# ECE 220 Computer Systems & Programming

## Lecture 2 – Repeated Code: TRAPs and Subroutines

ECE ILLINOIS

ILLINOIS

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

```
1  .ORIG x3000
2
3  KPOLL   LDI R0, KBSR ; Test For Character Input
4          BRzp KPOLL
5          LDI R0, KBDR
6  DPOLL   LDI R1, DSR  ; Test Display Regster is ready
7          BRzp DPOLL
8          STI R0, DDR
9  HALT
10
11 KBSR .FILL xFE00     ; Address of KBSR
12 KBDR .FILL xFE02     ; Address of KBDR
13 DSR  .FILL xFE04     ; Address of DSR
14 DDR  .FILL xFE06     ; Address of DDR
15 .END
```

**Drawbacks**

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

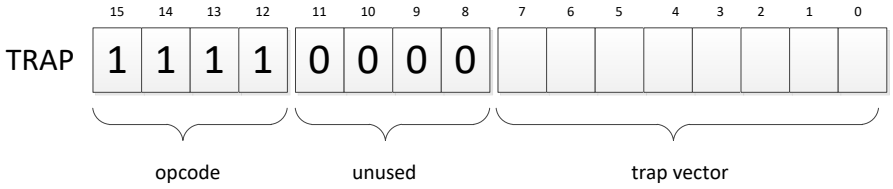
# Solution: **TRAP** Service Routine

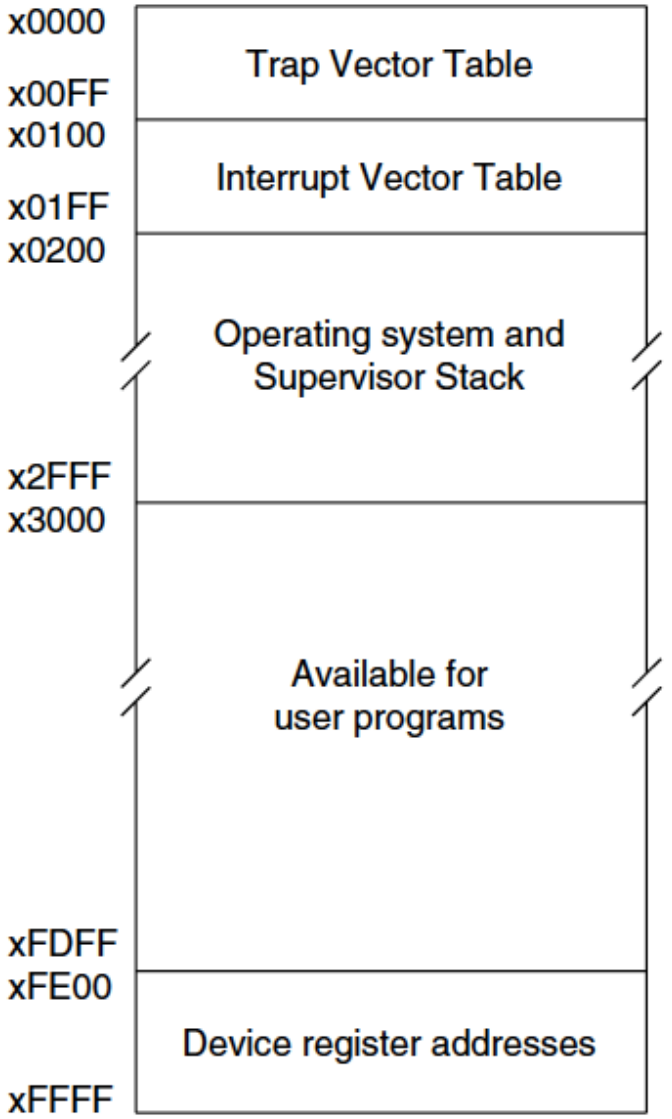
- It is desirable to provide *service routines* or *system calls* (part of operating system) to safely and conveniently perform low-level, privileged operations
  - User program invokes system call
  - Operating system code performs operation
  - Returns control to user program

# How to make this idea work?

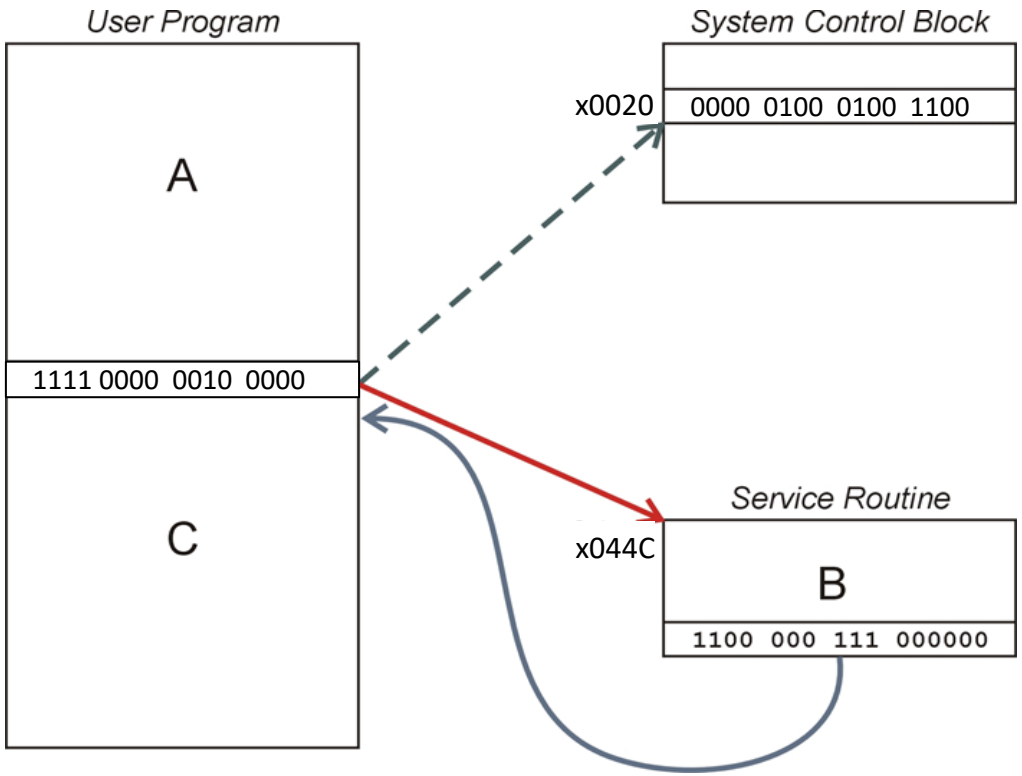
User program **invokes TRAP** subroutine; OS code performs operation; **Returns** control to user program

- The actual code of the service routine is referred indirectly in Trap Vector Table
- Mechanism for invocation
  - TRAP Instruction, e.g., TRAP x20
  - TRAP vector (8 bits)
  - How to find address service routine?



# TRAP Mechanism (TRAP x20)



# TRAP Mechanism (TRAP x20)

trap_test.asm

```
.ORIG X3000
TRAP x20
HALT
.END
```

trap_test.obj

```
Loaded "trap_test.obj" and set PC to x3000
(lc3sim) list x3000
                  x3000 xF020 GETC
                  x3001 xF025 HALT
```

# TRAP Mechanism (TRAP x20)

```
        x0020 x044C BRZ   x006D
        x0021 x0450 BRZ   x0072
        x0022 x0456 BRZ   x0079
        x0023 x0463 BRZ   x0087
```

trap_test.asm

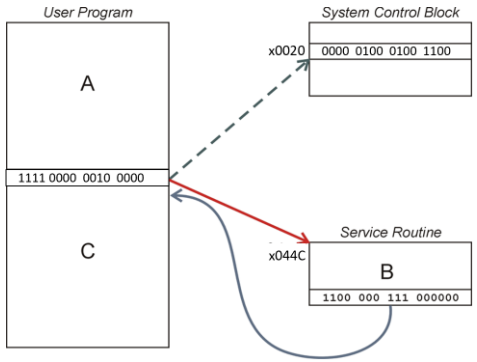
```
.ORIG X3000
TRAP x20
HALT
.END
```

```
TRAP_GETC x044C xA1F1 LDI   R0,OS_KBSR
          x044D x07FE BRZP  TRAP_GETC
          x044E xA1F0 LDI   R0,OS_KBDR
          x044F xC1C0 RET
```

```
  OS_KBSR x043E xFE00 .FILL xFE00
  OS_KBDR x043F xFE02 .FILL xFE02
   OS_DSR x0440 xFE04 .FILL xFE04
   OS_DDR x0441 xFE06 .FILL xFE06
```

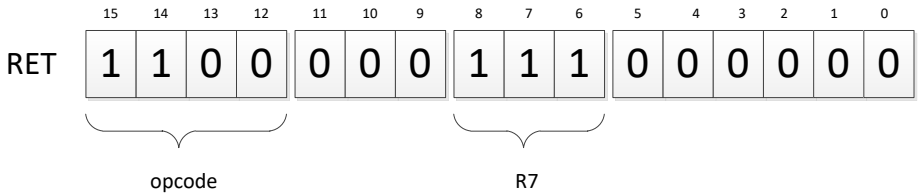
```
Loaded "trap_test.obj" and set PC to x3000
(lc3sim) s
PC=x044C IR=xF020 PSR=x0400 (ZERO)
R0=x0000 R1=x7FFF R2=x0000 R3=x0000 R4=x0000 R5=x0000 R6=x0000 R7=x3001
         TRAP_GETC x044C xA1F1 LDI   R0,OS_KBSR
```

# 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 RET instruction, here it does the same operation as JMP R7 instruction
  - PC←R7 (restore old PC to return to the user program)




  - **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

# TRAP x20 Mechanism – LC3 DEMO

trap_test.asm

```
.ORIG X3000
TRAP x20
HALT
.END
```

trap_test.obj

```
Loaded "trap_test.obj" and set PC to x3000
(lc3sim) list x3000
                    x3000 xF020 GETC
                    x3001 xF025 HALT
```

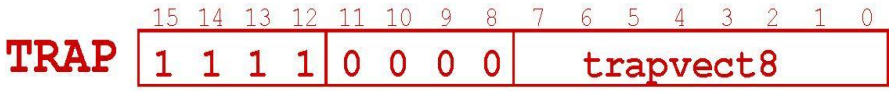
```
(lc3sim) break set x3001
Set breakpoint at x3001.
(lc3sim) s
PC=x044C IR=xF020 PSR=x0400 (ZERO)
R0=x0000 R1=x7FFF R2=x0000 R3=x0000 R4=x0000 R5=x0000 R6=x0000 R7=x3001
         TRAP_GETC x044C xA1F1 LDI   R0,OS_KBSR
(lc3sim) c
The LC-3 hit a breakpoint...
PC=x3001 IR=xC1C0 PSR=x0200 (POSITIVE)
R0=x0041 R1=x7FFF R2=x0000 R3=x0000 R4=x0000 R5=x0000 R6=x0000 R7=x3001
B                   x3001 xF025 HALT
(lc3sim)
```

# LC-3 TRAP Mechanism

- **TRAP instruction**
  - used by user program to transfer control to OS
  - 8-bit Trap vector names one of 256 service routines
- **Set of service routines**
  - part of OS
  - start at arbitrary addresses (within OS)
  - LC-3 uses only six TRAP service routines (x20 – x25)
- **Table of starting addresses**
  - stored at x0000 through x00FF in memory
  - called Trap Vector Table (or System Control Block)
- **Linkage**
  - return control back to user program

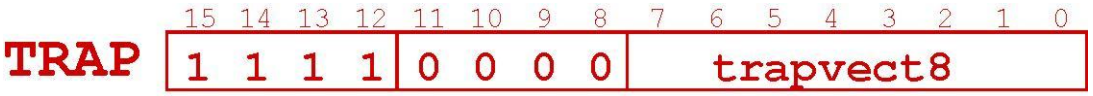
```
        OS_R2 x0449 x0000 NOP
        OS_R3 x044A x0000 NOP
        OS_R7 x044B x0490 BRZ    x04DC
    TRAP_GETC x044C xA1F1 LDI    R0,OS_KBSR
              x044D x07FE BRZP   TRAP_GETC
              x044E xA1F0 LDI    R0,OS_KBDR
              x044F xC1C0 RET
     TRAP_OUT x0450 x33F4 ST     R1,TOUT_R1
TRAP_OUT_WAIT x0451 xA3EE LDI    R1,OS_DSR
              x0452 x07FE BRZP   TRAP_OUT_WAIT
              x0453 xB1ED STI    R0,OS_DDR
              x0454 x23F0 LD     R1,TOUT_R1
              x0455 xC1C0 RET
```

```
x0020 x044C BRZ   x006D
x0021 x0450 BRZ   x0072
x0022 x0456 BRZ   x0079
x0023 x0463 BRZ   x0087
```




RET (a.k.a JMP R7)

# TRAP Instruction




- Trap vector (8-bit index)
  - Table of service routine addresses (x0000-x00FF)
  - Zero-extended into 16-bit memory address
  - **R0** is used to store the return value or to pass the argument.

| *vector* | *symbol* | *routine* |
|---|---|---|
| **x20** | **GETC** | read a single character into R0 (no echo) |
| **x21** | **OUT** | output a character in R0 to the monitor |
| **x22** | **PUTS** | write a string to the console (addr in R0) |
| **x23** | **IN** | print prompt to console, read and echo character from keyboard (R0) |
| **x24** | **PUTSP** | write a string to the console (2 characters per memory location) (addr in R0) |
| **x25** | **HALT** | halt the program |

# TRAP example

```
        .ORIG   x3000
        LEA     R3,Binary
        LD      R6,ASCII
        LD      R7,COUNT
AGAIN
        TRAP    x23
        ADD     R0,R0,R6
        STR     R0,R3,#0
        ADD     R3,R3,#1
        ADD     R7,R7,#-1
        BRp     AGAIN
        HALT
ASCII   .FILL   #-48
COUNT   .FILL   #3
Binary  .BLKW   #3
        .END
```

**Goal**

1. 3 decimal inputs (single digit) from keyboard
2. Convert ASCII into binary
3. Store the 3 binary values in memory

-What problem we have?

# TRAP example

```
      .ORIG   x3000
      LEA     R3,Binary
      LD      R6,ASCII
      LD      R7,COUNT
AGAIN
                ST R7, SAVER7
      TRAP    x23
      ADD     R0,R0,R6
      STR     R0,R3,#0
      ADD     R3,R3,#1
                LD R7, SAVER7
      ADD     R7,R7,#-1
      BRp     AGAIN
      HALT
ASCII   .FILL   #-48
COUNT   .FILL   #3
Binary  .BLKW   #3
SAVER7      .BLKW  #1
      .END
```

**Goal**

1. 3 decimal inputs from keyboard
2. Convert ASCII into binary
3. Store the 3 binary values in memory

-What problem we have?

# 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)

– last TRAP Example (i.e. get 3 decimal inputs from keyboard, convert them into binary, and store them)

- Save R7 before calling TRAP x23 and retrieve R7 after returning to the caller.

**2. Callee-saved** (callee knows what it alters, but does not know what will be needed by calling routine) – Example TRAP x21

```
        x0020 x044C BRZ   x006D
        x0021 x0450 BRZ   x0072
        x0022 x0456 BRZ   x0079
        x0023 x0463 BRZ   x0087
```

```
     TRAP_OUT x0450 x33F4 ST    R1,TOUT_R1
TRAP_OUT_WAIT x0451 xA3EE LDI   R1,OS_DSR
              x0452 x07FE BRZP  TRAP_OUT_WAIT
              x0453 xB1ED STI   R0,OS_DDR
              x0454 x23F0 LD    R1,TOUT_R1
              x0455 xC1C0 RET
```

# 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 resources 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?**

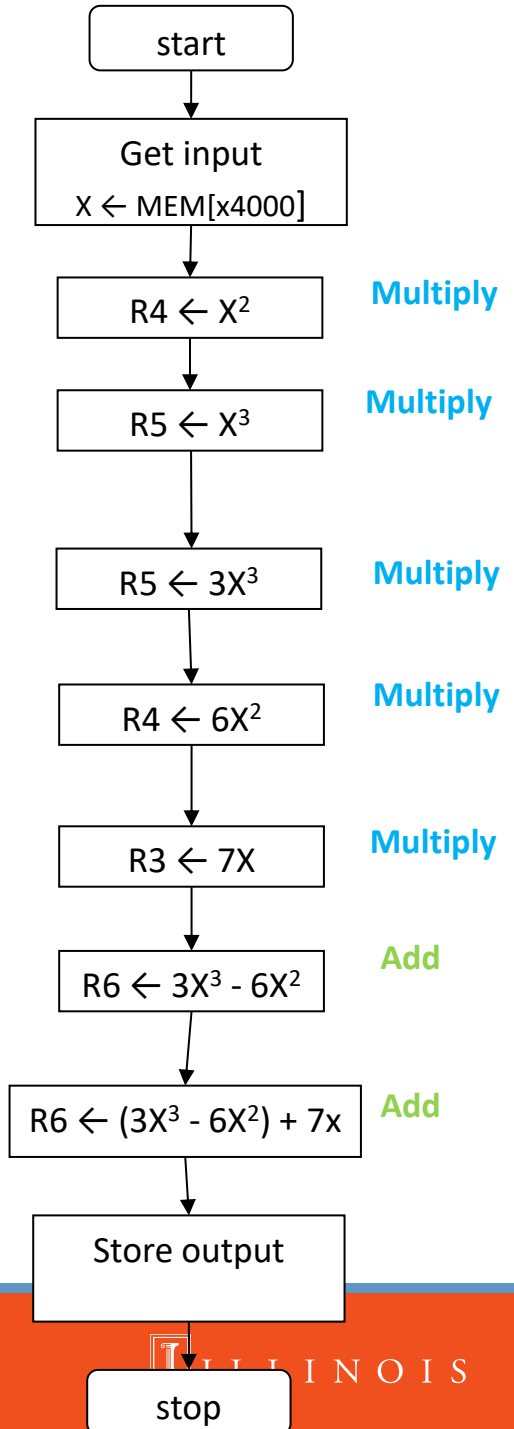
# Observation

Example problem:

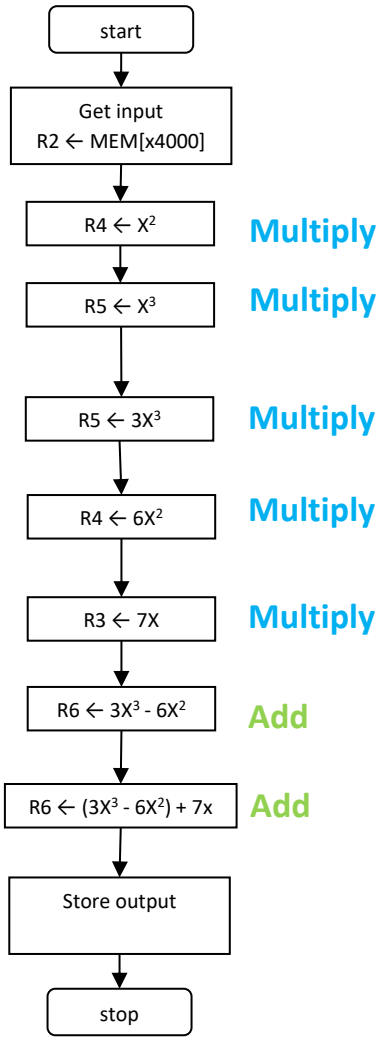
Compute $y=3x^3-6x^2+7x$ for any input $x > 0$

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
```



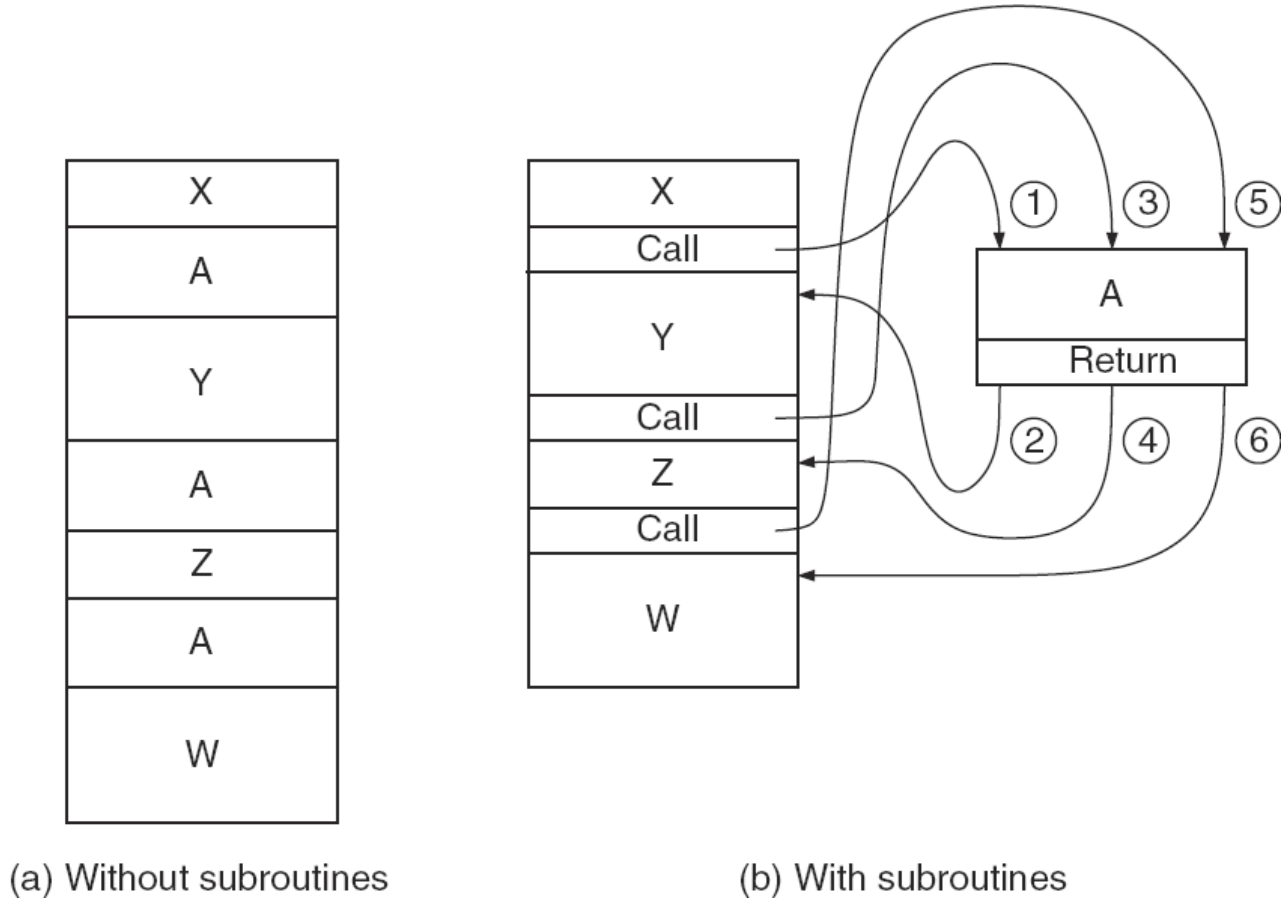
# Implementation Option




```
;; LC-3 Assembly Program
.ORIG x3000
LDI R2, Xaddr;  R2 ← x
ADD R1, R2, #0;
; Multiply R4 ← R1 * R2   (x^2)
...
...
; Multiply R5 ← R4 * R2   (x^3)
...
; Multiply R5 ← R5 * 3   (3x^3)
...
; Multiply R4 ← 6 * R4
```

Issues ?

# Idea

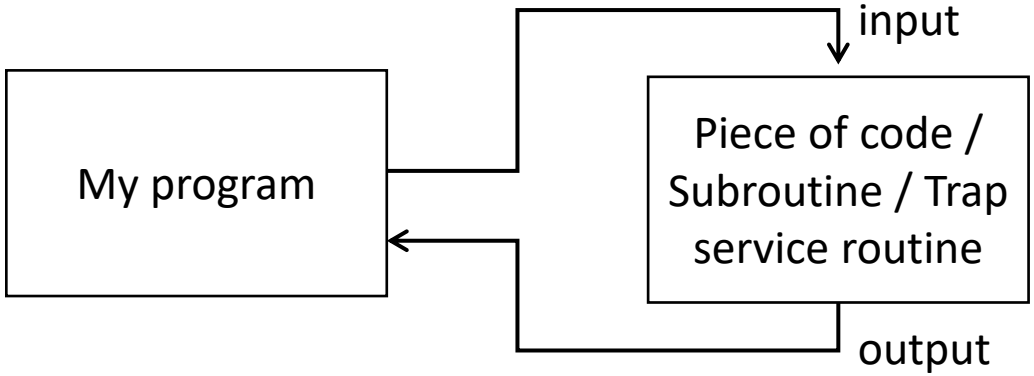



(a) Without subroutines

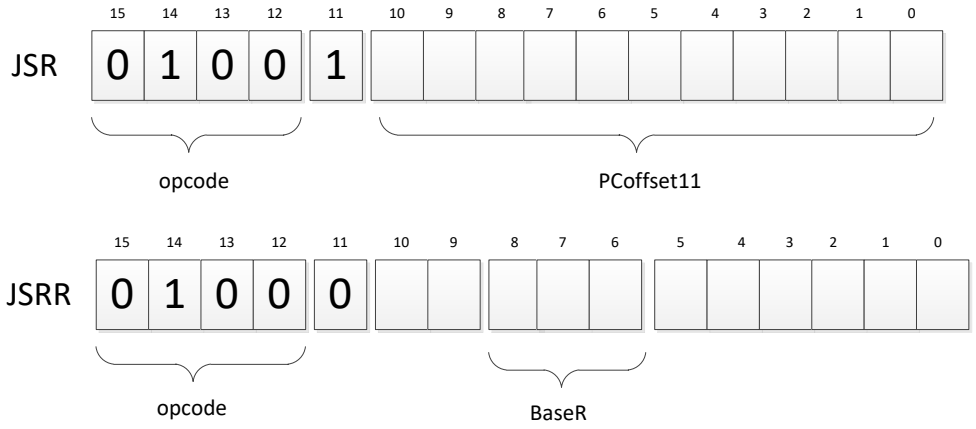
(b) With subroutines

# Subroutine

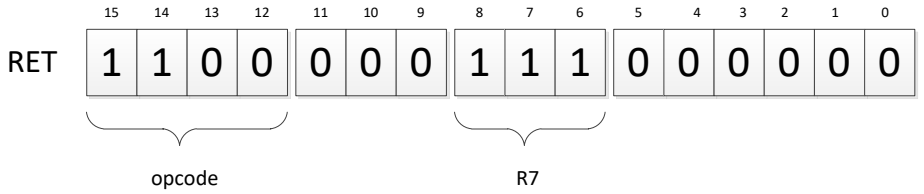
- User **invokes or calls** subroutine
- Subroutine code performs operation / task
- **Returns** control to user program with no other unexpected changes



# JSR and JSRR




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




RET ≡ JMP R7
PC ← R7

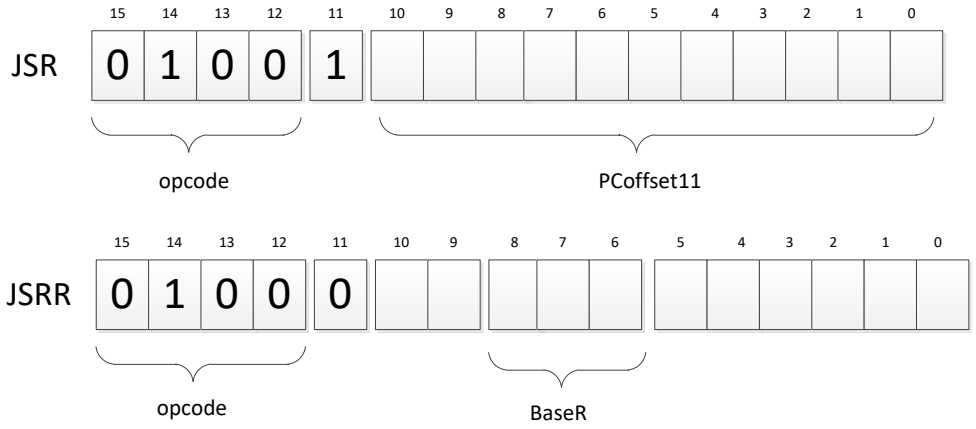
# JSR Example:

```
.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
```

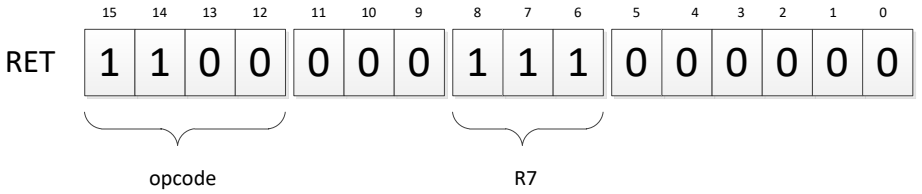
# JSRR Example:

```
.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.