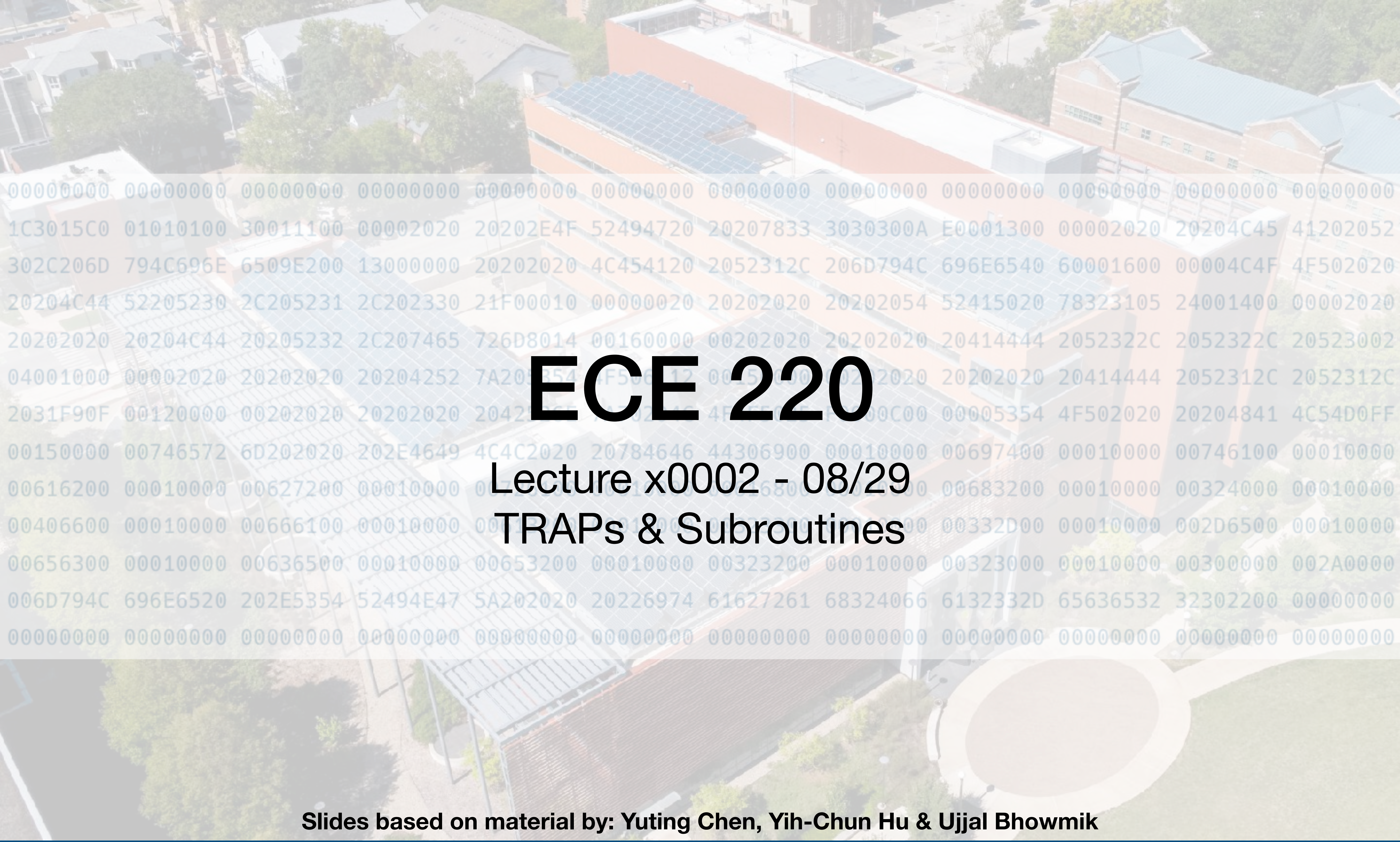


# ECE 220

## Lecture x0002 - 08/29

## TRAPs & Subroutines

**Slides based on material by: Yuting Chen, Yih-Chun Hu & Ujjal Bhowmik**

# Recap from “08/27”

- Consider “echo” routine:

```
KPOLL   LDI      R1, KBSR
        BRzp     KPOLL
        LDI      R0, KBDR
```

```
DPOLL   LDI      R1, DSR
        BRzp     DPOLL
        STI      R0, DDR
```

```
        BRnzp    NEXT_TASK
KBSR    .FILL    xFE00
KBDR    .FILL    xFE02
DSR     .FILL    xFE04
DSR     .FILL    xFE06
```

- Reading & writing from keyboard or display is common task
  - Inefficient to keep repeating this code
  - Need to free up `R1` and `R0` for use whenever blocks run
    - Save/restore current values before/after these blocks run

# Recap from “08/27”

- Consider “echo” routine:

```
;SAVE R0, 1
KPOLL  LDI     R1, KBSR
       BRzp    KPOLL
       LDI     R0, KBDR
;RESTORE R0, R1
```

```
;SAVE R0, R1
DPOLL  LDI     R1, DSR
       BRzp    DPOLL
       STI     R0, DDR
;RESTORE R0, R1
```

```
       BRnzp   NEXT_TASK
KBSR   .FILL   xFE00
KBDR   .FILL   xFE02
DSR    .FILL   xFE04
DDR    .FILL   xFE06
```

- Reading & writing from keyboard or display is common task
  - Inefficient to keep repeating this code
  - Need to free up `R1` and `R0` for use whenever blocks run
    - Save/restore current values before/after these blocks run

# Repeating code

- Consider $f(x) = x^4 + 4x^3 + 3x^2 + 2x + 1$
- Evaluate $f(2)$
  - How many multiplications?

# Repeating code

- Consider $f(x) = x^4 + 4x^3 + 3x^2 + 2x + 1$
- Evaluate $f(2)$
  - How many multiplications?
- Suppose we wish to evaluate $f(x)$ for many values of $x$
  - Why? E.g. Newton-Raphson method for finding roots of $f(x)$

**Note:** Information in “Asides” are not exam material but rather for your intellectual edification.

# Aside: NR method

**More multiplications!**

Suppose $f(x)$ such that $x, f(x) \in \mathbb{R}$ and $f'(x)$ is well defined. Let $x_0$ be an initial guess for some root $\bar{x}$ of $f(x)$. Then the iterates $x_n$

$$x_1 = x_0 - \frac{f(x)}{f'(x_0)} \quad \text{and} \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

successively improve on the guess $x_0$ as an approximation to $\bar{x}$ (roughly doubling the number of correct digits at each step).

**Note:** Information in “Asides” are not exam material but rather for your intellectual edification.

# Aside: Calculating $\sin(x)$

- Can you think of another instance where evaluating polynomials shows up?
  - ***Hint:*** Some power series from Calc 1 or 2
- It is one way most calculators *can* compute trigonometric values?

**Most don’t use the power series expansions but other more efficient methods (e.g. lookup + interpolation, CORDIC, etc.)**

- Example:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

# Subroutines

- Subroutines are blocks/pieces of code that do something specific. Examples:
  - Multiply two numbers
  - Sort a list of integers
  - Read keyboard press into a register
- Often called functions, methods, procedures, service calls, etc.
  - Different from *functions* in mathematics or functional programming languages

# Functions vs. subroutines

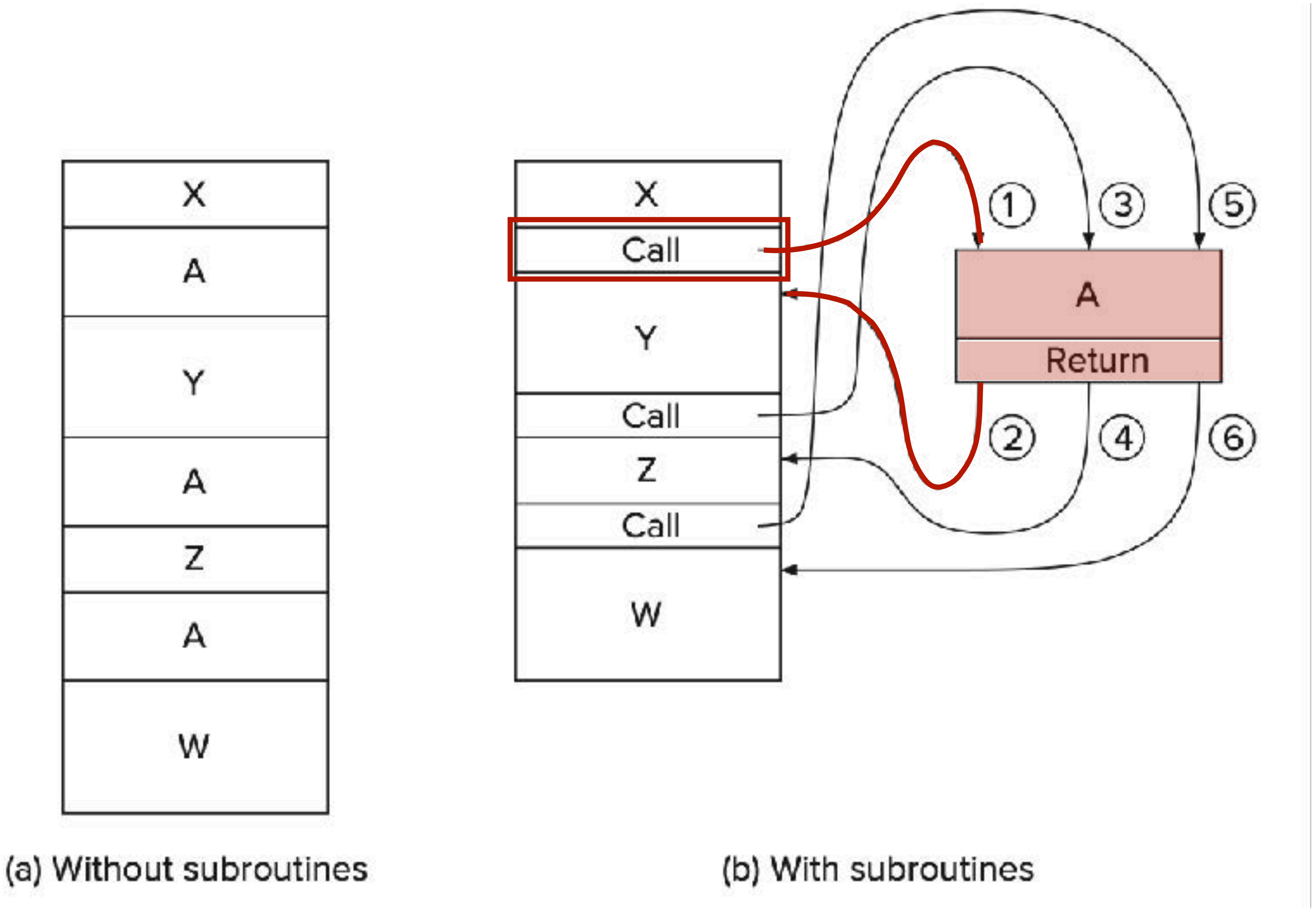
- In mathematics, a function $f(x)$ takes a value from a *set* and returns a value in a(nother) *set.* If you call $f$ with some particular value $x_0$ then it always returns $f(x_0)$.

- In CS/programming, a function `foo` is a piece of code that can be called, *perhaps* with inputs, and does some stuff and *maybe* returns something.

- In *functional* languages (in theory at least), you can replace a function call with its return value and nothing *should* break.

# Subroutines

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



(a) Without subroutines

(b) With subroutines

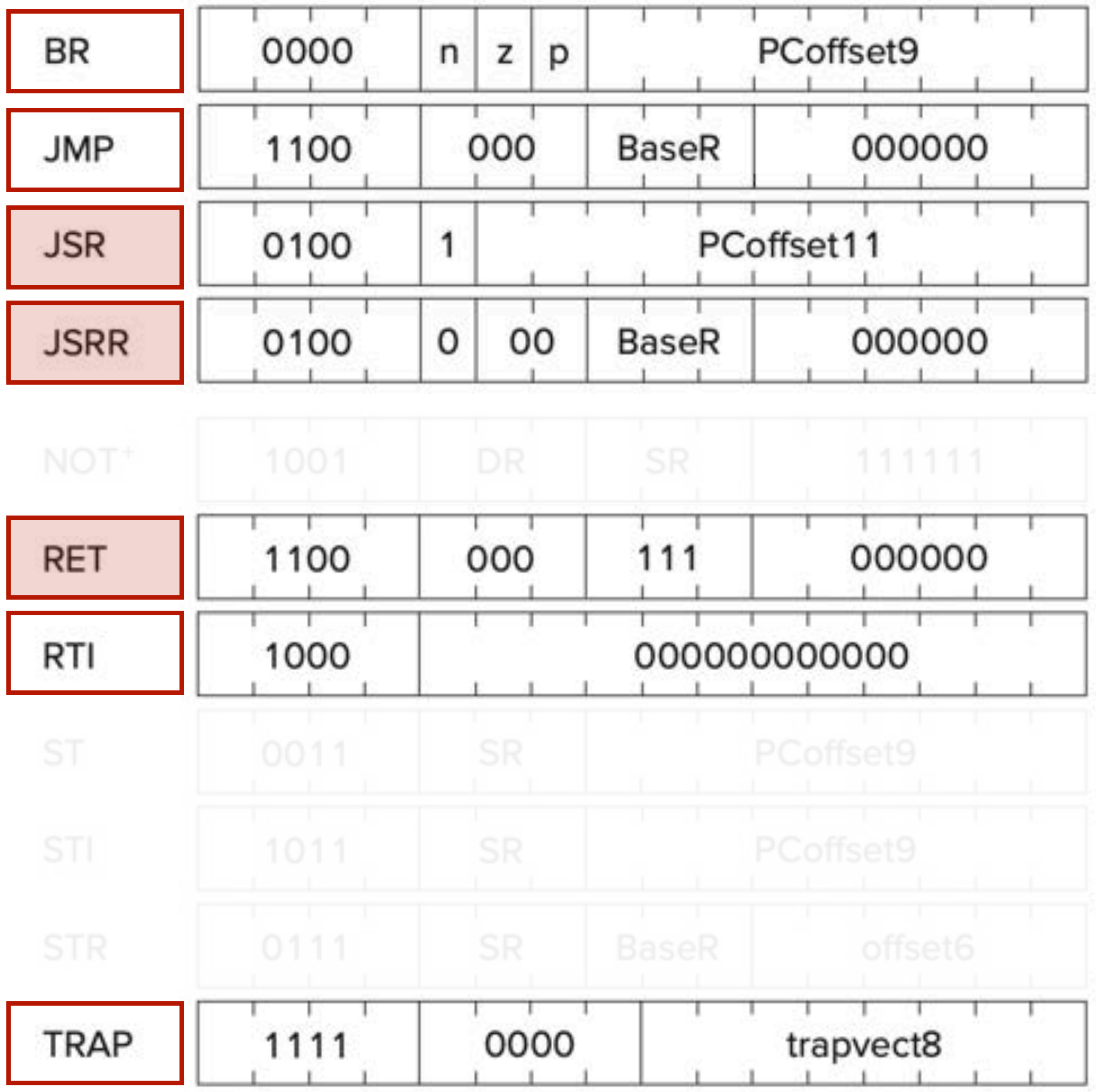
**Figure 8.2 - P&P 3rd Ed.**

# Subroutines in LC3

- Recall instructions that change program flow

- Subroutines make use of the `JSR(R)` and `RET` commands.

- *Exercise:* What is/are the difference(s) between `BR/JMP` and `JSR/JSRR`?

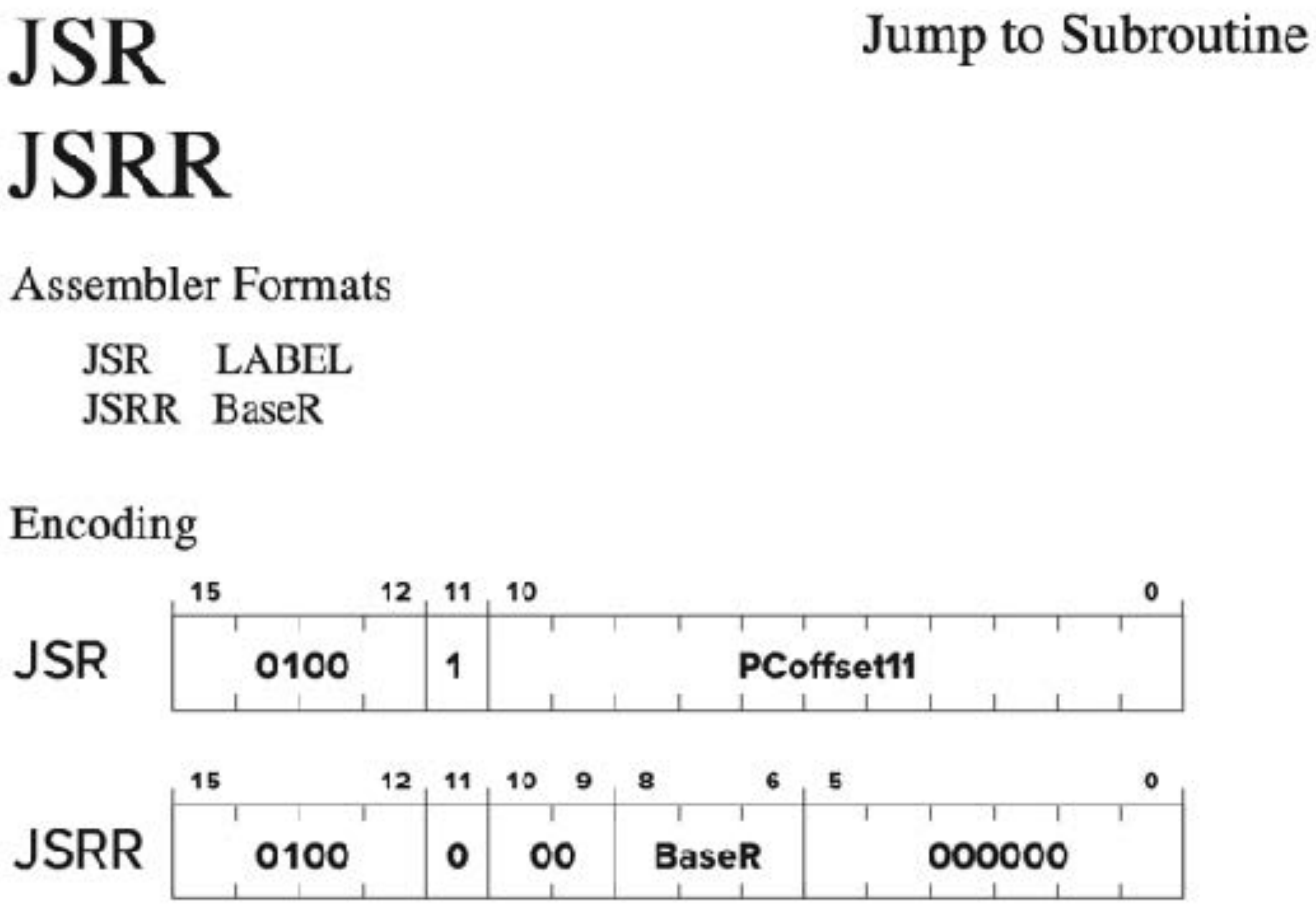


| | | | | |
|---|---|---|---|---|
| BR | 0000 | n z p | PCoffset9 | |
| JMP | 1100 | 000 | BaseR | 000000 |
| JSR | 0100 | 1 | PCoffset11 | |
| JSRR | 0100 | 0 00 | BaseR | 000000 |
| NOT+ | 1001 | DR | SR | 111111 |
| RET | 1100 | 000 | 111 | 000000 |
| RTI | 1000 | 000000000000 | | |
| ST | 0011 | SR | PCoffset9 | |
| STI | 1011 | SR | PCoffset9 | |
| STR | 0111 | SR | BaseR | offset6 |
| TRAP | 1111 | 0000 | trapvect8 | |

**Figure “A.2” - P&P 3rd Ed.**

# JSR & JSRR

- When `JSR(R)` is encountered `R7` is loaded with `PC`[+] and then `PC` is set in one of two ways:
- `JSR` and `JSRR` differ in addressing modes (signified by bit #11).
  - `PC ← PC + SEXT(PCoffset11)`
  - `PC ← BaseR`
- After subroutine ends, `RET` is used to return to caller



**JSR**
**JSRR**

Jump to Subroutine

Assembler Formats

JSR LABEL
JSRR BaseR

Encoding

JSR

| 15–12 | 11 | 10–0 |
|---|---|---|
| 0100 | 1 | PCoffset11 |

JSRR

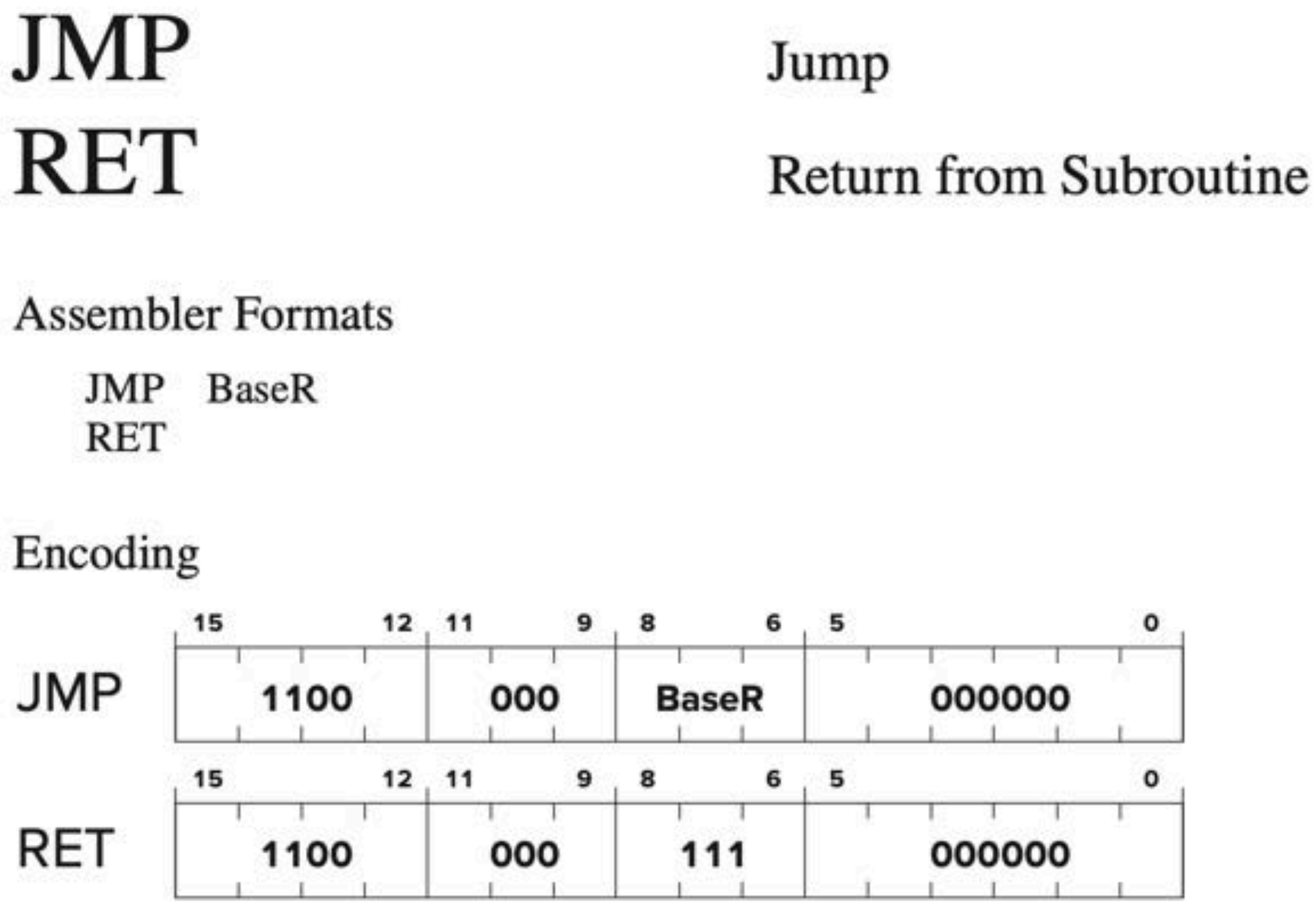
| 15–12 | 11 | 10–9 | 8–6 | 5–0 |
|---|---|---|---|---|
| 0100 | 0 | 00 | BaseR | 000000 |

**Appendix A, P&P 3rd Ed.**

[+] Recall PC is incremented after FETCH.

# RET & JMP

- `JMP` & `RET` are relatives; op-code is the same
  - `JMP: PC ← BaseR`
  - `RET: PC ← R7`
- <u>Note:</u> `JSR(R)` & `RET` rely on `R7` to provide *return-linkage.*
- What if `R7` was being used?



## JMP
## RET

Jump

Return from Subroutine

**Assembler Formats**

JMP BaseR
RET

**Encoding**

| | 15 12 | 11 9 | 8 6 | 5 0 |
|---|---|---|---|---|
| JMP | 1100 | 000 | BaseR | 000000 |
| RET | 1100 | 000 | 111 | 000000 |

# Using subroutines

## Saving & restoring registers

- To use a subroutine the user must know:
  - It’s address (or label)
  - It’s *arguments* (where to pass in data, if any)
  - It’s *return values* (where to *put* computed data, if any)
  - What it does 🙄
    - Maybe not all the gory details but definitely registers it may use or overwrite!

# Using subroutines

## Saving & restoring registers

Generally we have two strategies depending on who saves/restores registers:

- **Caller-saved:** Onus on user to save/restore registers that will be needed later; may not know what registers subroutine will use
    - User saves/restore registers they will need (or know could get destroyed)

- **Callee-saved:** Subroutine knows registers it will alter/use, but cannot know what the user will need later
    - Subroutine saves/restores registers it will use

# Using subroutines

## Saving & restoring registers

Good practices:

- Keep `R7` unused, especially for *nested* subroutines
- Use callee-save, except for return values (should be caller saved)
- Restore incoming *arguments* to their original values unless intended to be overwritten by return value

# Example

## Multiplication

Try to complete `MULTIPLY` subroutine by filling in the missing piece.

```
; LC3 subroutine to multiply two numbers
; Inputs: R0 (multiplicand), R1 (multiplier)
; Output:  R2 (result)

MULTIPLY:
    ST R0, MulSaveR0          ; Callee save registers
    ST R1, MulSaveR1
    AND R2, R2, #0            ; Clear R2 to be used as result
    ADD R2, R0, #0            ; Load multiplicand into R2
    ADD R1, R1, #-1           ; Use R1 as counter

MUL_LOOP:





MUL_DONE:
    LD R0, MulSaveR0          ; Restore registers
    LD R1, MulSaveR1
    RET                       ; Return from the subroutine
```

# Example

## Multiplication

Try to complete `MULTIPLY` subroutine by filling in the missing piece.

```
; LC3 subroutine to multiply two numbers
; Inputs: R0 (multiplicand), R1 (multiplier)
; Output:  R2 (result)

MULTIPLY:
    ST R0, MulSaveR0            ; Callee save registers
    ST R1, MulSaveR1
    AND R2, R2, #0              ; Clear R2 to be used as result
    ADD R2, R0, #0              ; Load multiplicand into R2
    ADD R1, R1, #-1             ; Use R1 as counter

MUL_LOOP:
    BRz MUL_DONE                ; If R1 == 0, multiplication done
    ADD R2, R0, R2
    ADD R1, R1, #-1             ; Decrement the counter in R1
    BR MUL_LOOP                 ; Jump back to MUL_LOOP

MUL_DONE:
    LD R0, MulSaveR0            ; Restore registers
    LD R1, MulSaveR1
    RET                         ; Return from the subroutine
```

# Exercise

## Exponentiation

Use the `MULTIPLY` subroutine in the previous slide to write an LC3 subroutine that performs exponentiation.

```
; LC3 subroutine to that performs exponentiation
; Inputs: R0 (base), R1 (exponent)
; Loop counter: R2
; Output: R2 (result)
; POW knows it should call MULTIPLY and it knows
; MULTIPLY overwrites the value in R2

POW:







POW_LOOP:
    BRz POW_DONE            ; If R2==0, loop complete
    ST R2, PowSaveR2        ; Caller save
    JSR MULTIPLY            ; Result in R2
    ADD R1, R2, #0          ; Copy result for next multiply
    LD R2, PowSaveR2        ; Caller restore
    ADD R2, R2, #-1         ; Decrement counter
    BR POW_LOOP

POW_DONE:
```

# Exercise

## Exponentiation

Use the `MULTIPLY` subroutine in the previous slide to write an LC3 subroutine that performs exponentiation.

**Will this program halt? Why? Why not?**

```
; LC3 subroutine to that performs exponentiation
; Inputs: R0 (base), R1 (exponent)
; Loop counter: R2
; Output: R2 (result)
; POW knows it should call MULTIPLY and it knows
; MULTIPLY overwrites the value in R2

POW:
    ST R0, PowSaveR0      ; Callee save registers
    ST R1, PowSaveR1
    ADD R2, R1, #-1       ; Initialize counter
                          ; Why can't we use R1 as counter?
    ADD R1, R0, #0        ; Set up to call MULTIPLY

POW_LOOP:
    BRz POW_DONE          ; If R2==0, loop complete
    ST R2, PowSaveR2      ; Caller save
    JSR MULTIPLY          ; Result in R2
    ADD R1, R2, #0        ; Copy result for JSR to multiply
    LD R2, PowSaveR2      ; Caller restore
    ADD R2, R2, #-1       ; Decrement counter
    BR POW_LOOP

POW_DONE:
    ADD R2, R1, #0        ; Move result to R2
    LD R0, PowSaveR0      ; Callee restore
    LD R1, PowSaveR1
    RET
```

# Exercise

## Exponentiation

Use the `MULTIPLY` subroutine in the previous slide to write an LC3 subroutine that performs exponentiation.

```
; LC3 subroutine to that performs exponentiation
; Inputs: R0 (base), R1 (exponent)
; Loop counter: R2
; Output: R2 (result)
; POW knows it should call MULTIPLY and it knows
; MULTIPLY overwrites the value in R2

POW:
    ST R0, PowSaveR0    ; Callee save registers
    ST R1, PowSaveR1
    ADD R2, R1, #-1     ; Initialize counter
                        ; Why can't we use R1 as counter?
    ADD R1, R0, #0      ; Set up to call MULTIPLY

POW_LOOP:
    BRz POW_DONE        ; If R2==0, loop complete
    ST R2, PowSaveR2    ; Caller save
    JSR MULTIPLY        ; Result in R2
    ADD R1, R2, #0      ; Copy result for JSR to multiply
    LD R2, PowSaveR2    ; Caller restore
    ADD R2, R2, #-1     ; Decrement counter
    BR POW_LOOP

POW_DONE:
    ADD R2, R1, #0      ; Move result to R2
    LD R0, PowSaveR0    ; Callee restore
    LD R1, PowSaveR1
    RET
```

**Nested subroutines better save R7!**

# User routine vs. service routine

- Consider keyboard input:
  - It’s used often and has too many specific details for most programmers
  - Improper usage could breach security of the system or mess up keyboard usage for other users/programs
- Solution: make this part of the OS
  - User program → invokes service routine (a.k.a OS call) → OS performs operation → returns control to user program

# TRAP mechanism

System calls in LC3 are achieved using the `TRAP` mechanism

**TRAP** System Call

Assembler Format

TRAP trapvector8

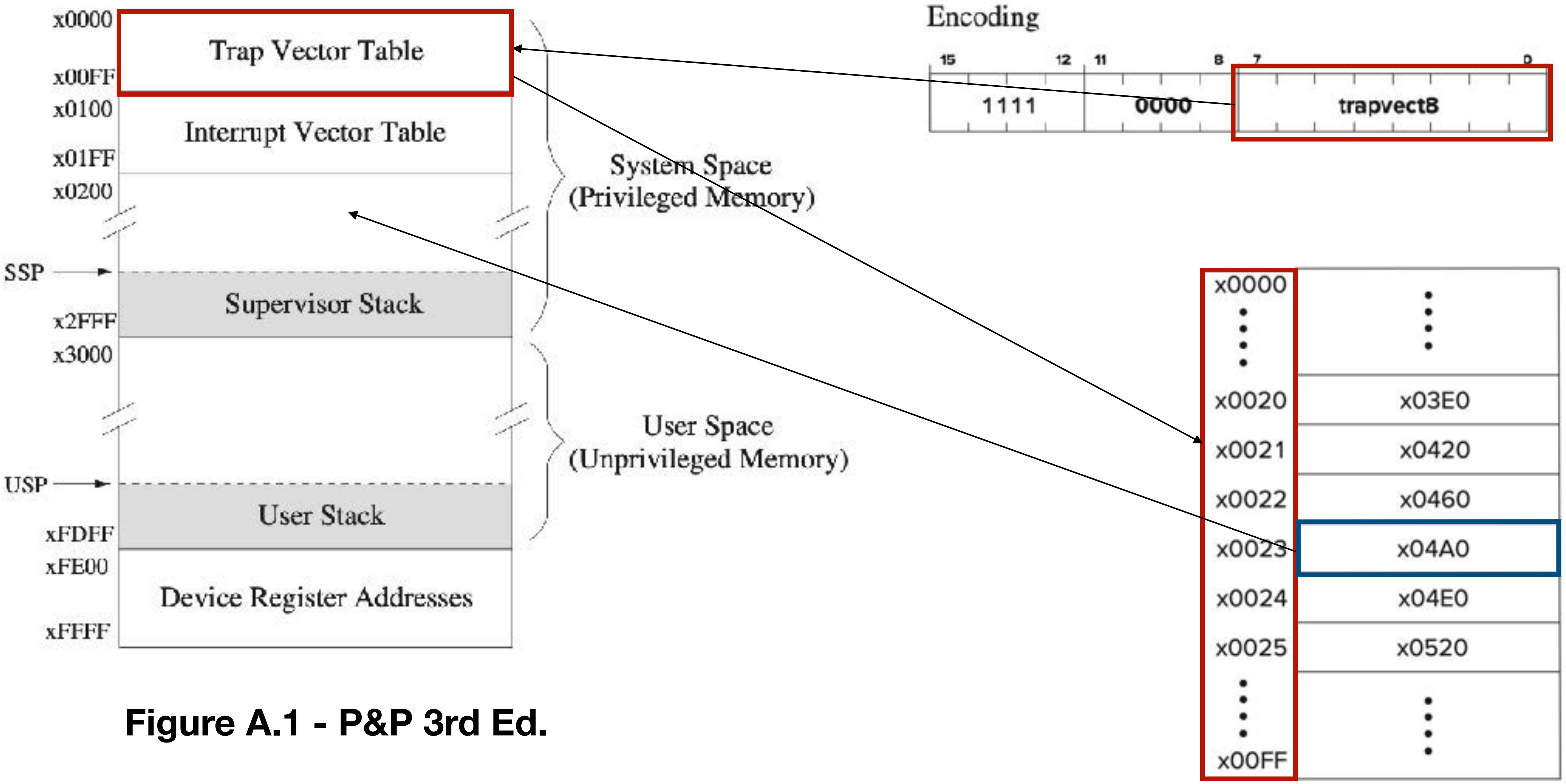


**Figure A.1 - P&P 3rd Ed.**

# TRAP mechanism

**Table A.3 of P&P 3rd Ed.**

| Vector | Symbol | Routine |
|---|---|---|
| x20 | GETC | Read a single character (no echo) |
| x21 | OUT | Output character to monitor |
| x22 | PUTS | Write a string to monitor |
| x23 | IN | Print prompt to monitor, read and echo character from keyboard |
| x24 | PUTSP | Write a string to monitor, two characters per memory location |
| x25 | HALT | Halt program |
| x26 |  | Write a number to monitor (undocumented) |

*Exercise*: Try using each of these!

# TRAP: Flow Control

- Slight difference between editions of the textbook
- **Edition 2**: Last statement in `TRAP` is `JMP R7` (i.e. `RET`)
- **Edition 3**: Last statement is `RTI`

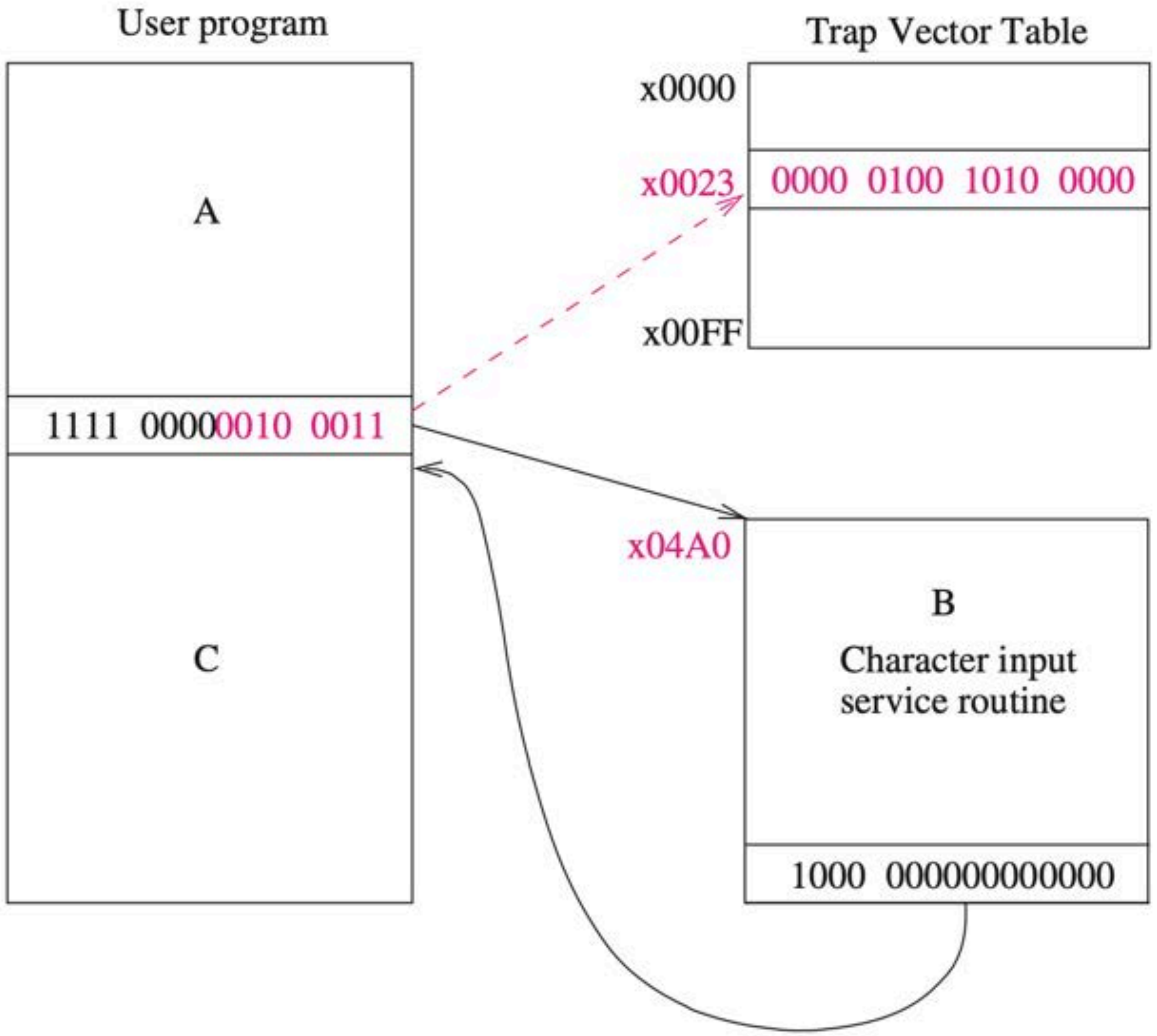
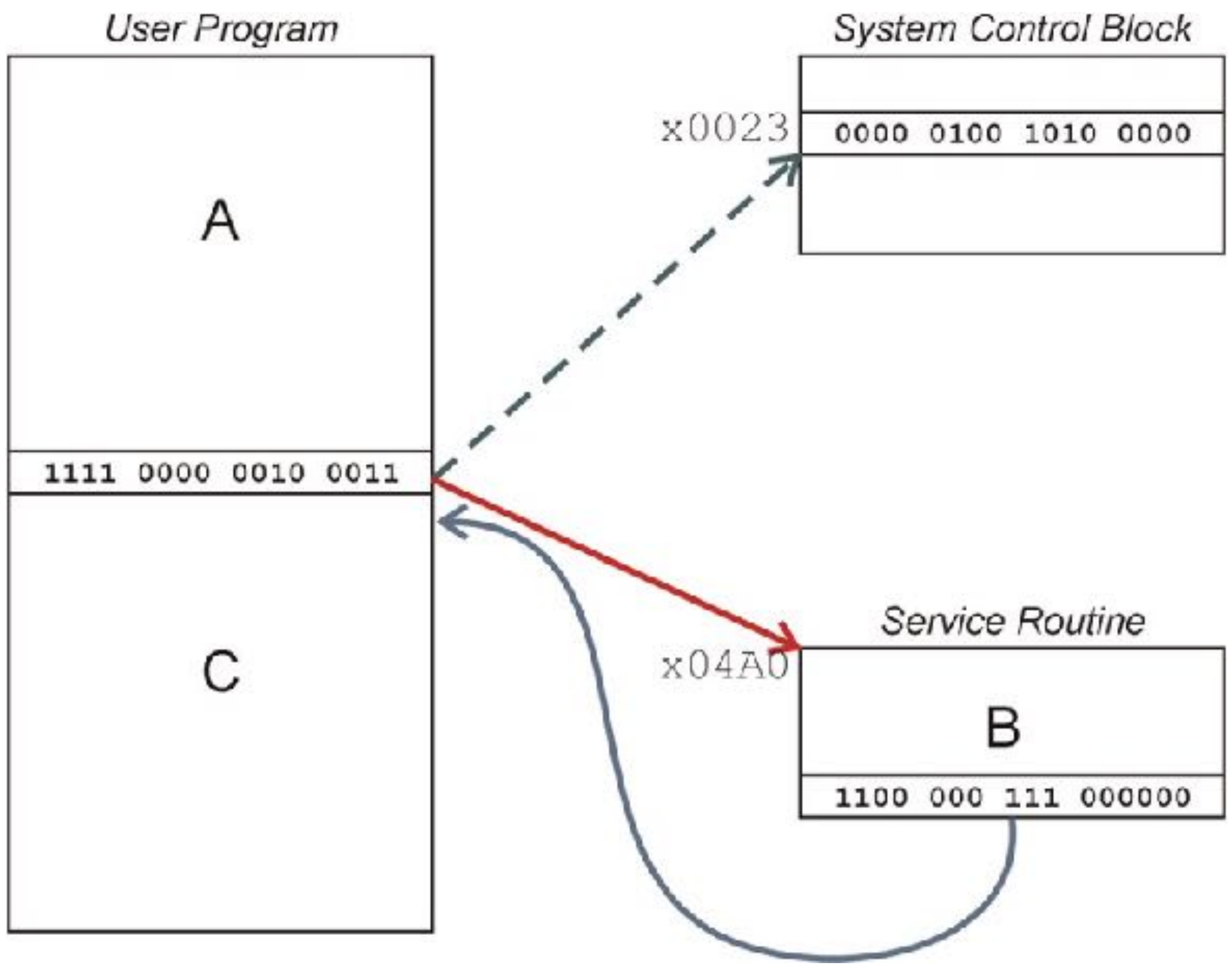


**Figure 9.11 In P&P 3rd Ed.**

# TRAP Mechanism: 2nd Ed.

- `MAR ← ZEXT(trapvect8)`
- `MDR ← MEM[MAR]`
- `R7 ← PC`
- `PC ← MDR`
- `....`
- `JMP R7`



# TRAP example

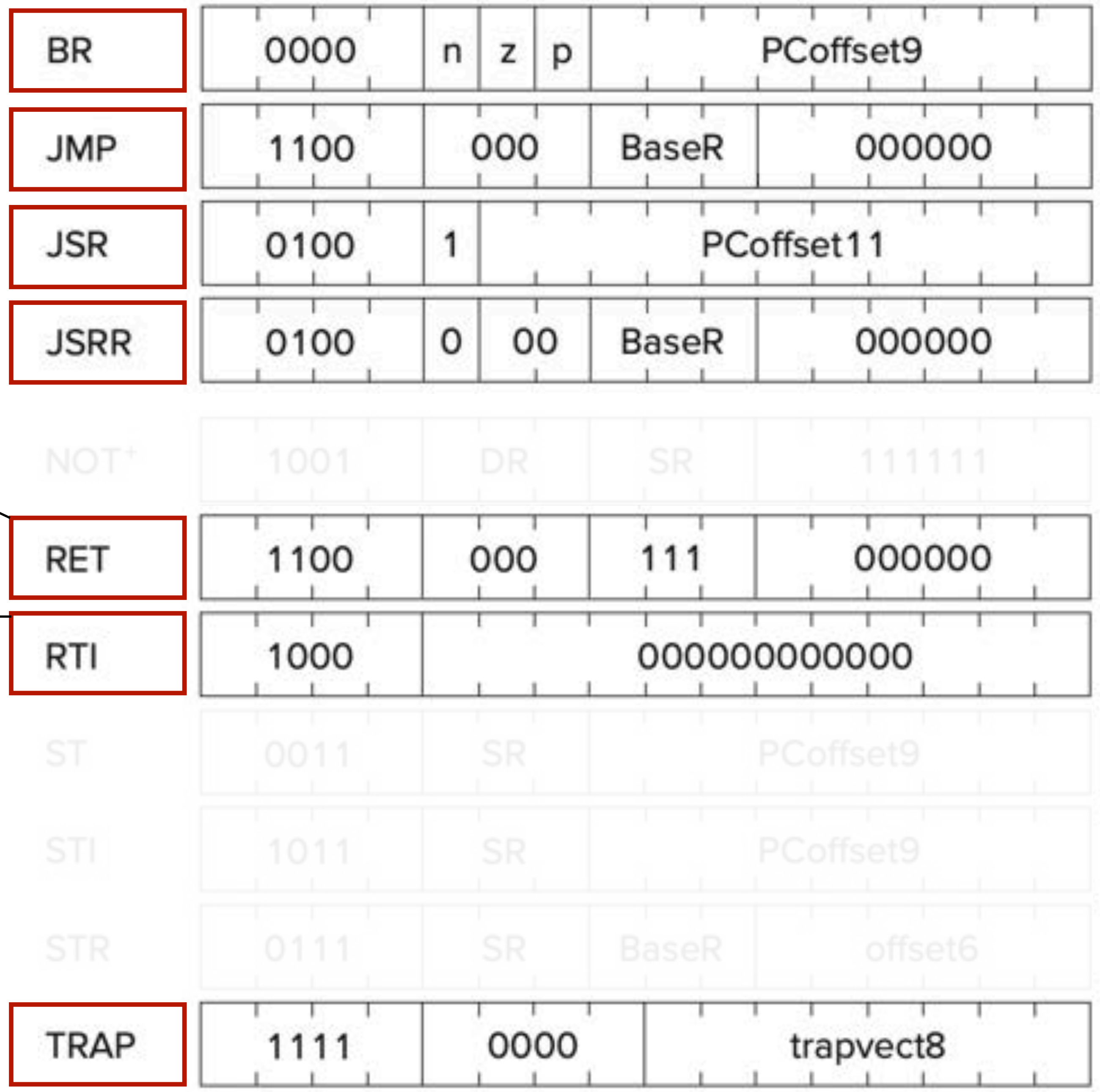
- What are the values in `R0` and `R7` right before `IN`?
- How about right before `HALT`?

```
.ORIG x3000


AND R0, R0, #0          ;init R0
ADD R0, R0, #3          ;set R0 to 3
ADD R7, R0, #4          ;set R7 to 7
ADD R0, R0, #1          ;increment R0
ADD R7, R7, #1          ;increment R7

IN                      ;same as 'TRAP x23'

ADD R0, R0, #1          ;increment R0
ADD R7, R7, #0          ;increment R7

HALT
.END
```

# RTI: Return from TRAP/Interrupt

- 2nd edition: LC3 will overwrite `R7`
- 3rd edition: `R7` will be left unchanged.
    - Mechanism? **Uses *stacks* → next lecture.**

**Which one does EWS use?**



| | | | | | |
|---|---|---|---|---|---|
| BR | 0000 | n | z | p | PCoffset9 |
| JMP | 1100 | 000 | BaseR | 000000 | |
| JSR | 0100 | 1 | PCoffset11 | | |
| JSRR | 0100 | 0 | 00 | BaseR | 000000 |
| NOT+ | 1001 | DR | SR | 111111 | |
| RET | 1100 | 000 | 111 | 000000 | |
| RTI | 1000 | 000000000000 | | | |
| ST | 0011 | SR | PCoffset9 | | |
| STI | 1011 | SR | PCoffset9 | | |
| STR | 0111 | SR | BaseR | offset6 | |
| TRAP | 1111 | 0000 | trapvect8 | | |

**Figure "A.2" - P&P 3rd Ed.**

# TRAP vs. subroutines

- Service routines (`TRAP`) provide 3 main functions
  - Shield programmers from system-specific details (`KBDR`, `KBSR`, etc.)
  - Write frequently-used code just once
  - Protect system recourses from malicious/clumsy programmers
- Subroutines provide the same functions for non-system (user) code
  - Lives in user space
  - Performs a well-defined task
  - Is invoked (called) by another user program
  - Returns control to the calling program when finished