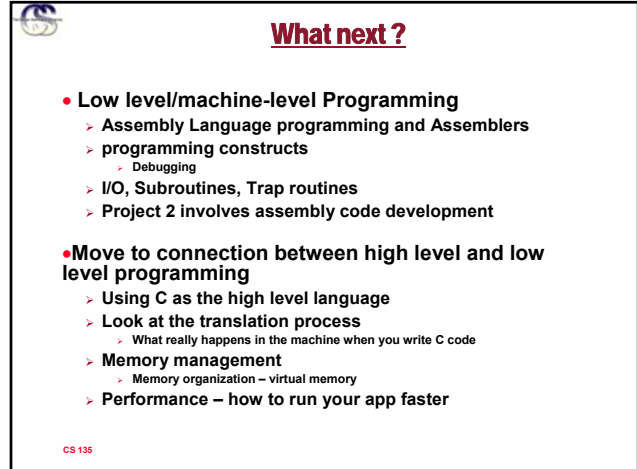


# CS 135: Computer Architecture I

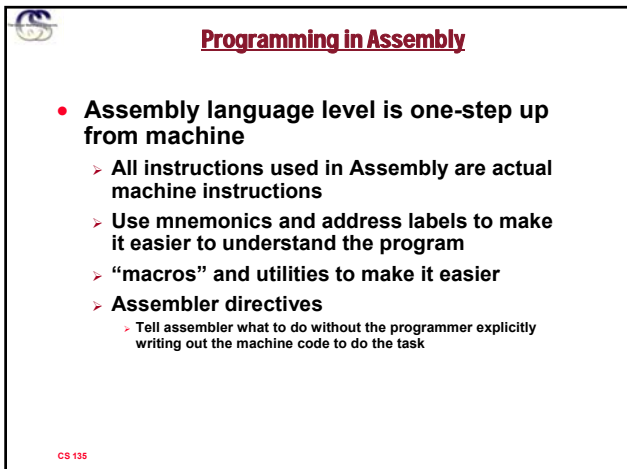
**Instructor: Prof. Bhagi Narahari**  
 Dept. of Computer Science  
 Course URL: [www.seas.gwu.edu/~bhagiweb/cs135/](http://www.seas.gwu.edu/~bhagiweb/cs135/)



## What next ?

- **Low level/machine-level Programming**
  - Assembly Language programming and Assemblers
  - programming constructs
    - Debugging
  - I/O, Subroutines, Trap routines
  - Project 2 involves assembly code development
- **Move to connection between high level and low level programming**
  - Using C as the high level language
  - Look at the translation process
    - What really happens in the machine when you write C code
  - **Memory management**
    - Memory organization – virtual memory
  - Performance – how to run your app faster

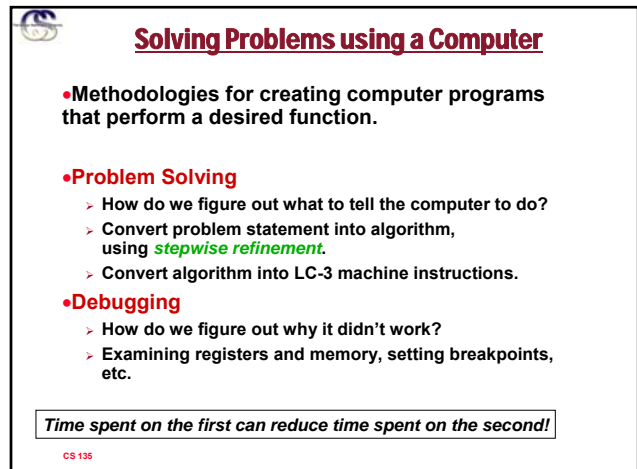
CS 135



## Programming in Assembly

- **Assembly language level is one-step up from machine**
  - All instructions used in Assembly are actual machine instructions
  - Use mnemonics and address labels to make it easier to understand the program
  - “macros” and utilities to make it easier
  - **Assembler directives**
    - Tell assembler what to do without the programmer explicitly writing out the machine code to do the task

CS 135



## Solving Problems using a Computer

- **Methodologies for creating computer programs that perform a desired function.**
- **Problem Solving**
  - How do we figure out what to tell the computer to do?
  - Convert problem statement into algorithm, using *stepwise refinement*.
  - Convert algorithm into LC-3 machine instructions.
- **Debugging**
  - How do we figure out why it didn't work?
  - Examining registers and memory, setting breakpoints, etc.

*Time spent on the first can reduce time spent on the second!*

CS 135

**Recap: Problem Solving and Problem Decomposition**

- With an eye towards writing assembly programming/low-level software

CS 135

**Stepwise Refinement**

- Also known as **systematic decomposition**.
- Start with problem statement:
 

“We wish to count the number of occurrences of a character in a file. The character in question is to be input from the keyboard; the result is to be displayed on the monitor.”
- **Decompose** task into a few simpler **subtasks**.
- Decompose each subtask into **smaller subtasks**, and these into **even smaller subtasks**, etc.... until you get to the machine instruction level.
- **Ambiguous statements ?**
  - Ask person who wants problem solved
  - Or, make decision and document it

CS 135

**Three Basic Constructs**

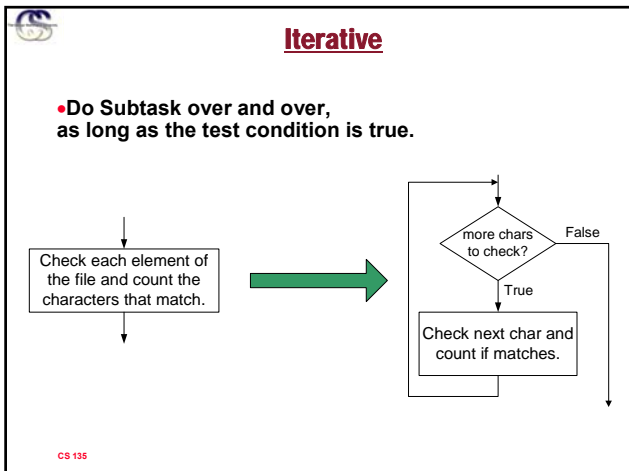
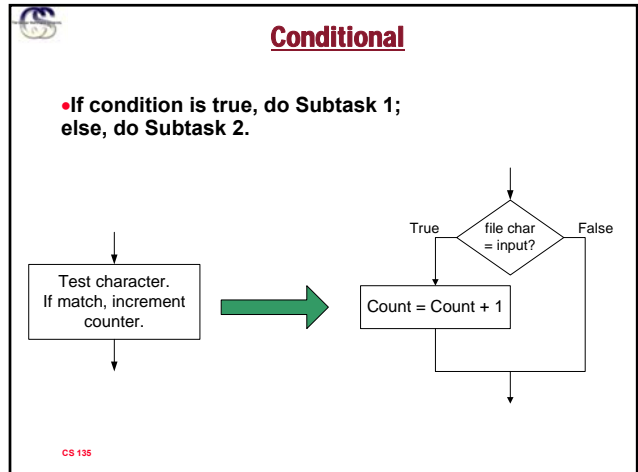
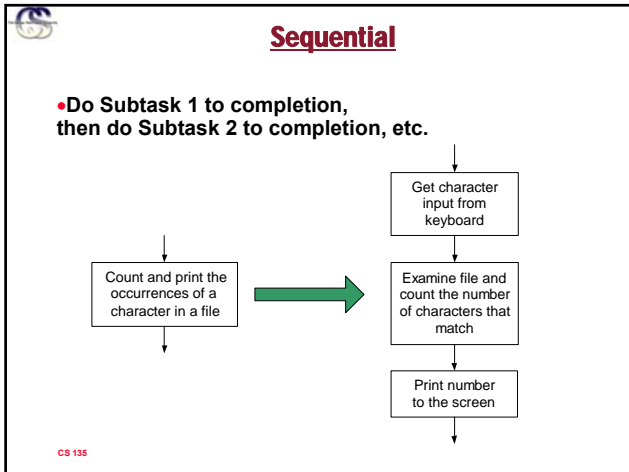
• There are three basic ways to decompose a task:

CS 135 **Sequential** **Conditional** **Iterative**

**Problem Solving Skills**

- Learn to convert problem statement into step-by-step description of subtasks.
  - Like a puzzle, or a “word problem” from grammar school math.
    - What is the starting state of the system?
    - What is the desired ending state?
    - How do we move from one state to another?
  - **Recognize English words that correlate to three basic constructs:**
    - “do A then do B” ⇒ **sequential**
    - “if G, then do H” ⇒ **conditional**
    - “for each X, do Y” ⇒ **iterative**
    - “do Z until W” ⇒ **iterative**

CS 135

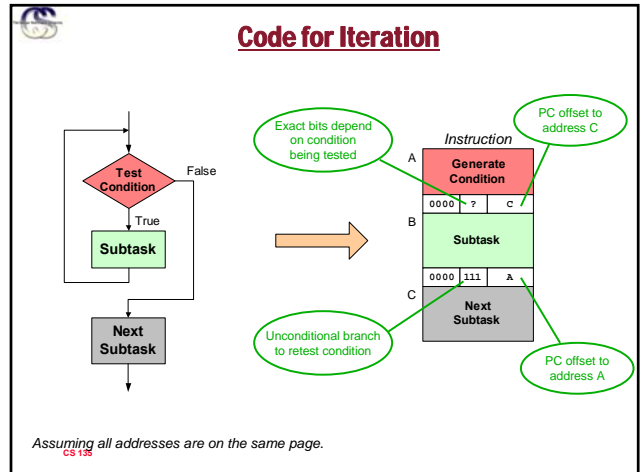
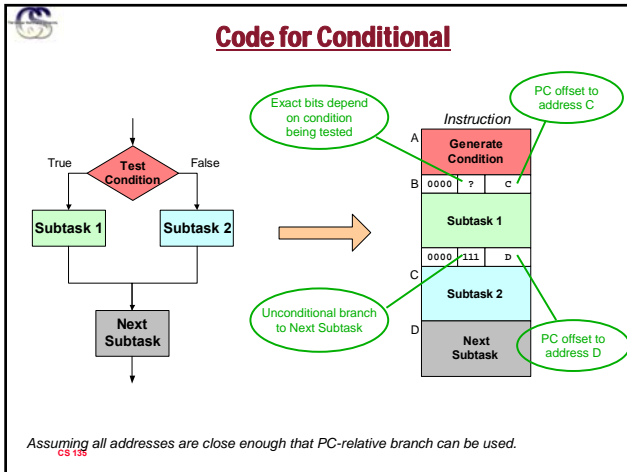


### LC-3 Control Instructions

•How do we use LC-3 instructions to encode the three basic constructs?

- Sequential**
  - > Instructions naturally flow from one to the next, so no special instruction needed to go from one sequential subtask to the next.
- Conditional and Iterative**
  - > Create code that converts condition into N, Z, or P.  
 Example:  
 Condition: "Is R0 = R1?"  
 Code: Subtract R1 from R0; if equal, Z bit will be set.
  - > Then use BR instruction to transfer control to the proper subtask.

CS 135



### Assembly Language: 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.

CS 135

### Programming in Assembly

- Assembly language level is one-step up from machine
  - All instructions used in Assembly are actual machine instructions
  - Use mnemonics and address labels to make it easier to understand the program
    - Labels converted to addresses and offsets by assembler
  - “macros” and utilities to make it easier
  - Assembler directives
    - Tell assembler what to do without the programmer explicitly writing out the machine code to do the task
    - Allocating storage
    - Initializing data

CS 135

## 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

```

CS 135

## LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  - > an instruction
  - > an assembler 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

CS 135

## 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

```

CS 135

## 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

CS 135

## 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

CS 135

## 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.

CS 135

```

•; Multiply a number by 6
•
• .ORIG x3050
• LD R1, SIX ; R1 is loop counter
• LD R2, NUMBER
• AND R3, R3, #0 ; Clr R3, will hold product
•; The Loop
•AGAIN ADD R3, R3, R2 ; Summing into R3
• ADD R1, R1, #-1 ; Dec loop counter
• BRp AGAIN
• HALT
•;
•NUMBER .BLKW 1
•SIX .FILL x0006
• .END
  
```

CS 135

```

•; Multiply a number by 6
•
• .ORIG x3050
• LD R1, RYAN ; R1 is loop counter
• LD R2, NUMBER
• AND R3, R3, #0 ; Clr R3, will hold product
•; The Loop
•AGAIN ADD R3, R3, R2 ; Summing into R3
• ADD R1, R1, #-1 ; Dec loop counter
• BRp AGAIN
• HALT
•;
•NUMBER .BLKW 1
•RYAN .FILL x0006
• .END
  
```

CS 135

```

•; Multiply a number by 6
•
• .ORIG x3050
• LD R1, SIX ; R1 is loop counter
• LD R2, NUMBER
• AND R3, R3, #0 ; Clr R3, will hold product
•; The Loop
•AGAIN ADD R3, R3, R2 ; Summing into R3
• ADD R1, R1, #-1 ; Dec loop counter
• BRp AGAIN
• HALT
•;
•NUMBER .BLKW 1
•SIX .FILL x0006
• .END

```

Must have Opcode and Operands

CS 135

```

•; Multiply a number by 6
•
• .ORIG x3050
• LD R1, SIX ; R1 is loop counter
• LD R2, NUMBER
• AND R3, R3, #0 ; Clr R3, will hold product
•; The Loop
•AGAIN ADD R3, R3, R2 ; Summing into R3
• ADD R1, R1, #-1 ; Dec loop counter
• BRp AGAIN
• HALT
•;
•NUMBER .BLKW 1
•SIX .FILL x0006
• .END

```

Label and Comments optional

CS 135

```

•; Multiply a number by 6
•
• .ORIG x3050
• LD R1, SIX ; R1 is loop counter
• LD R2, NUMBER
• AND R3, R3, #0 ; Clr R3, will hold product
•; The Loop
•AGAIN ADD R3, R3, R2 ; Summing into R3
• ADD R1, R1, #-1
• BRp AGAIN
• HALT
•;
•NUMBER .BLKW 1
•SIX .FILL x0006
• .END

```

	Decimal	Binary	Hex
#		b	x

Decimal #  
Binary b  
Hex x

CS 135

```

•; Multiply a number by 6
•
• .ORIG x3050
• LD R1, SIX ; R1 is loop counter
• LD R2, NUMBER
• AND R3, R3, #0 ; Clr R3, will hold product
•; The Loop
•AGAIN ADD R3, R3, R2 ; Summing into R3
• ADD R1, R1, #-1
• BRp AGAIN
• HALT
•;
•NUMBER .BLKW 1
•SIX .FILL x0006
• .END

```

More than just code, need to worry about memory allocation

CS 135

**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.

CS 135

**Assembly Process**

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

```

graph LR
  A[Assembly Language Program] --> B[1st Pass]
  B --> C[2nd Pass]
  C --> D[Executable Image]
  B <--> E[Symbol Table]
  C <--> E
  
```

- **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

CS 135

**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.

CS 135

**Pass 1**

- Construct the symbol table for the program

Symbol	Address
SIX	
NUMBER	
AGAIN	

CS 135

## 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

CS 135

## Pass 2

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

Statement	Machine Language
LD R1, SIX	
BRp AGAIN	
LD R2, NUMBER	

CS 135

## LC-3 Assembler

- Using “assemble” (Unix) or LC3Edit (Windows), generates several different output files.

```

graph LR
    A[Assembly Language Program (.asm)] --> B((Assembler))
    B --> C[Binary Listing (.bin)]
    B --> D[Hex Listing (.hex)]
    B --> E[Object File (.obj)]
    B --> F[Symbol Table (.sym)]
    B --> G[Listing File (.lst)]
    
```

CS 135

## Object File Format

- LC-3 object file contains
  - Starting address (location where program must be loaded), followed by...
  - Machine instructions
- Example
  - Beginning of object file has origin command

CS 135

**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 xFDFE
  - each object file includes a starting address
  - be careful not to load overlapping object files

CS 135

**Linking and Loading**

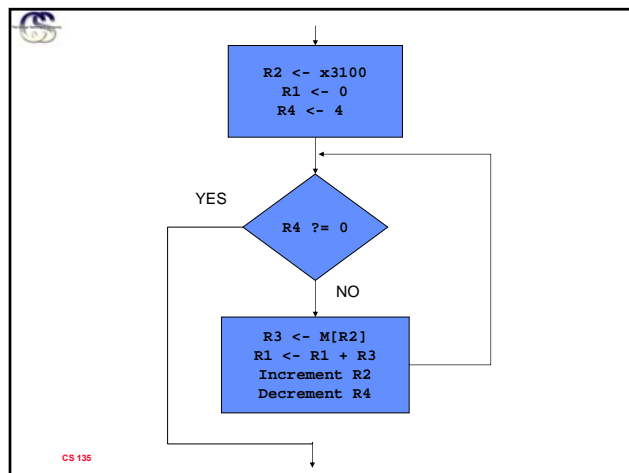
- **Loading** is the process of copying an executable image into memory.
  - more sophisticated loaders are able to relocate images to fit into available memory
    - must readjust branch targets, load/store addresses
- **Linking** is the process of resolving symbols between independent object files.
  - 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


CS 135

**Example: Addition of array of numbers**

- **Compute sum of 4 integers.**  
Numbers start at location x3100. Program starts at location x3000.
  - Add numbers from location x3100 to x3104
  - Store first address in R2
  - R4 has "counter" – counts down from 12 to 0
  - R1 will store the running Sum
  - Store result in memory location

CS 135






## Converting Code to Assembly

- Can use a standard template approach
- Typical Constructs
  - > if/else
  - > while
  - > do/while
  - > for


CS 135



## if/else

<ul style="list-style-type: none"> <li>• if(x &gt; 0)</li> <li>• {</li> <li style="padding-left: 20px;">• r2 = r3 + r4;</li> <li>• }</li> <li>• else</li> <li>• {</li> <li style="padding-left: 20px;">• r5 = r6 + r7;</li> <li>• }</li> </ul>	<ul style="list-style-type: none"> <li>• LD R1, X</li> <li>• BRP THEN</li> <li>• ADD R5,R6,R7</li> <li>• BRNZP DONE</li> <li>• THEN ADD R2,R3,R4</li> <li>• DONE ...</li> </ul>
--	---


CS 135



## if/else

<ul style="list-style-type: none"> <li>• if(x &gt; 0)</li> <li>• {</li> <li style="padding-left: 20px;">• r2 = r3 + r4;</li> <li>• }</li> <li>• else</li> <li>• {</li> <li style="padding-left: 20px;">• r5 = r6 + r7;</li> <li>• }</li> </ul>	<ul style="list-style-type: none"> <li>• LD R1,X</li> <li>• BRNZ ELSE</li> <li>• ADD R2,R3,R4</li> <li>• BRNZP DONE</li> <li>• ELSE ADD R5,R6,R7</li> <li>• DONE ...</li> </ul>
--	---


CS 135



## while

<ul style="list-style-type: none"> <li>• i = 10;</li> <li>• x = 0;</li> <li>• while(i &gt; 0)</li> <li>• {</li> <li style="padding-left: 20px;">• x = x + i;</li> <li style="padding-left: 20px;">• i--;</li> <li>• }</li> </ul>	<ul style="list-style-type: none"> <li>• AND R1,R1,#0</li> <li>• ADD R1,R1,#10</li> <li>• AND R2,R2,#0</li> <li>• WHL → BRNZ DONE</li> <li>• ADD R2,R2,R1</li> <li>• ADD R1,R1,#-1</li> <li>• BRNZP WHL</li> </ul>
--	--


CS 135



### Course Announcements

- **Mid-semester grade will be given out next class**
  - If you are not B or higher then you **MUST** meet with me to review your performance and figure out how to get more from the course
- **Slight change in Class lecture style in future lectures**
  - In-class exercises
  - Start playing with LC3 editor and simulator
    - We expect you to become familiar with it when you come to class
- **Study habits**
  - Textbook
  - Regular reading – time management!
- **Team evaluations: is it working?**

CS 135




### Why so many Load Instructions?

- **LD**
  - $R_x \leftarrow \text{MEM}[\text{PC}+1+\text{Offset}]$
  - Gets the contents of a memory location into a register
  - Memory location must be physically close to the instruction

```
LD R1, DATA
```

```
DATA .FILL #42
```

CS 135




### Why so many Load Instructions?

- **LDI**
  - $R_x \leftarrow \text{MEM}[\text{MEM}[\text{PC} + 1 + \text{Offset}]]$
  - Gets the contents of the memory location pointed to by another memory location
  - Typical usage device register operations

```
LDI R0, KBSR
```

```
KBSR .FILL xFE00
```

CS 135




### Why so many Load Instructions?

- **LDR**
  - $R_x \leftarrow \text{MEM}[R_{\text{base}} + \text{Offset}]$
  - Used to load register from an address contained in another register (+ offset)

```
LDR R1, R2, #3
```

CS 135




### Why so many Load Instructions?

- **LEA**
  - $R \leftarrow PC + 1 + \text{Offset}$
  - **Used to get an address into a register**
    - Use before LDR or STR
    - Use with PUTS

```
LEA    R0, MSG
PUTS
```

```
MSG    .STRINGZ "Hello, World!"
```


CS 135



### Thought Question

- **Do we need all of these load instructions?**


CS 135



### Need LD?

<ul style="list-style-type: none"> <li>• LD R1, DATA</li> </ul>	<ul style="list-style-type: none"> <li>• LEA R2, DATA</li> <li>• LDR R1, R2, #0</li> </ul>
---	--

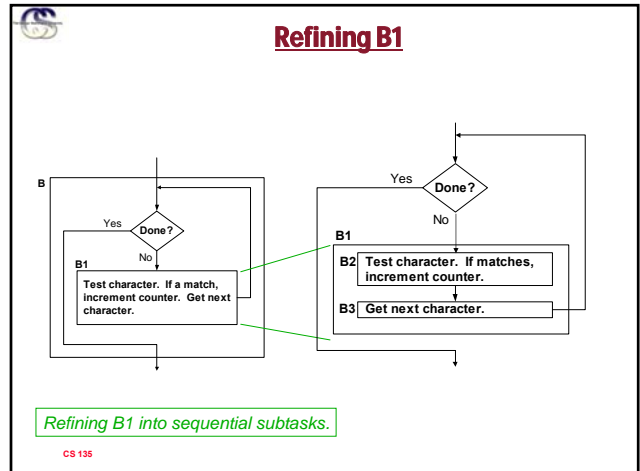
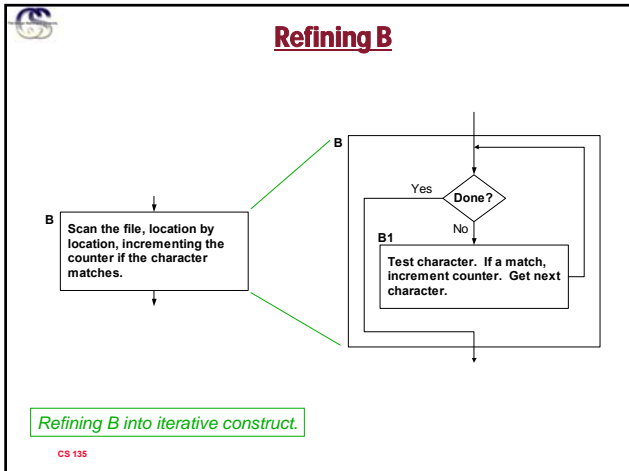
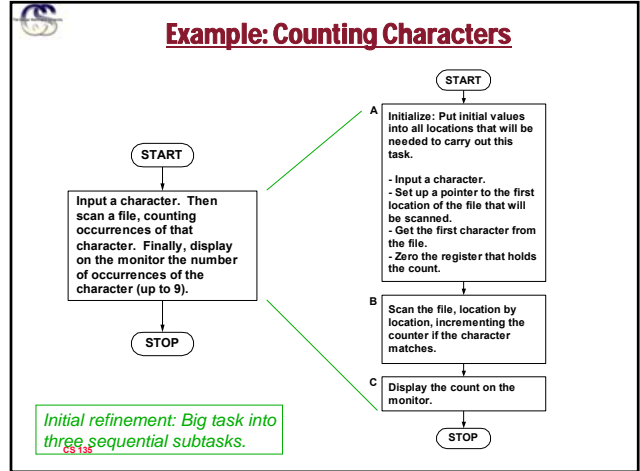
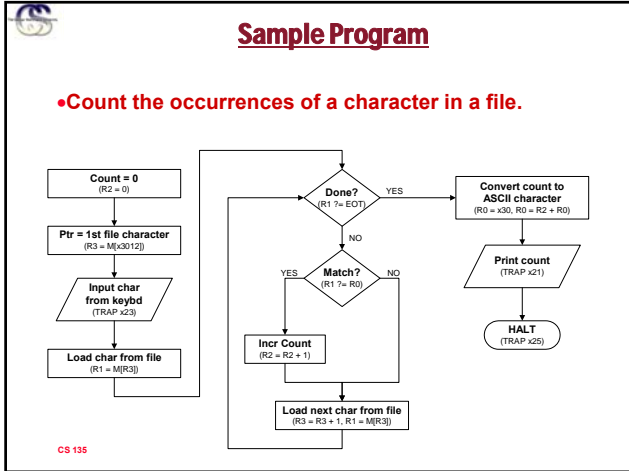
CS 135

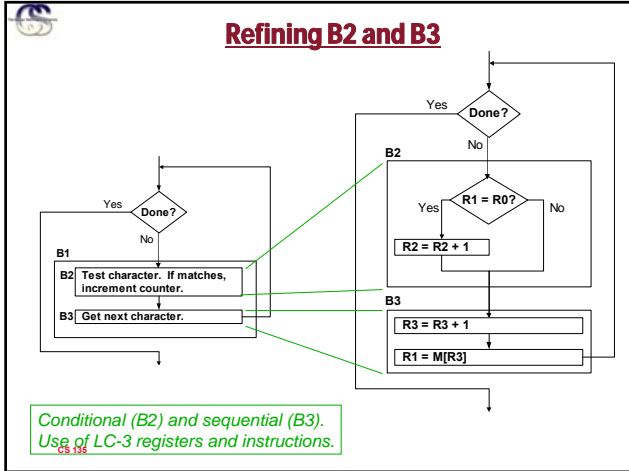


### Need LDI?

<ul style="list-style-type: none"> <li>• LDI R1, KBSR</li> <li>• KBSR .FILL xFE00</li> </ul>	<ul style="list-style-type: none"> <li>• LEA R2, KBSR</li> <li>• LDR R1, R2, #0</li> <li>• LDR R1, R1, #0</li> <li>• KBSR .FILL xFE00</li> </ul>
--	--

CS 135





### The Last Step: LC-3 Instructions

• Use comments to separate into modules and to document your code.

```

; Look at each char in file.
0001100001111100 ; is R1 = EOT?
0000010xxxxxxxxx ; if so, exit loop
; Check for match with R0.
1001001001100001 ; R1 = -char
000101000000001 ; R1 = R0 - char
000101xxxxxxxxxx ; no match, skip incr
00010100100001 ; R2 = R2 + 1
; Incr file ptr and get next char
000101101100001 ; R3 = R3 + 1
0110001011000000 ; R1 = M[R3]
  
```

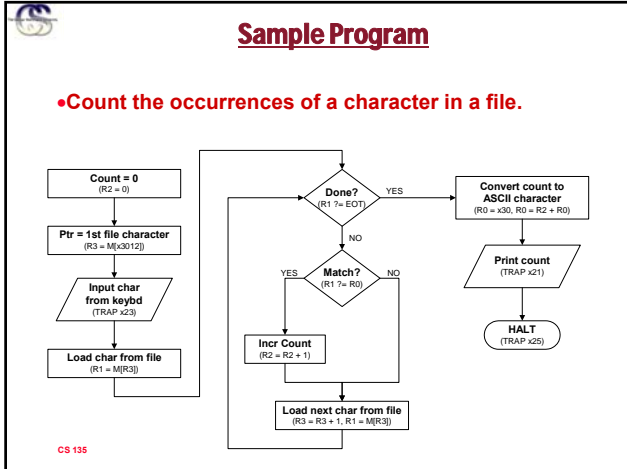
Don't know PCOffset bits until all the code is done

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			

+ Indicates instructions that modify condition codes

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					

+ Indicates instructions that modify condition codes



### 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
  
```

CS 135

### 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
  
```

CS 135

### Char Count in Assembly Language (3 of 3)

```

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

CS 135

