# ECE 220 Computer Systems & Programming

## Lecture 2 – Repeated Code: TRAPs and Subroutines

ECE ILLINOIS

ILLINOIS

# **Last Class Example** (memory Mapped I/O)

```
.ORIG x3000

KPOLL   LDI R0, KBSR ; Test For Character Input
        BRzp KPOLL
        LDI R0, KBDR
DPOLL   LDI R1, DSR  ; Test Display Regster is ready
        BRzp DPOLL
        STI R0, DDR
HALT

KBSR .FILL xFE00     ; Address of KBSR
KBDR .FILL xFE02     ; Address of KBDR
DSR  .FILL xFE04     ; Address of DSR
DDR  .FILL xFE06     ; Address of DDR
.END
```

**Drawbacks**

- **Requires knowledge of the hardware**
- **One could mess up hardware registers**

# Solution: **TRAP** Service Routine

- It is desirable to provide *service routines* or *system calls* (part of operating system) to safely and conveniently perform low-level, privileged operations
  - User program invokes system call
  - Operating system code performs operation
  - Returns control to user program

# TRAP Vector Table for LC3

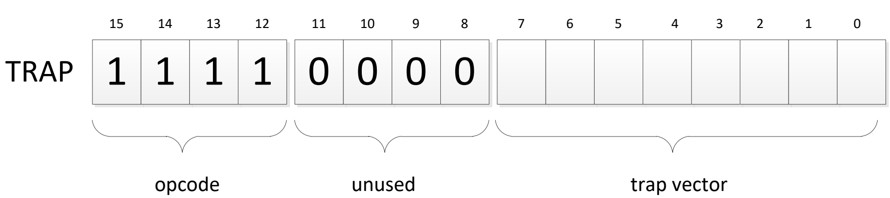
| vector | address | symbol | routine |
|---|---|---|---|
| … | | | |
| x20 | x…. | GETC | read a single character (no echo) |
| x21 | x…. | OUT | output a character to the monitor |
| x22 | x…. | PUTS | write a string to the console |
| x23 | x…. | IN | print prompt to console, read and echo character from keyboard |
| X24 | x…. | PUTSP | write a string to the console; two chars per memory location |
| x25 | x…. | HALT | halt the program |
| … | | | |

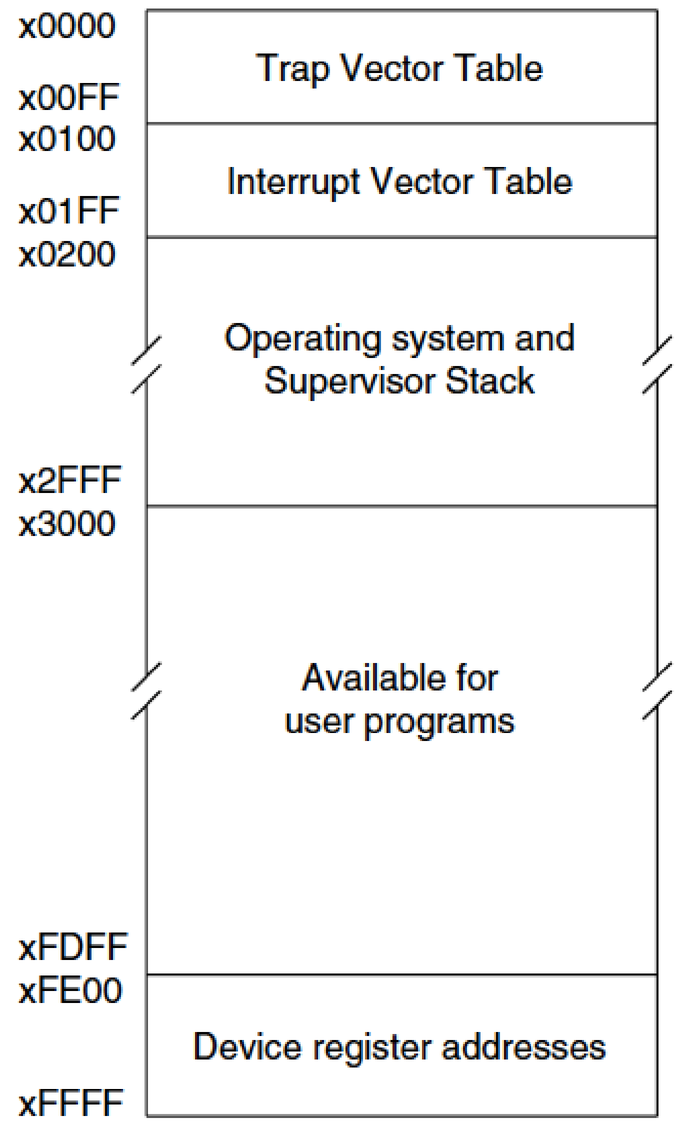
Look-up table decouples names of subroutines (GETC) from the location of its implementation in memory

# How to make this idea work?

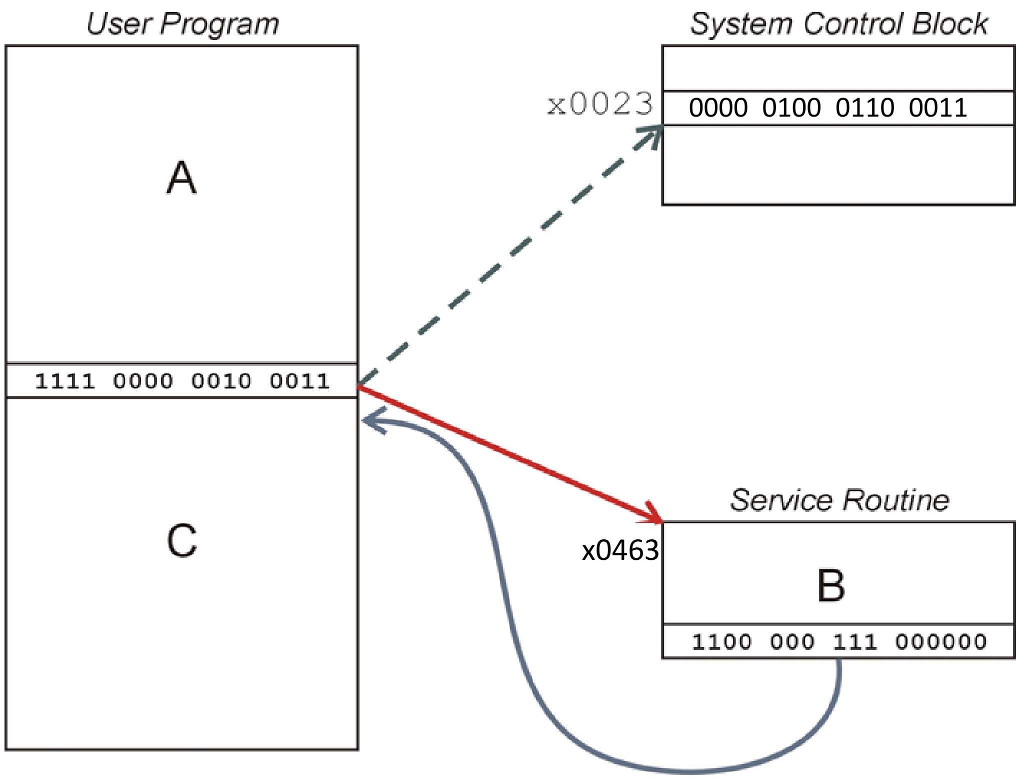
User program **invokes TRAP** subroutine; OS code performs operation; **Returns** control to user program

- The actual code of the service routine is referred indirectly
- Mechanism for invocation
  - TRAP Instruction, e.g., TRAP x23
  - TRAP vector (8 bits)
  - How to find address service routine?

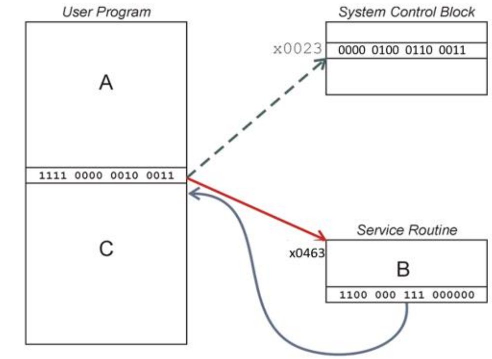


# TRAP Mechanism



# TRAP Mechanism




- PC is loaded with the address of the first instruction of the corresponding service routine
  - MAR←ZEXT(trapvector)
  - MDR←MEM[MAR]
  - R7←PC (note that R7 is loaded with the current content of the PC to provide a way back to the user program)
  - PC←MDR

- Once the service routine is done, control is passed back to the user program using <u>RET</u> instruction, here it does the same operation as <u>JMP R7</u> instruction

  - PC←R7 (restore old PC to return to the user program)

| | 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 |
| | opcode | | | | | | | R7 | | | | | | | | |

- **must make sure that service routine does not change R7, or we won't know where to return**
- also, must make sure R7 does not have a useful value that will be overwritten in the process of calling a TRAP

# LC3 Demo

# TRAP Example (Needs special attention)

```
.ORIG x3000

AND R0, R0, #0
ADD R0, R0, #5     ;init R0 and set it to 5
LD R7, COUNT       ;Initialize to 10

IN                 ;same as 'TRAP x23'

ADD R0, R0, #1     ;increment R0
ADD R7, R7, #-1    ;decrement COUNT

HALT
.END
COUNT   .FILL #10
```

- **Question: What could go wrong?**
- **What are the values in R0 and R7 before and after IN statement?**

# Remedy: Save & Restore Registers

**We must save the value of a register** if its value will be destroyed by a subsequent action (e.g. service routine) and we will need to use the value after that action.

**Two Conventions for Saving & Restoring Registers:**

**1. Caller-saved** (caller knows what it needs later, but may not know what gets altered by callee routine)

-

-

**2. Callee-saved** (callee knows what it alters, but does not know what will be needed by calling routine)

-

-

# Service Routine Features

**Three main features of Service routines (TRAP):**

- Abstract away the system-specific details from the user program
- **Write frequently-used code just once**
- Protect system recourses from malicious/inept programmers

## Subroutines:

**User (non-system) defined routines, i.e. subroutines perform the same functions as service routine but without accessing privileged area of memory.**

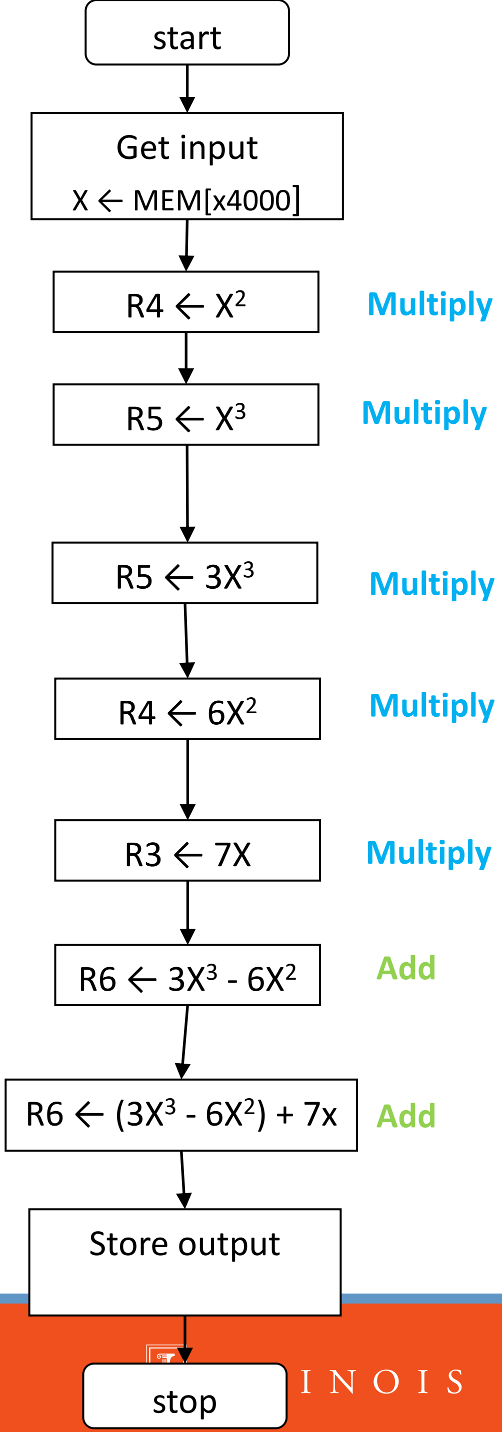
**When we use subroutines?**

# Observation

Example problem: Compute $y=3x^3-6x^2+7x$ for any input $x > 0$

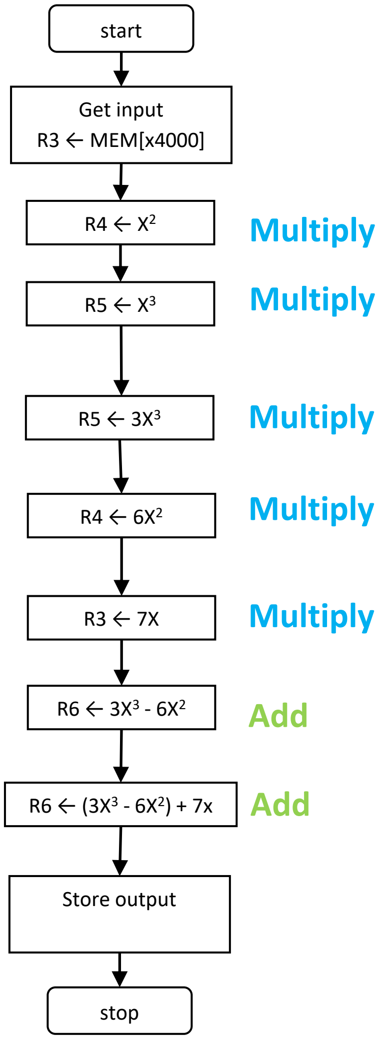
Programs have lots of repetitive code fragments

```
; multiply R0 ←R1 * R2
MULT     AND R0, R0, #0      ; R0 = 0
LOOP     ADD R0, R0, R2      ; R0 = R0 + R2
         ADD R1, R1, #-1     ; decrease counter
         BRp LOOP
```



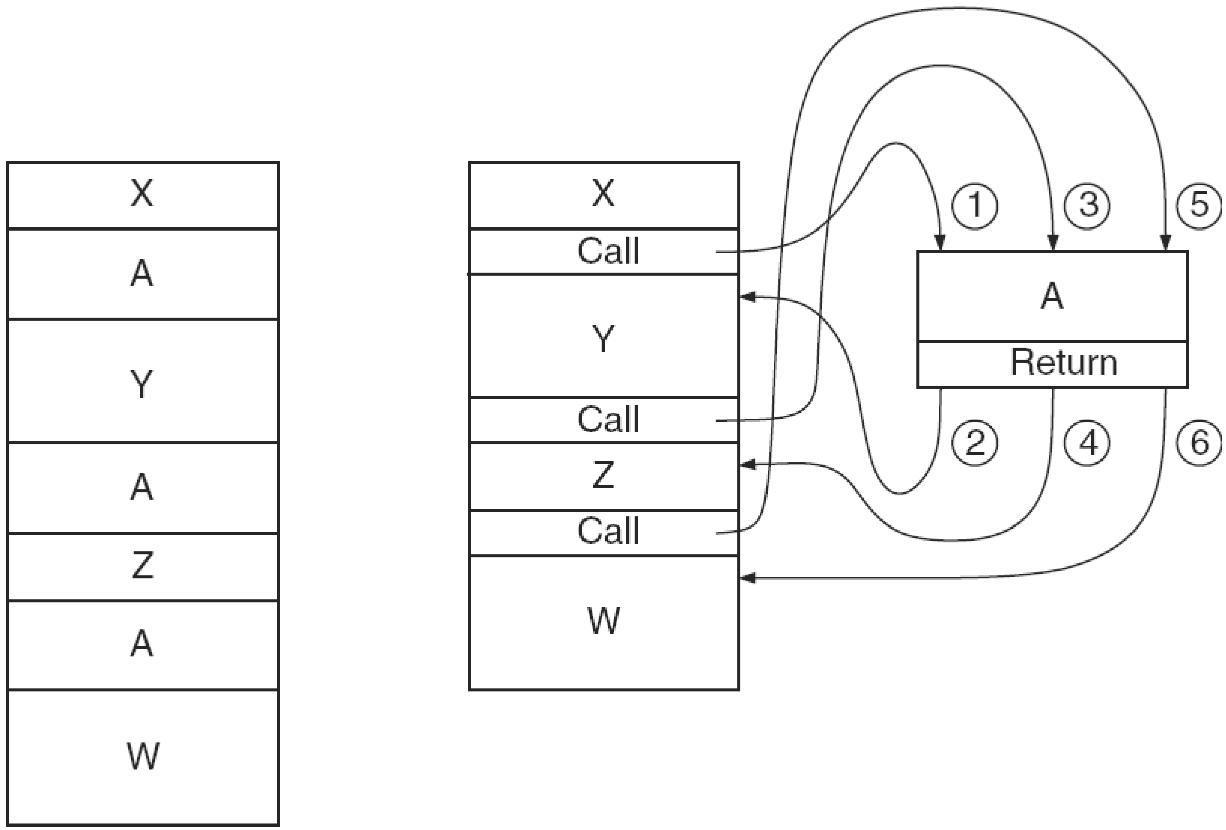
# Implementation Option

Issues ?




```
;; LC-3 Assembly Program
.ORIG x3000
LDI R3, Xaddr;  R3 ← x
ADD R1, R3, #0;
; Multiply R4 ← R1 * R3  (x²)
...
...
; Multiply R5 ← R4 * R3  (x³)
...
; Multiply R5 ← R5 * 3  (3x³)
...
; Multiply R4 ← 6 * R4
```

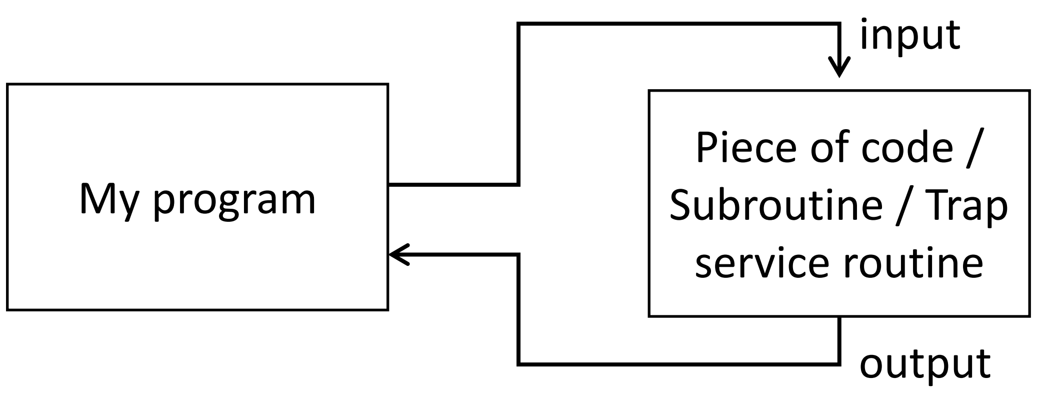
# Idea



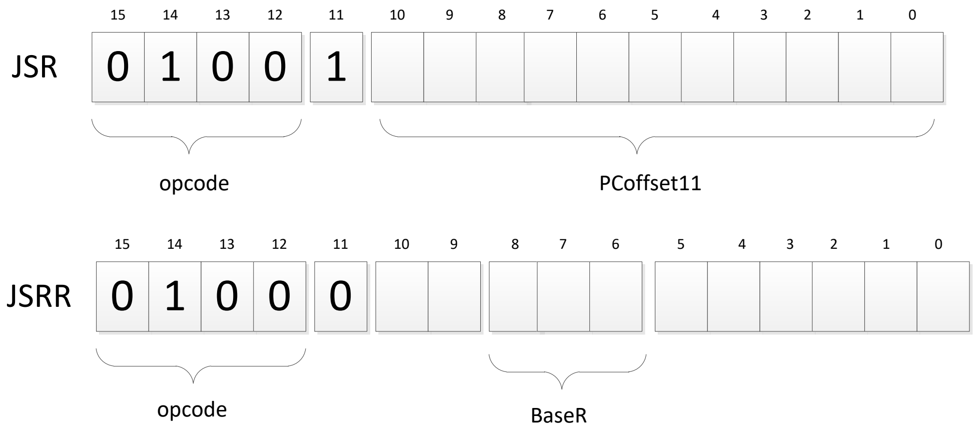

(a) Without subroutines (b) With subroutines

# Idea

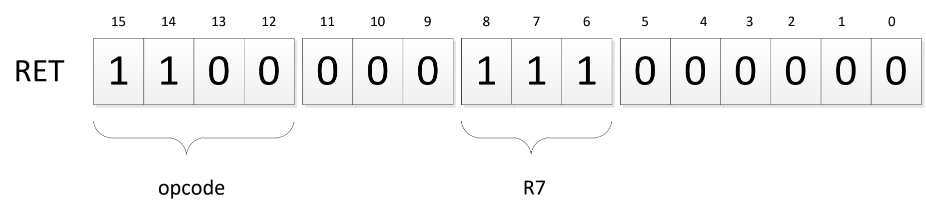
- User **invokes or calls** subroutine
- Subroutine code performs operation / task
- **Returns** control to user program with no other unexpected changes



# JSR and JSRR




R7←PC
If (IR[11] == 0) PC←BaseR
Else PC←PC+SEXT(IR[10:0])




RET ≡ JMP R7
PC ← R7

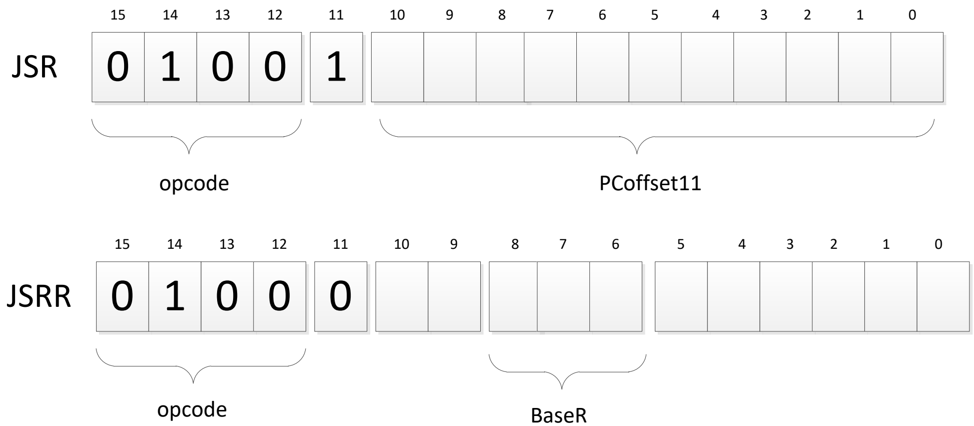
# JSR Example:

```
.ORIG x3000
; perform C=A-B

LD R1, A
LD R2, B
JSR SUB
HALT

;Subroutine: SUB
;input arguments: R1 and R2
;Output: R0 = R1-R2

SUB
    NOT R2, R2
    ADD R2, R2, #1
    ADD R0, R1, R2
    RET


A .FILL #4
B .FILL #2

.END
```

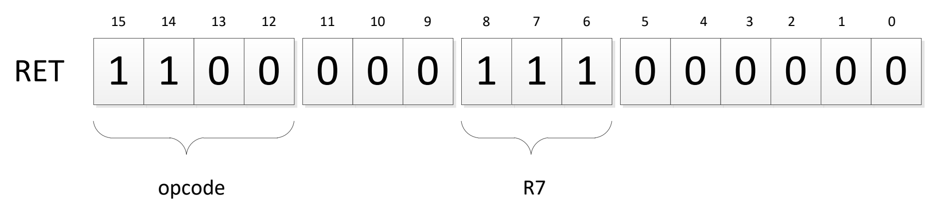
# JSRR Example:

```
.ORIG x3000
; perform C=A-B

LD R1, A
LD R2, B
LEA R4, SUB
JSRR R4
HALT

A .FILL #4
B .FILL #2

;Subroutine: SUB
;input arguments: R1 and R2
;Output: R0 = R1-R2

SUB
   NOT R2, R2
   ADD R2, R2, #1
   ADD R0, R1, R2
   RET

.END
```

# JSR and JSRR – When do we use JSRR?




R7←PC
If (IR[11] == 0) PC←BaseR
Else PC←PC+SEXT(IR[10:0])




RET ≡ JMP R7
PC ← R7

# Subroutine is in a separate file

```
.ORIG x4000
; Subroutine: SUB
   NOT R2, R2
   ADD R2, R2, #1
   ADD R0, R1, R2
   RET
   .END
```

```
.ORIG x3000
; perform C=A-B
;Call Subroutine at x4000
;input arguments: R1 and R2
;Output: R0 = R1-R2

LD R1, A
LD R2, B
LD R4, SUB
JSRR R4
HALT

A .FILL #4
B .FILL #2
SUB .FILL x4000

.END
```

# To use a subroutine,

- A programmer must know
  1. its address (or at least a label)
  2. its function
  3. its arguments (where to pass data in, if any)

     Example:

     - In OUT service routine, R0 is the character to be printed.
     - In PUTS service routine, R0 is the address of string to be printed.
  4. its return value (where to get computed data, if any)
     - In GETC service routine, character read from the keyboard is returned in R0.

# NESTED SUB ROUTINE:

## Check whether the result of C=A-B, is ODD or EVEN?

**Anything wrong??**

```
.ORIG x3000
; perform C=A-B
;Check the result ODD or EVEN

LD R1, A
LD R2, B
JSR SUB
HALT

;Subroutine: SUB
;input arguments: R1 and R2
;Output: R0 = R1-R2

SUB
   NOT R2, R2
   ADD R2, R2, #1
   ADD R0, R1, R2
   ADD R3, R0, #0
   JSR ODD_EVEN
   RET

; Subroutine: ODD_EVEN
;input arguments: R3
;output R4=1; if ODD
;output R4=0; if EVEN

ODD_EVEN
        AND R4, R4, #0
        ADD R4, R4, #1
        AND R4, R3, R4
        RET

A .FILL #4
B .FILL #2

.END
```

# Corrected Code:

**Save R7 before calling ODD_EVEN**

**and**

**Restore R7 after return from ODD_EVEN**

**Nested subroutine –> Save R7**

```
.ORIG x3000
; perform C=A-B
;Check the result ODD or EVEN

LD R1, A
LD R2, B
JSR SUB
HALT

;Subroutine: SUB
;input arguments: R1 and R2
;Output: R0 = R1-R2

SUB
    NOT R2, R2
    ADD R2, R2, #1
    ADD R0, R1, R2
    ST R7, SAVER7
    ADD R3,R0,#0
    JSR ODD_EVEN
    LD R7,SAVER7
    RET

; Subroutine: ODD_EVEN
;input arguments: R3
;output R4=1; if ODD
;output R4=0; if EVEN

ODD_EVEN
        AND R4, R4, #0
        ADD R4, R4, #1
        AND R4, R3, R4
        RET

A .FILL #4
B .FILL #3
SAVER7 .BLKW #1
.END
```

# Saving/Restoring Registers in Subroutines

1. Generally, use callee-save strategy, except for return values
2. Save anything that the subroutine will alter internally
3. It's good practice to restore incoming arguments to their original values.