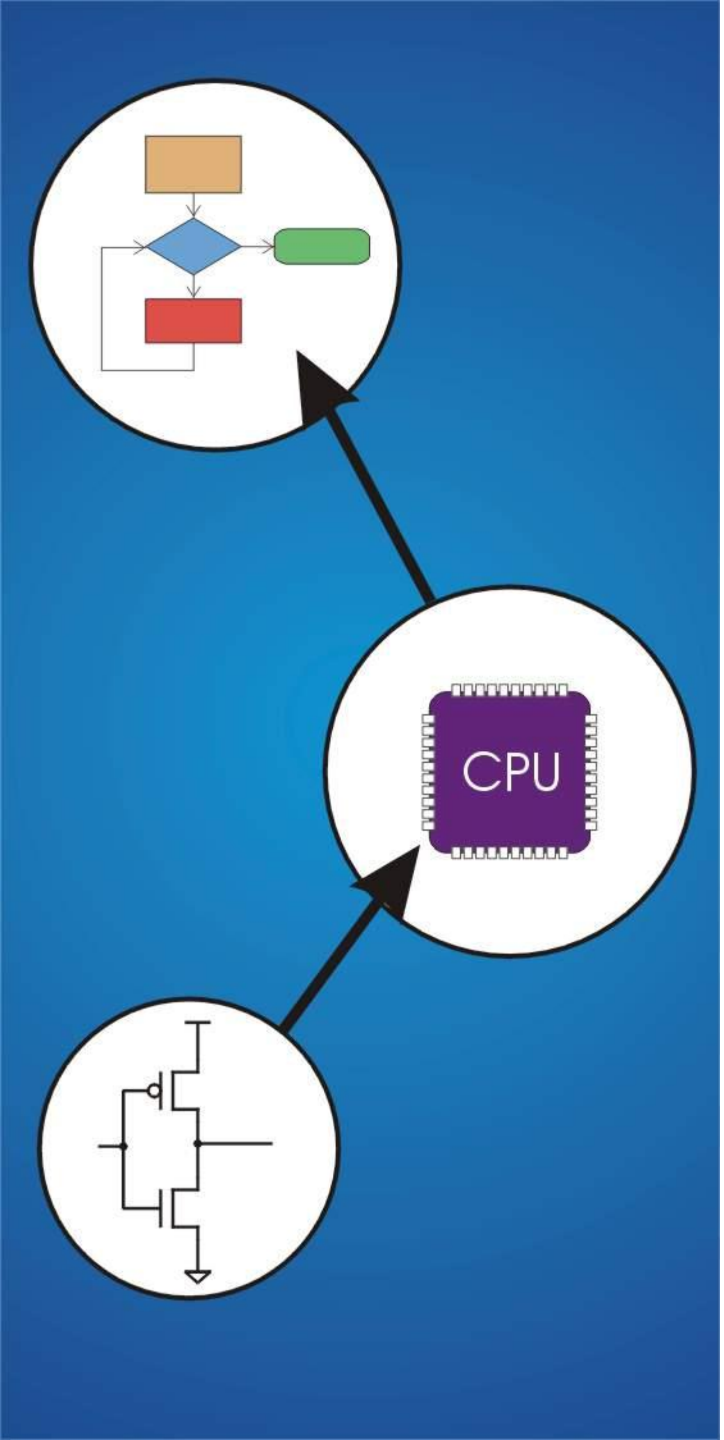
Introduction to Computer Engineering

CS/ECE 252, Spring 2013

Prof. Mark D. Hill
Computer Sciences Department
University of Wisconsin – Madison



Chapter 7 & 9.2
Assembly Language and Subroutines

Human-Readable Machine Language

Computers like ones and zeros…

```
0001110010000110
```

Humans like symbols…

```
ADD  R6,R2,R6   ; increment index reg.
```

Assembler is a program that turns symbols into machine instructions.

- **ISA-specific:**
  close correspondence between symbols and instruction set
  - **mnemonics for opcodes**
  - **labels for memory locations**
- **additional operations for allocating storage and initializing data**

An Assembly Language Program

```
;
; Program to multiply a number by the constant 6
;
        .ORIG  x3050
        LD     R1, SIX
        LD     R2, NUMBER
        AND    R3, R3, #0    ; Clear R3.  It will
                             ; contain the product.
; The inner loop
;
AGAIN   ADD    R3, R3, R2
        ADD    R1, R1, #-1   ; R1 keeps track of
        BRp    AGAIN         ; the iteration.
;
        HALT
;
NUMBER  .BLKW  1
SIX     .FILL  x0006
;
        .END
```

LC-3 Assembly Language Syntax

Each line of a program is one of the following:

- **an instruction**
- **an assember directive (or pseudo-op)**
- **a comment**

Whitespace (between symbols) and case are ignored.

Comments (beginning with ";") are also ignored.

An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```

optional   *mandatory*

Opcodes and Operands

Opcodes

- reserved symbols that correspond to LC-3 instructions
- listed in Appendix A
  - ex: `ADD`, `AND`, `LD`, `LDR`, ...

Operands

- registers -- specified by Rn, where n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - ex:

```
ADD R1,R1,R3
ADD R1,R1,#3
LD  R6,NUMBER
BRz LOOP
```

Labels and Comments

Label

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  - ex:

```
LOOP  ADD  R1,R1,#-1
      BRp  LOOP
```

Comment

- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs
- tips for useful comments:
  - avoid restating the obvious, as “decrement R1”
  - provide additional insight, as in “accumulate product in R6”
  - use comments to separate pieces of program

Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but "opcode" starts with dot

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of program
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Trap Codes

LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.

Code	*Equivalent*	*Description*
`HALT`	`TRAP x25`	Halt execution and print message to console.
`IN`	`TRAP x23`	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
`OUT`	`TRAP x21`	Write one character (in R0[7:0]) to console.
`GETC`	`TRAP x20`	Read one character from keyboard. Character stored in R0[7:0].
`PUTS`	`TRAP x22`	Write null-terminated string to console. Address of string is in R0.

Style Guidelines

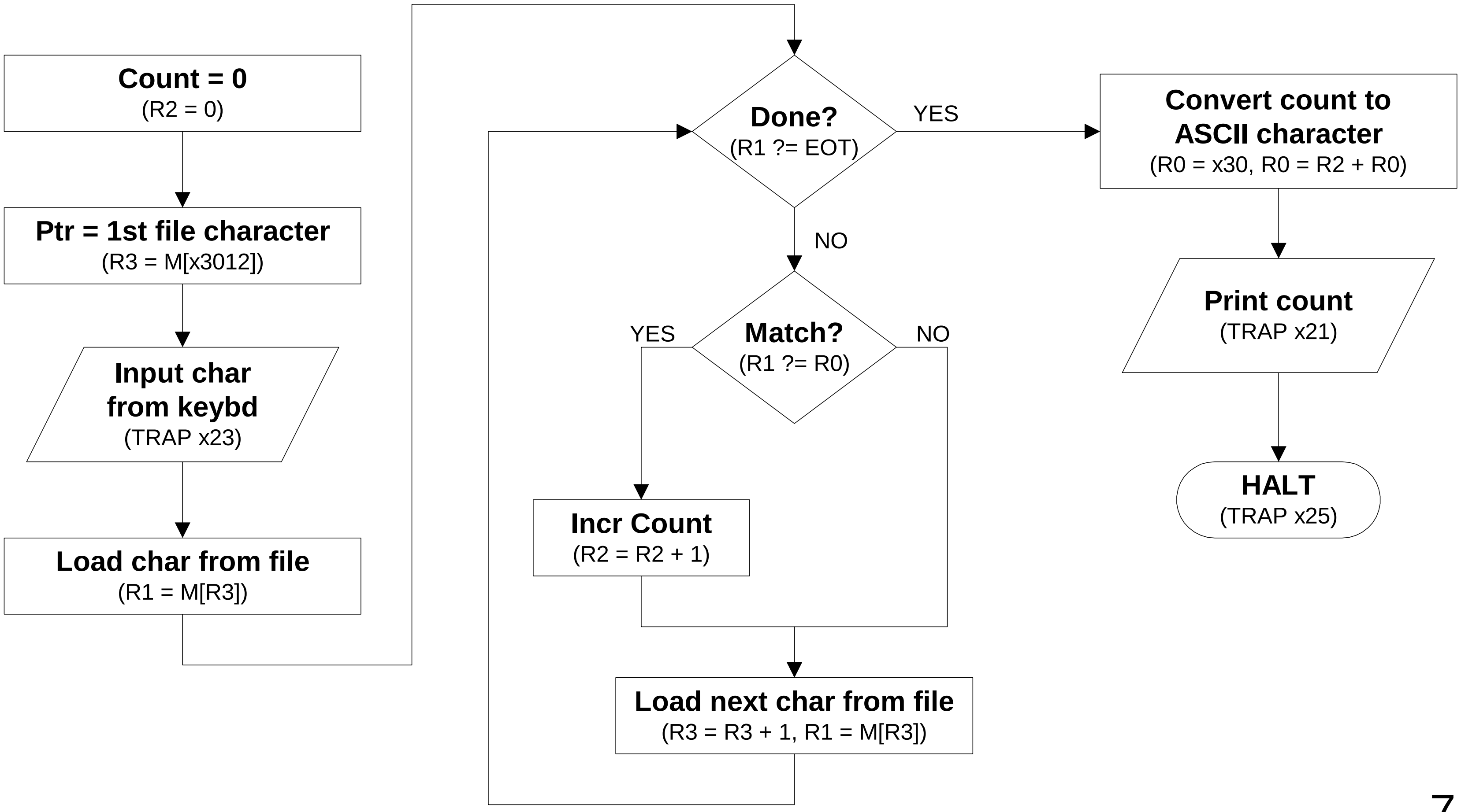
Use the following style guidelines to improve the readability and understandability of your programs:

1. **Provide a program header, with author's name, date, etc., and purpose of program.**
2. **Start labels, opcode, operands, and comments in same column for each line.** (Unless entire line is a comment.)
3. **Use comments to explain what each register does.**
4. **Give explanatory comment for most instructions.**
5. **Use meaningful symbolic names.**
    - **Mixed upper and lower case for readability.**
    - ASCIItoBinary, InputRoutine, SaveR1
6. **Provide comments between program sections.**
7. **Each line must fit on the page -- no wraparound or truncations.**
    - **Long statements split in aesthetically pleasing manner.**

Sample Program

Count the occurrences of a character in a file.

Remember this?



Char Count in Assembly Language (1 of 3)

```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG   x3000
        AND     R2, R2, #0      ; R2 is counter, initially 0
        LD      R3, PTR         ; R3 is pointer to characters
        GETC                    ; R0 gets character input
        LDR     R1, R3, #0      ; R1 gets first character
;
; Test character for end of file
;
TEST    ADD     R4, R1, #-4     ; Test for EOT (ASCII x04)
        BRz     OUTPUT          ; If done, prepare the output
```

Char Count in Assembly Language (2 of 3)

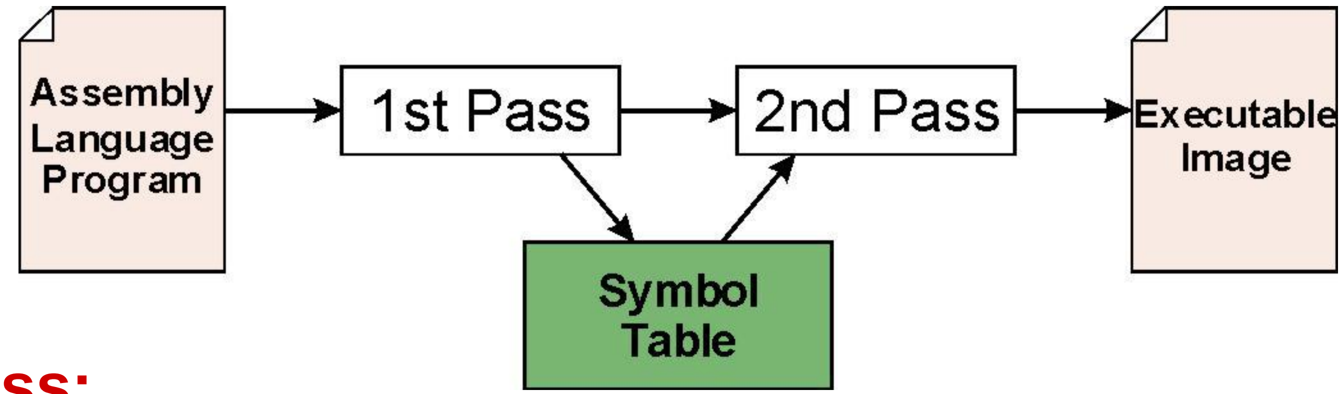
```
;
; Test character for match.  If a match, increment count.
;
        NOT     R1, R1
        ADD     R1, R1, R0  ; If match, R1 = xFFFF
        NOT     R1, R1      ; If match, R1 = x0000
        BRnp    GETCHAR     ; If no match, do not increment
        ADD     R2, R2, #1
;
; Get next character from file.
;
GETCHAR ADD     R3, R3, #1  ; Point to next character.
        LDR     R1, R3, #0  ; R1 gets next char to test
        BRnzp   TEST
;
; Output the count.
;
OUTPUT  LD      R0, ASCII   ; Load the ASCII template
        ADD     R0, R0, R2  ; Covert binary count to ASCII
        OUT                 ; ASCII code in R0 is displayed.
        HALT                ; Halt machine
```

Char Count in Assembly Language (3 of 3)

```
;
; Storage for pointer and ASCII template
;
ASCII    .FILL   x0030
PTR      .FILL   x4000
         .END
```

Assembly Process

Convert assembly language file (.asm)
into an executable file (.obj) for the LC-3 simulator.



First Pass:

- **scan program file**
- **find all labels and calculate the corresponding addresses; this is called the *symbol table***

Second Pass:

- **convert instructions to machine language, using information from symbol table**

First Pass: Constructing the Symbol Table

1. **Find the `.ORIG` statement, which tells us the address of the first instruction.**
   - **Initialize location counter (LC), which keeps track of the current instruction.**

2. **For each non-empty line in the program:**
   a) **If line contains a label, add label and LC to symbol table.**
   b) **Increment LC.**
      – **NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.**

3. **Stop when `.END` statement is reached.**

NOTE: A line that contains only a comment is considered an empty line.

Practice

Construct the symbol table for the program in Figure 7.1 (Slides 7-11 through 7-13).

Symbol	Address

Second Pass: Generating Machine Language

For each executable assembly language statement, generate the corresponding machine language instruction.

- **If operand is a label, look up the address from the symbol table.**

Potential problems:

- **Improper number or type of arguments**
  - **ex:**
    ```
    NOT  R1,#7
    ADD  R1,R2
    ADD  R3,R3,NUMBER
    ```
- **Immediate argument too large**
  - **ex:** `ADD R1,R2,#1023`
- **Address (associated with label) more than 256 from instruction**
  - **can't use PC-relative addressing mode**

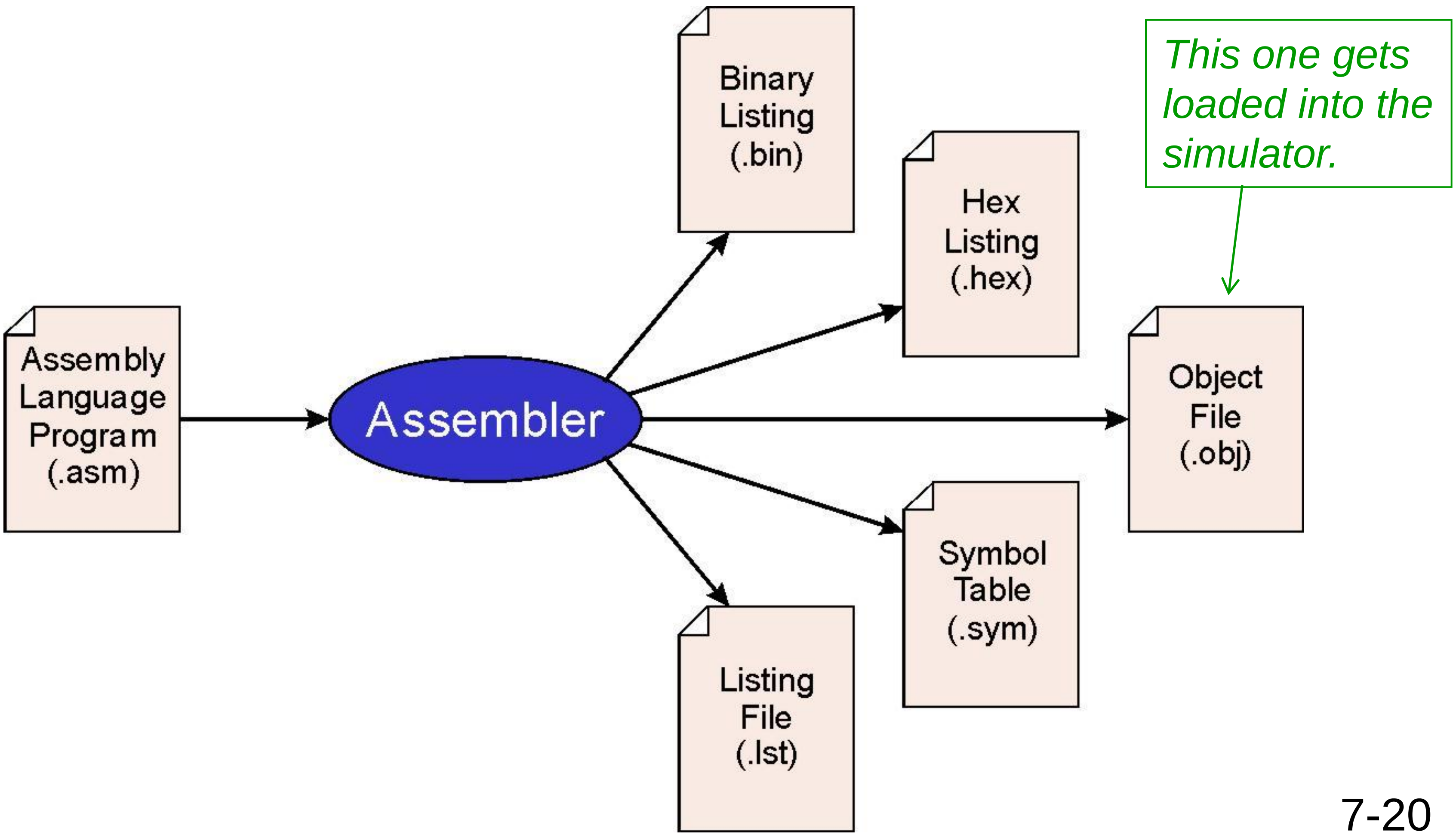
Practice

Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

Statement	Machine Language
`LD    R3,PTR`	
`ADD   R4,R1,#-4`	
`LDR   R1,R3,#0`	
`BRnp  GETCHAR`	

LC-3 Assembler

Using "lc3as" (Unix) or LC3Edit (Windows), generates several different output files.



Object File Format

LC-3 object file contains

- Starting address (location where program must be loaded), followed by…
- Machine instructions

Example

- Beginning of "count character" object file looks like this:

0011000000000000	←	.ORIG x3000
0101010010100000	←	AND R2, R2, #0
0010011000010001	←	LD R3, PTR
1111000000100011	←	TRAP x23
.		
.		
.		

Multiple Object Files

An object file is not necessarily a complete program.

- **system-provided library routines**
- **code blocks written by multiple developers**

For LC-3 simulator,
can load multiple object files into memory,
then start executing at a desired address.

- **system routines, such as keyboard input, are loaded automatically**
  - **loaded into “system memory,” below x3000**
  - **user code should be loaded between x3000 and xFDFF**
- **each object file includes a starting address**
- **be careful not to load overlapping object files**

Linking and Loading

Loading **is the process of copying an executable image into memory.**

- **more sophisticated loaders are able to *relocate* images to fit into available memory**
- **must readjust branch targets, load/store addresses**

Linking **is the process of resolving symbols between independent object files.**

- **suppose we define a symbol in one module, and want to use it in another**
- **some notation, such as `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module**
- **linker will search symbol tables of other modules to resolve symbols and complete code generation before loading**

Skipping Ahead to Chapter 9

You will need to use subroutines for programming assignments

- **Read Section 9.2**

A subroutine is a program fragment that:

- **performs a well-defined task**
- **is invoked (called) by another user program**
- **returns control to the calling program when finished**

Reasons for subroutines:

- **reuse useful (and debugged!) code without having to keep typing it in**
- **divide task among multiple programmers**
- **use vendor-supplied *library* of useful routines**

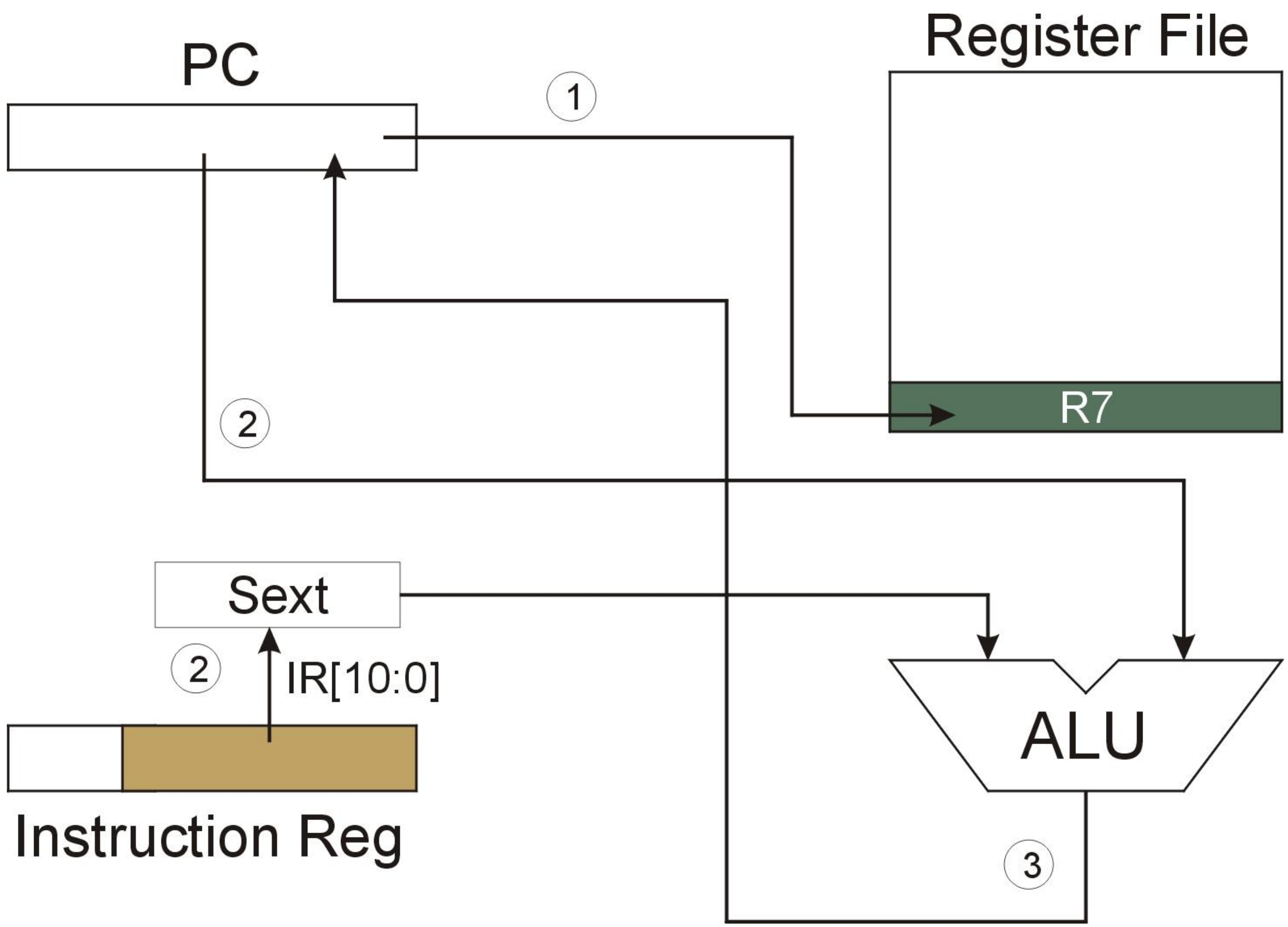
JSR Instruction

	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]**

JSR



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

JSRR Instruction

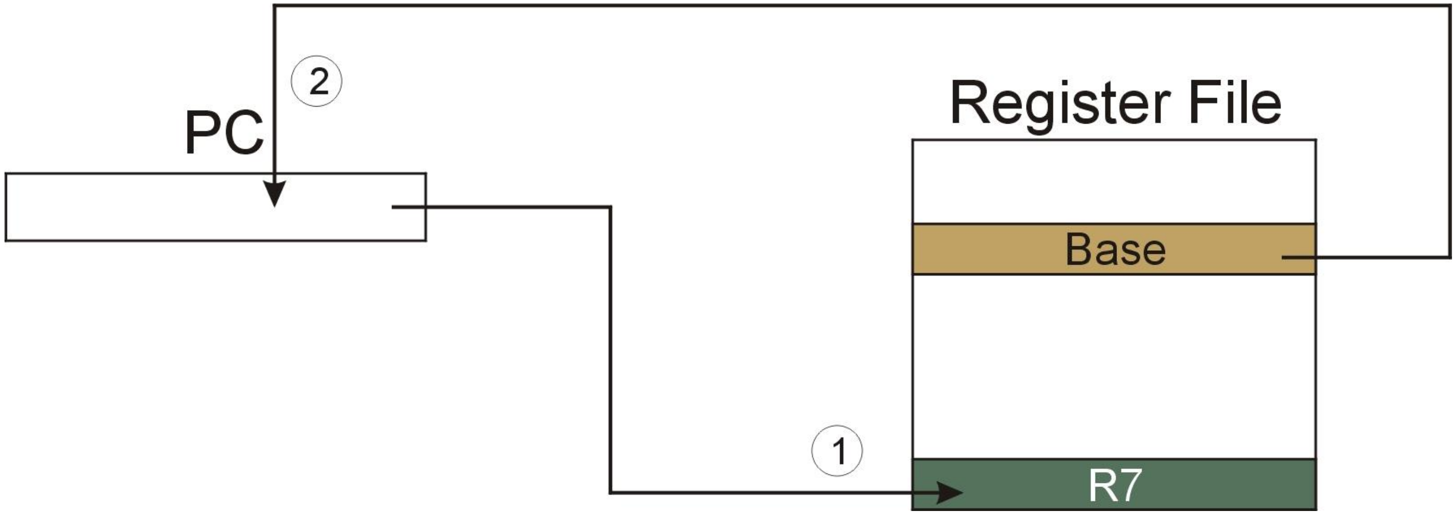
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSRR	0	1	0	0	0	0	0	Base			0	0	0	0	0	0

Just like JSR, except Register addressing mode.

- **target address is Base Register**
- **bit 11 specifies addressing mode**

What important feature does JSRR provide that JSR does not?

JSRR



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

Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.

- **just like TRAP**

Example: Negate the value in R0

```
2sComp    NOT  R0, R0       ; flip bits
          ADD  R0, R0, #1   ; add one
          RET               ; return to caller
```

To call from a program (within 1024 instructions):

```
; need to compute R4 = R1 - R3
          ADD  R0, R3, #0   ; copy R3 to R0
          JSR  2sComp       ; negate
          ADD  R4, R1, R0   ; add to R1
          ...
```

Note: Caller should save R0 if we'll need it later!

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 2sComp 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 2sComp 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.**

Example

(1) Write a subroutine FirstChar to:

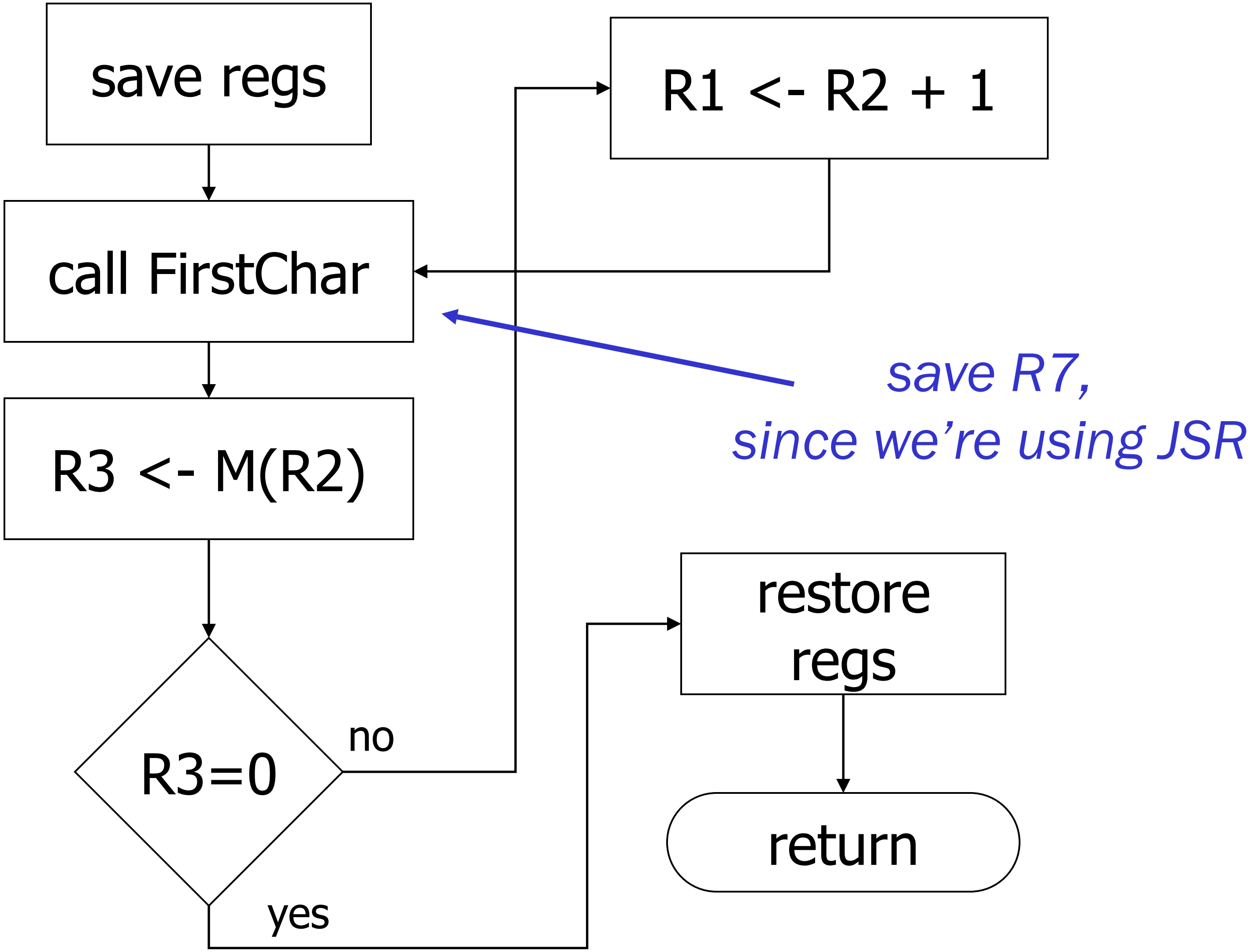
find the <u>first</u> occurrence
of a particular character (in R0)
in a string (pointed to by R1);
return pointer to character or to end of string (NULL) in R2.

(2) Use FirstChar to write CountChar, which:

counts the <u>number</u> 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!

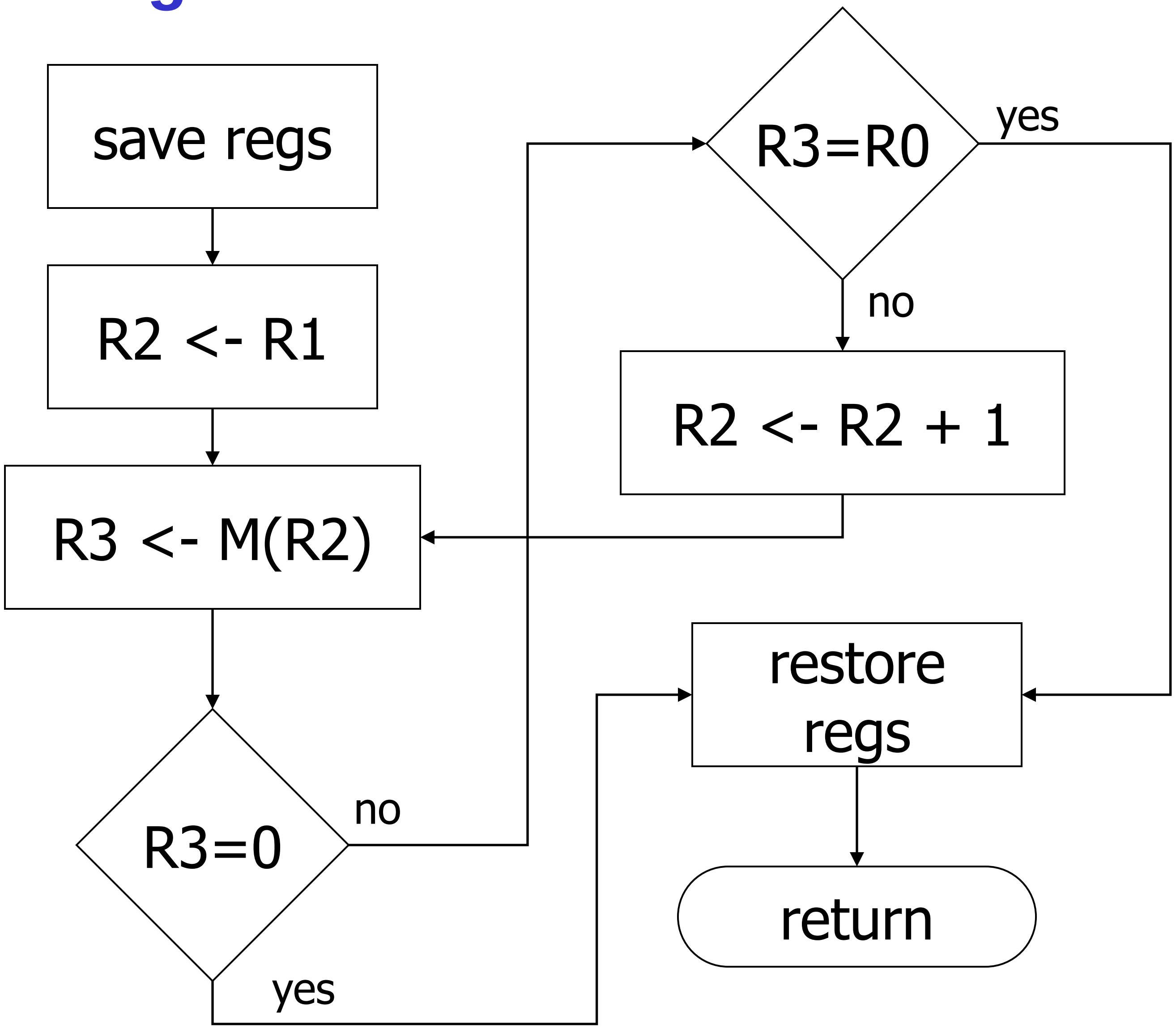
CountChar Algorithm (using FirstChar)



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        ; negate R0 for comparisons
        ADD   R4, R4, #1
        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
```