Chapter 9
TRAP Routines and Subroutines

System Calls

Some ops. require specialized knowledge and protection

- **Abstract I/O device registers and how to use them**
  Programmers don't want to know this!
- **Protection for shared I/O resources - isolate programs from OS**
- **Reuse of common code**

Solution: *service routines* or *system calls*

- **Low-level, privileged operations performed by operating system**

1. **User program invokes system call**
2. **Operating system code:**
   - **Saves registers**
   - **Performs operation**
   - **Restores registers**
3. **Returns control to user program**

LC-3 TRAP Mechanism

Provides set of service routines

- **Part of operating system -- routines start at arbitrary addresses**
  (by convention system code is x0200 through x2FFF)
- **Up to 256 routines**

Requires table of starting addresses

- **Stored in memory (x0000 through x00FF)**
- **Used to associate code with trap number**
- **Called System Control Block or Trap Vector Table**

Uses TRAP instruction

- **Used by program to transfer control to operating system (w/ privileges)**
- **8-bit trap vector names one of the 256 service routines**

Uses "RTT" instruction

- **Returns control to the user program (w/o privileges)**
- **Execution resumes immediately after the TRAP instruction**

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

- **Identifies which system call to invoke**
- **Serves as index into table of service routine addresses**
  - **LC-3: table stored in memory at 0x0000 – 0x00FF**
  - **8-bit trap vector zero-extended to form 16-bit address**
- **Enters privileged mode**

Where to go

- **Lookup starting address from table; place in PC**

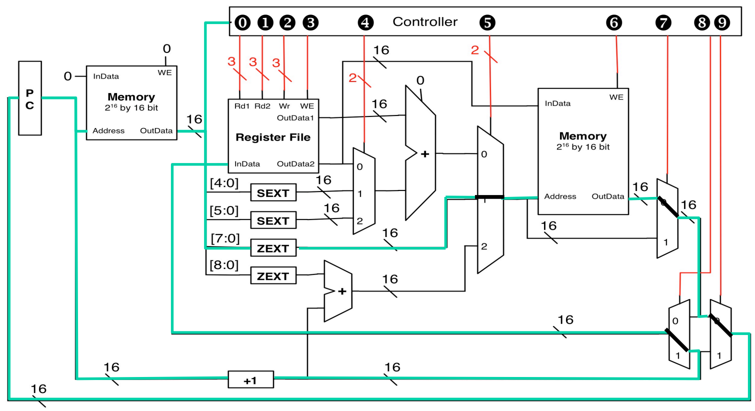
Enabling return

- **Save address of next instruction (current PC) in R7**

How to return

- **Place address in R7 in PC**

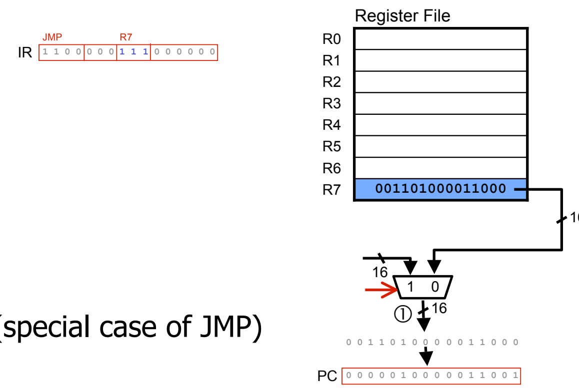
TRAP



	Opcode		Control									
Instr	I[15:12]	I[5]	⓿	❶	❷	❸	❹	❺	❻	❼	❽	❾
TRAP	1111	-	-	-	7	1	xx	01	0	0	1	0

RET

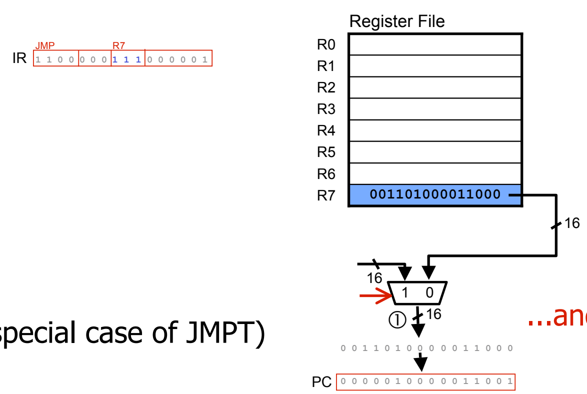
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RET	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0



(special case of JMP)

RTT

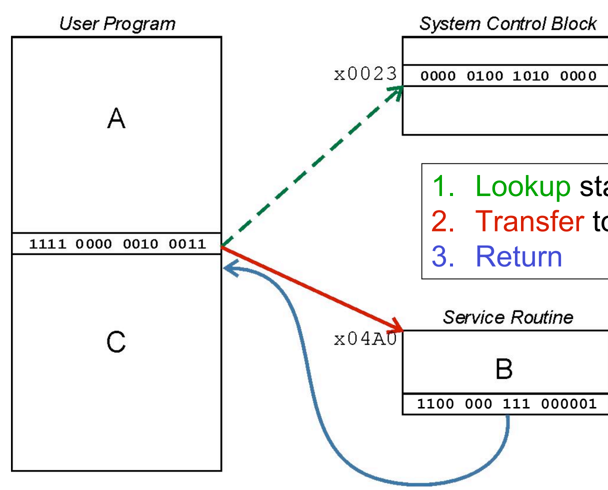
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTT	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1



(special case of JMPT)

...and exit priv. mode

TRAP Mechanism Operation



1. Lookup starting address
2. Transfer to service routine
3. Return

TRAP Routine Template (From HW6)

```
DRAW_BLOCK:
    ; Register Saving
    ST R0, DB_R0
    ST R1, DB_R1
    ...
    ST R6, DB_R6
    ST R7, DB_R7  ; return address

    ; *** Code ***

    ; Register Restoring
    LD R0, DB_R0
    LD R1, DB_R1
    ....
    LD R6, DB_R6
    LD R7, DB_R7  ; return address
    RTT
```

```
; Register Saves
DB_R0: .FILL x0
DB_R1: .FILL x0
DB_R2: .FILL x0
DB_R3: .FILL x0
DB_R4: .FILL x0
DB_R5: .FILL x0
DB_R6: .FILL x0
DB_R7: .FILL x0
```

TRAP routine interface:

- Reads input registers
- Writes output registers
- Value in R7 is destroyed
- All other registers preserved
- Condition codes not preserved

TRAP x40

Example: Character Output Service Routine (OUT)

```
                .ORIG x0430       ; Syscall x21 address
Out:            ST    R1, SaveR1  ; Save R1

; Write character
TryWrite:       LDI   R1, DSR     ; Get status
                BRzp  TryWrite    ; Bit 15 says not ready?
WriteIt:        STI   R0, DDR     ; Write char
; Return from TRAP
Return:         LD    R1, SaveR1  ; Restore R1
                RTT               ; Return from trap

DSR             .FILL xFE04
DDR             .FILL xFE06
SaveR1          .FILL 0
                .END
```

stored in table, location x21

Caution Using TRAPs

```
        LEA   R3, Block     ; Init. to first loc.
        LD    R6, ASCII     ; Char->digit template
        LD    R7, COUNT     ; Init. 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_TASK
ASCII   .FILL xFFD0         ; Negative of x0030
COUNT   .FILL #10
Block   .BLKW #10
```

What's wrong with this code?

Saving and Restoring Registers

Called routine ⇒ *"callee-save"*

- **Before start, save registers that will be altered** (except output regs)
- **Before return, restore those same registers** (again, except output regs)
- **Values are saved by storing them in memory**

Calling routine ⇒ *"caller-save"*

- **If register value needed later, save register destroyed by own instructions or by called routines (if known)**
    - **Save R7 before TRAP**
- **Or avoid using those registers altogether**

LC-3: By convention, callee-saved when possible

- **Other ISAs use a more efficient combination of caller- and callee-save**

Privilege

Goal: Isolation

- OS performs I/O (in traps)
- Application can't perform I/O directly

How is this enforced?

Privilege: Processor modes

- Privileged (supervisor)
- Unprivileged (user)
- Encoded in 15th bit of processor status register (PSR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P													N	Z	P

Supervisor Mode Versus User Mode

Supervisor mode

- Program has access to resources not available to user programs
- LC-3: memory (including memory-mapped I/O devices)

User mode in LC-3

- Memory access is limited by memory protection register (MPR)
- Each MPR bit corresponds to 4K memory segment
- 1 indicates that users can access memory in this segment

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

Cannot access x0 to x2FFF

Can access x3000 to xBFFF

Cannot access >= xC000

MPR

Note: MPR not in book!

Set (only) by OS

- OS decides policy, HW enforces it

Prevents user from. . .

- Updating trap table
- Changing OS code (*i.e.,* trap handlers)
- Accessing video memory
- Accessing memory-mapped I/O registers (*e.g.,* DDR, DSR)
- Could be different for each application

Managing Privilege

What sets privilege bit in PSR?

- TRAP instruction

What clears privilege bit?

- JMPT/RTT (Note: not in book!)

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

Subroutines

A subroutine is a program fragment that. . .

- Resides in user space (*i.e,* not in OS)
- Performs a well-defined task
- Is invoked (called) by a user program
- Returns control to the calling program when finished

Like a TRAP routine, but not part of the OS

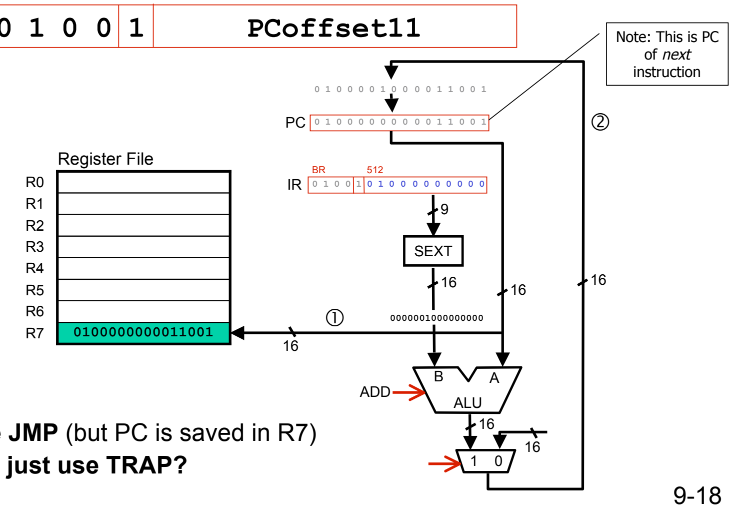
- Not concerned with protecting hardware resources
- No special privilege required

Virtues

- Reuse code without re-typing it (and debugging it!)
- Divide task into parts (or among multiple programmers)
- Use vendor-supplied *library* of useful routines

JSR

	15	14	13	12	11	10–0
JSR	0	1	0	0	1	PCoffset11

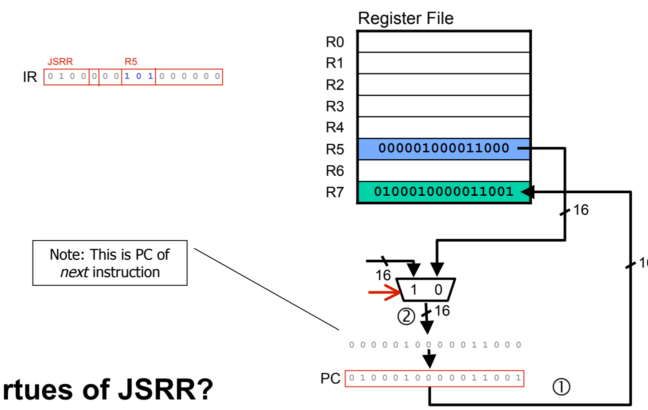


Just like JMP (but PC is saved in R7)
Why not just use TRAP?

JSRR

this zero means "register mode"

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



Virtues of JSRR?

Subroutine Template

```
SUB_NAME:
    ; Register Saving
    ST R0, SUB_R0
    ST R1, SUB_R1
    …
    ST R6, SUB_R6
    ST R7, SUB_R7  ; return address

    ; *** Code ***

    ; Register Restoring
    LD R0, SUB_R0
    LD R1, SUB_R1
    ….
    LD R6, SUB_R6
    LD R7, SUB_R7  ; return address
    RET
```

```
; Register Saves
SUB_R0: .FILL x0
SUB_R1: .FILL x0
SUB_R2: .FILL x0
SUB_R3: .FILL x0
SUB_R4: .FILL x0
SUB_R5: .FILL x0
SUB_R6: .FILL x0
SUB_R7: .FILL x0
```

Subroutine interface:

- Reads input registers
- Writes output registers
- Value in R7 is destroyed
- All other registers preserved
- Condition codes not preserved

JSR **SUB_NAME**

Note: we'll add support for recursion later

Example: Negate the value in R0

```
TwosComp  NOT  R0, R0      ; flip bits
          ADD  R0, R0, #1  ; add one
          RET              ; return to caller
```

To call from a program

```
; need to compute R4 = R1 - R3
          ADD  R0, R3, #0  ; copy R3 to R0
          JSR  TwosComp    ; negate
          ADD  R4, R1, R0  ; add to R1
          ...
```

Note: TwosComp overwrites R0

Using Subroutines

Programmer must know

- **Address:** or at least a label that will be bound to its address
- **Function:** what it does
  - NOTE: The programmer does not need to know ***how*** the subroutine works, but what changes are visible in the machine's state after the routine has run
- **Arguments:** what they are and where they are placed
- **Return values:** what they are and where they are placed

Passing Information To Subroutines

Argument(s)

- **Value passed in to a subroutine is called an argument**
- **This is a value needed by the subroutine to do its job**
- **Examples**
  - **TwosComp: R0 is number to be negated**
  - **OUT: R0 is character to be printed**
  - **PUTS: R0 is *address* of string to be printed**

How?

- **In registers (simple, fast, but limited number)**
- **In memory (many, but awkward, expensive)**
- **Both**

Getting Values From Subroutines

Return Values

- **A value passed out of a subroutine is called a return value**
- **This is the value that you called the subroutine to compute**
- **Examples**
  - **TwosComp: negated value is returned in R0**
  - **GETC: character read from the keyboard is returned in R0**

How?

- **Registers, memory, or both**
- **Single return value in register most common**

Saving and Restore Registers

Like service routines, must save and restore registers

- Who saves what is part of the calling convention

Generally use "callee-save" strategy, except for ret vals

- Same as trap service routines
- Save anything that subroutine alters internally that shouldn't be visible when the subroutine returns
- Restore incoming arguments to original values (unless overwritten by return value)

Remember

- You MUST save R7 if you call any other subroutine or trap
- Otherwise, you won't be able to return!

Local Variables

Goal: keep values in register (simple and efficient)

More variables than register?

- Keep values in memory (load from memory to compute on them)

Example

```
      .ORIG x3000
Foo:  . . .
      LD    R3, Val1
      ADD   R3, R3, #1
      ST    R3, Val1
      . . .
Val1: .FILL #0
      . . .
      .END
```

What prevents another subroutine from accessing your local variables?

Global Variables

Just like local variables (labeled memory)

Problem: LD only supports 9-bit offsets (-256 to 255)

Solution: Keep *references* near subroutine, use indirect addressing

Example:

```
         .ORIG x3000
Foo:     . . .
         LDI   R3, Val1Ref
         ADD   R3, R3, #1
         STI   R3, Val1Ref
         . . .
Val1Ref:       .FILL Val1
         . . .
Val1:    .FILL #0
         . . .
         .END
```

Note: All labels must be unique!

Note: Can be more than one reference to single datum

Alternative: reserve register to always point to start of "globals"

Example

(1) Write a subroutine FirstChar to. . .

Find <u>first</u> occurrence of particular character (in R0) in a string (pointed to by R1); return pointer to character or to end of string (NULL) in R5

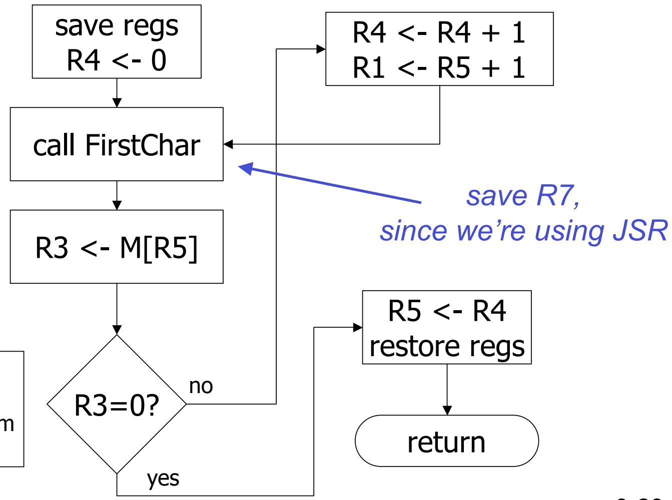
(2) Use FirstChar to write CountChar, which. . .

Counts <u>number</u> of occurrences of particular character (in R0) in a string (pointed to by R1); return count in R5

Strategy

- **Write second subroutine first, without knowing the implementation of FirstChar!**

CountChar Algorithm (using FirstChar)



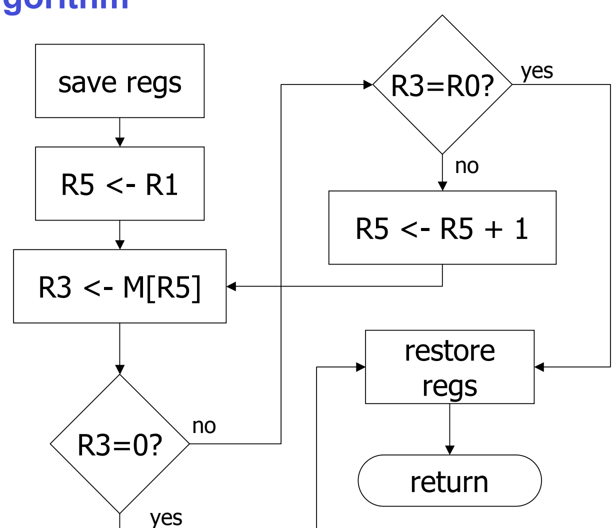
R0 Character
R1 Str. Ptr.
R3 Char in mem
R4 Count

CountChar Implementation

R0 Character
R1 Str. Ptr.
R3 Char in mem
R4 Count

```
; CountChar: subroutine to count occurrences of a char
CountChar:
        ST     R1, CCR1      ; save regs
        ST     R3, CCR3
        ST     R4, CCR4
        ST     R7, CCR7      ; JSR alters R7
        AND    R4, R4, #0    ; initialize count to zero
CC1:    JSR    FirstChar     ; find next occurrence (ptr in R1)
        LDR    R3, R5, #0    ; see if char or null
        BRz    CC2           ; if null, no more chars
        ADD    R4, R4, #1    ; increment count
        ADD    R1, R5, #1    ; point to next char in string
        BRnzp  CC1
CC2:    ADD    R5, R4, #0    ; move return val (count) to R5
        LD     R1, CCR1      ; restore regs
        LD     R3, CCR3
        LD     R4, CCR4
        LD     R7, CCR7
        RET                  ; and return
```


FirstChar Algorithm



R0 Character
R1 Str. Ptr.
R3 Char in mem
R4 -Character
R5 Str. Prt.

FirstChar Implementation

R0 Character
R1 Str. Ptr.
R3 Char in mem
R4 -Character
R5 Str. Prt.

```
; FirstChar: subroutine to find first occurrence of a char
FirstChar:
        ST     R3, FCR3      ; save registers
        ST     R4, FCR4      ; save original char
        NOT    R4, R0        ; negate R0 for comparisons
        ADD    R4, R4, #1
        ADD    R5, R1, #0    ; initialize ptr to beginning of string
FC1:    LDR    R3, R5, #0    ; read character
        BRz    FC2           ; if null, we're done
        ADD    R3, R3, R4    ; see if matches input char
        BRz    FC2           ; if yes, we're done
        ADD    R5, R5, #1    ; increment pointer
        BRnzp  FC1
FC2:    LD     R3, FCR3      ; restore registers
        LD     R4, FCR4      ;
        RET                  ; and return
```

What if we used CCR3?

Library Routines

Call subroutines in other object files (or library)

- Assembler/linker must support EXTERNAL symbols
- Extra "linking" step will fill in value of SQAddr

```
            . . .
            .EXTERNAL SQRT
            . . .
            LD   R2, SQAddr    ; load SQRT addr
            JSRR R2
            . . .
SQAddr      .FILL SQRT
```

Using JSRR, because SQRT likely not "nearby"

Problems?

What's the problem with. . . recursion?

```
Main: . . .
      JSR   Foo
Next: . . .


Foo:  ST    R7, SaveR7
      AND   R0, R0, #0
      . . .
      JSR   Foo
After:. . .
      LD    R7, SaveR7
      Ret
SaveR7:.FILL #0
```

- First call to Foo (SaveR7 contains address of Next)
- Second call to Foo (SaveR7 contains address of After)
- First return from Foo (returns to After)
- Second return from Foo (returns to After again!!!)

Recursion

Need

- Per-subroutine-invocation data space (*activation record*)

Approach

- Allocate new *activation record* for each call
- Subroutine uses its own activation record to hold invocation-specific data (*e.g.,* local variables, saved registers)
- Organized like a stack (named "the call stack")

Note

- As SnakeOS/Snake is not recursive, we won't need to do this for HW 6 and 7!