ECE 220 Computer Systems & Programming

Lecture 3 – Repeated Code: TRAPs and Subroutines

September 3, 2019

- MP1 due Thursday, 9/5, by 10pm
- Schedule mock quiz for extra credit

Outline

- Chapter 9
- Repeated code: TRAPs and Subroutines

- Key concepts
  – Lookup table: for starting address of subroutines/TRAPS (vector table)
  – Preserving register and PC values
- Instructions
  – TRAP
  – RET
  – JSR, JSRR

Last Class Example (memory Mapped I/O)

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

```
.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
```

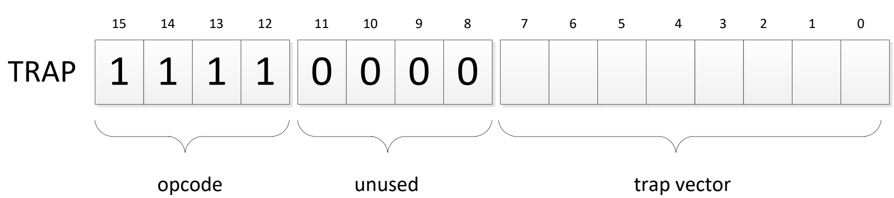
Solution: **TRAP** Service Routine

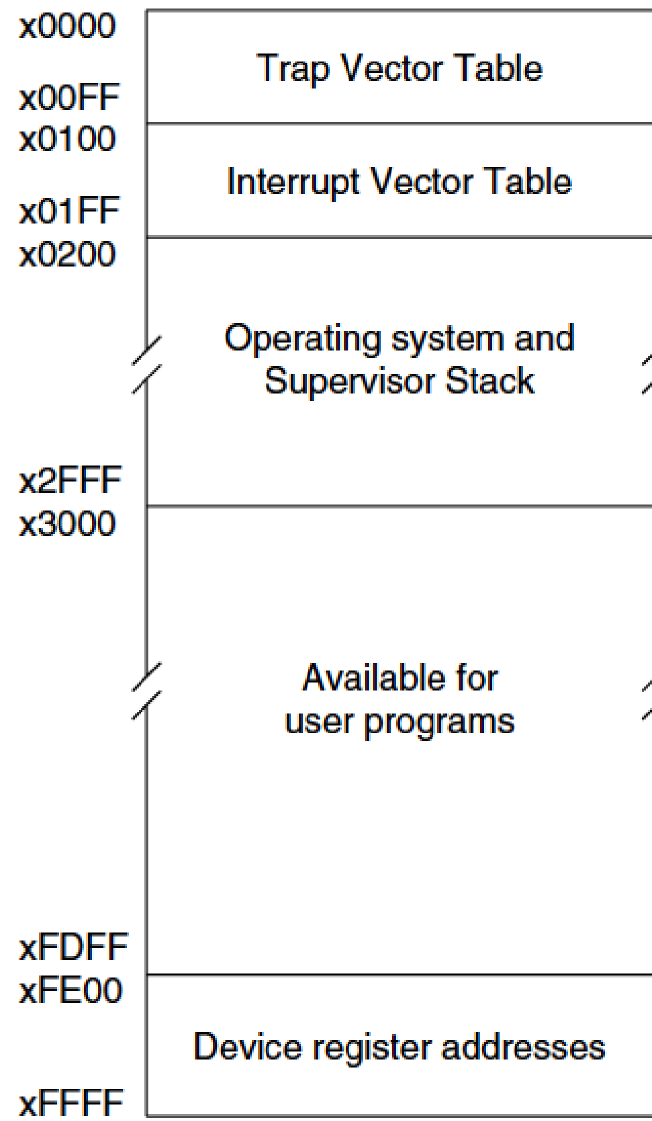
- It is desirable to provide *service routines* or *system calls* (part of operating system) to safely and conveniently perform low-level, privileged operations
  - User program invokes system call
  - Operating system code performs operation
  - Returns control to user program

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
- Mechanism for invocation
  - TRAP Instruction, e.g., TRAP x23
  - TRAP vector (8 bits)
  - How to find address service routine?



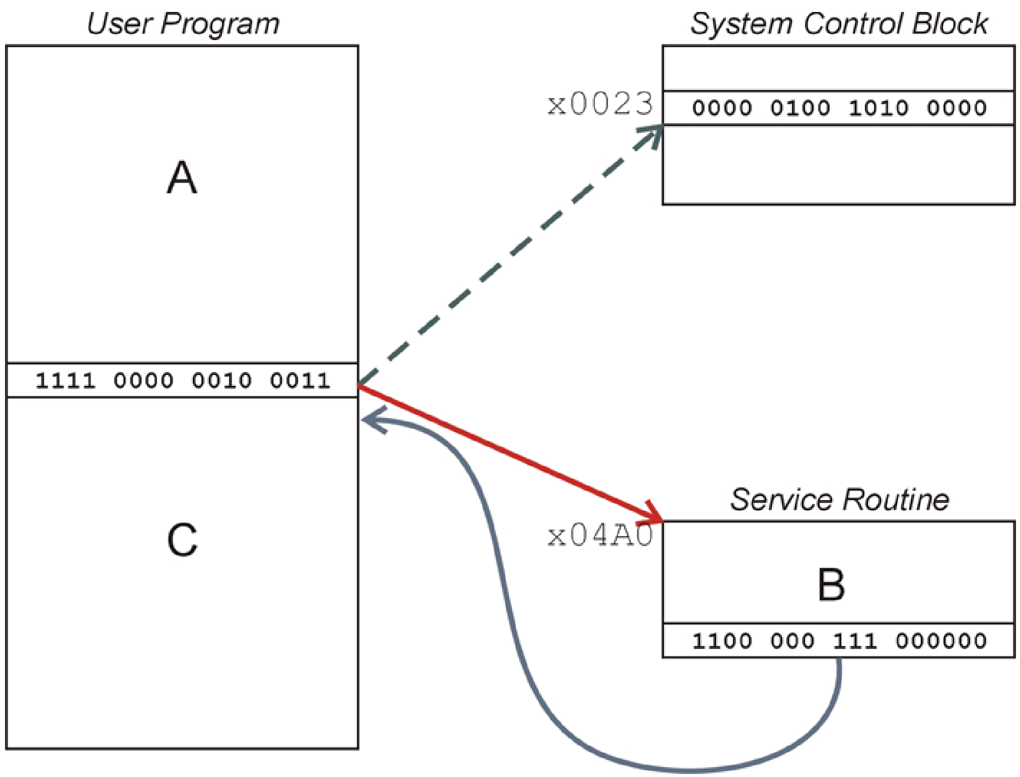


TRAP Vector Table for LC3

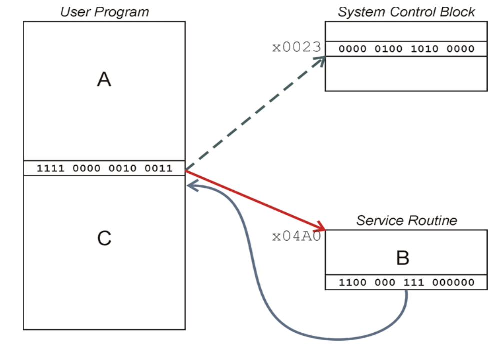
vector	address	symbol	routine
…			
x20	x….	GETC	read a single character (no echo)
x21	x….	OUT	output a character to the monitor
x22	x….	PUTS	write a string to the console
x23	x….	IN	print prompt to console, read and echo character from keyboard
X23	x….	PUTSP	write a string to the console; two chars per memory location
x25	x….	HALT	halt the program
…			

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

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 we save old PC in R7)
  - PC←MDR

- Once the service routine is done, control is passed back to the user program suing <u>RET</u> instruction, which is just another name for <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

What do we need to make this work?

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

- The actual code of the service routine
- Mechanism for invocation
  - TRAP Instruction, e.g., TRAP x23
  - TRAP vector
  - MAR ←ZEXT[trapvector]
  - MDR ←MEM[MAR]
  - PC←MDR
- How to return to user program after execution of OUT completes?

Address	Contents	Comments
x0000		;system space;
X0023	**x04A0**	**; Trap vector table**
x00FF		; End of trap vect[illegible]
x04A0		; code for IN
….		
x04…		; code for OUT
…		
x3000		; user program
…		
	TRAP x23	; call to
xFE00		; Device registers

What do we need to make <u>this</u> work?

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

- The actual code of the service routine
- Mechanism for invocation
  - TRAP Instruction, e.g., TRAP x23
  - TRAP vector
  - MAR ←ZEXT[trapvector]
  - MDR ←MEM[MAR]
  - R7 ← PC
  - PC←MDR
- Mechanism for resuming user program
  - RET ≡ JMP R7

	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								

Address	Contents	Comments
x0000		;system space;
x0023	**x04A0**	**; Trap vector table**
x00FF		; End of trap vect
x04A0		; code for IN
….	**RET**	
x04..		; code for OUT
…		
x3000		; user program
…		
	TRAP x23	; call to
xFE00		; Device registers

Putting it all together: 4 Things make TRAPs work

1. TRAP instruction
   - used by program to transfer control to OS subroutines
   - 8-bit trap vector names one of the 256 subroutines
2. Trap vector table: stores starting addresses of OS subroutines
   - stored at x0000 through x00FF in memory
3. A set of OS subroutines
   - part of operating system -- routines start at arbitrary addresses (convention is that system code is below x3000) up to 256 routines
4. A linkage back to the user program (RET)
   - want execution of the user program to resume immediately after the TRAP instruction

Example:

Convert lowercase input characters to uppercase characters.
The program uses a sentinel of 1 to terminate the program

```
        .ORIG x3000
;Convert Lowercase Letter to Uppercase
;Program terminate When press 1

        LD R2, TERM
        LD R3, ASCII_DIFF
        NOT R3, R3;
        ADD R3, R3, #1 ;2's complement of x0020
AGAIN   TRAP x23
        ADD R1,R2,R0 ; Test for Terminating Character
        BRz EXIT
        ADD R0,R0,R3
        TRAP x21 ;Out
        BR AGAIN

EXIT    TRAP x25 ;HALT

TERM        .FILL   xFFCF  ;1 is used as sentinel to terminate
ASCII_DIFF  .FILL   x0020  ;Difference between the lowercase and Uppercase ASCII
.END
```

Please see the Modified version (which check invalid inputs) posted on Github

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?

Suggested approach:

- Caller of service routine can save (and restore): **Caller-save**
- Called service routine saves (and restore): **Callee-save**

- Saving and restoring values of registers is an example of a task computers need to perform in **context switching**

```
; Caller-save user program
…
ST R0, SaveR0        ; store R0 in memory
ST R7, SaveR7        ; store R7 in memory
IN                   ; call TRAP which
                     ; destroys R0 and R7
LD R7, SaveR7        ; restore R7
…                    ; consume input in R0
LD R0, SaveR0        ; restore R0
…
HALT

SaveR0 .BLKW 1
SaveR7 .BLKW 1
```

What do we need to make this work? (3RD Edition Book)

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

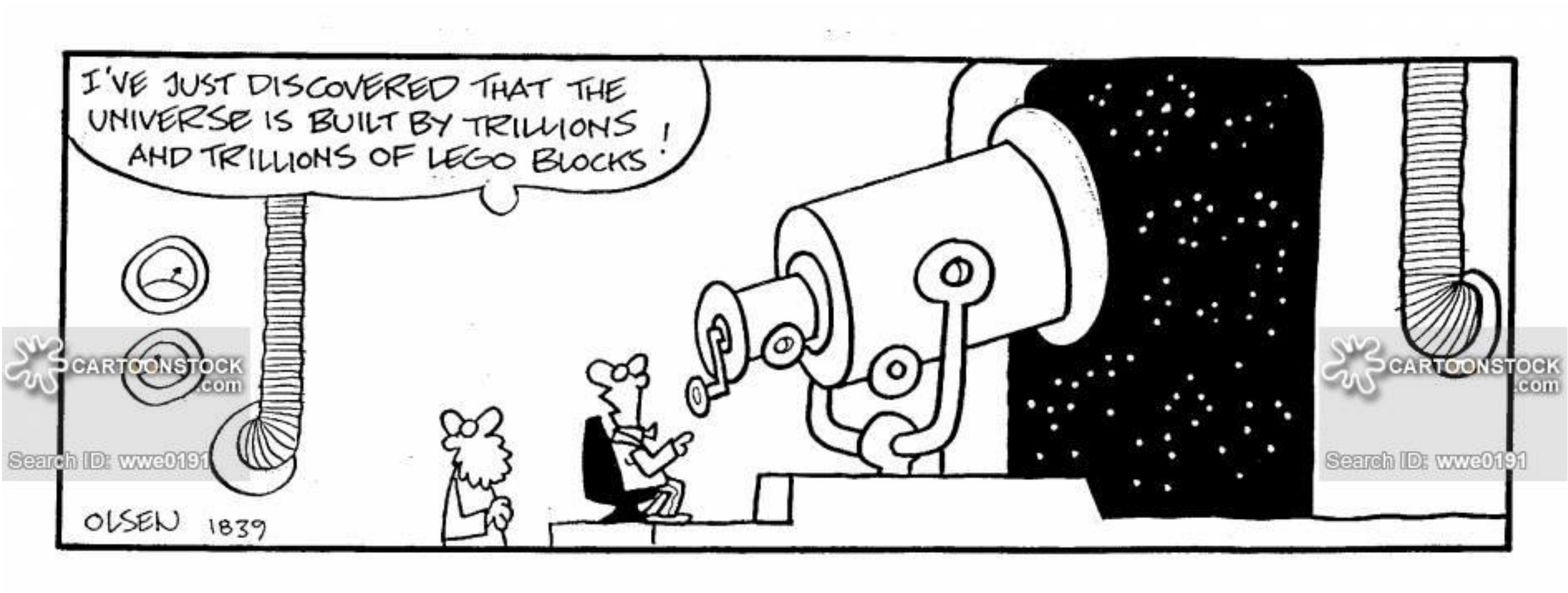
- The actual code of the service routine
- Mechanism for invocation
  - Fetch TRAP Instruction, e.g., TRAP x23
  - System Stack ← PC, PSR
  - PSR[15] is set to zero and save R6 and load R6 ← content of Saved_SSP
  - MAR ←ZEXT[trapvector]
  - MDR ←MEM[MAR]
  - PC←MDR
- Mechanism for resuming user program
  - RTI

  -pops the two values of system stack into PC and PSR.

  - Check PSR[15] and R6 is restored accordingly

Address	Contents	Comments
x0000		;system space;
x0023	**x04A0**	**; Trap vector table**
x00FF		; End of trap vect[illegible]
x04A0		; code for IN
….	**RET**	
x04..		; code for OUT
…		
x3000		; user program
…		
	TRAP x23	; call to
xFE00		; Device registers

Key Concept: Abstraction

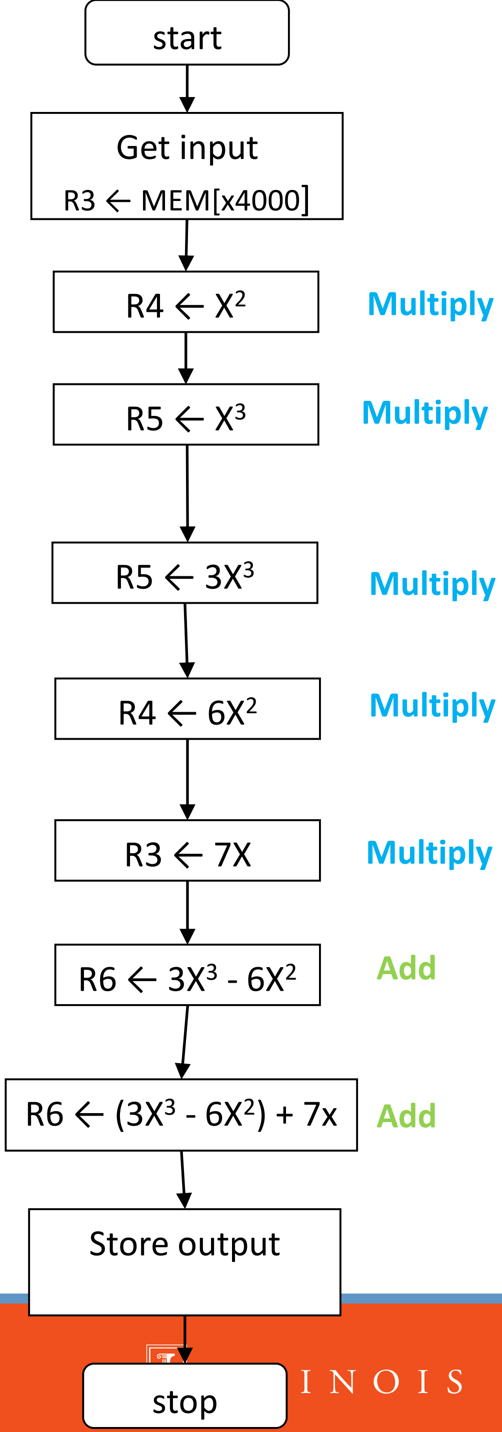


Observation

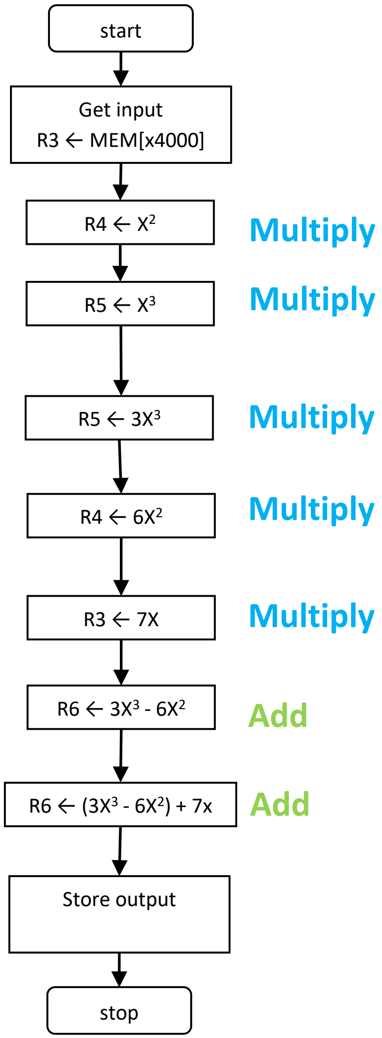
Example problem: Compute $y=3x^3-6x^2+7x$ for any input $x > 0$

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 x^2
...
...
; Multiply R5 ← R4 * R3 x^3
...
; Multiply R5 ← R5 * 3 (3x^3)
...
; Multiply R4 ← 6 * R4
```


*Sometimes programs are compiled to look like this for better performance: inline functions

Subroutines

- A sequence of instructions that performs a specific task (and nothing else---no side effects). This unit can then be used in programs wherever that particular task should be performed.
- Hide details of code and package them with an interface
  - Abstract away details
  - **Needs only Arguments** and **return values**
- Why is this a good idea in programming?
  - Reuse; shorter programs
  - Simplify; code comprehension
  - Teamwork; allows multiple developers to work on different pieces; libraries
- TRAPs are examples of OS subroutines

Idea

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

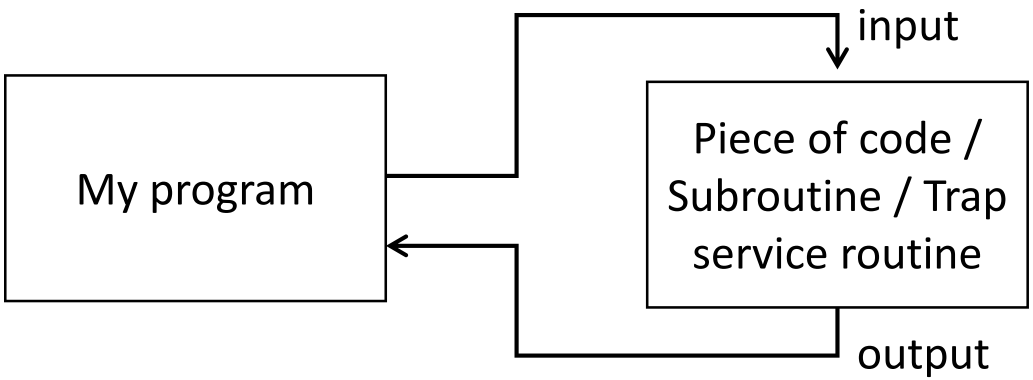
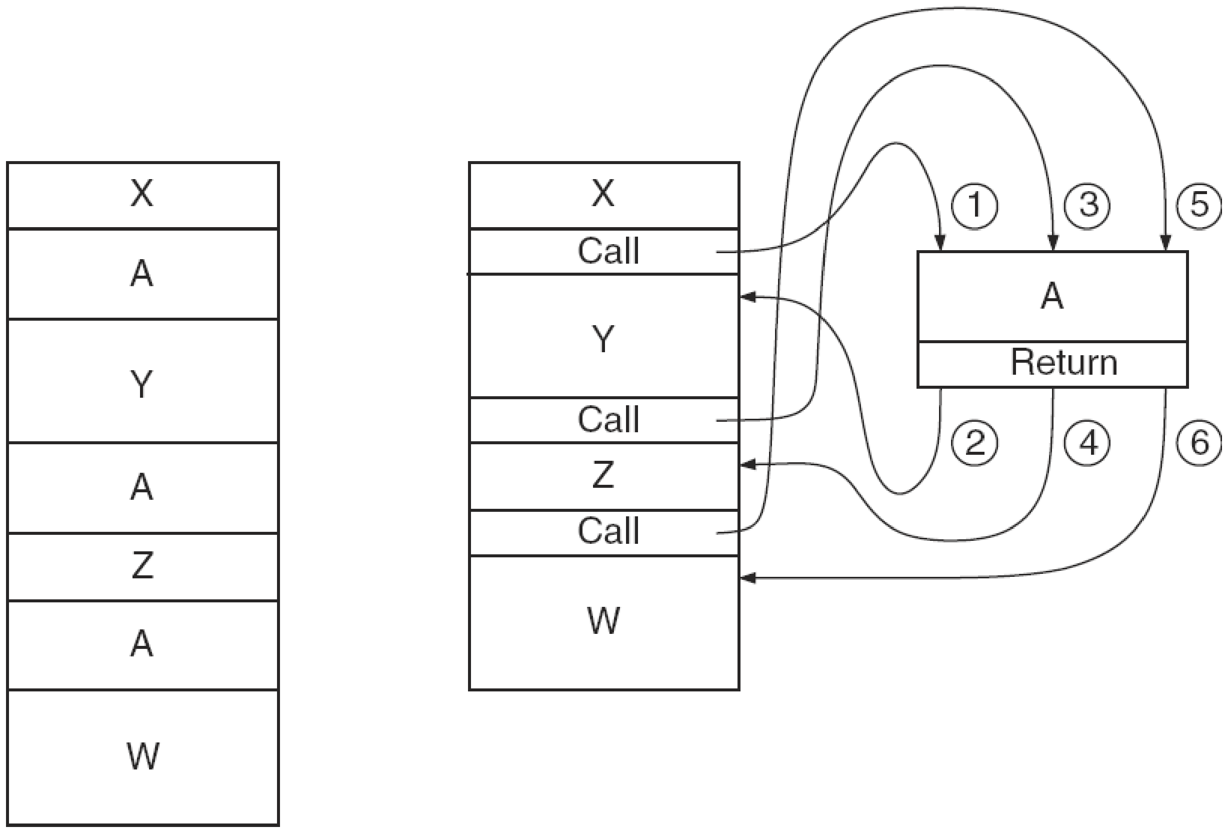


Figure 9.7



(a) Without subroutines

(b) With subroutines

JSR and JSRR

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

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

R7←PC

If (IR[11] == 0) PC←BaseR

Else PC←PC+SEXT(IR[10:0])

	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								

RET ≡ JMP R7

PC ← R7

Subroutine Examples

- Can you find the bugs in the following piece of code?

```
; SUBTR subroutine computes difference of two 2's complement numbers
; IN: R1, R2
; JSR SUBTR
OUT: R0 <- R1-R2
SUBTR   NOT R2, R2
        ADD R7, R2, #1
        ADD R0, R1, R7
        RET
```

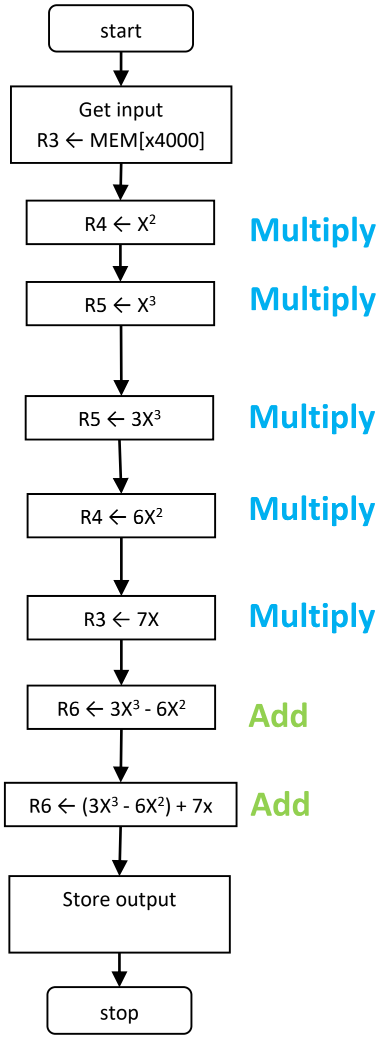
Subroutine Examples

- **Can you find the bugs in the following piece of code?**

```
; SUBTR subroutine computes difference of two 2's complement numbers
; IN: R1, R2
; JSR SUBTR
OUT: R0 <- R1-R2
SUBTR  NOT R2, R2
       ADD R7, R2, #1
       ADD R0, R1, R7
       RET
```

- Issues with this implementation
  - Argument in R2 will be changed on exit
    - Fix: save/restore R2 in the subroutine
  - R7 value is modified in the subroutine, thus, we will not be able to return from it
    - Fix: do not use R7, or save/restore it in the subroutine
  - If R7 holds a useful value in the main program, it will be overwritten during the subroutine call
    - Fix: save/restore R7 in the main program
  - If SUBTR is located far enough in the memory from the point where it is called, we may not be able to reach it with JSR instruction
    - Fix: use JSRR instruction instead

Implementation (Code is on Github)




```
;; 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
```


*Sometimes programs are compiled to look like this for better performance: inline functions