

Chapter 9

TRAP Routines and Subroutines

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetia

How to Perform I/O?

Require specialized knowledge and protection

- **Knowledge** of I/O device registers and how to use them
Programmers don't want to know this!
- **Protection** for shared I/O resources e.g. Hard- Disk
Want process isolation

Solution: *service routines* or *system calls*

- Low-level, privileged operations performed by operating system
- Used by almost all applications and hence made a routine (regular course of procedure)
- Example: keyboard & display polling routine for LC-3

CIT 593

2/37

System Call

1. User program invokes system call
 - i.e. calls the part of the system code that deals with I/O
2. Operating system code performs operation
3. Returns control to user program

In LC-3: done via *TRAP mechanism*

CIT 593

3/37

LC3 Trap Mechanism

Consists of Trap Vector Table

- Used to associate code with trap number
 - The table contains the address of where the service routine is located
- The table is stored in memory (x0000 through x00FF in LC3)

Vector Table (Memory location)	Routine location	Routine
x21	x0430	output a character to the monitor
x23	x04A0	input a character from the keyboard
x25	xFD70	halt the program (HALT)

Note: The service routines are located at location at x0200 - x2FFF in memory except the HALT routine

CIT 593

4/37

How do we get back to user program?

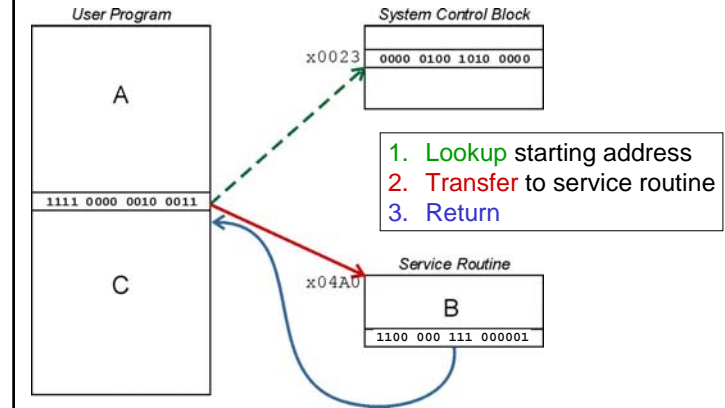
By using RETURN instruction

- Returns control to the user program
- Execution resumes immediately after the TRAP instruction

CIT 593

5/37

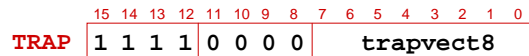
TRAP Mechanism Operation



CIT 593

6/37

TRAP Instruction



Trap vector

- Identifies which system call to invoke
- Serves as index into table of service routine addresses
 - LC-3: table stored in memory at `0x0000 – 0x00FF`
 - 8-bit trap vector zero-extended to form 16-bit address, thus allows upto 256 service routines

Where to go

- Lookup starting address from table; place in PC

Enabling return

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

How to return

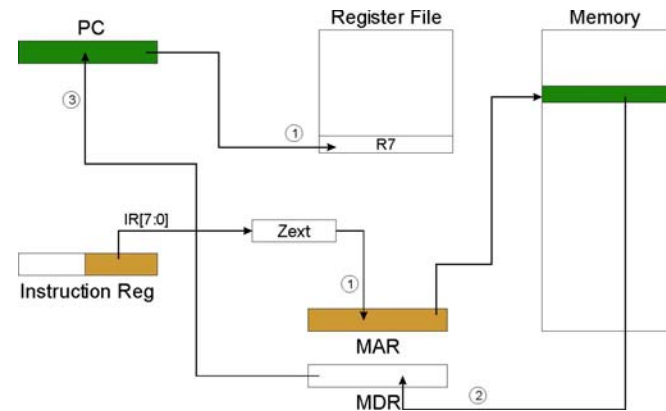
- Place address in R7 in PC

CIT 593

7/37

TRAP

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



CIT 593

8/37

RET

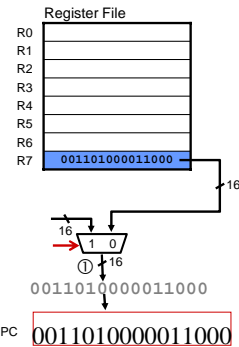
```

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

```

JUMP

R7



(special case of JMP)

CIT 593

9/37

Generic Trap Service Routine

```
.ORIG address ; Where routine is located in memory
```

```

;Do some operations using local registers
;Check status registers if they ready for I/O
;If Device Ready: Perform the I/O

```

```
RET
```

```
.END
```

CIT 593

10/37

Example1: Using the TRAP Instruction

```

.ORIG x3000
LD R2, TERM ; Load negative ASCII '7'
LD R3, ASCII ; Load ASCII difference
AGAIN TRAP x23 ; Input character
;After returning from TRAP, R0 contains the character
ADD R1, R2, R0 ; Test for terminating char
BRz EXIT ; Exit if done
ADD R0, R0, R3 ; Change to lowercase
TRAP x21 ; Output to monitor...
BRnzp AGAIN ; ... and again repeat...
EXIT TRAP x25 ; Halt

TERM .FILL xFFC9 ; -7'
ASCII .FILL x0020 ; Lowercase bit
.END

```

CIT 593

11/37

Example2: Using the TRAP Instruction

```

LEA R3, Block ; Init. to first loc.
LD R6, ASCII ; Char->digit template
LD R7, COUNT ; Init. 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?
HALT

ASCII .FILL xFFD0 ; - x0030
COUNT .FILL #10
Block .BLKW #10

```

What's wrong with this code?

CIT 593

12/37

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 later

Who saves?

- Caller i.e the program that calls the service routine
OR
- Called service routine i.e. the program invoked to perform the service routine

CIT 593

13/37

Caller of Service Routine

What does the caller know?

- Knows what it needs later
- May/should not know what gets altered by service routine

Example

```
LEA R3, Block ; Init. to first loc.
LD R6, ASCII ; Char->digit template
LD R7, COUNT ; Init. 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?
HALT
ASCII .FILL xFFD0 ; Negative of x0030
COUNT .FILL #10
Block .BLKW #10
```

CIT 593

14/37

Called Service Routine (Callee)

What does the callee know?

- Knows what it alters
- Does not know what will be needed later (by calling routine)

Example

```
.ORIG x0430 ; Syscall x21 address
TryWrite LDI R1, DSR ; Get status
BRzp TryWrite ; Bit 15 says not ready?
WriteIt STI R0, DDR ; Write char
Return RET ; Return from trap
DSR .FILL xFE04
DDR .FILL xFE06
.END
```

CIT 593

15/37

Saving and Restoring Registers

Called routine ⇒ “callee-save”

- Before start, save registers that will be altered (unless altered value is desired by calling program!)
- Before return, restore those same registers
- Values are saved by storing them in memory

Calling routine ⇒ “caller-save”

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

LC-3: By convention, callee-saved

CIT 593

16/37

Example: Character Output Service Routine

```

        .ORIG x0430      ; Syscall x21 address
        ST  R1, SaveR1  ; Save R1

; Write character
TryWrite LDI  R1, DSR    ; Get status
        BRzp TryWrite  ; Bit 15 says not ready?
WriteIt  STI  R0, DDR    ; Write char
; Return from TRAP
Return   LD   R1, SaveR1 ; Restore R1
        RET   ; Return from trap

DSR      .FILL xFE04
DDR      .FILL xFE06
SaveR1   .FILL #0

        .END
    
```

stored in table,
location x21

CIT 593

17/37

Privilege

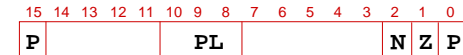
Goal: Isolation

- OS performs I/O (in traps)
- Application can't perform I/O directly

How is this enforced?

Privilege: Processor modes

- Privileged (supervisor)
- Unprivileged (user)
- Encoded in 15th bit of processor status register (PSR)



CIT 593

18/37

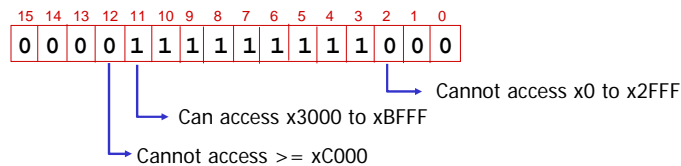
Supervisor Mode Versus User Mode

Supervisor mode

- Program has access to resources not available to user programs

User mode in LC-3

- Memory access is limited by memory protection register (MPR)
- Each MPR bit corresponds to 4K memory segment
- 1 indicates that users can access memory in this segment



CIT 593

19/37

MPR

MPR Prevents user from. . .

- Updating trap table
- Changing OS code
- Accessing video memory
- Accessing memory-mapped I/O registers (e.g., DDR, DSR)
- Could be different for each application

CIT 593

20/37

Managing Privilege

Who sets privilege bit in PSR?

- TRAP instruction

Who clears privilege bit?

- However RET (in simulator) does not unset the privilege
 - So if return from trap still in privilege mode and user can access restricted memory now !!
- New instruction RTT (Note: not in book!)
 - So to return from TRAP use RTT instead of RET
 - More on this next

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

CIT 593

21/37

What about User Code?

Service routines provide three main functions

1. Protect system resources from malicious/clumsy programmers
2. Shield programmers from system-specific details
3. Write frequently-used code just once

Do these benefits apply to user code, too?

CIT 593

22/37

Subroutines

A **subroutine** is a program fragment that. . .

- Resides in user space (i.e, not in OS)
- Performs a well-defined task
- Is invoked (called) by a 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

Virtues

- Reuse useful code without having to keep typing it in (and debugging it!)
- Divide task among multiple programmers
- Use vendor-supplied *library* of useful routines

CIT 593

23/37

Opcode for CALLING a Subroutine

JSR/JSRR – saves the return address in R7 and computes the the starting address of the subroutine and loads it into PC

JSR

- PC-relative mode (just like PC-relative LD/ST)
- Target address of the subroutine is incremented PC + offset

JSR 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 1 0 0 1 PCoffset11

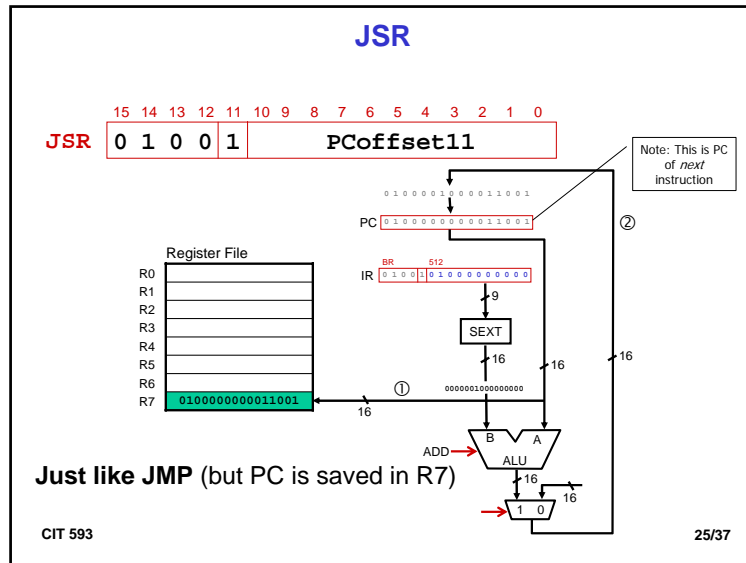
JSRR

- Base addressing mode
- Target address of the subroutine is obtained from Base Register

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

CIT 593

24/37



Returning From a Subroutine

Same idea as returning from TRAPS

- Use RET
 - Just a special case of JMP i.e. RET == JMP R7

Note

- If we use JMP to call subroutine instead of JSR/JSRR, we can't use RET to return from subroutine!
- Why not?

CIT 593 26/37

Example: Routine that negates a number

```

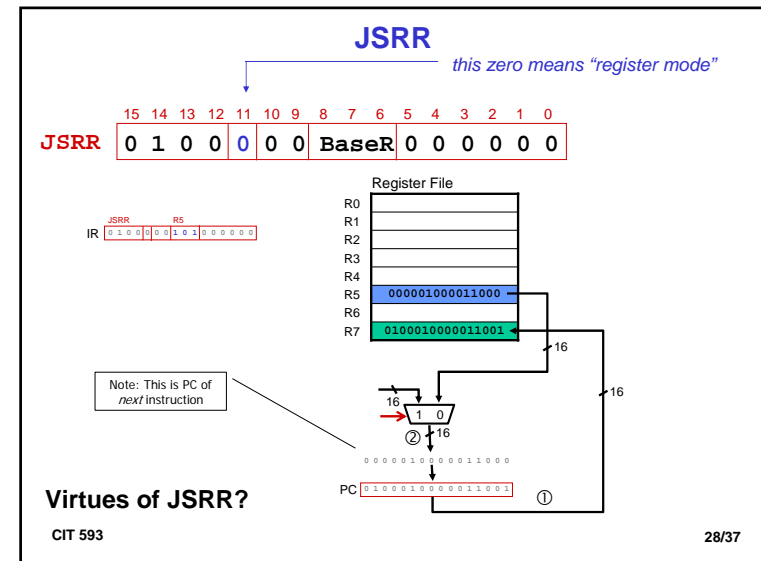
.ORIG x3000
DoSomething1 .
.
.
.
;need to compute R4 = R1 - R3
x3005 ADD R0, R3, #0 ; copy R3 to R0
x3006 JSR TwosComp ; negate
x3007 ADD R4, R1, R0 ; add to R1
x3008 BRz DoSomething2
TwosComp NOT R0, R0 ; R0 is the input to the routine
ADD R0, R0, #1 ; add one
RET ; return to caller

DoSomething2 .
.
.

```

**Note: Assume that in this example R0 is the input to be negated
Therefore, Caller should save R0 if need it later!**

CIT 593 27/37



Passing Information To Subroutines

Argument(s)

- Value **passed in** to a subroutine is called an argument
- This is a value needed by the subroutine to do its job
- Examples
 - TwosComp: R0 is number to be negated
 - OUT: R0 is character to be printed

How?

- In registers (simple, fast, but limited number)
- In memory (many, but expensive)
- Both

CIT 593

29/37

Getting Values From Subroutines

Return Values

- A value **passed out** of a subroutine is called a return value
- This is the value that you called the subroutine to compute
- Examples
 - TwosComp: negated value is returned in R0
 - GETC: character read from the keyboard is returned in R0

How?

- Registers, memory, or both
- Single return value in register most common

CIT 593

30/37

Using Subroutines

Programmer must know

- **Address:** or at least a label that will be bound to its address
- **Function:** what it does
 - NOTE: The programmer does not need to know *how* the subroutine works, but what changes are visible in the machine's state after the routine has run
- **Arguments:** what they are and where they are placed
- **Return values:** what they are and where they are placed

CIT 593

31/37

Calling Conventions for subroutines

Caller/Callee must agree on argument/ret-val location

Approach 1

- Every subroutine does what it likes
- Program needs to look at documentation for each one

Approach 2

- Define a consistent *calling convention*

LC-3 argument/ret-val location

- First 4 arguments passed in R0, R1, R2, R3
- Single value returned in register other than R7
- Subsequent arguments passed in memory

CIT 593

32/37

Saving and Restore Registers

Like service routines, must save and restore registers

- Who saves what is part of the calling convention

Generally use “callee-save” strategy, except for ret vals

- Same as trap service routines
- Save anything that subroutine alters internally that shouldn't be visible when the subroutine returns
- Restore incoming arguments to original values (unless overwritten by return value)

CIT 593

33/37

Example of Callee Save for subroutines

```
TwosComp:  ;good to R7 even if u don't use it
           ST  R7, SAVER7;save R7
           ST  R1, Saver1;save R1
           NOT R0, R0    ;R0 is the input to the routine
           AND R1, R1, #0
           ADD R1, R1, #1
           ADD R0, R0,R1
           LD  R7, SAVER7 ;restore R7
           LD  R1, SAVER1
           RET
SAVER7:   .FILL x0000
SAVER0:   .FILL x0000
```

CIT 593

34/37

Library Routines

Vendor provide object files containing useful subroutines

- Don't want to provide source code (intellectual property)
- Assembler/linker must support EXTERNAL symbols

```
...
.EXTERNAL SQRT
...
LD  R2, SQAddr    ; load SQRT addr
JSRR R2
...
SQAddr .FILL SQRT
```

Using JSRR, because SQRT likely not “nearby”

CIT 593

35/37

Linking and Loading

Linking is the process of resolving symbols between independent object files

- Notation, such as .EXTERNAL, is used to tell assembler that a symbol is defined in another module
- Linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

Loading is the process of copying an executable (machine code) image into memory

CIT 593

36/37

Next

Still need to talk about:

- Local vs. global data
- What extra things need to be done for recursion i.e. a method calling itself?
- Answers in Chapter 14

Next Time chapter 10