14. TRAP and I/O Service Routines (Chapter 9)
October 17, 2018

- **LC-3 TRAP Routines**
  - **TRAP mechanism**
  - **TRAP instruction**
  - **Handling I/O**
  - **Halting the computer**
  - **Saving and restoring registers**

EXAM 1: WEDNESDAY, OCT. 31

REVIEW SESSION:
SUNDAY, OCT. 28, 2-5 PM
IN THIS CLASSROOM

System Calls

Certain operations require specialized knowledge and protection:

- **specific knowledge of I/O device registers and the sequence of operations needed to use them**
- **I/O resources shared among multiple users/programs; a mistake could affect lots of other users!**

Not every programmer knows (or wants to know) this level of detail

Provide *service routines* or *system calls* (part of operating system) to safely and conveniently perform low-level, privileged operations

System Call

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

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

LC-3 TRAP Mechanism

1. A set of service routines.

- **part of operating system -- routines start at arbitrary addresses**
  (convention is that system code is below x3000)
- **up to 256 routines**

2. Table of starting addresses.

- **stored at x0000 through x00FF in memory**
- **called System Control Block in some architectures**

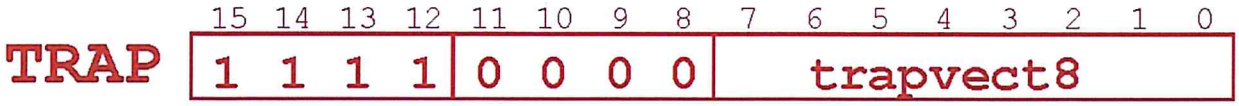
3. TRAP instruction.

- **used by program to transfer control to operating system**
- **8-bit trap vector names one of the 256 service routines**

4. A linkage back to the user program.

- **want execution to resume immediately after the TRAP instruction**

TRAP Instruction



Trap vector

- identifies which system call to invoke
- 8-bit index into table of service routine addresses
  - in LC-3, this table is stored in memory at 0x0000 – 0x00FF
  - 8-bit trap vector is zero-extended into 16-bit memory address

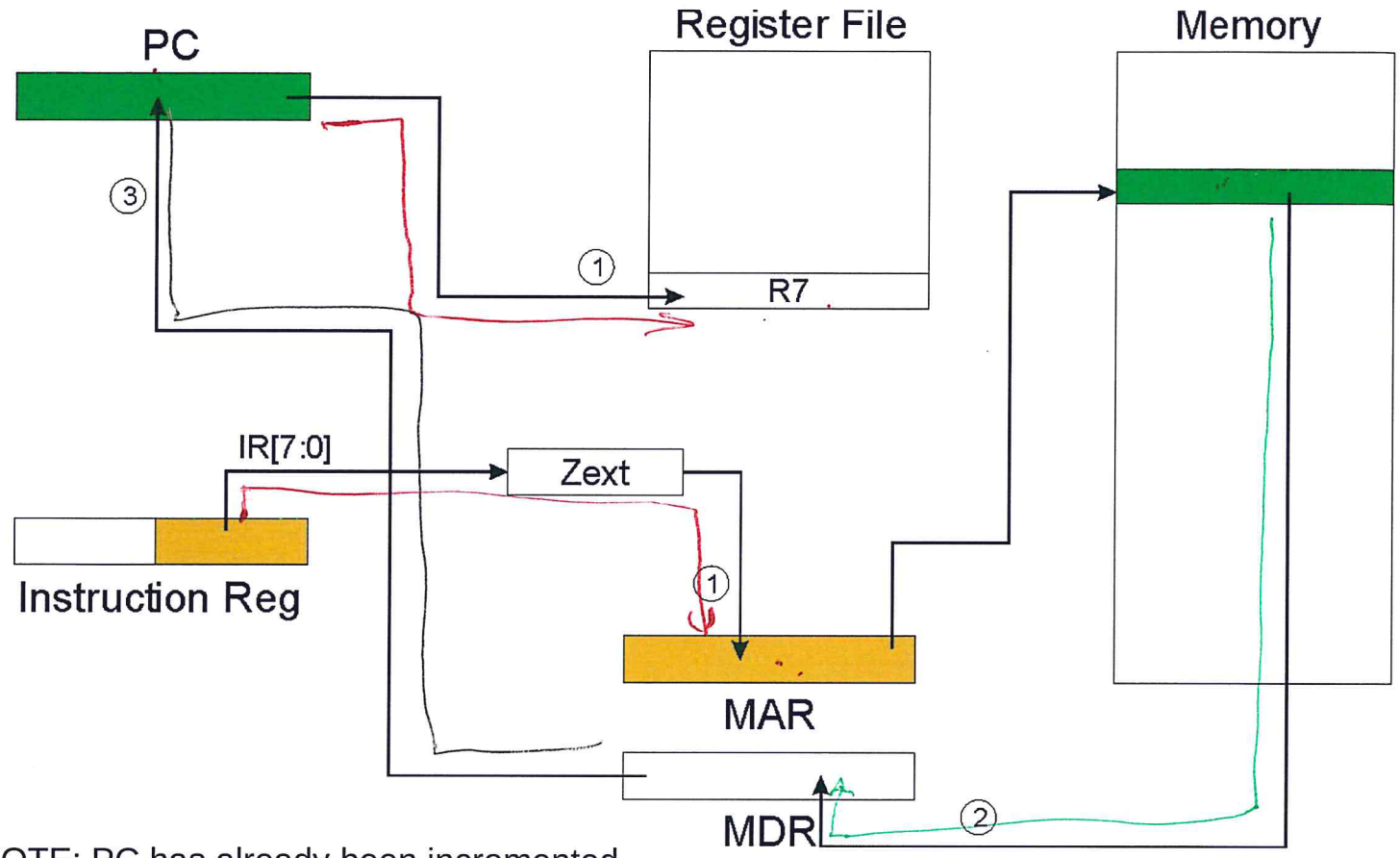
Where to go

- lookup starting address from table; place in PC

How to get back

- save address of next instruction (current PC) in R7

TRAP



NOTE: PC has already been incremented during instruction fetch stage.

RET (JMP R7)

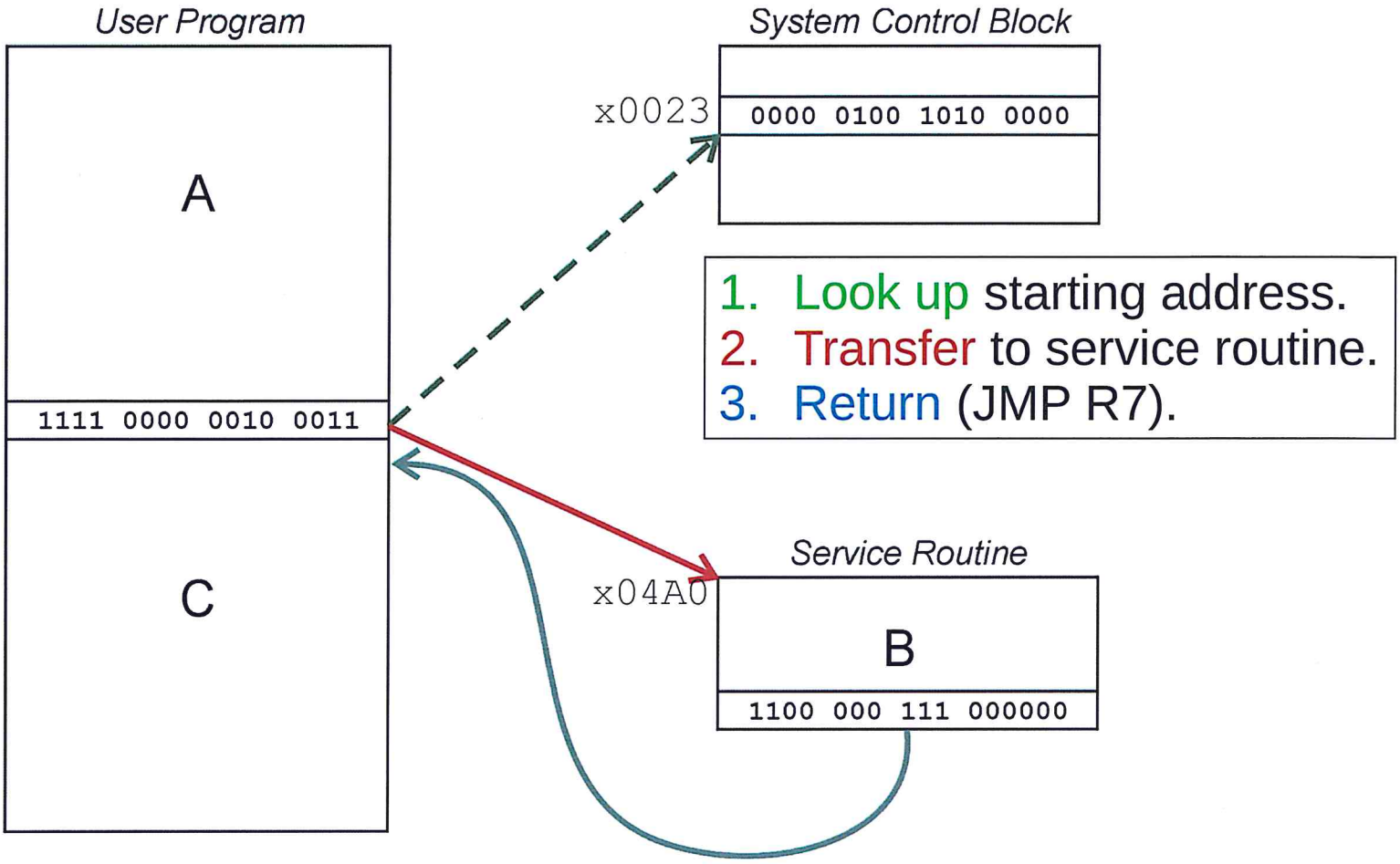
How do we transfer control back to instruction following the TRAP?

We saved old PC in R7.

- **JMP R7** gets us back to the user program at the right spot.
- LC-3 assembly language lets us use **RET** (return) in place of "JMP R7".

Must make sure that service routine does not change R7, or we won't know where to return.

TRAP Mechanism Operation

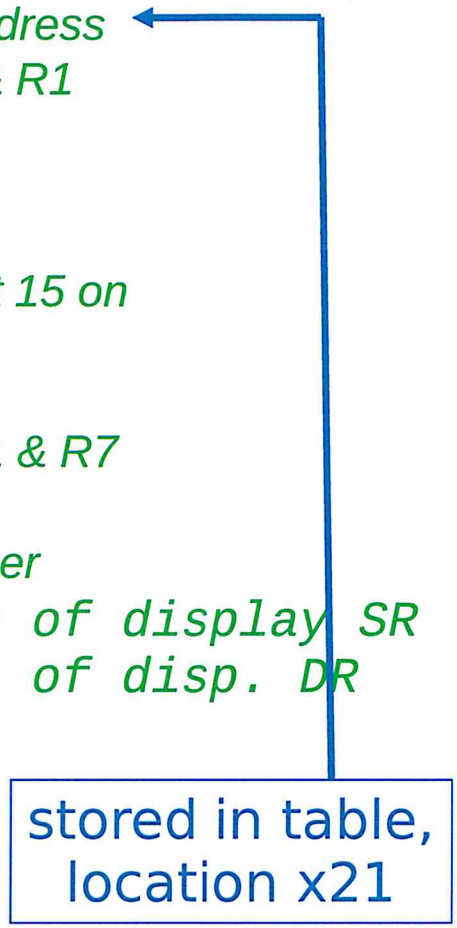


Example: Using the TRAP Instruction

```
            .ORIG x3000
            LD    R2, TERM
            LD    R3, ASCII
AGAIN       TRAP  x23
            ADD   R1, R2, R0
            BRz   EXIT
            ADD   R0, R0, R3
            TRAP  x21
            BRnzp AGAIN
TERM        .FILL     xFFC9
ASCII       .FILL     x0020
EXIT        TRAP  x25
            .END
```

Example: Output Service Routine

```
            .ORIG x0430          ; syscall address
            ST    R7, SaveR7     ; save R7 & R1
            ST    R1, SaveR1
; ----- Write character
TryWrite    LDI   R1, DSR        ; get status
            BRzp  TryWrite       ; look for bit 15 on
WriteIt     STI   R0, DDR        ; write char
; ----- Return from TRAP
Return      LD    R1, SaveR1     ; restore R1 & R7
            LD    R7, SaveR7
            RET                  ; back to user
DSR         .FILL xFE04          ; Address of display SR
DDR         .FILL xFE06          ; Address of disp. DR
SaveR1      .BLKW 1
SaveR7      .BLKW 1
            .END
```



TRAP Routines and their Assembler Names

vector	*symbol*	*routine*
x20	GETC	read a single character (no echo)
x21	OUT	output a character to the monitor
x22	PUTS	write a string to the console
x23	IN	print prompt to console, read and echo character from keyboard
x25	HALT	halt the program

Saving and Restoring Registers

Must save the value of a register if:

- **Its value will be destroyed by service routine, and**
- **We will need to use the value after that action.**

Who saves?

- **caller of service routine?**
  - ➢ **knows what it needs later, but may not know what gets altered by called routine**
- **called service routine?**
  - ➢ **knows what it alters, but does not know what will be needed later by calling routine**

Example

```
        LEA   R3, Binary
        LD    R6, ASCII   ; char->digit template
        LD    R7, COUNT   ; initialize to 10
AGAIN   TRAP  x23         ; Get char
        ADD   R0, R0, R6      ; convert to number
        STR   R0, R3, #0      ; store number
        ADD   R3, R3, #1      ; incr pointer
        ADD   R7, R7, -1      ; decr counter
        BRp   AGAIN           ; more?
        BRnzp NEXT
ASCII   .FILL xFFD0
COUNT   .FILL #10
Binary  .BLKW #10
```

What's wrong with this routine?
What happens to R7?

Saving and Restoring Registers

Called routine -- *"callee-save"*

- **Before start, save any registers that will be altered (unless altered value is desired by calling program!)**
- **Before return, restore those same registers**

Calling routine -- *"caller-save"*

- **Save registers destroyed by own instructions or by called routines (if known), if values needed later**
  - **save R7 before TRAP**
  - **save R0 before TRAP x23 (input character)**
- **Or avoid using those registers altogether**

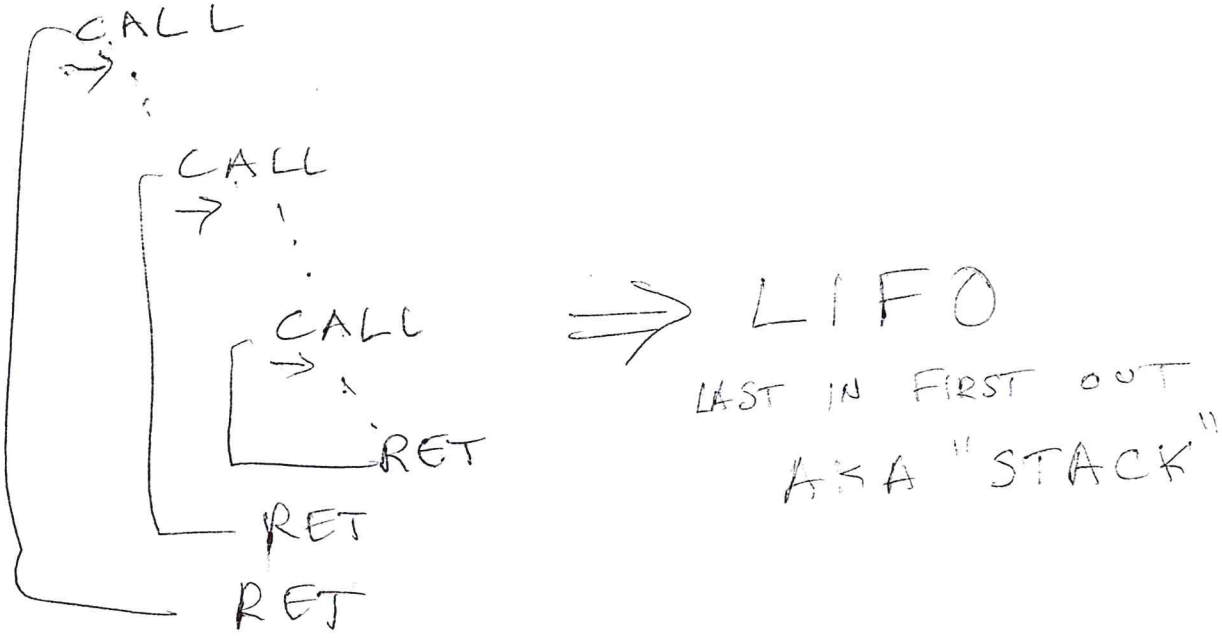
Values are saved by storing them in memory.

Question

Can a service routine call another service routine?

NOT DIRECTLY (R7!)

If so, is there anything special the calling service routine must do?

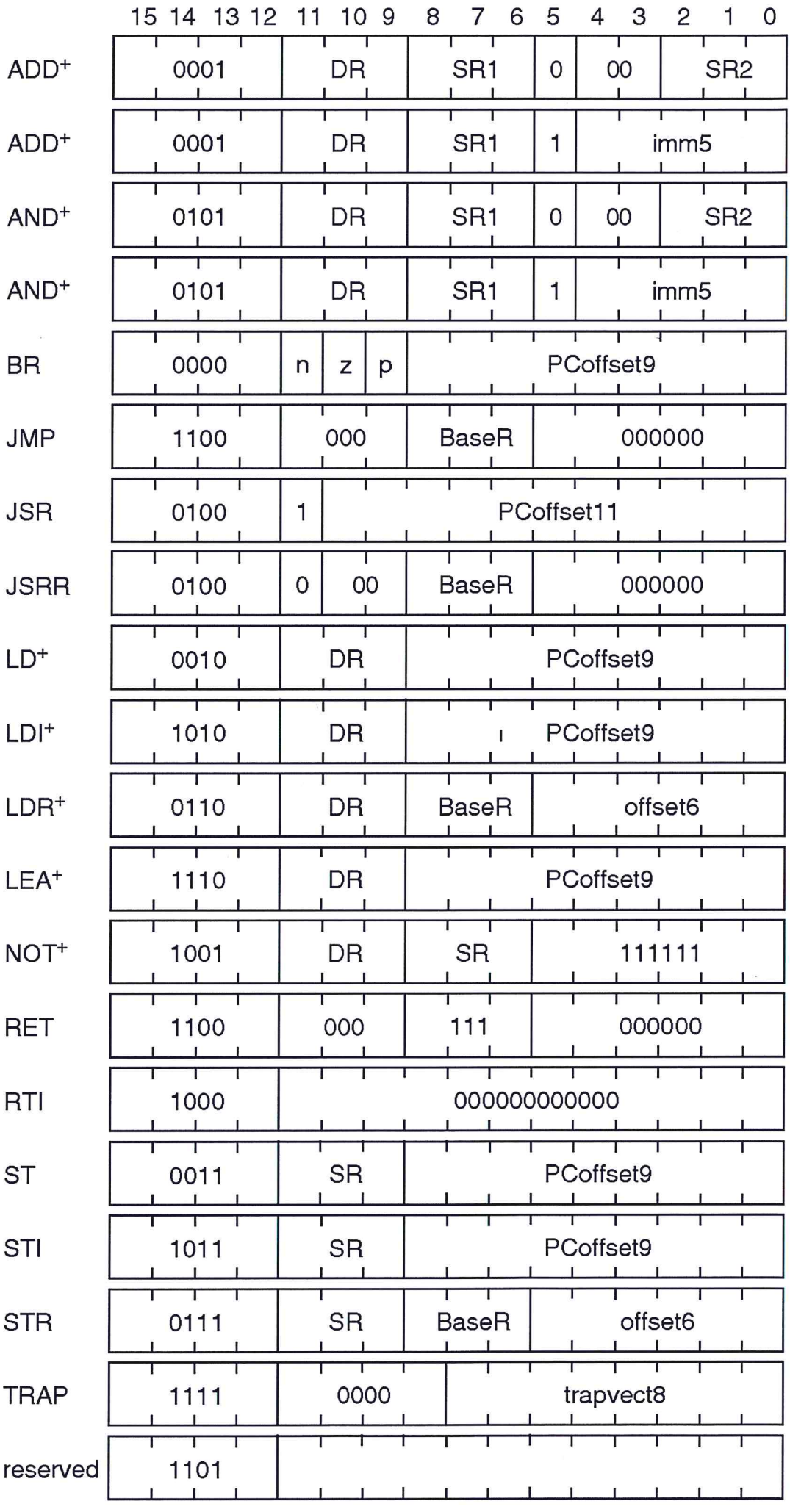


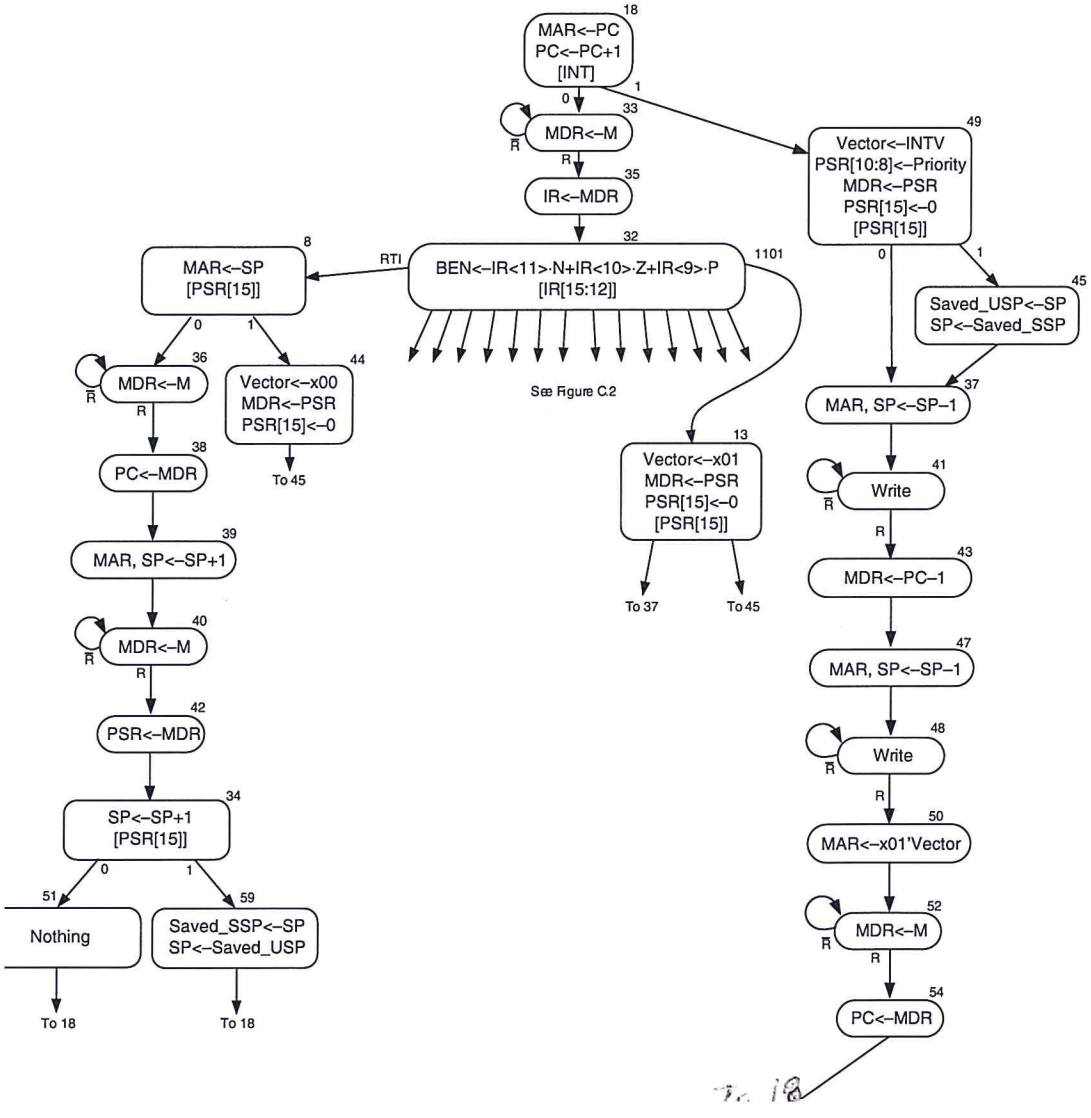
What about User Code?

Service routines provide three main functions:

1. **Shield programmers from system-specific details.**
2. **Write frequently-used code just once.**
3. **Protect system resources from malicious/clumsy programmers.**

Are there any reasons to provide the same functions for non-system (user) code?


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ADD+ 0001 DR SR1 0 00 SR2
ADD+ 0001 DR SR1 1 imm5
AND+ 0101 DR SR1 0 00 SR2
AND+ 0101 DR SR1 1 imm5
BR 0000 n z p PCoffset9
JMP 1100 000 BaseR 000000
JSR 0100 1 PCoffset11
JSRR 0100 0 00 BaseR 000000
LD+ 0010 DR PCoffset9
LDI+ 1010 DR PCoffset9
LDR+ 0110 DR BaseR offset6
LEA+ 1110 DR PCoffset9
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
reserved 1101


18
MAR<–PC
PC<–PC+1
[INT]
0
1
33
MDR<–M
R
R
35
IR<–MDR
32
BEN<–IR<11>·N+IR<10>·Z+IR<9>·P
[IR[15:12]]
1101
RTI
8
MAR<–SP
[PSR[15]]
0
1
36
MDR<–M
R
R
38
PC<–MDR
39
MAR, SP<–SP+1
40
MDR<–M
R
R
42
PSR<–MDR
34
SP<–SP+1
[PSR[15]]
0
1
51
Nothing
59
Saved_SSP<–SP
SP<–Saved_USP
To 18
To 18
44
Vector<–x00
MDR<–PSR
PSR[15]<–0
To 45
See Figure C.2
13
Vector<–x01
MDR<–PSR
PSR[15]<–0
[PSR[15]]
To 37
To 45
49
Vector<–INTV
PSR[10:8]<–Priority
MDR<–PSR
PSR[15]<–0
[PSR[15]]
0
1
45
Saved_USP<–SP
SP<–Saved_SSP
37
MAR, SP<–SP–1
41
Write
R
R
43
MDR<–PC–1
47
MAR, SP<–SP–1
48
Write
R
R
50
MAR<–x01'Vector
52
MDR<–M
R
R
54
PC<–MDR

To 18