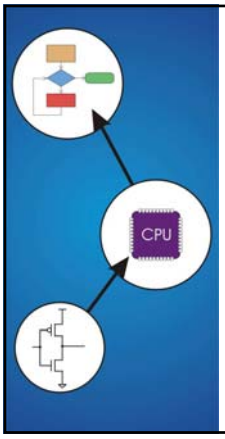
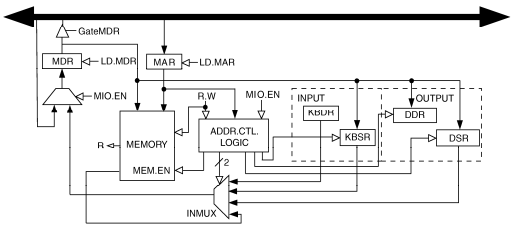
Introduction to Computer Engineering

ECE/CS 252, Fall 2010
Prof. Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin – Madison



Chapter 9.1 Traps

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



Very simple I/O system, yet many messy details

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**

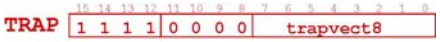
3. TRAP instruction.

- **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

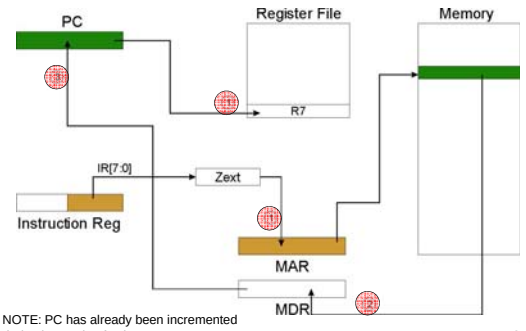
Where to go

- 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)

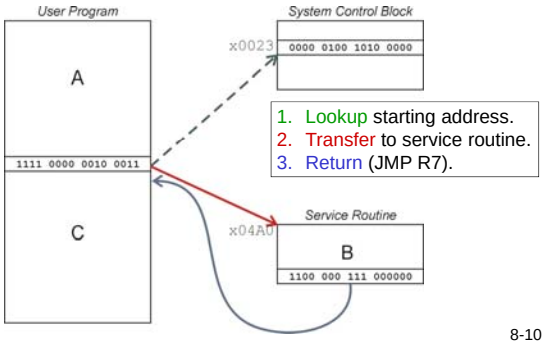
How do we transfer control back to instruction following the TRAP?

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, DSR    ; get status
            BRzp  TryWrite   ; look for bit 15 on
WriteIt     STI   R0, DDR    ; write char
; ----- Return from TRAP
Return      LD    R1, SaveR1 ; restore R1 & R7
            LD    R7, SaveR7
            RET              ; back to user
DSR         .FILL xFE04
DDR         .FILL xFE06
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?

Question

- **Can a service routine call another service routine?**
  - **Yes, of course, just as with nested subroutines**
- **If so, is there anything special the calling service routine must do?**
  - **Make sure its return address is not "lost"**
  - **On entry to service routine, R7 contains its return address**
  - **When it invokes TRAP again, R7 is overwritten**
  - **Must save R7 in memory (store) before TRAP**
  - **Must restore R7 from memory (load) after TRAP**

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.

What about User Code?

- **Service routines provide three main functions:**
- **1. Shield programmers from system-specific details.**
- **2. Write frequently-used code just once.**
- **3. Protect system resources from malicious/clumsy programmers.**
- **Are there any reasons to provide the same functions for non-system (user) code?**

Review: Subroutines (Chapter 9.2)

- A **subroutine** is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by another user program
  - returns control to the calling program when finished
- Like a service routine, but not part of the OS
  - not concerned with protecting hardware resources
  - no special privilege required
- Reasons for subroutines:
  - reuse useful (and debugged!) code without having to keep typing it in
  - divide task among multiple programmers
  - use vendor-supplied *library* of useful routines

Library Routines

- Vendor may provide object files containing useful subroutines
  - don't want to provide source code -- intellectual property
  - assembler/linker must support EXTERNAL symbols (or starting address of routine must be supplied to user)
- ```
  ...
  .EXTERNAL SQRT
  ```
- ```
  ...
  LD   R2, SQAddr   ; load SQRT addr
  JSRR R2
  ...
  SQAddr  .FILL SQRT
  ```
- Using JSRR, because we don't know whether SQRT is within 1024 instructions.

Summary

Hide details of and protect I/O device interaction

TRAP/RET instructions

Caller- vs callee-saved registers

Library Routines