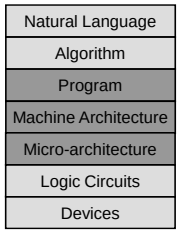
These notes are based on Dr Delmas' and many other people's notes

Chapter 9

TRAP Routines and Subroutines



Agenda & Reading

- System call
- Subroutines

- Read Chapter 9 of the textbook

Agenda

- **System call**
  - **What are system calls**
  - **How are system calls handled**
  - **Issues that need to be aware of**
- Subroutines

System Calls

Certain operations require specialized knowledge and protection:

- specific knowledge of I/O device registers and the sequence of operations is needed to use them
- I/O resources are 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

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

The Steps in Making 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

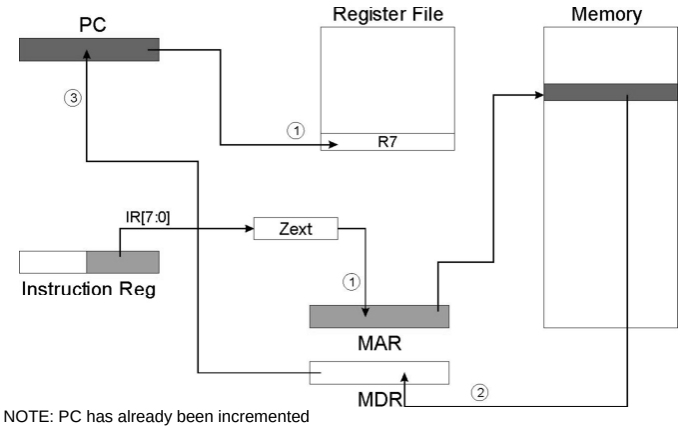
TRAP Instruction

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; then place the address 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) instruction

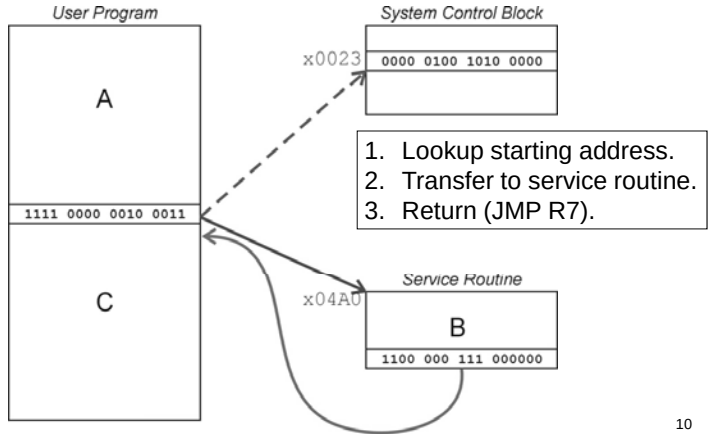
How do we transfer control back to instruction following the TRAP?

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
        LEA     R0,     Msg
        PUTS
        LEA     R1, Num         ; R1 points to the first location for
                                ; storing the input value
Next    LD      R2, Mq
        TRAP    x23             ; read in one character
        ADD     R3, R2, R0      ; check whether the input is "q"
        BRz     End             ; if input is "q", terminate
        LD      R2, M30
        ADD     R0, R0, R2      ; obtain the value of the input digit
        STR     R0, R1, 0       ; store the value in memory
        ADD     R1, R1, 1       ; R1 points to the next location for
                                ; storing the value
        BRnzp   Next
End     TRAP    x25
```



```
Msg     .STRINGZ        "Example:"
Num     .BLKW  10
Mq      .FILL   -113    ; -1 * the ASCII value of "q"
M30     .FILL   -48     ; -1 * the ASCII value of "0"
        .END
```

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 calling the service 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?
          RET
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.

Agenda

- System call
- **Subroutines**
  - **What are subroutines**
  - **How to pass information to/from subroutines**
  - **How to preserve register values**

Subroutines

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

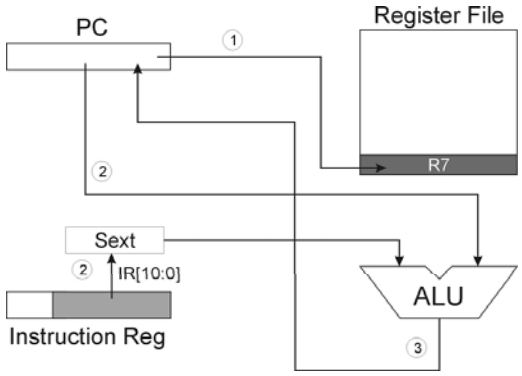
Like a service routine, but not part of the OS
- not concerned with protecting hardware resources
- no special privilege required

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

JSR Instruction

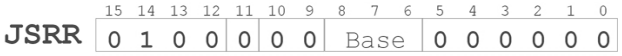
Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.
- saving the return address is called "linking"
- target address is PC-relative **(PC + Sext(IR[10:0]))**
- bit 11 specifies addressing mode
  - if =1, PC-relative: target address = PC + Sext(IR[10:0])
  - if =0, register: target address = contents of register IR[8:6]

JSR



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

JSRR Instruction

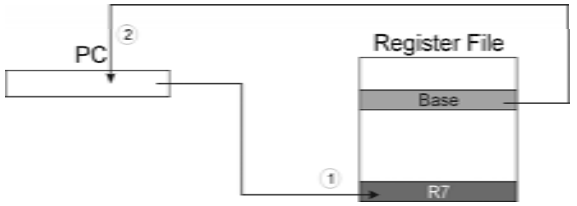


Just like JSR, except Register addressing mode.

- target address is Base Register
- bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?

JSRR



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

Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.

JSR (Jump to SubRoutine)

- The JSR instruction is used to call a subroutine
- JSR Label
  - Label is the name of the subroutine. It is the label assigned to the first instruction of the subroutine.
  - JSR Negate

RET (RETurn)

- The RET instruction is used as the last instruction in a subroutine. It allows the execution returns to the caller of the subroutine.
- RET


```
                .ORIG x3000
; This program evaluates expression: R0 ← (R1)-(R2)
                LD      R1, N1
                LD      R2, N2
                JSR     Negate
                ADD     R0, R1, R2
                HALT

; A subroutine for negating the value stored in R2
; The negated number is stored in R2
Negate          NOT     R2, R2
                ADD     R2, R2, #1
                RET

N1              .FILL 12
N2              .FILL 11
                .END
```


JSRR (Jump to SubRoutine Register-mode)

- The JSRR instruction is used to call a subroutine
  - It is similar to JSR except the address of the subroutine is given in a register
- JSRR BR
  - BR is the register holding the address of the first instruction of the subroutine.
  - JSRR R4


```
                .ORIG x3000

                LD      R1, N1
                LD      R2, N2
                LD      R4, Address     ; load the address of the
                                        ; subroutine to R4
                JSRR    R4
                ADD     R0, R1, R2
                HALT

Negate          NOT     R2, R2          ; the address is x3006
                ADD     R2, R2, #1
                RET

N1              .FILL 12
N2              .FILL 11
Address         .FILL x3006     ; the address of the subroutine is
                                ; x3006
                .END
```

Passing Information to/from Subroutines

Arguments

- A value passed in to a subroutine is called an argument.
- This is a value needed by the subroutine to do its job.
- Examples:
  - In Negate routine, R2 is the number to be negated
  - In OUT service routine, R0 is the character to be printed.
  - In PUTS routine, R0 is *address* of string to be printed.

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:
  - In Negate routine, negated value is returned in R2.
  - In GETC service routine, character read from the keyboard is returned in R0.

Using Subroutines

In order to use a subroutine, a programmer must know:

- its address (or at least a label that will be bound to its address)
- its function (what does it do?)
  - 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.
- its arguments (where to pass data in, if any)
- its return values (where to get computed data, if any)

Saving and Restore Registers

Since subroutines are just like service routines, we also need to save and restore registers, if needed.

Generally use "callee-save" strategy, except for return values.

- Save anything that the subroutine will alter internally that shouldn't be visible when the subroutine returns.
- It's good practice to restore incoming arguments to their original values (unless overwritten by return value).

Remember: You MUST save R7 if you call any other subroutine or service routine (TRAP).

- Otherwise, you won't be able to return to caller.


```
            .ORIG x3000
; This program evaluates expression: R0 = (R1)-(R2)
            LD      R1, N1
            LD      R2, N2
            JSR     Negate
            ADD     R0, R1, R2
            HALT

; A subroutine for negating the value stored in R2
; The negated number is stored in R2
Negate      ST      R3, SavedR3     ; save the value in R3
            NOT     R3, R2
            ADD     R2, R3, #1
            LD      R3, SavedR3     ; restore the value in R3
            RET

N1          .FILL 12
N2          .FILL 11
SavedR3     .FILL 0         ; location for holding the save value
            .END
```

reviews

- What are system calls
- How system calls are handled
  - TRAP, RET
- What are subroutines
- How to write subroutines
  - Pass information to/from subroutines
  - Save and restore values
  - JSR, JSRR, RET