Lecture 24 11/3/20

Sunday, November 01, 2020 2:25 PM

Check Your Grades! 1 Week Rule!

Problem Session #8 This Week!

HW #9 Due 11/5 Quiz 4 11/10/20

Program #3 Due 11/12/20 Chps Instrn Flow charts

Late Assignments -10% per day! 7, 8, 9 all.

Today:

Chapter 9 Subroutines

From Syllabus

Regrading Requests

If you have discussed your grading on an assignment with your TA and are still not satisfied, you may submit a request to me within **one (1) week** of the graded assignment being returned to you. You must write a cover sheet explaining why you feel you deserve additional points on a given problem, attach it to the front of your graded paper, and give it to me either in class or my office. Regrading requests will **NOT** be considered more than one week after the assignments are returned to the students.

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**

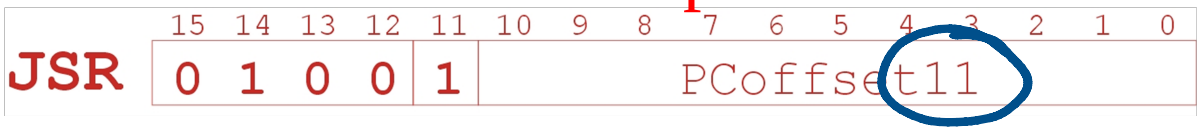
Like a service routine, but not part of the OS

- **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**

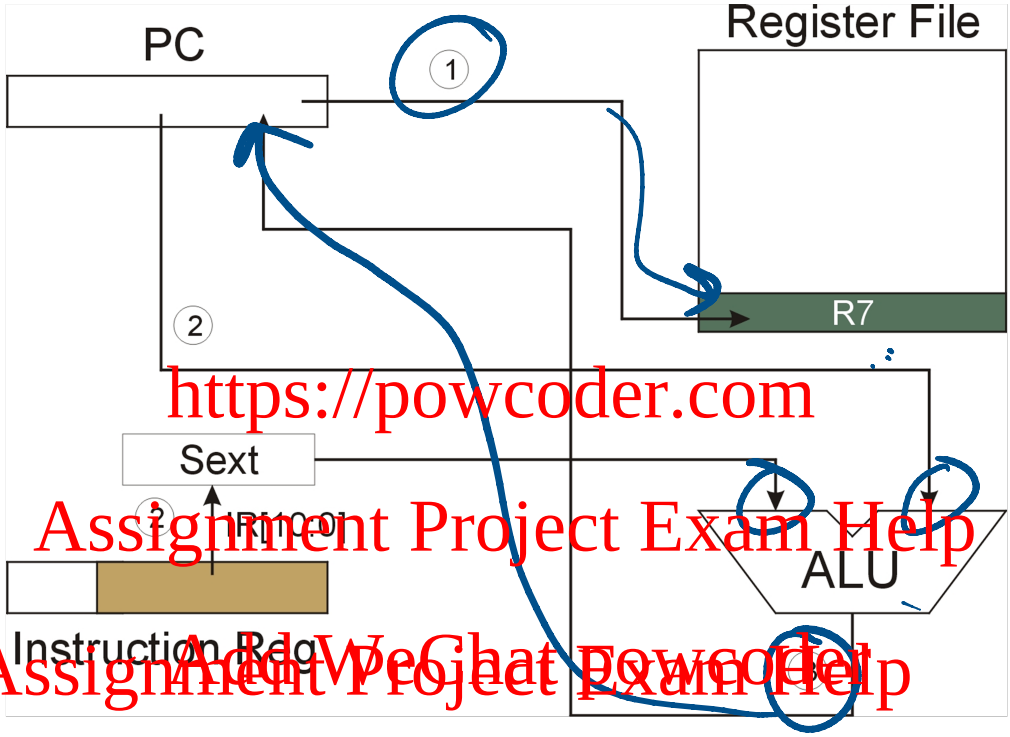
JSR Instruction



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] (called "JSRR" in assembly language)**

JSR



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

"Pirate" Add WeChat powcoder

JSRR Instruction

if bit range

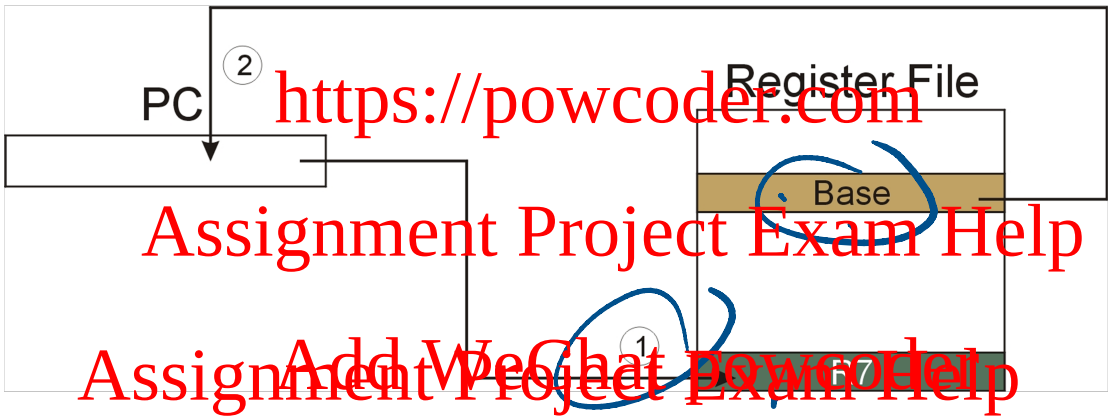
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSRR	0	1	0	0	0	0	0	Base			0	0	0	0	0	0

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.

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.

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.

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: Negate the value in R0

```
2sComp      NOT   R0, R0       ; flip bits
            ADD   R0, R0, #1   ; add one
            RET                ; return to caller
```

To call from a program (within 1024 instructions):

```
; need to compute R4 = R1 - R3
            ADD   R0, R3, #0   ; copy R3 to R0
            JSR   2sComp       ; negate
            ADD   R4, R1, R0   ; add to R1
            ...
```

Note: Caller should save R0 if we'll need it later!

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 2sComp routine, R0 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 2sComp routine, negated value is returned in R0.
  - 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.**

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.

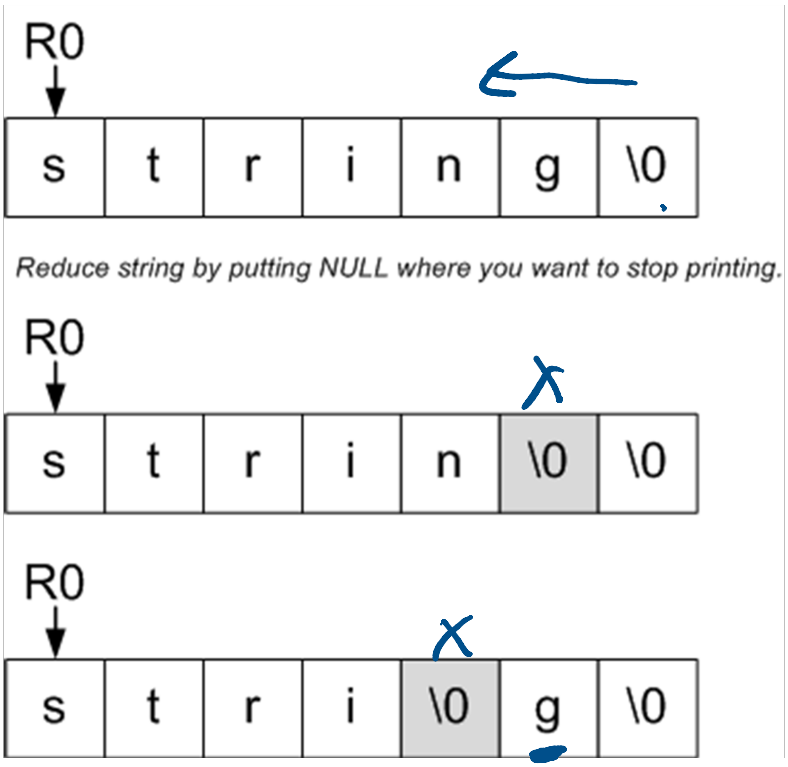
Example Subroutine

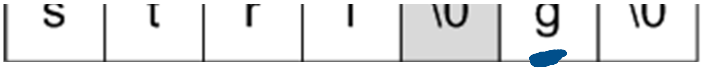
Write an LC-3 subroutine that does the following:

- **User provides the address of a string in R0.**
- **Subroutine prints the string, followed by linefeed (ASCII x0A).**
- **Prints the string again, but only up to the next-to-last char.**
- **Keeps printing the string, dropping one character off the end, until there is nothing left to print.**
- **When the subroutine returns, string should be the same as when we started.**
- **Example: String is "Go State!", result is:**

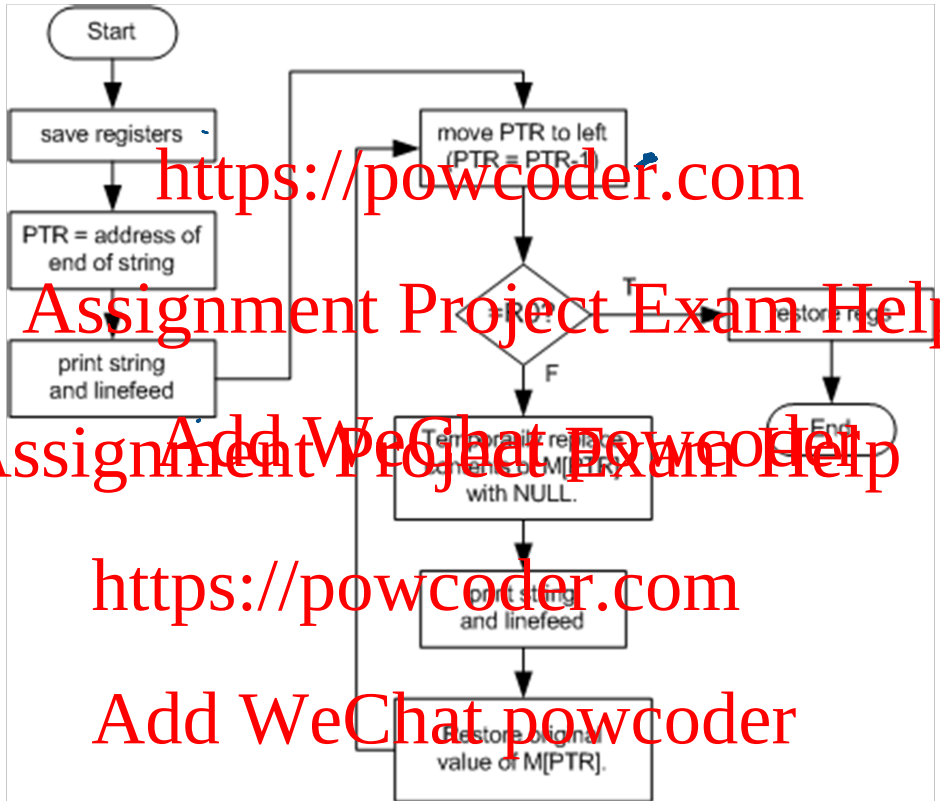
```
Go State!
Go State
Go Stat
Go Sta
Go St
Go S
Go
Go
G
```


Dataflow





Flowchart



Code (1 of 3)

```
RevPr   ST R1, RPR1     ; save regs
        ST R2, RPR2
        ST R3, RPR3
        ST R7, RPR7

        ; make R1 point to end of string
        ADD R1, R0, #0
Rev1    LDR R3, R1, #0  ; check for null
        BRz Rev2
        ADD R1, R1, #1  ; move to next char
        BRnzp Rev1      ; and try again
```

```
        ADD R1, R1, #1   ; move to next char
        BRnzp Rev1       ; and try again

Rev2    PUTS             ; print entire string
        ADD R3, R0, #0   ; save R0 and print LF
        AND R0, R0, #0
        ADD R0, R0, xA
        OUT
        ADD R0, R3, #0   ; restore R0
```


Code (2 of 3)


```
Rev3    ADD R2, R1, #-1  ; move ptr to left
        NOT R3, R1       ; subtract from R0
        ADD R3, R3, #1
        ADD R3, R3, R0
        BRz RevDone      ; exit if equal
```



```
        LDR R2, R1, #0   ; save char
        AND R3, R3, #0   ; put null char in its spot
        STR R3, R1, #0
        PUTS             ; print string and LF
        ADD R3, R0, #0
        AND R0, R0, #0
        ADD R0, R0, xA
        OUT
        ADD R0, R3, #0
        STR R2, R1, #0   ; restore char
        BRnzp Rev3       ; and repeat
```

Code (3 of 3)

```
RevDone     LD R1, RPR1   ; restore regs
            LD R2, RPR2
            LD R3, RPR3
            LD R7, RPR7
            RET           ; return

RPR1        .BLKW 1
RPR2        .BLKW 1
RPR3        .BLKW 1
RPR7        .BLKW 1
```


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

(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

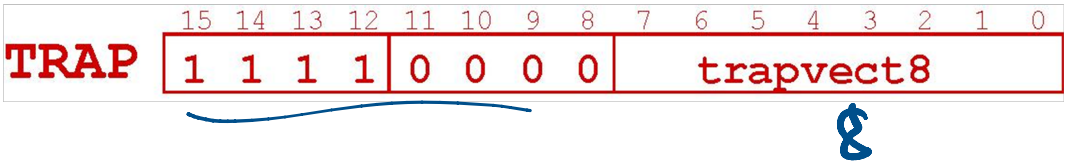
4. A linkage back to the user program.

- want execution to resume immediately after the TRAP instruction

- want execution to resume immediately after the TRAP instruction

Uses R7 for Return!

TRAP Instruction



8

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

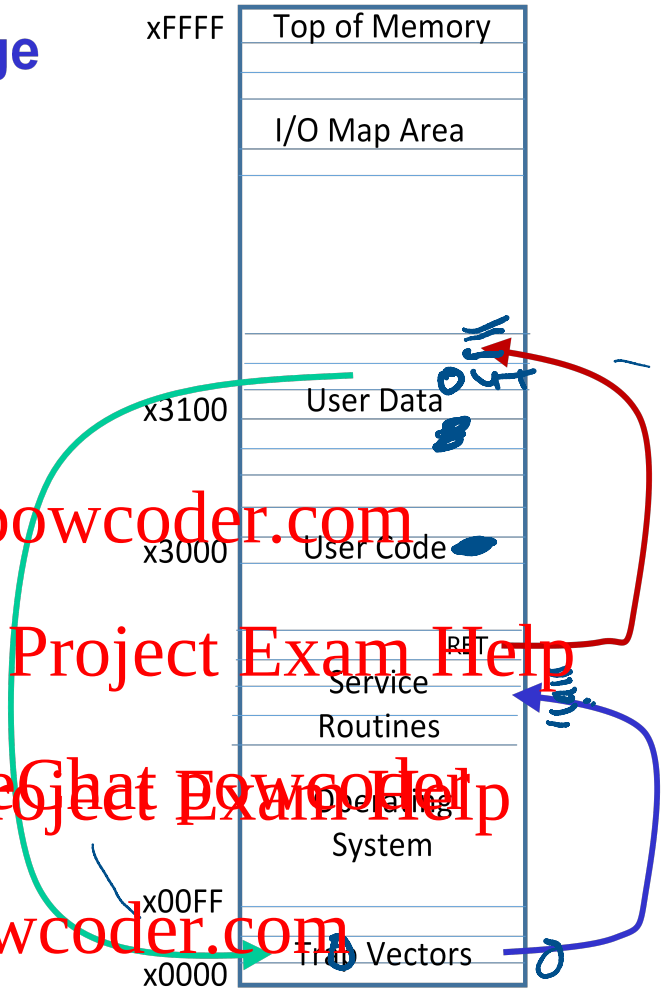
- lookup starting address from table; place in PC

How to get back

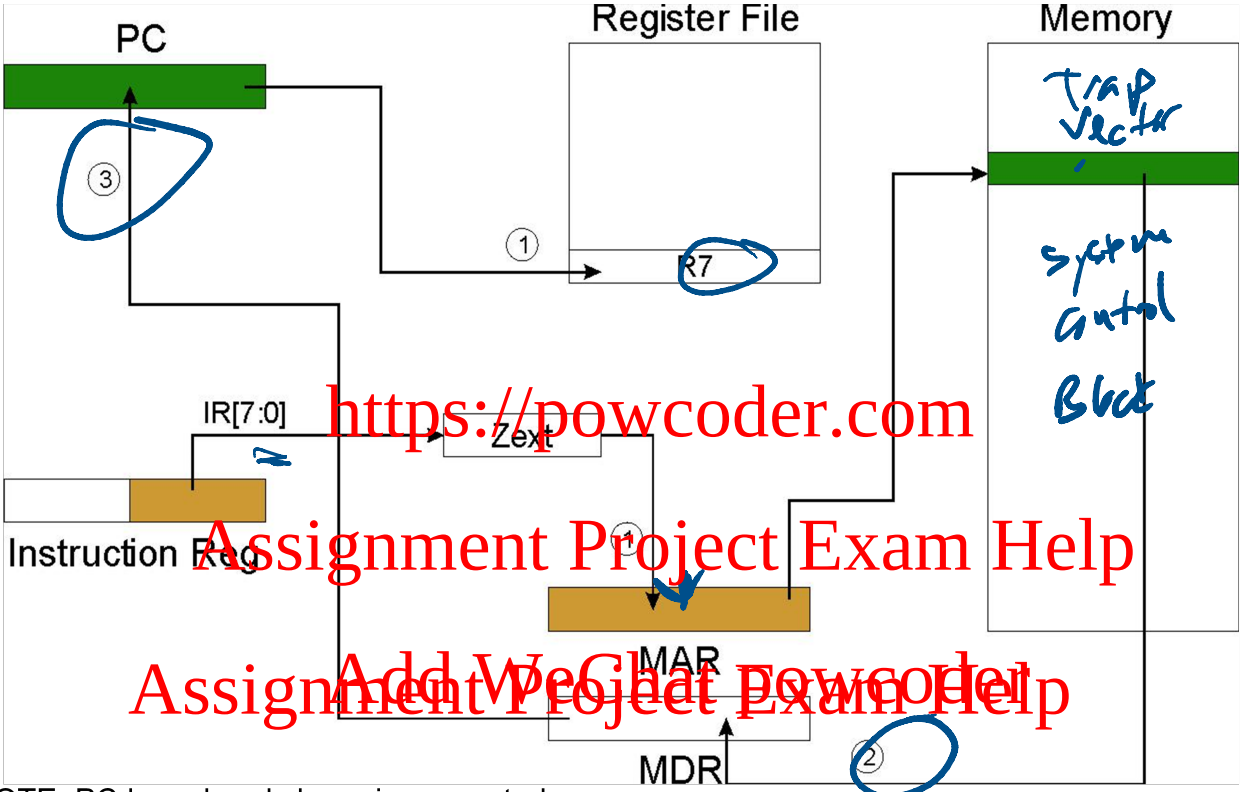
- save address of next instruction (current PC) in R7

Common Memory Usage

16-bit Memory Addressing



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?

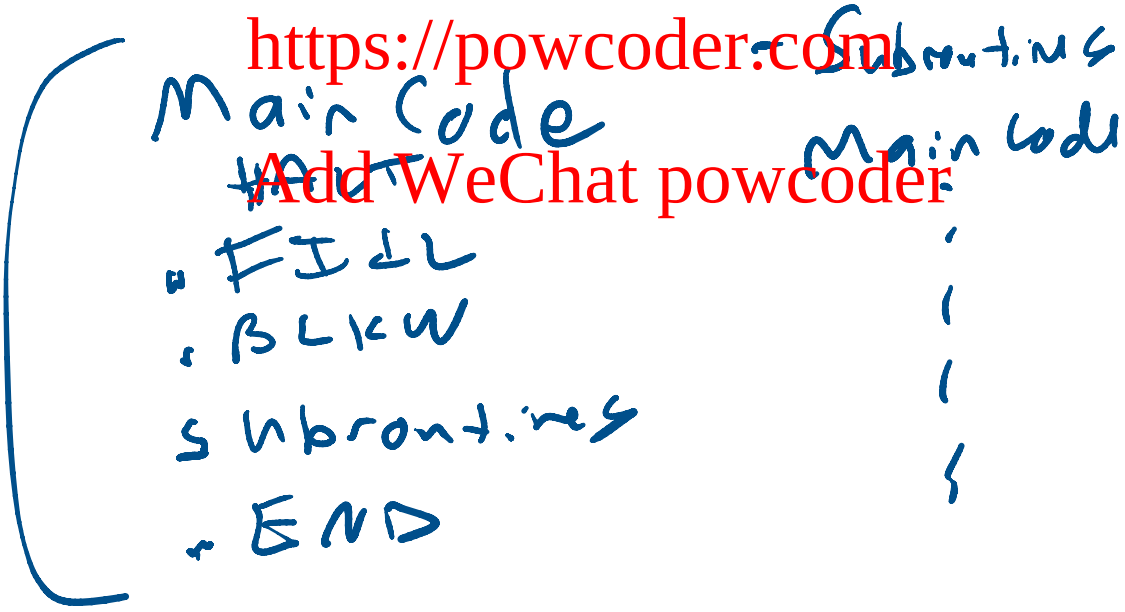
We saved old PC in R7.

- **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 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



Nested Subroutine
JSR GUPDA