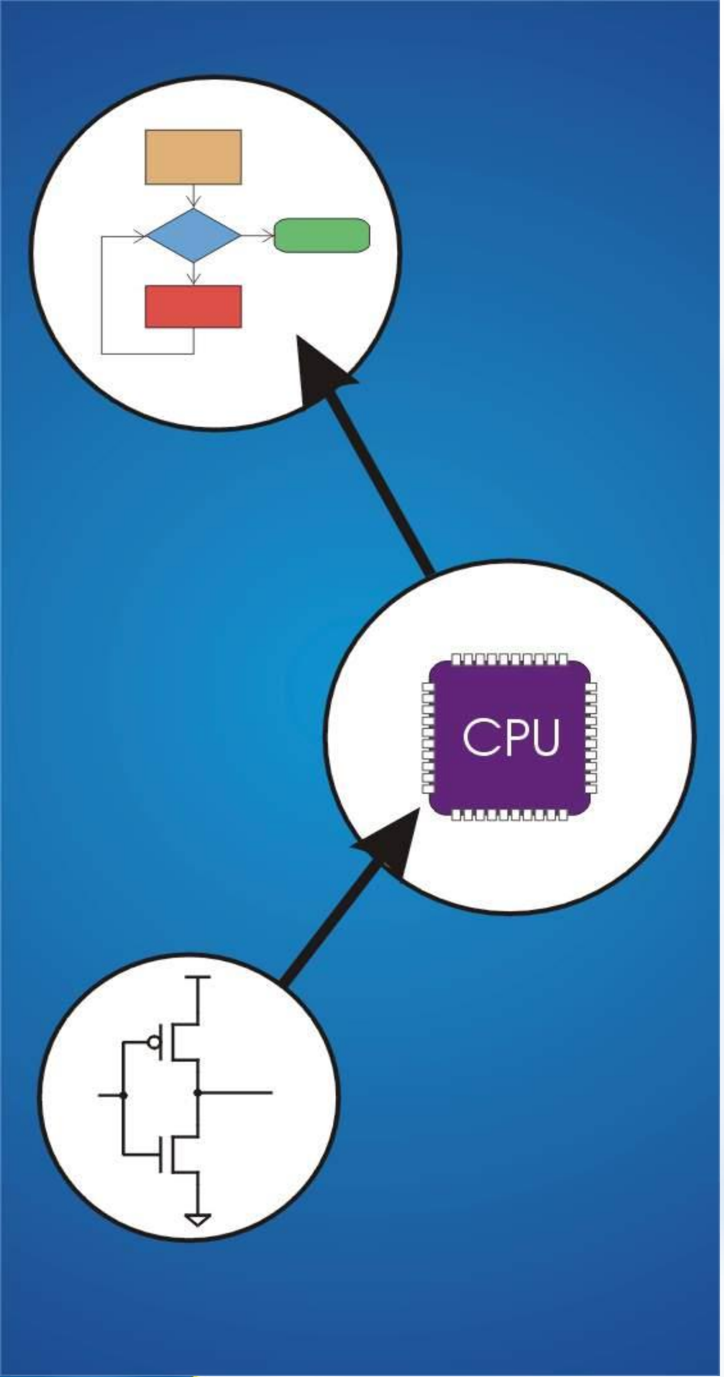
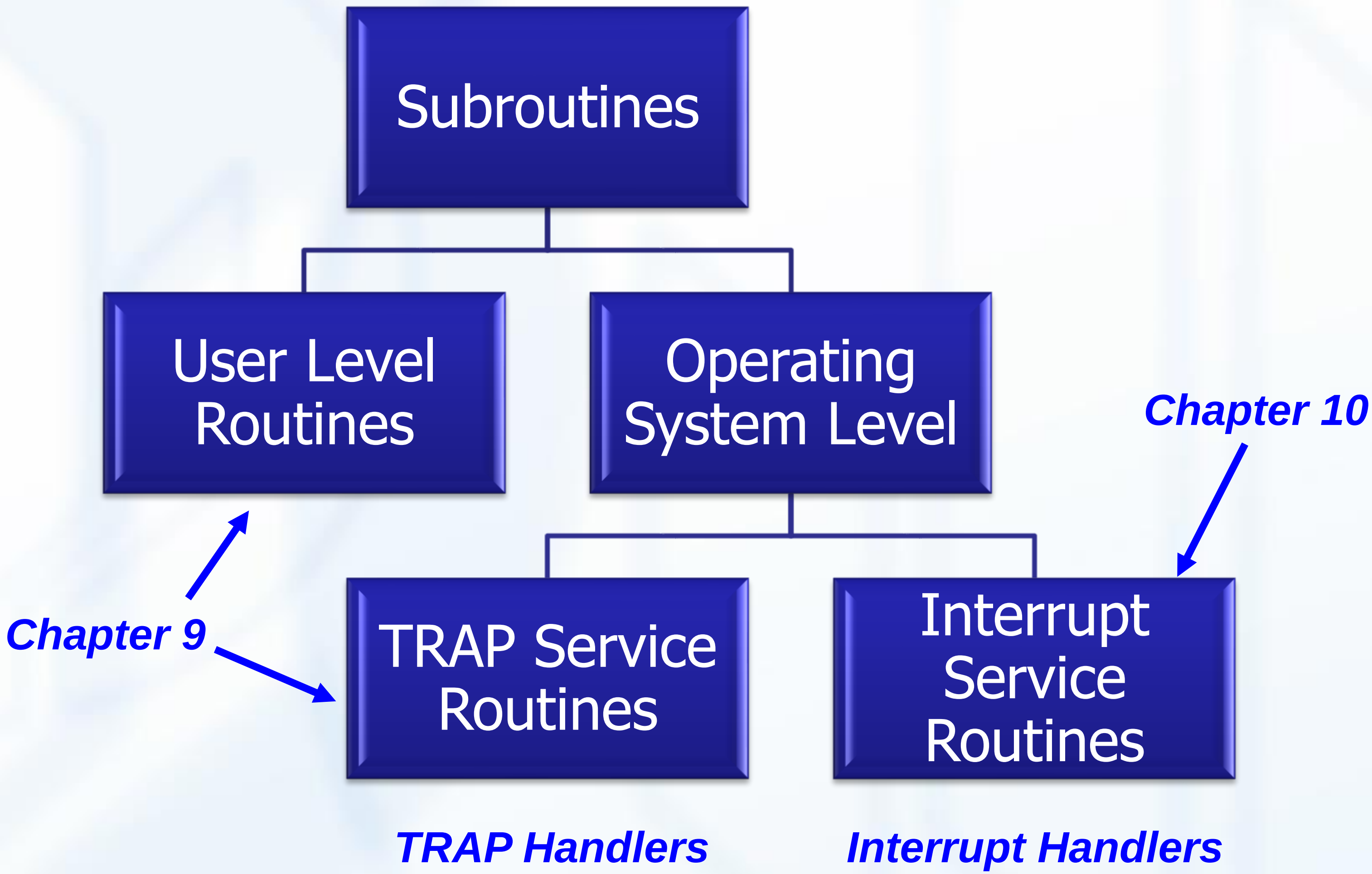


Chapter 9 TRAP Routines and Subroutines

ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

Subroutines, TRAPs, and Interrupts



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

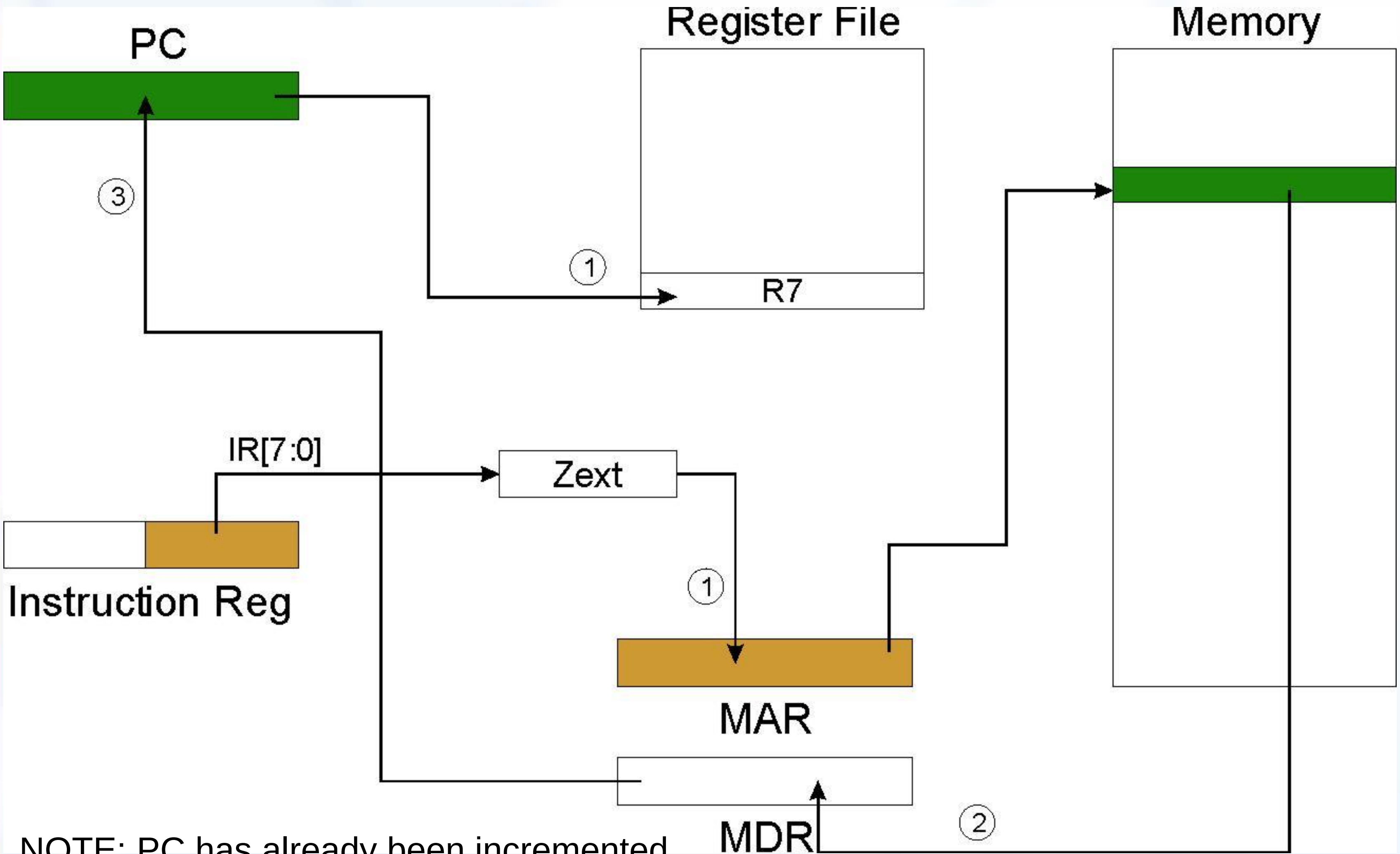
1. **A set of service routines.**
   - part of operating system -- routines start at arbitrary addresses
     - convention is that system code is below x3000
   - up to 256 routines
2. **Trap Vector Table (table of service routine addresses)**
   - stored at x0000 through x00FF in memory
   - called 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

	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
  - 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.

TRAP: Invoke a system routine

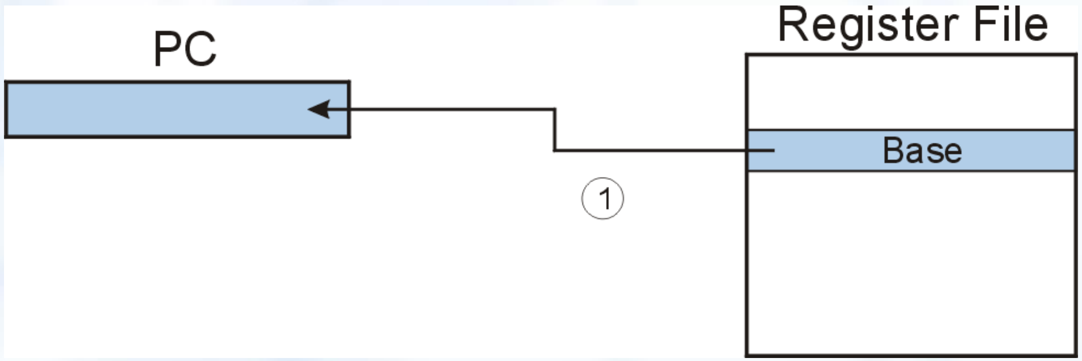
- **Assembler Instruction:**
  - TRAP trapvec
- **Encoding:**
  - 1111 0000 trapvect8
- **Example:**
  - TRAP x23
- **Note:**
  - R7 <= (PC) (for eventual return)
  - PC <= Mem[Zext(trapvect8)] (Zext means 0 extent)

Start here. Go anywhere.

Remember JMP (Register) Instruction?

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

- **Jump is an unconditional branch (always taken)**
  - Target address is the contents of a register.
  - Allows any target address.



RET Instruction (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

- **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.**

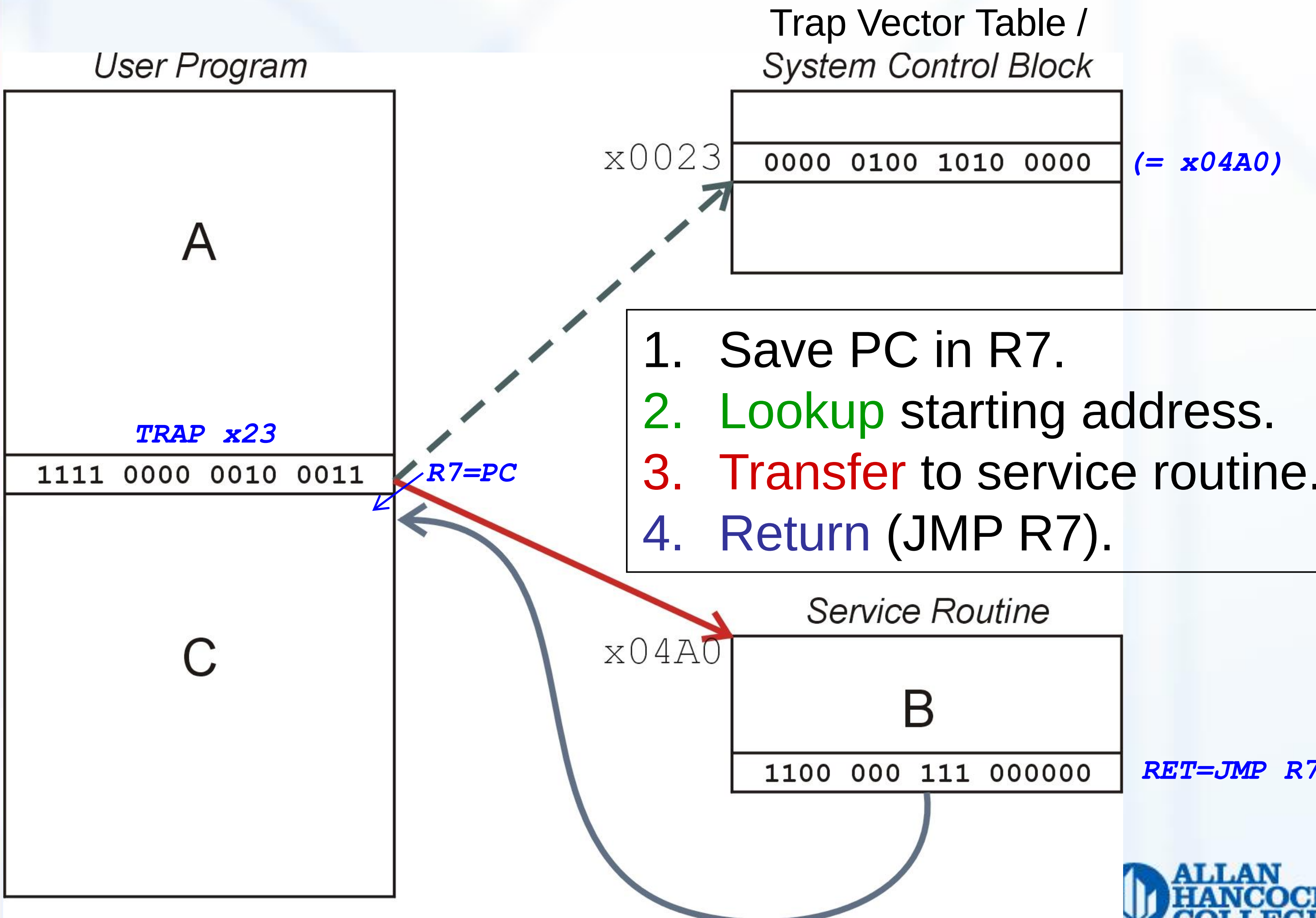
ALLAN HANCOCK COLLEGE Start here. Go anywhere.

RET: Return from a subroutine

- **Assembler Instruction:**
  - RET
- **Encoding:**
  - 1100 000 111 000000
- **Example:**
  - RET ; PC <- R7
- **Note:**
  - Return from a previous TRAP / JSR instruction

ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

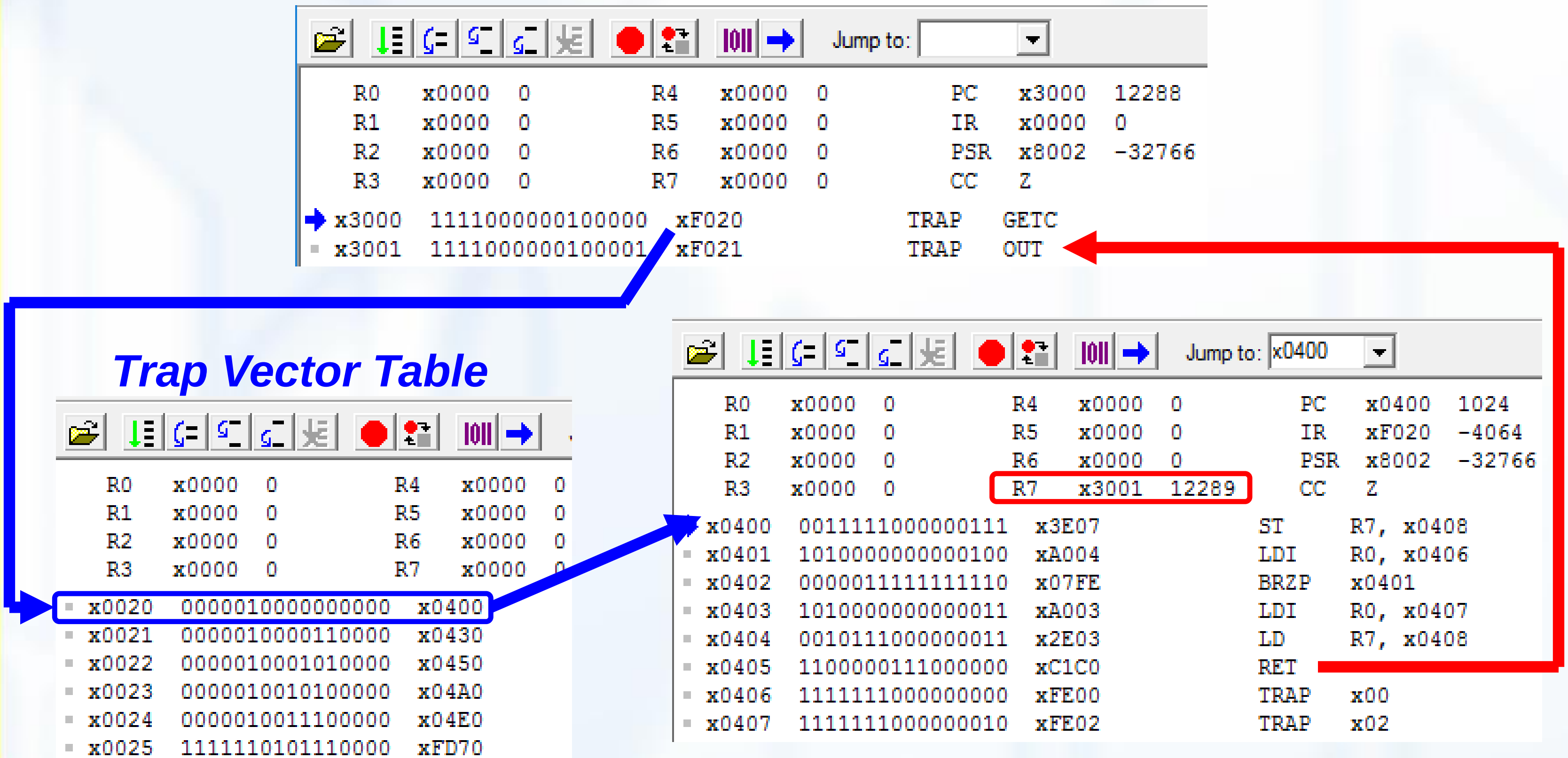
TRAP Mechanism Operation



1. Save PC in R7.
2. Lookup starting address.
3. Transfer to service routine.
4. Return (JMP R7).

ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

TRAP x20: Input a Character



Location	*I/O Register*	*Function*
xFE00	Keyboard Status Register (KBSR)	Bit [15] is one when keyboard has received a new character.
xFE02	Keyboard Data Register (KBDR)	Bits [7:0] contain the last character typed on keyboard.

TRAP x26: Get and Echo a Character

```
; File:  trapx26.asm
; Description:   Get Character and Echo TRAP
;
; Get character input console.
; Display character on console.
; Return character in R0.
;
         .ORIG x600        ; must match TRAPX26 in trapx26install.asm

         ST R7,X26_R7
         GETC
         OUT
         LD R7,X26_R7
         RET

X26_R7   .BLKW 1           ; Saved R7
         .END
```

TRAP x26 Example

Install TRAP x26 in Trap Vector Table

```
; File:  trapx26install.asm
; Description:   Install TRAP x26
;
         .ORIG x26
TRAPX26  .FILL x600        ; TRAP x26 starts at x600
         .END
```

Program that executes TRAP x26

```
; File:  trapx26echo.asm
; Description:   Get Character and Echo using TRAP x26
;
         .ORIG x3000
         TRAP x26
         HALT
         .END
```

TRAP Routines and Assembler Symbols

See Table A.2

Symbol	*TRAP Vector*	*Description*
GETC	x20	Read one character from keyboard. Character stored in R0[7:0], upper 8 bits are cleared.
OUT	x21	Write one character (in R0[7:0]) to console.
PUTS	x22	Write null-terminated string to console. Address of string is in R0.
IN	x23	Print "`Input a character>` " on console, then read (and echo) one character from keyboard. Address of string is in R0; character stored in R0[7:0], upper 8 bits are cleared.
PUTSP	x24	"Packed" PUTS. Same as PUTS except two ASCII characters per word: bits [7:0] of memory location is written first, then bits [15:8]. Termination value is x0000 or x00 in bits [15:8] if bits [7:0] contain a value.
HALT	x25	Halt execution and print message to console.

ALLAN HANCOCK COLLEGE Start here. Go anywhere.

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         ; xFFD0 = -x30
COUNT           .FILL  #10
Binary          .BLKW  #10
```

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

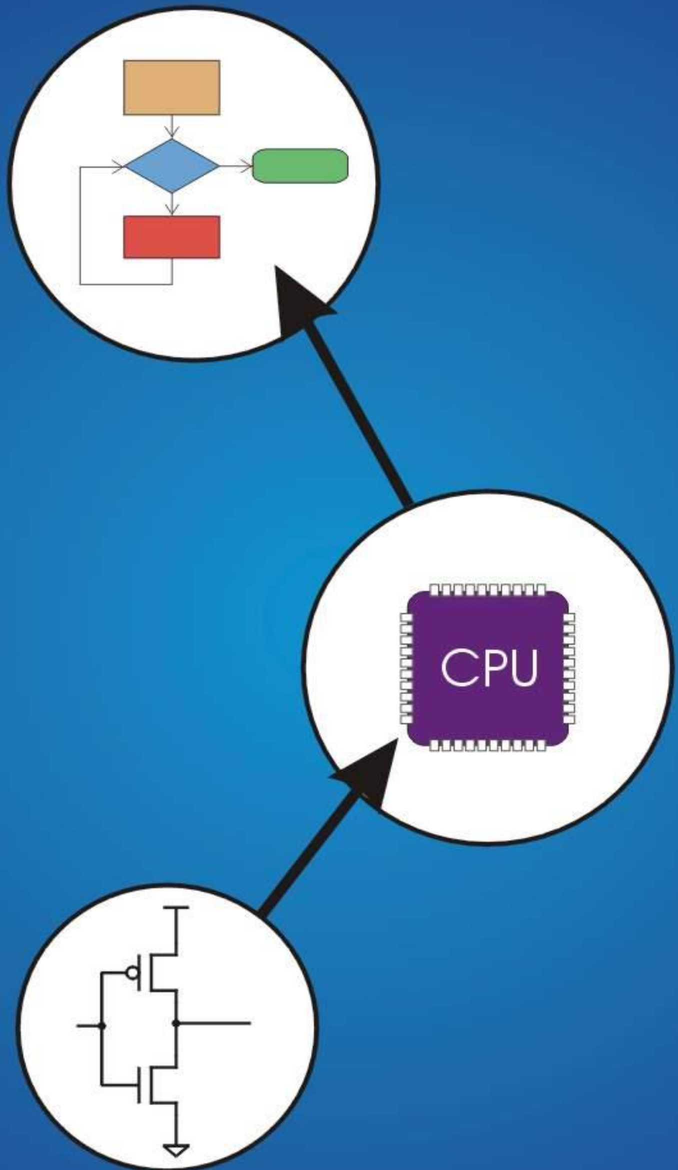
Saving and Restoring Registers

- **Called routine -- "callee-save"**
  - *"Called routine saves registers"*
  - 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"**
  - *"You save registers prior to calling routine"*
  - 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?**

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



Subroutines

ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

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?**

ALLAN HANCOCK COLLEGE Start here. Go anywhere.

Subroutines

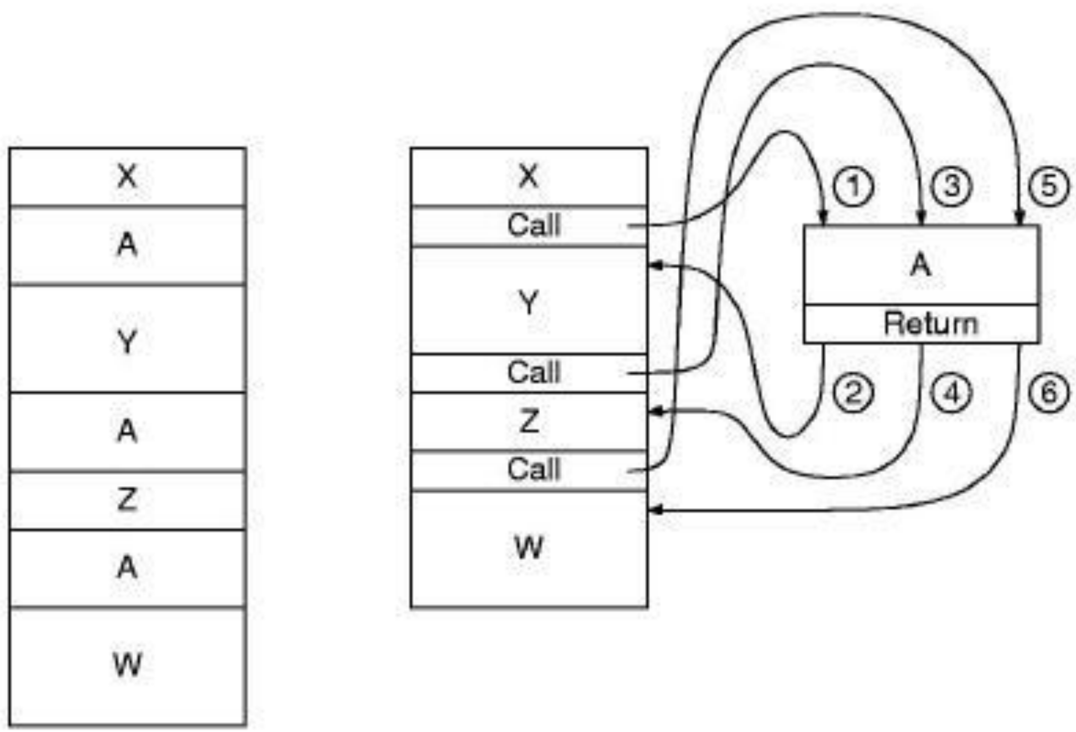
- **A subroutine is a program fragment that:**
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by another user program
  - returns control to the calling program when finished
- **Like a service routine, but not part of the OS**
  - not concerned with protecting hardware resources
  - no special privilege required

Subroutines (continued)

- **Used for**
  - Frequently executed code segments
    - Reuse useful (and debugged!) code without having to keep typing it in
  - Library routines
  - Team-developed systems
    - in other words, all the same reasons for using subroutines in higher level languages, where they may be called functions, procedures, methods, etc.
- **Requirements:**
  - Pass parameters and return values, via registers or memory.
  - Call from any point & return control to the same point.

The Call / Return mechanism

- **The figure illustrates the execution of a program comprising code fragments A, W, X, Y and Z.**
  - Note that fragment A is repeated several times, and so is well suited for packaging as a subroutine:



(a) Without subroutines (b) With subroutines

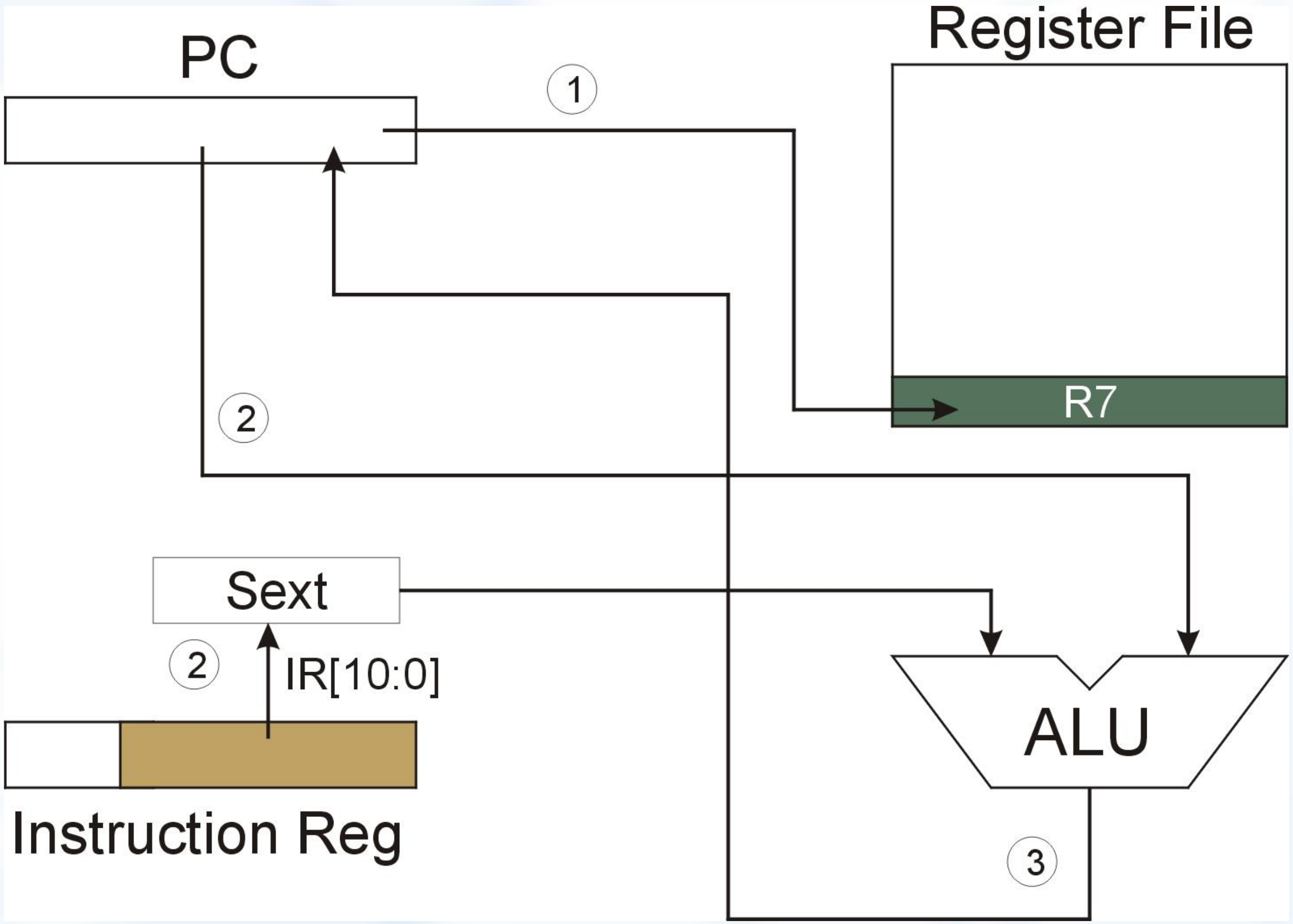
JSR Instruction

this one means "PC-relative mode"

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

- **Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.**
  - saving the return address is called "linking"
  - target address is PC-relative (PC + Sext(IR[10:0]))
  - bit 11 specifies addressing mode
    - if =1, PC-relative: target address = PC + Sext(IR[10:0])
    - if =0, register: target address = contents of register IR[8:6]
      - See JSRR Instruction

JSR



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

JSR: Jump to Subroutine

- **Assembler Instruction:**
  - JSR LABEL
- **Encoding:**
  - 0100 1 PCoffset11
- **Example:**
  - JSR GOSUB
- **Note:**
  - R7 <= (PC) (for eventual return)
  - PC <= Mem[Sext(PCoffset11)]

JSRR Instruction

this zero means "base mode"

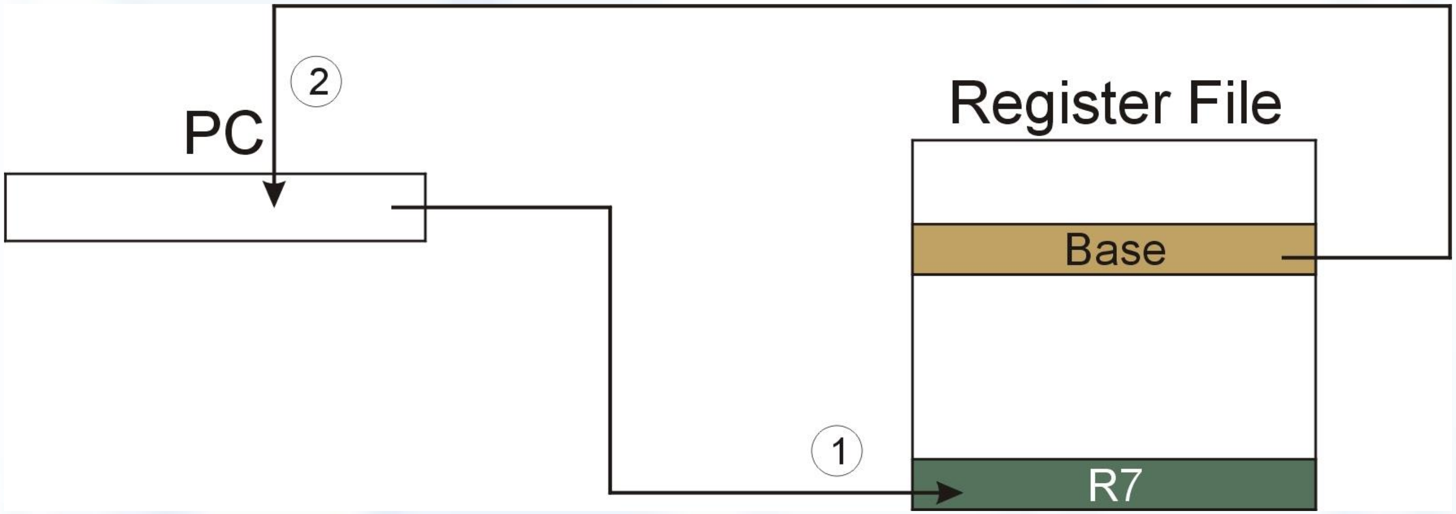
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSRR	0	1	0	0	0	0	0		BaseR		0	0	0	0	0	0

- **Just like JSR, except Base addressing mode.**
  - target address is Base Register
  - bit 11 specifies addressing mode
    - if =1, PC-relative: target address = PC + Sext(IR[10:0])
      - See JSR instruction
    - if =0, register: target address = contents of register IR[8:6]
  - offset is 0

- **What important feature does JSRR provide that JSR does not?**

ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

JSRR



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

JSR: Jump to Subroutine

- **Assembler Instruction:**
  - JSRR BaseR
- **Encoding:**
  - 0100 0 00 BaseR 000000
- **Example:**
  - JSRR R3 ; PC <= R3
- **Note:**
  - R7 <= (PC) (for eventual return)

Returning from a Subroutine

- **RET (JMP R7) gets us back to the calling routine.**
  - just like TRAP

Example: Negate the value in R0

```
; Program: need to compute R3 = R1 - R2
; Note: Caller should save R0 if needed later!
      ...
      ADD   R0, R2, #0   ; copy R2 to R0
      JSR   TwosComp     ; take 2's comp
      ADD   R3, R1, R0   ; add to R1
      ...
; Two's Complement Subroutine
TwosComp
      NOT   R0, R0       ; flip bits
      ADD   R0, R0, #1   ; add one
      RET
```

```
; File:         twoscomp.asm
; Description:  Example of Two's Complement Subroutine (using both JSR and JSRR)
;
; Need to compute R3 = R1 - R2 and then R5 = R3 - R4
; Note: Caller should save R0 if we'll need it later!
            .ORIG x3000
; Populate registers with test values
            LD          R1, R1value
            LD          R2, R2value
            LD          R4, R4value
; Perform R3 = R1 - R2 using JSR
            ADD         R0, R2, #0  ; copy R2 to R0
            JSR         TwosComp    ; take 2's comp
            ADD         R3, R1, R0  ; add to R1
            NOP                     ; just to provide some visual space when viewing in simulator

; Perform R5 = R3 - R4 using JSRR
            ADD         R0, R4, #0  ; copy R4 to R0
            LD          R6, TC_Adr  ; load TwosComp address in R6
            JSRR        R6          ; take 2's comp (TwosComp address is in R6)
            ADD         R5, R3, R0  ; add to R3
;           ...
            HALT
            NOP                     ; just to provide some visual space when viewing in simulator

; Pointer Table
TC_Adr      .FILL TwosComp          ; create memory pointer to TwosComp
            NOP                     ; just to provide some visual space when viewing in simulator

; Two's Complement Subroutine
;  Input:   R0, contains value to be converted
;  Output:  R0, contains 2's complement of input value
TwosComp
            NOT         R0, R0      ; flip bits
            ADD         R0, R0, #1  ; add one
            RET                     ; return to caller
            NOP                     ; just to provide some visual space when viewing in simulator

R1value     .FILL #9
R2value     .FILL #2
R4value     .FILL #4
            .END
```

Passing Information to/from Subroutines

- **Arguments**
  - A value passed in to a subroutine is called an argument.
  - This is a value needed by the subroutine to do its job.
  - Examples:
    - In TwosComp routine, R0 is the number to be negated
    - In OUT service routine, R0 is the character to be printed.
    - In PUTS routine, R0 is address of string to be printed.
- **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:
    - In TwosComp routine, negated value is returned in R0.
    - In GETC service routine, character read from the keyboard is returned in R0.

Using Subroutines

- **In order to use a subroutine, a programmer must know:**
  - its address (or at least a label that will be bound to its address)
  - its function (what does it do?)
    - 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.
  - its arguments (where to pass data in, if any)
  - its return values (where to get computed data, if any)

Saving and Restore Registers

- **Since subroutines are just like service routines, we also need to save and restore registers, if needed.**
- **Generally use "callee-save" strategy, except for return values.**
  - Save anything that the subroutine will alter internally that shouldn't be visible when the subroutine returns.
  - It's good practice to restore incoming arguments to their original values (unless overwritten by return value).
- **Remember: You MUST save R7 if you call any other subroutine or service routine (TRAP).**
  - Otherwise, you won't be able to return to caller.

Library Routines

- **Vendor may provide object files containing useful subroutines**
  - don't want to provide source code -- intellectual property
  - assembler/linker must support EXTERNAL symbols (or starting address of routine must be supplied to user)

```
   . . .
   .EXTERNAL   TwosComp
               . . .
               LD     R6, TC_Adr
               JSRR   R6
               . . .
  TC_Adr       .FILL  TwosComp
```

- **Using JSRR, because we don't know whether TwosComp is within 1024 instructions.**

ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

Linking Example

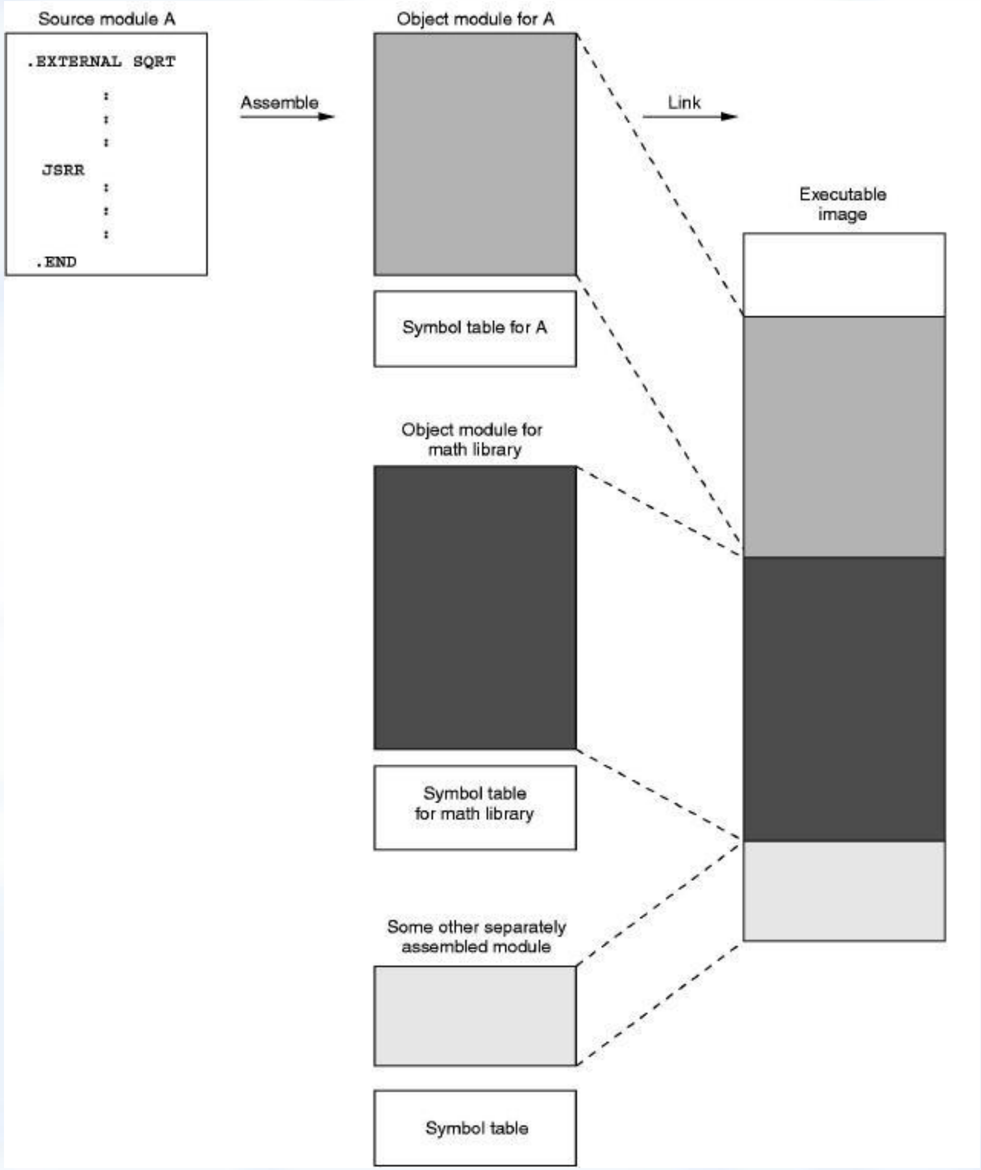
Before Linking

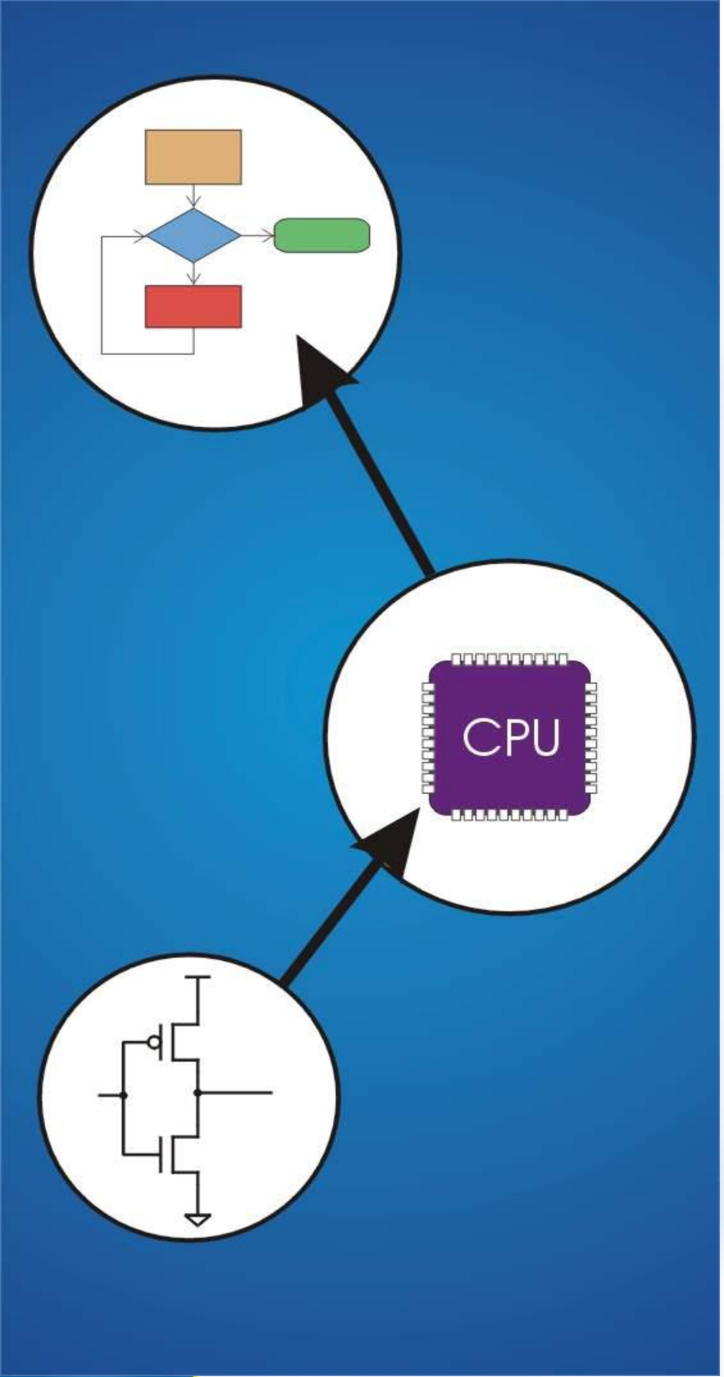
```
; Program loaded at x3000:
.EXTERNAL TwosComp
;                    ...
x3014                LD   R6, TC_Adr
x3015                JSRR R6
;                    ...
x3040 TC_Adr         .FILL TwosComp
                     .END
Symbol Table:
TC_Adr      x3040
TwosComp   x????


; Library Module loaded at x7000:
;                    ...
x7345 TwosComp       NOT  R0, R0
x7346                ADD  R0, R0, #1
x7347                RET
;                    ...
                     .END
Symbol Table:
TwosComp   x7345
```

After Linking

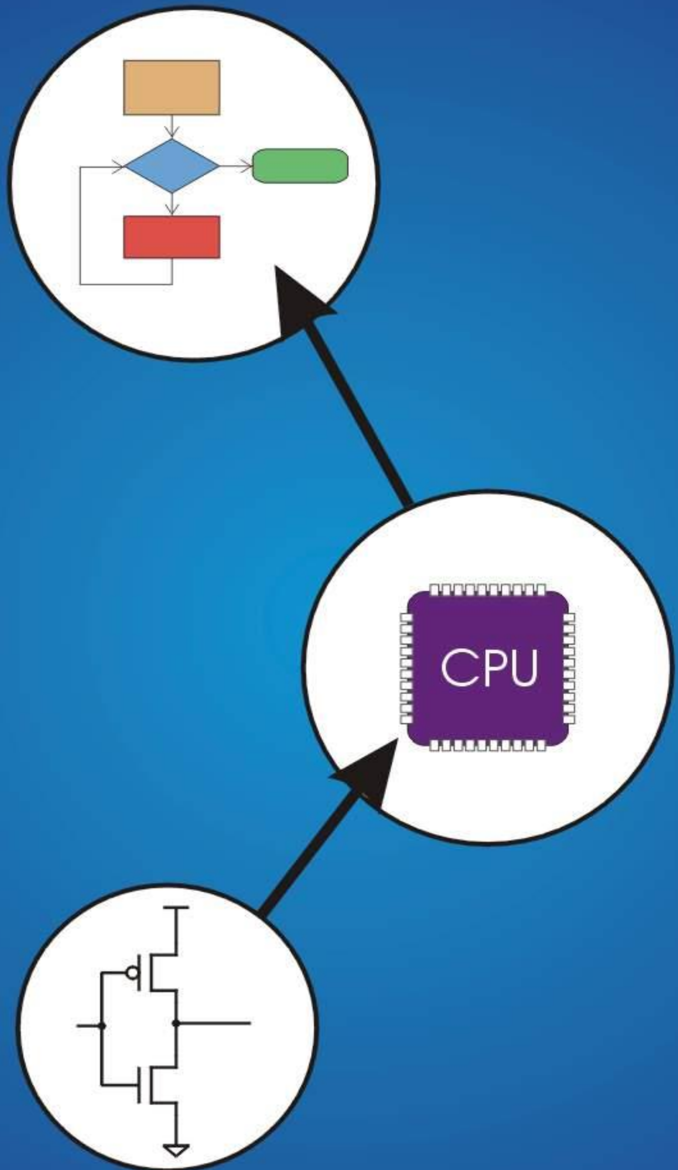
```
;                    ...
x3014                LD   R6, x25
x3015                JSRR R6
;                    ...
x3040 TC_Adr         .FILL x7345
;                    ...
x7345 TwosComp       NOT  R0, R0
x7346                ADD  R0, R0, #1
x7347                RET
;                    ...
```

Linking multiple files





End of Chapter 9 TRAP Routines and Subroutines



Chapter 9
TRAP Routines and Subroutines
Backup Slides

TRAP Example: Convert Character from Upper Case to Lower Case

```
; File:         trapex.asm
; Description: See book Example 9.1
;
; Converts any upper case character to lower case.
; Program prompts for input character, converts character to
; lower case, and repeats until the number 7 is entered.
;
         .ORIG x3000
         LD       R2, TERM    ; Load negative ASCII ‘7’ (7 is sentinel)
         LD       R3, ASCII  ; Load ASCII difference
AGAIN    TRAP      x23        ; input character
         ADD      R1, R2, R0  ; Test for termination character
         BRz      EXIT        ; Exit if done
         ADD      R0, R0, R3  ; Change to lowercase
         TRAP      x21        ; Output to monitor...
         BRnzp  AGAIN         ; ... again and again...
TERM     .FILL     xFFC9      ; xFFC9 is -‘7’ (7 is ASCII code x55)
ASCII    .FILL     x0020      ; lowercase bit
EXIT     TRAP      x25        ; halt
         .END
```


ALLAN HANCOCK COLLEGE Start here. Go anywhere.

TRAP Example: LC-3 Output Character TRAP Routine

```
; File:             trap21.asm
; Description:      LC-3 character output service routine (fig 9.5)
;
; Wait for Display Status Register to be ready for output.
; Write character to Display Data Register.
;
; Install service routine at address x21.
;
                    .ORIG x0430           ; system call starting address
                    ST        R7, SaveR7  ; save R7 & R1
                    ST        R1, SaveR1  ; R1 used for DSR polling
; Write character
TryWrite LDI        R1, DSR               ; get status
                    BRzp      TryWrite    ; look for bit 15 (display is ready)
WriteIt             STI       R0, DDR     ; write character
; Return from TRAP
Return              LD        R1, SaveR1  ; restore R1 & R7
                    LD        R7, SaveR7
                    RET                   ; return from trap (JMP R7)
DSR                 .FILL     xFE04       ; address of Display Status Register
DDR                 .FILL     xFE06       ; address of Display Data Register
SaveR1              .FILL     0
SaveR7              .FILL     0
                    .END
```


ALLAN HANCOCK COLLEGE
Start here. Go anywhere.

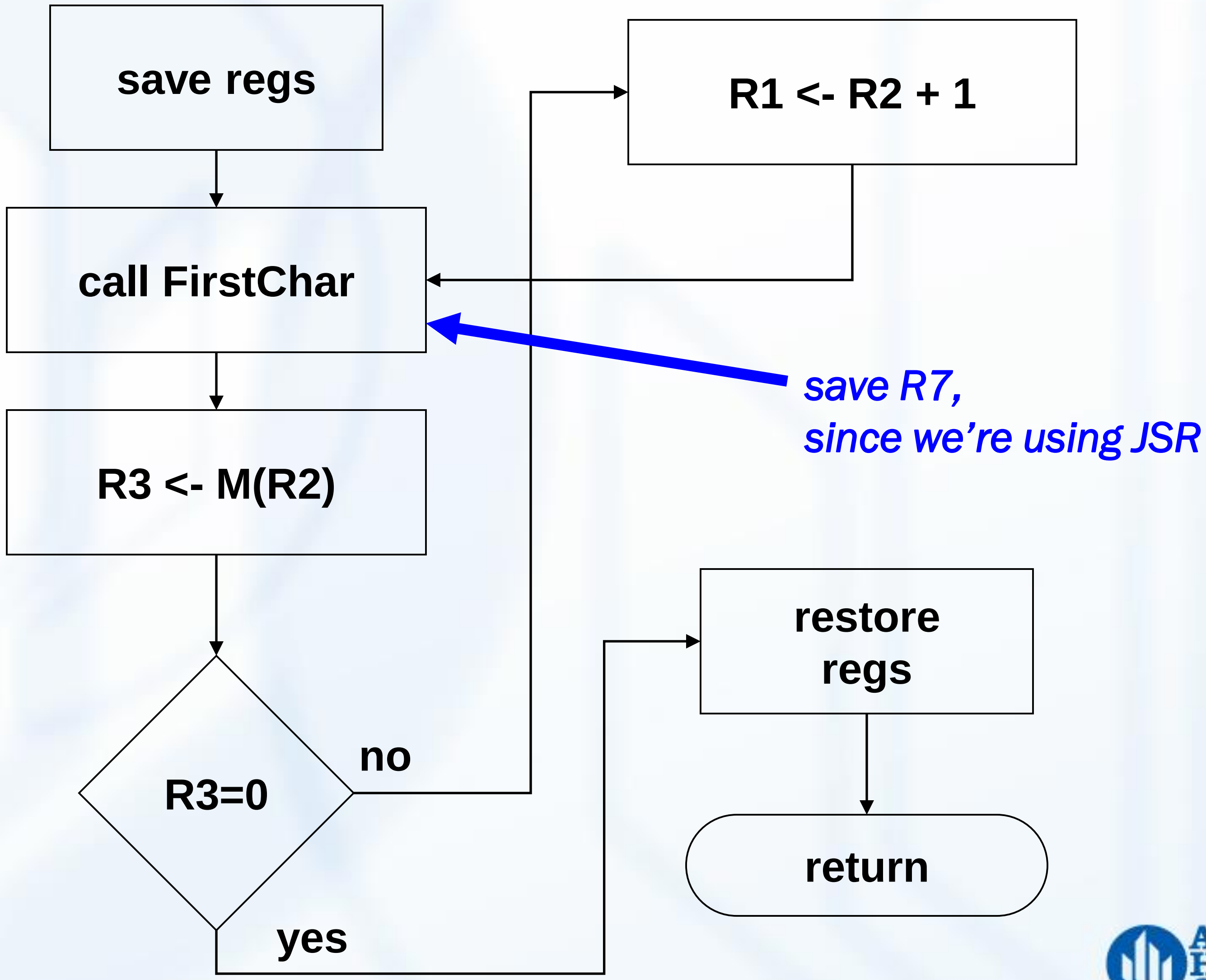
Example

1. **Write a subroutine FirstChar to:**
   - find the first occurrence
     of a particular character (in R0)
     in a string (pointed to by R1);
     return pointer to character or to end of string (NULL) in R2.

2. **Use FirstChar to write CountChar, which:**
   - counts the number of occurrences
     of a particular character (in R0)
     in a string (pointed to by R1);
     return count in R2.

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

CountChar Algorithm (using FirstChar)

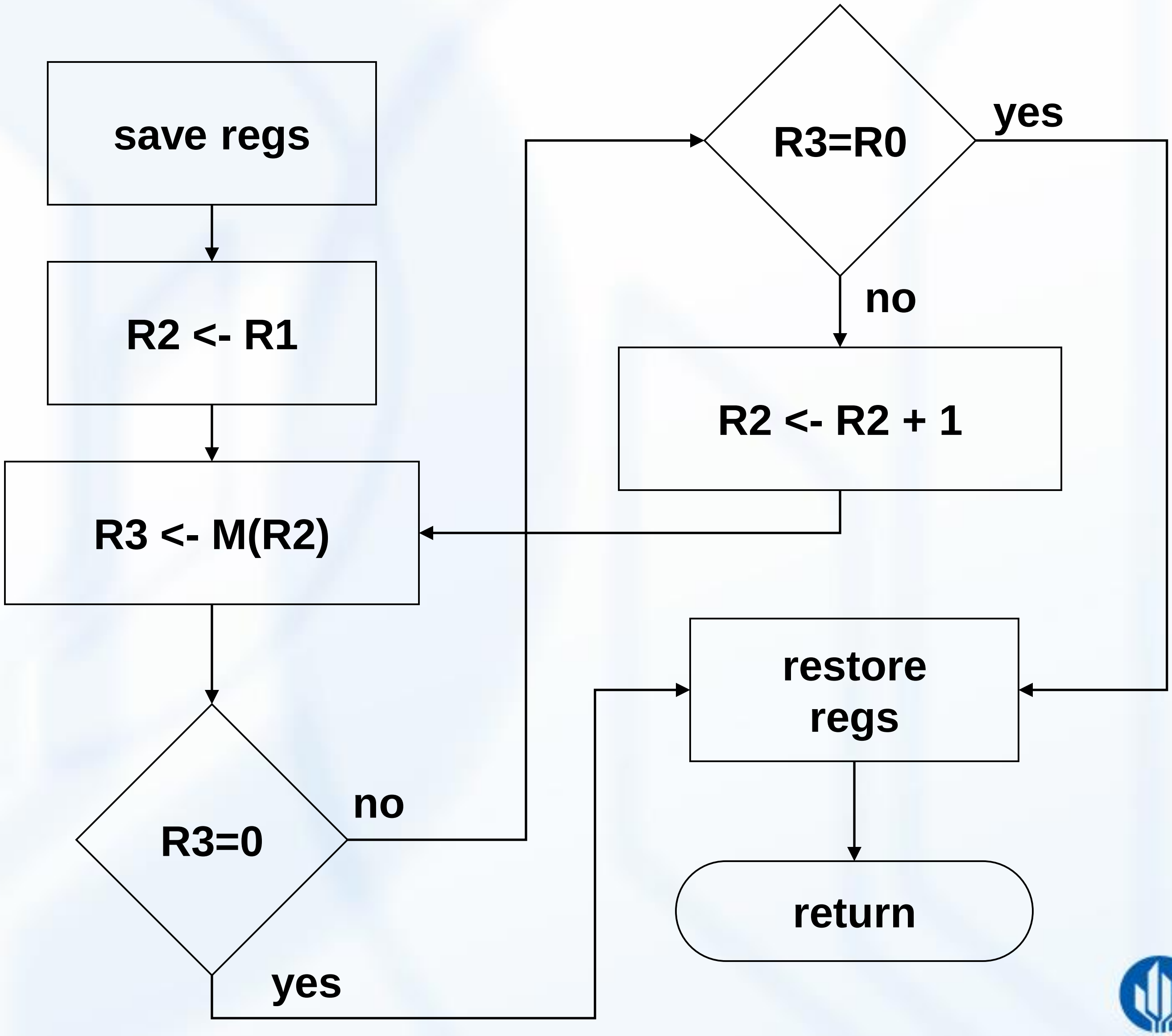


Start here. Go anywhere.

CountChar Implementation

```
; CountChar: subroutine to count occurrences of a char
CountChar
        ST      R3, CCR3        ; save registers
        ST      R4, CCR4
        ST      R7, CCR7        ; JSR alters R7
        ST      R1, CCR1        ; save original string ptr
        AND     R4, R4, #0      ; initialize count to zero
CC1     JSR     FirstChar       ; find next occurrence (ptr in R2)
        LDR     R3, R2, #0      ; see if char or null
        BRz     CC2             ; if null, no more chars
        ADD     R4, R4, #1      ; increment count
        ADD     R1, R2, #1      ; point to next char in string
                BRnzp   CC1
CC2     ADD     R2, R4, #0      ; move return val (count) to R2
        LD      R3, CCR3        ; restore regs
        LD      R4, CCR4
        LD      R1, CCR1
        LD      R7, CCR7
        RET                     ; and return
```

FirstChar Algorithm



FirstChar Implementation

```
; FirstChar: subroutine to find first occurrence of a
char
FirstChar
        ST      R3, FCR3        ; save registers
        ST      R4, FCR4        ; save original char
        NOT     R4, R0          ; set R4 to 2's complement
        ADD     R4, R4, #1      ;   of input for comparison
        ADD     R2, R1, #0      ; initialize ptr to
                                ; beginning of string
FC1     LDR     R3, R2, #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     R2, R2, #1      ; increment pointer
        BRnzp   FC1
FC2     LD      R3, FCR3        ; restore registers
        LD      R4, FCR4        ;
        RET                     ; and return
```


ALLAN HANCOCK COLLEGE
Start here. Go anywhere.