


# 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/)



## Programming

- **Low level/machine-level Programming**
  - Quick review of Stepwise Refinement and programming constructs
    - debugging

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



## 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.

CS 135

**Problem Statement**

- Because problem statements are written in English, they are sometimes ambiguous and/or incomplete.
- How do you resolve these issues?
  - Ask the person who wants the problem solved, or
  - Make a decision and document it.

CS 135

**Three Basic Constructs**

• There are three basic ways to decompose a task:

CS 135

**Sequential**

• Do Subtask 1 to completion, then do Subtask 2 to completion, etc.

CS 135

**Conditional**

• If condition is true, do Subtask 1; else, do Subtask 2.

CS 135

### Iterative

- Do Subtask over and over, as long as the test condition is true.

CS 135

### 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

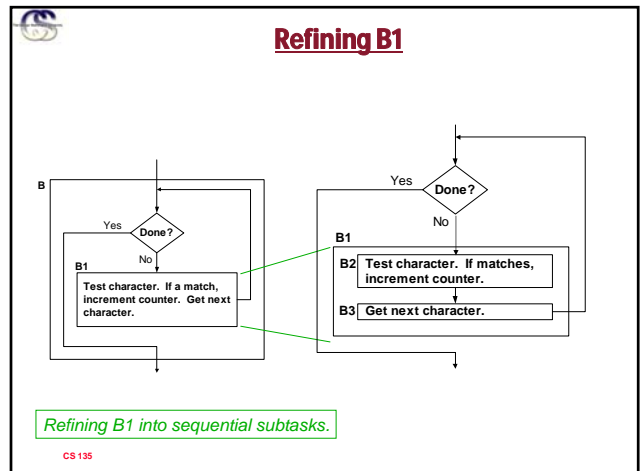
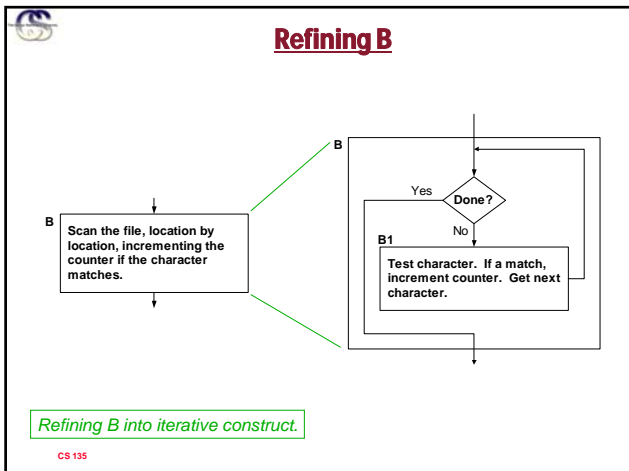
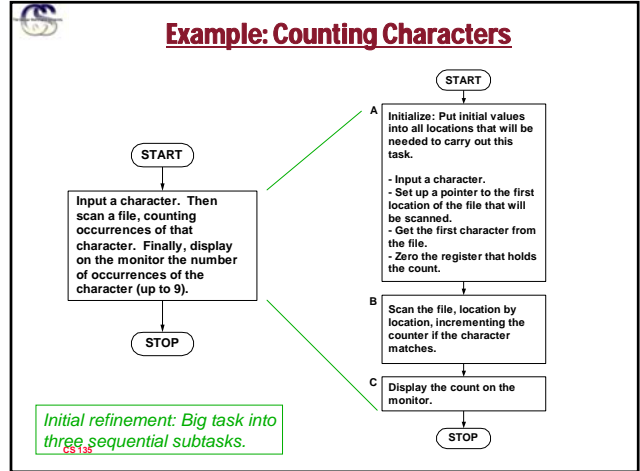
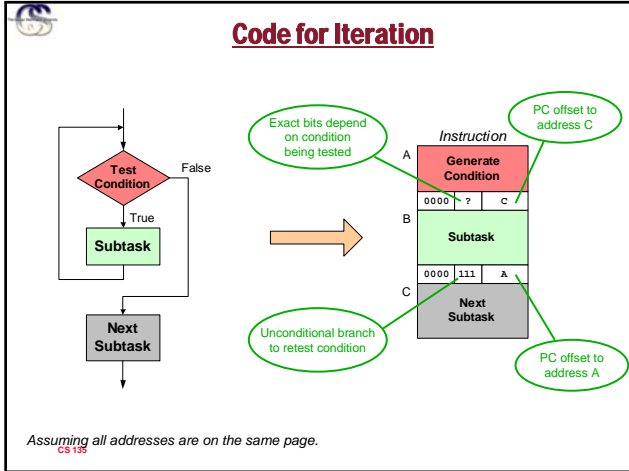
- How do we use LC-3 instructions to encode the three basic constructs?
- **Sequential**
  - > Instructions naturally flow from one to the next, so no special instruction needed to go from one sequential subtask to the next.
- **Conditional and Iterative**
  - > Create code that converts condition into N, Z, or P.
    - Example: “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.

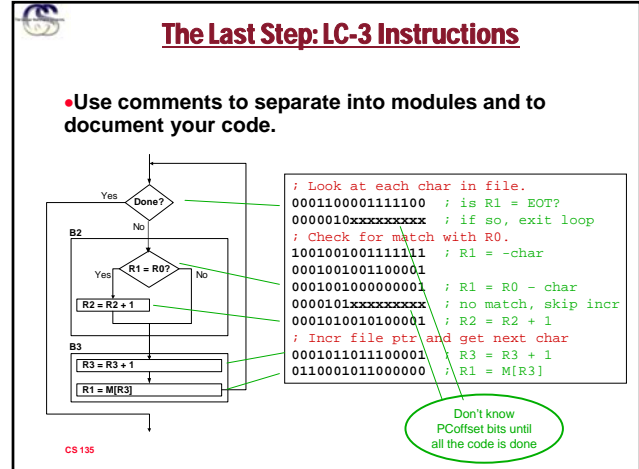
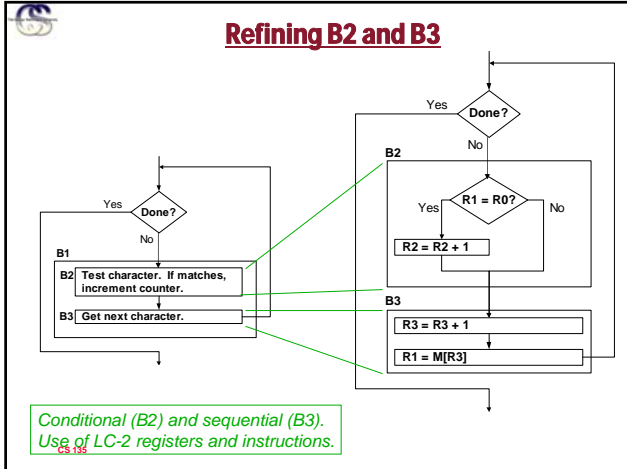
CS 135

### Code for Conditional

Assuming all addresses are close enough that PC-relative branch can be used.

CS 135





### Debugging

- You've written your program and it doesn't work.
- Now what?
- What do you do when you're lost in a city?
  - ✗ Drive around randomly and hope you find it?
  - ✓ Return to a known point and look at a map?
- ✓ In debugging, the equivalent to looking at a map is **tracing** your program.
  - Examine the sequence of instructions being executed.
  - Keep track of results being produced.
  - Compare result from each instruction to the expected result.

CS 135

### Debugging Operations

- Any debugging environment should provide means to:
  1. Display values in memory and registers.
  2. Deposit values in memory and registers.
  3. Execute instruction sequence in a program.
  4. Stop execution when desired.
- Different programming levels offer different tools.
  - > High-level languages (C, Java, ...) usually have source-code debugging tools.
  - > For debugging at the machine instruction level:
    - > simulators
    - > operating system "monitor" tools
    - > in-circuit emulators (ICE)
      - > plug-in hardware replacements that give instruction-level control

CS 135

### LC-3 Simulator

stop execution, set breakpoints

execute instruction sequences

set/display registers and memory

```

R0 x0000 0   R4 x0000 0   PC x3200 12800
R1 x0000 0   R5 x0000 0   PSR x8002 -3276
R2 x0000 0   R6 x0000 0   CC  Z
R3 x0000 0   R7 x0000 0
+ x3200 0101010010100000 x54A0  AND  R2, R2, #0
+ x3201 0001010010000100 x1484  ADD  R2, R2, R4
+ x3202 0001101101111111 x1B7F  ADD  R5, R5, #-1
+ x3203 0000011111111101 x07FD  BRZP x3201
+ x3204 1111000000100101 xF025  TRAP HALT
+ x3205 0000000000000000 x0000  NOP
+ x3206 0000000000000000 x0000  NOP
multiply.obj      0 instructions executed      idle
  
```

CS 135

### Types of Errors

- Syntax Errors
- Logic Errors
- Data Errors
  - Input data is different than what you expected.
  - Test the program with a wide variety of inputs.

CS 135

### Tracing the Program

- Execute the program one piece at a time, examining register and memory to see results at each step.
- Single-Stepping
  - Execute one instruction at a time.
  - Tedious, but useful to help you verify each step of your program.
- Breakpoints
  - Tell the simulator to stop executing when it reaches a specific instruction.
  - Check overall results at specific points in the program.
    - Lets you quickly execute sequences to get a high-level overview of the execution behavior.
    - Quickly execute sequences that you believe are correct.
- Watchpoints
  - Tell the simulator to stop when a register or memory location changes or when it equals a specific value.
  - Useful when you don't know where or when a value is changed.

CS 135

### Example 1: Multiply

- This program is supposed to multiply the two unsigned integers in R4 and R5.

```

clear R2
add R4 to R2
decrement R5
R5 = 0?
No -> add R4 to R2
Yes -> HALT
  
```

```

x3200 0101010010100000
x3201 0001010010000100
x3202 0001101101111111
x3203 0000011111111101
x3204 1111000000100101
  
```

Set R4 = 10, R5 = 3.  
Run program.  
Result: R2 = 40, not 30.

CS 135

## Debugging the Multiply Program

PC and registers at the beginning of each instruction

| PC    | R2 | R4 | R5 |
|-------|----|----|----|
| x3200 | -- | 10 | 3  |
| x3201 | 0  | 10 | 3  |
| x3202 | 10 | 10 | 3  |
| x3203 | 10 | 10 | 2  |
| x3201 | 10 | 10 | 2  |
| x3202 | 20 | 10 | 2  |
| x3203 | 20 | 10 | 1  |
| x3201 | 20 | 10 | 1  |
| x3202 | 30 | 10 | 1  |
| x3203 | 30 | 10 | 0  |
| x3201 | 30 | 10 | 0  |
| x3202 | 40 | 10 | 0  |
| x3203 | 40 | 10 | -1 |
| x3204 | 40 | 10 | -1 |
|       | 40 | 10 | -1 |

Single-stepping  
Breakpoint at branch (x3203)

| PC    | R2 | R4 | R5 |
|-------|----|----|----|
| x3203 | 10 | 10 | 2  |
| x3203 | 20 | 10 | 1  |
| x3203 | 30 | 10 | 0  |
| x3203 | 40 | 10 | -1 |
|       | 40 | 10 | -1 |

Should stop looping here!

Executing loop one time too many.  
Branch at x3203 should be based on Z bit only, not Z and P.

CS 135

## Debugging: Lessons

- Trace program to see what's going on.
  - Breakpoints, single-stepping
- When tracing, make sure to notice what's really happening, not what you think should happen.
- Test your program using a variety of input data.
  - Be sure to test extreme cases

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