ECE 220: Computer Systems & Programming

Lecture 2: Repeated code- TRAPs and Subroutines

Thomas Moon

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Previous lecture

- I/O basics, I/O types
- Input from keyboard/Output to monitor
- Memory-mapped I/O, Handshaking (ready-bit), Polling

Today's lecture

- TRAPs: `GETC, IN, OUT, PUTS, PUTSP, HALT`
- Subroutines: `JSR, JSRR`
- Demystify `R7`
    - You may use any registers, but we recommend that you avoid using `R7`. ← Why?

From Lec 1

Input/Output routines by **USER**

```
POLL        LDI     R1, KBSR_ADDR
            BRzp    POLL
            LDI     R0, KBDR_ADDR
POLL2       LDI     R1, DSR_ADDR
            BRzp    POLL2
            STI     R0, DDR_ADDR

KBSR_ADDR          .FILL xFE00
KBDR_ADDR          .FILL xFE02
DSR_ADDR           .FILL xFE04
DDR_ADDR           .FILL xFE06
```

by **TRAP**

```
GETC
OUT
```

or

```
TRAP x20
TRAP x21
```

User Program Accessing I/O

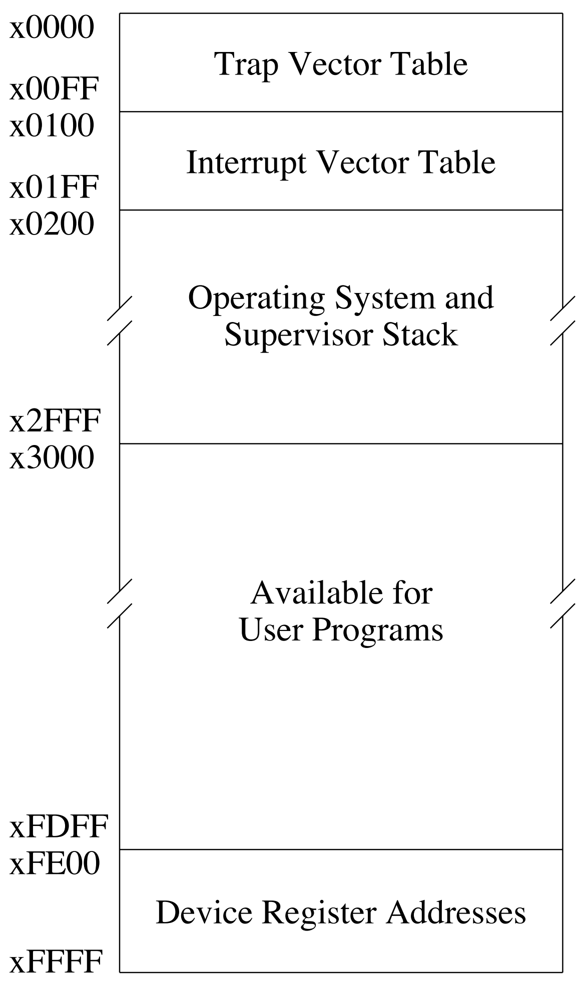
- Problem
  - It requires too many specific details for programmer (device regs, memory-mapped, handshaking protocols, etc)
  - Security issue: I/O resources shared with multiple programs

- Solution: make this part of OS

  Service routines or **system calls**

  1. User program invokes system call
  2. OS code performs operation
  3. Returns control to user program

- In LC-3, this is done through the TRAP mechanism.



TRAP Instruction

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

- 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

PUTS vs PUTSP

```
.ORIG x3000
LEA    R0, LB
PUTS
HALT
LB    .STRINGZ "abcd"
.END
```

```
   x3000 xE002 LEA    R0,LB
   x3001 xF022 PUTS
   x3002 xF025 HALT
LB x3003 x0061 .FILL 'a'
   x3004 x0062 .FILL 'b'
   x3005 x0063 .FILL 'c'
   x3006 x0064 .FILL 'd'
   x3007 x0000 NOP
   x3008 x0000 NOP
```

```
.ORIG x3000
LEA R0, LB
PUTSP
HALT
LB   .FILL x6261
.FILL x6463
.FILL x0
.END
```

```
   x3000 xE002 LEA    R0,LB
   x3001 xF024 PUTSP
   x3002 xF025 HALT
LB x3003 x6261 LDR    R1,R1,#-31
   x3004 x6463 LDR    R2,R1,#-29
   x3005 x0000 NOP
```

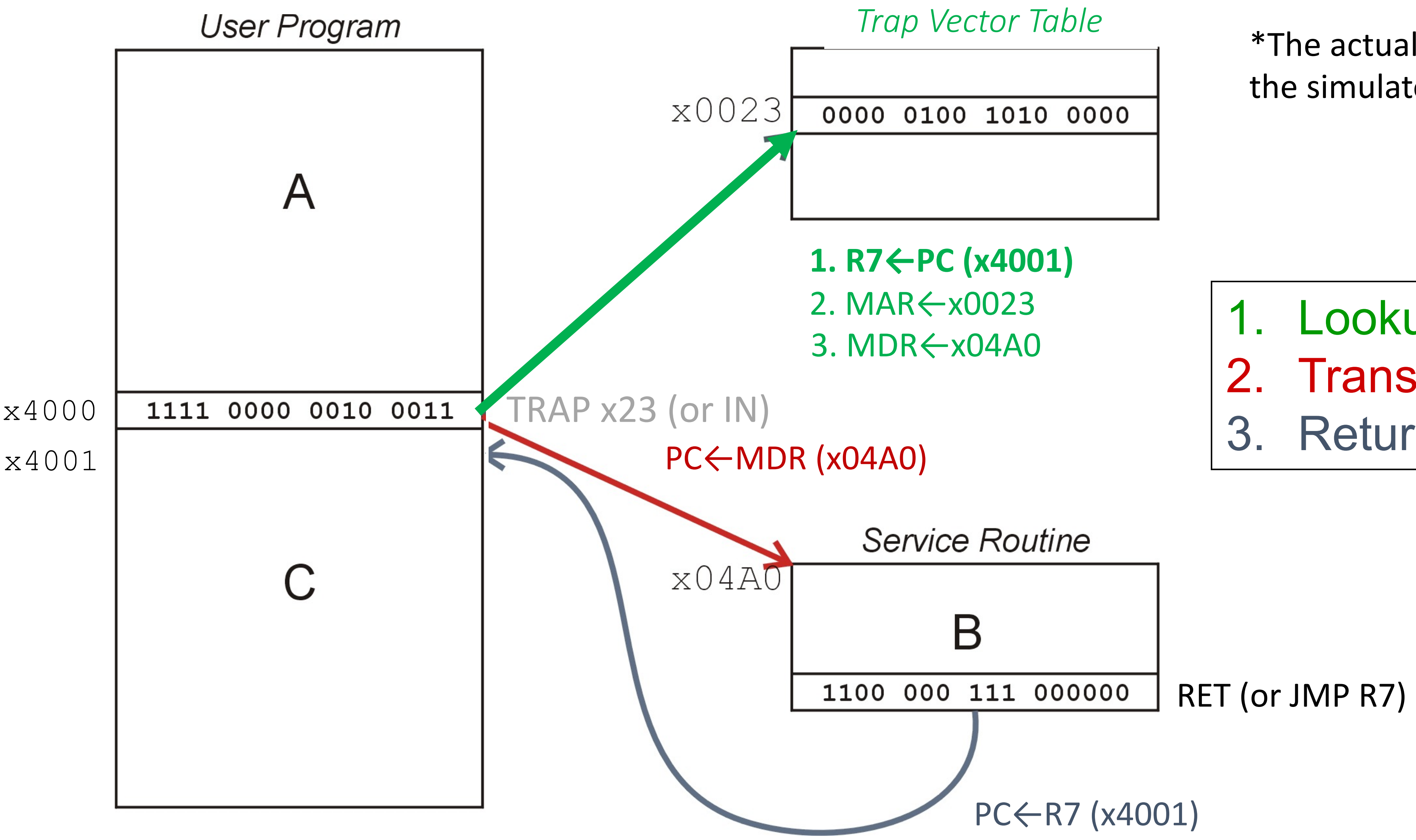
They both prints

`abcd`

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

Q. How many different TRAP routines can be implemented?

TRAP Mechanism Operation



*The actual value of TVT is subjective to the simulator.

1. Lookup starting address.
2. Transfer to service routine.
3. Return (RET = JMP R7).

LC-3 TRAP Mechanism

1. **TRAP instruction**
   - used by user program to transfer control to OS
   - 8-bit Trap vector names one of 256 service routines

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

2. **Table of starting addresses**
   - stored at x0000 through x00FF in memory
   - called Trap Vector Table (or System Control Block)

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

3. **Set of service routines**
   - part of OS
   - start at arbitrary addresses (within OS)
   - LC-3 is designed to have upto 256 routines

```
          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
```

4. **Linkage**
   - return control back to user program

RET (a.k.a JMP R7)

TRAP example

```
        .ORIG   x3000
        LD      R0, CAP_A
        LD      R1, CNT
LOOP
        OUT
        ADD     R1, R1, #-1
        BRp     LOOP
        HALT

CNT     .FILL   #3
CAP_A   .FILL   x41
        .END
```

Describe the program.

TRAP example

```
        .ORIG   x3000
        LD      R0, CAP_A
        LD      R7, CNT
LOOP
        OUT
        ADD     R7, R7, #-1
        BRp     LOOP
        HALT

CNT     .FILL   #3
CAP_A   .FILL   x41
        .END
```

Describe the program.

→ *If we have to use R7, what will be the solution?*

Saving and Restoring Registers

- Called routine – “**callee-save**”
  - Before start, save any registers that will be altered
  - Before return, restore the registers

- Calling routine - “**caller-save**”
  - Save registers destroyed by called routines, if values needed later
    - Save R7 before any TRAP
    - Save R0 before IN or GETC (what about OUT or PUTS?)
  - Or avoid using those registers

TRAP: Callee-save Example

```
     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
```

R1 is callee-saved because it will be changed.

Subroutines

- **Service routines** (TRAP) provides 3 main functions:
  - Shield programmers from system-specific details
  - Write frequently-used code just once
  - Protect system resources from malicious/clumsy programmers

- A **subroutine** is a program fragment that:
  - performs a well-defined task
  - is called by another user program
  - returns control to the calling program when finished
  - lives in user space (not part of OS, not concerned with protecting hardware resources)

JSR/JSRR – Jump to Subroutine

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

	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

- Jumps to a location (like a branch but unconditional) and saves current PC (addr of next instruction) in **R7**

```
TEMP = PC
if (bit[11] == 0)
        PC = baseR;
else
        PC = PC + SEXT(PCoffset11);
R7 = TEMP;
```

- To return form a subroutine, use RET (just like TRAP).

JSR Example

```
        .ORIG   x3000
        LD      R1, VAL1
        LD      R2, VAL2
        LD      R3, VAL3
        JSR     ADD3
        HALT
; ADD3 subroutine: R0 = R1 + R2 + R3
ADD3
        AND     R0, R0, #0
        ADD     R0, R0, R1
        ADD     R0, R0, R2
        ADD     R0, R0, R3
        RET
VAL1    .FILL   #2
VAL2    .FILL   #3
VAL3    .FILL   #4
        .END
```

JSRR Example

```
        .ORIG   x3000
        LD      R1, VAL1
        LD      R2, VAL2
        LD      R3, VAL3
        LEA     R4, ADD3
        JSRR    R4
        HALT
; ADD3 subroutine: R0 = R1 + R2 + R3
ADD3
        AND     R0, R0, #0
        ADD     R0, R0, R1
        ADD     R0, R0, R2
        ADD     R0, R0, R3
        RET
VAL1    .FILL   #2
VAL2    .FILL   #3
VAL3    .FILL   #4
        .END
```

- When do you use JSRR?

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.

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.

4. If Nested subroutine, Caller-Save R7

Example: Subtraction

```
        .ORIG   x3000
        LD      R2,Value1    ;load a value into R2
        LD      R3,Value2    ;load a value into R3
        JSR     SUBTR        ;jump to subroutine
        HALT
;NEG: R6 = −R0
NEG     ST      R0,SaveR0_NEG
        NOT     R0,R0
        ADD     R6,R0,#1
        LD      R0,SaveR0_NEG
        RET
;SUBTR: R1 = R2 − R3
SUBTR   ST      R0, SaveR0_SUB
        ST      R6, SaveR6_SUB
        ADD     R0, R3, #0
        JSR     NEG
        ADD     R1, R2, R6
        LD      R0, SaveR0_SUB
        LD      R6, SaveR6_SUB
        RET
```

-What problem we have?

Example: Subtraction

```
               .ORIG   x3000
               LD      R2,Value1    ;load a value into R2
               LD      R3,Value2    ;load a value into R3
               JSR     SUBTR        ;jump to subroutine
x3003 -------> HALT
        ;NEG: R6 = -R0
        NEG    ST      R0,SaveR0_NEG
               NOT     R0,R0
               ADD     R6,R0,#1
               LD      R0,SaveR0_NEG
               RET
        ;SUBTR: R1 = R2 - R3
        SUBTR  ST      R0, SaveR0_SUB
               ST      R6, SaveR6_SUB
               ADD     R0, R3, #0
               JSR     NEG
x300D -------> ADD     R1, R2, R6
               LD      R0, SaveR0_SUB
               LD      R6, SaveR6_SUB
               RET
```

2. R7 = x300D → R7 is overwritten

1. R7 = x3003

3. R7 = x300D → Never return to x3003

```
                .ORIG   x3000
                LD      R2,Value1   ;load a value into R2
                LD      R3,Value2   ;load a value into R3
                JSR     SUBTR       ;jump to subroutine
x3003 --------> HALT
        ;NEG: R6 = -R0
        NEG     ST      R0,SaveR0_NEG
                NOT     R0,R0
                ADD     R6,R0,#1
                LD      R0,SaveR0_NEG
                RET
        ;SUBTR: R1 = R2 - R3
        SUBTR   ST      R0, SaveR0_SUB
                ST      R6, SaveR6_SUB
                ST      R7, SaveR7_SUB
                ADD     R0, R3, #0
                JSR     NEG
x300D --------> ADD     R1, R2, R6
                LD      R0, SaveR0_SUB
                LD      R6, SaveR6_SUB
                LD      R7, SaveR7_SUB
                RET
```

2. R7 = x300D

1. R7 = x3003

3. R7 = x300D

4. R7 = x3003

→ Return to x3003

Callee-save vs Caller-save???

```
     TRAP_PUTS x0456 x31F0 ST    R0,OS_R0
               x0457 x33F0 ST    R1,OS_R1
               x0458 x3FF2 ST    R7,OS_R7
               x0459 x1220 ADD   R1,R0,#0
TRAP_PUTS_LOOP x045A x6040 LDR   R0,R1,#0
               x045B x0403 BRZ   TRAP_PUTS_DONE
               x045C xF021 OUT
               x045D x1261 ADD   R1,R1,#1
               x045E x0FFB BRNZP TRAP_PUTS_LOOP
TRAP_PUTS_DONE x045F x21E7 LD    R0,OS_R0
               x0460 x23E7 LD    R1,OS_R1
               x0461 x2FE9 LD    R7,OS_R7
               x0462 xC1C0 RET
```

Callee-save R?, R?

Caller-save R?