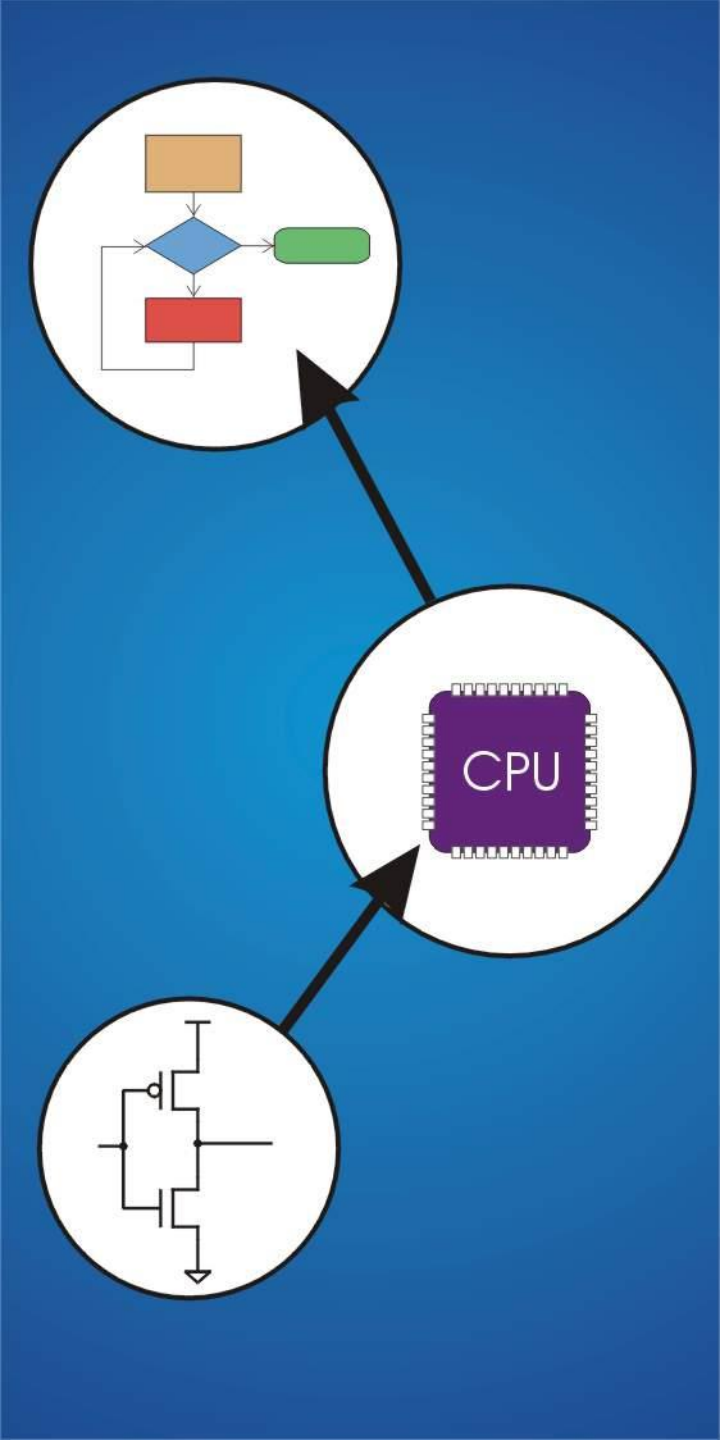


Chapter 6
Programming

Solving Problems using a Computer

Methodologies for creating a computer program that solves a given problem.

Problem Solving

- **Convert the problem statement into an algorithm, using *stepwise refinement***
- **Convert the algorithm into LC-3 machine instructions.**

Debugging

- **How do we figure out why the program didn't work?**
- **Examining registers and memory, setting breakpoints, etc.**

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

Stepwise Refinement

Also known as systematic decomposition.

Start with the 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.

Problem Statement

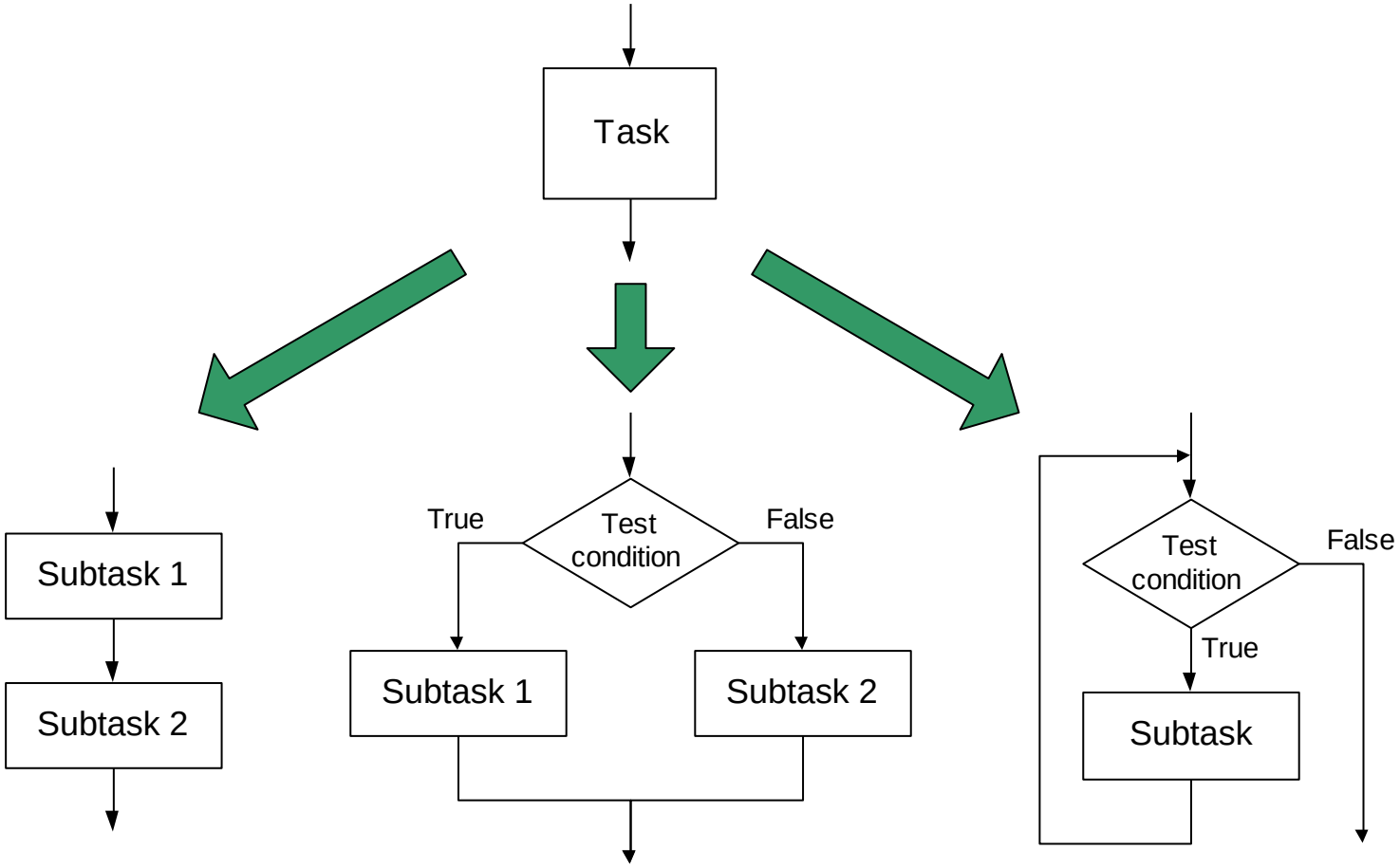
Because problem statements are written in English, they are sometimes ambiguous and/or incomplete.

- **Where is "file" located? How big is it, or how do I know when I've reached the end?**
- **How should the final count be printed? A decimal number?**
- **If the character is a letter, should I count both upper-case and lower-case occurrences?**

How do you resolve these issues?

- **Ask the person who wants the problem solved,**
- **Make a decision and document it, or**
- **Parameterize the issue and let the user decide when he/she runs the program.**

Stepwise Refinement



Sequential ***Conditional*** ***Iterative***

모두 single entry, single exit – implication?

Problem Solving Skills

Learn to convert problem statement into step-by-step description of subtasks.

- **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**

LC-3 Control Instructions

How do we use LC-3 instructions to express the three basic control structures?

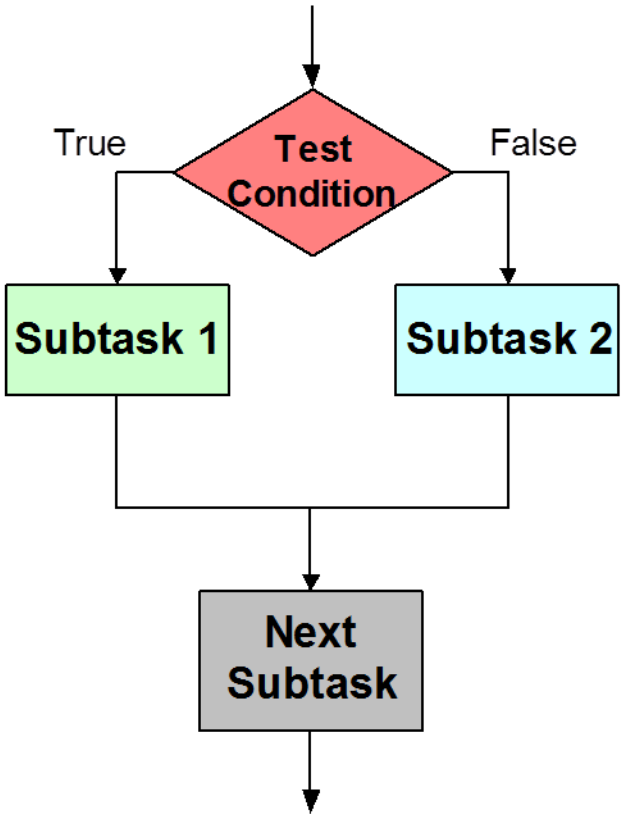
Sequential

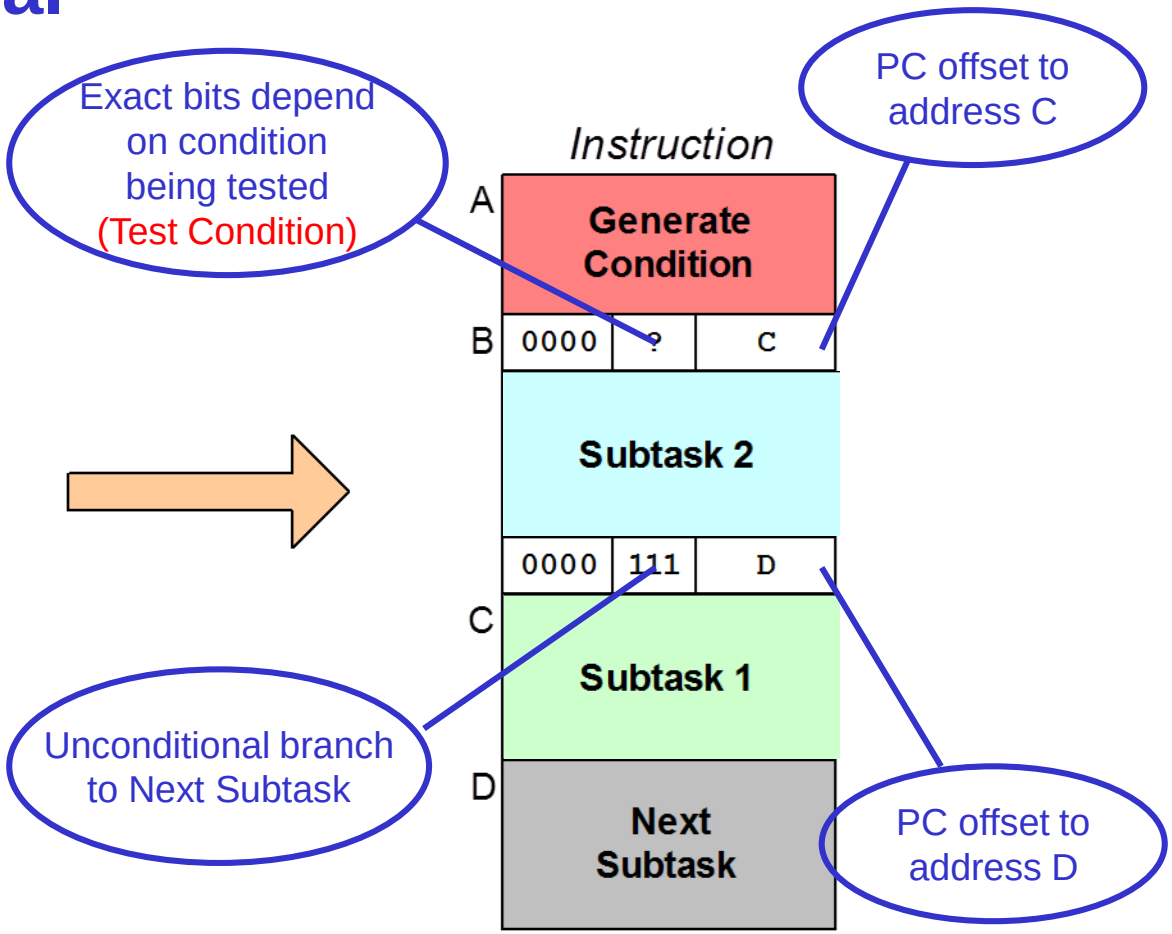
- **Instructions naturally flow from one to the next, so no special instruction needed to go from one sequential subtask to the next.**

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.**

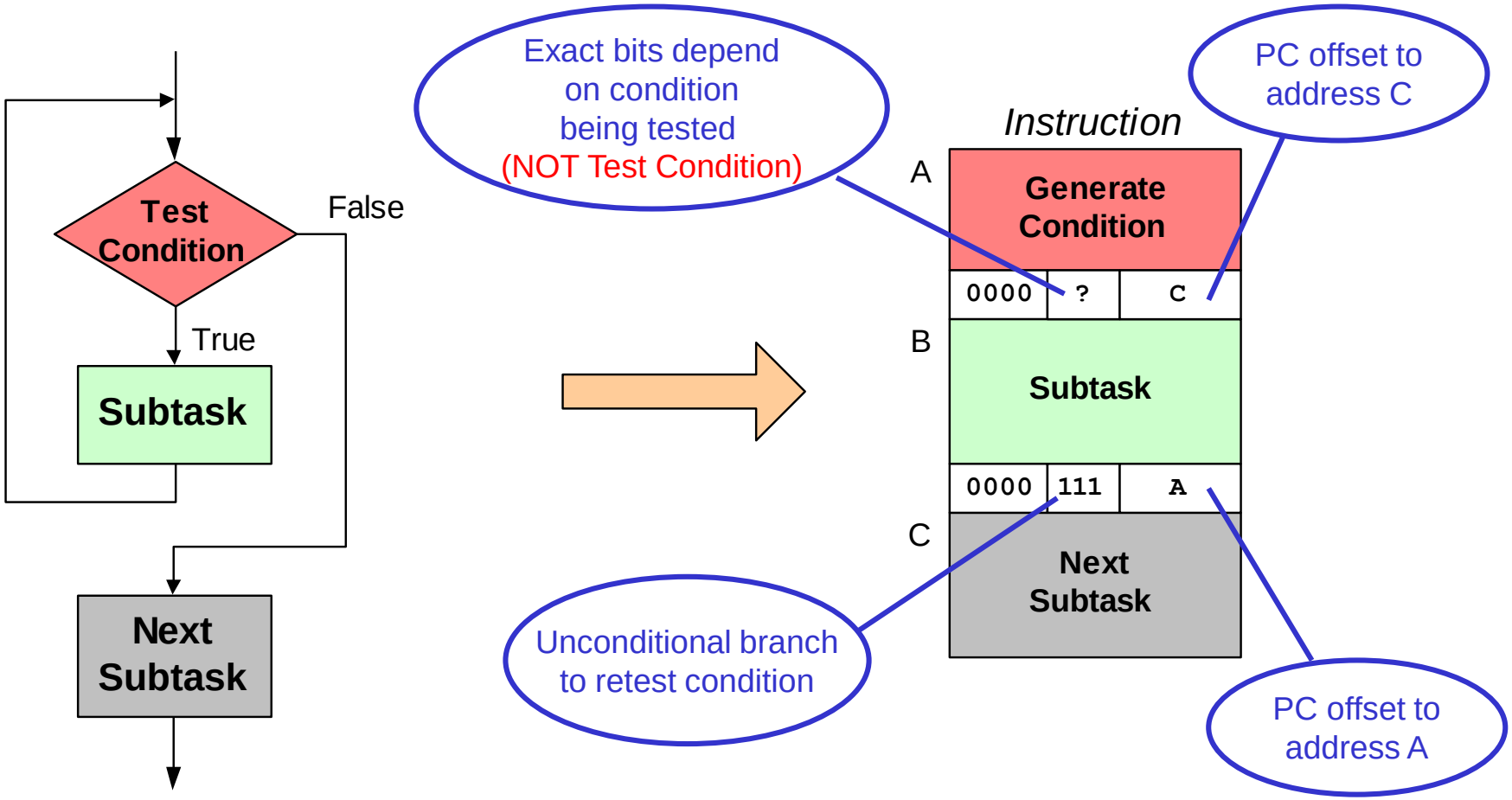
Code for Conditional





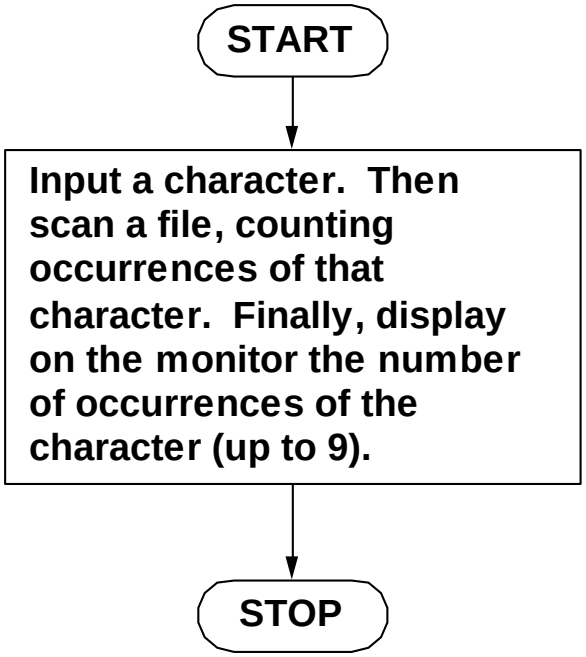
Assuming all addresses are close enough that PC-relative branch can be used.

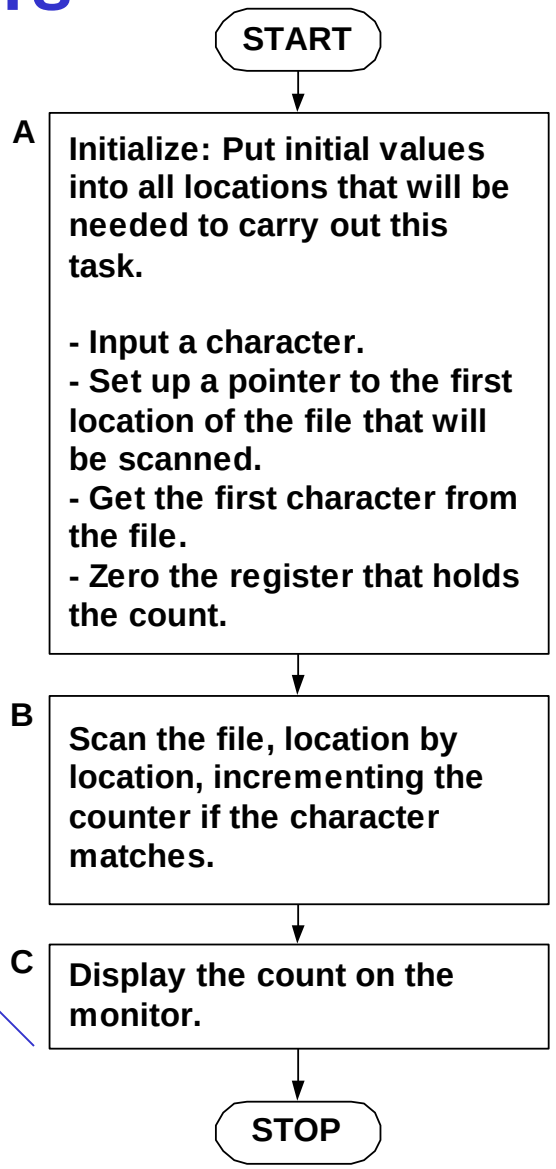
Code for Iteration



Assuming all addresses are on the same page.

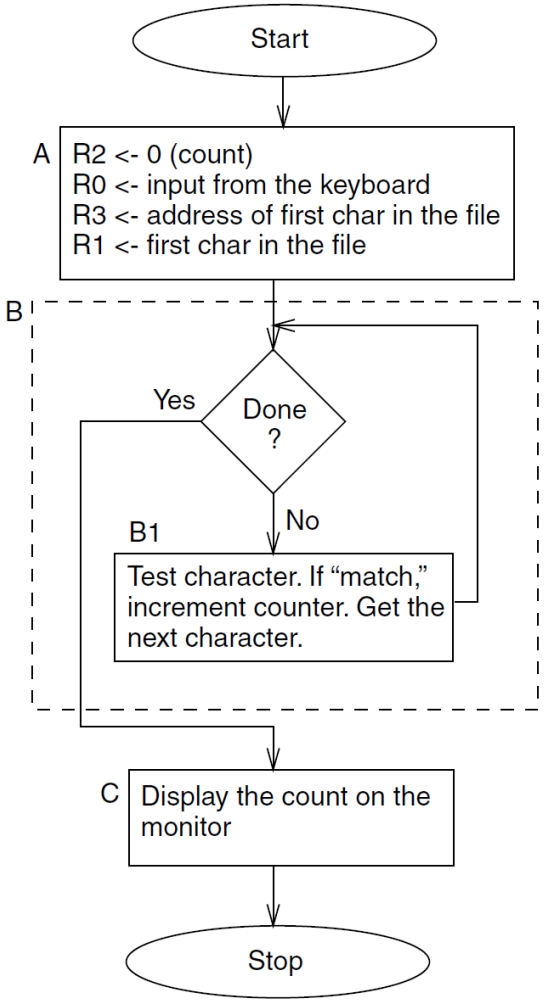
Example: Counting Characters



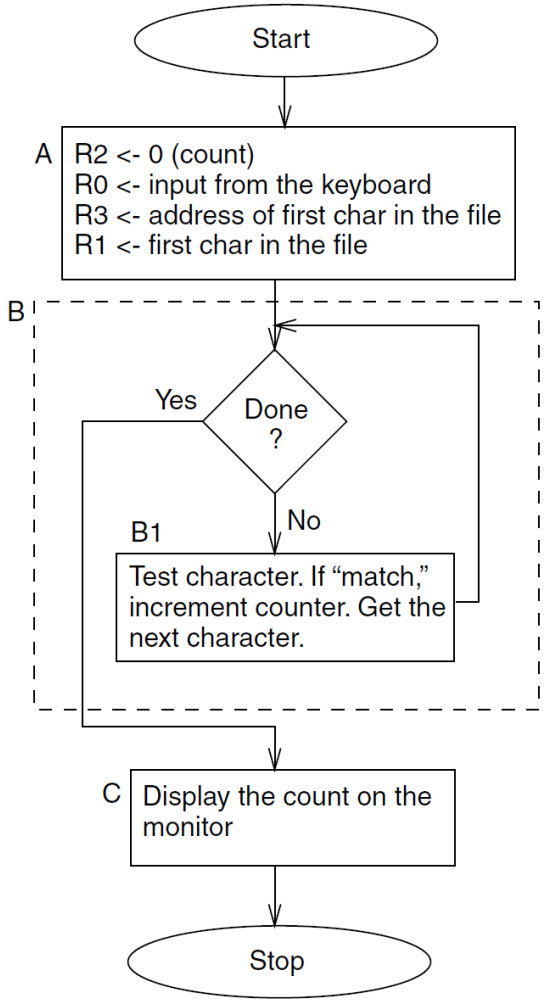


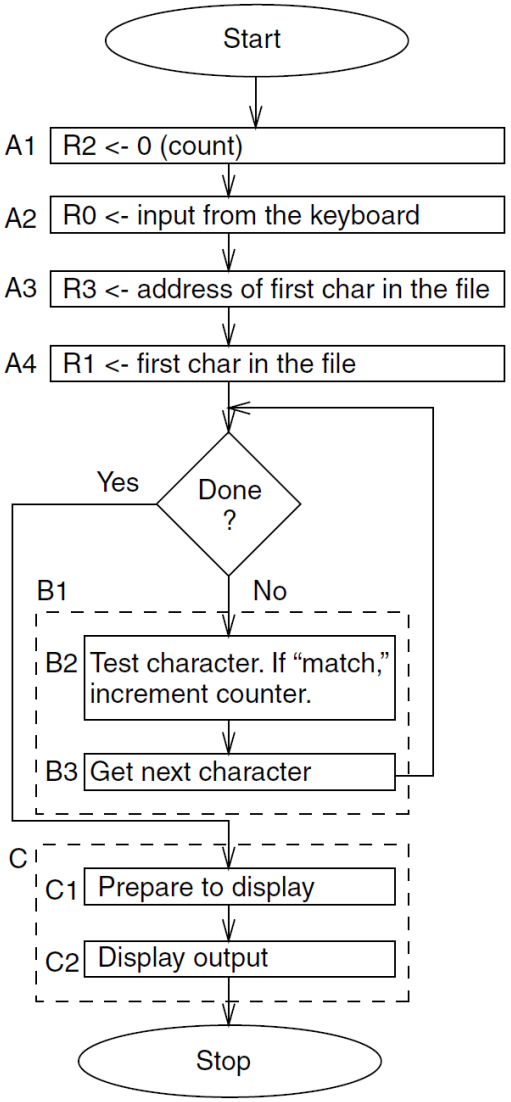
Initial refinement: Big task into three sequential subtasks.

Initial Code in LC-3 Instructions

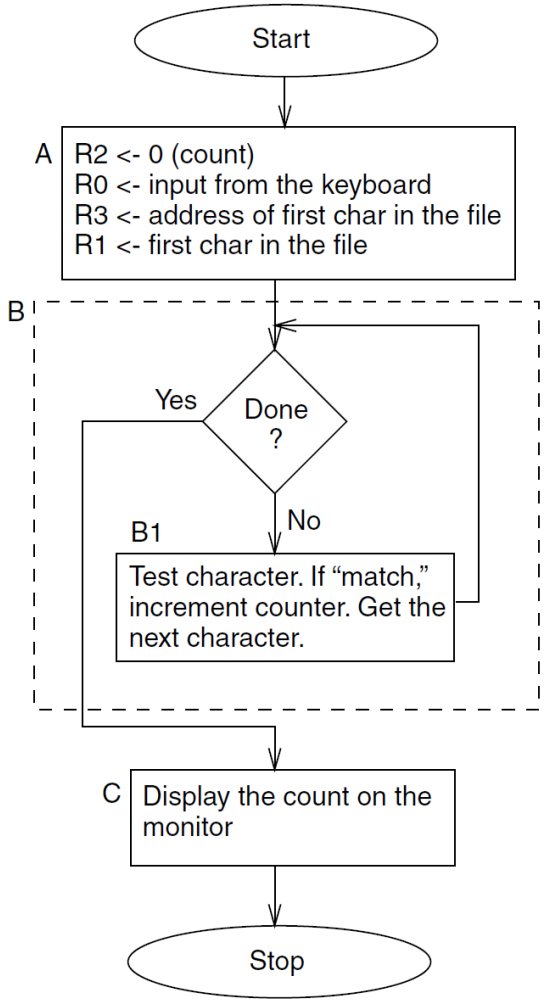


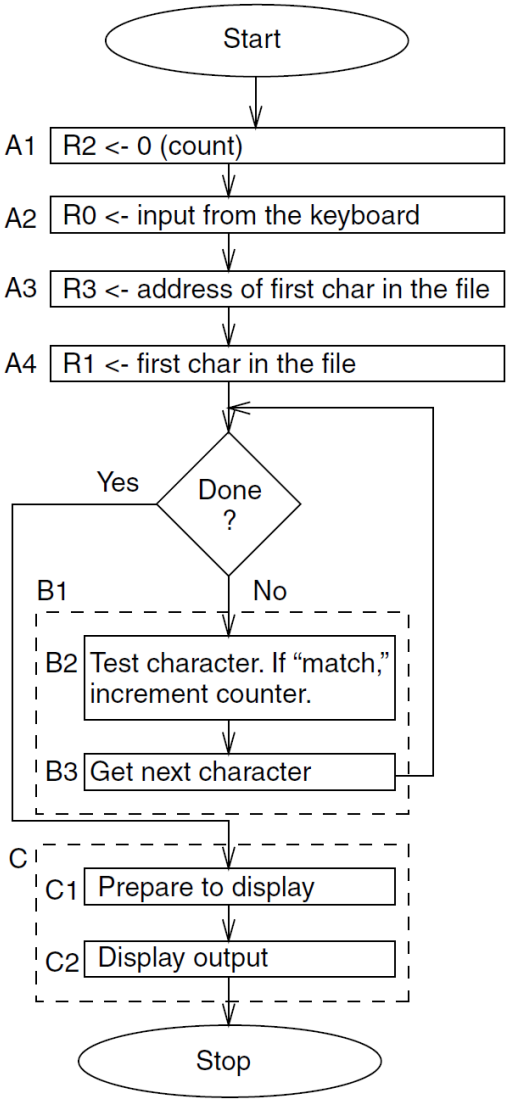
Stepwise Refinement

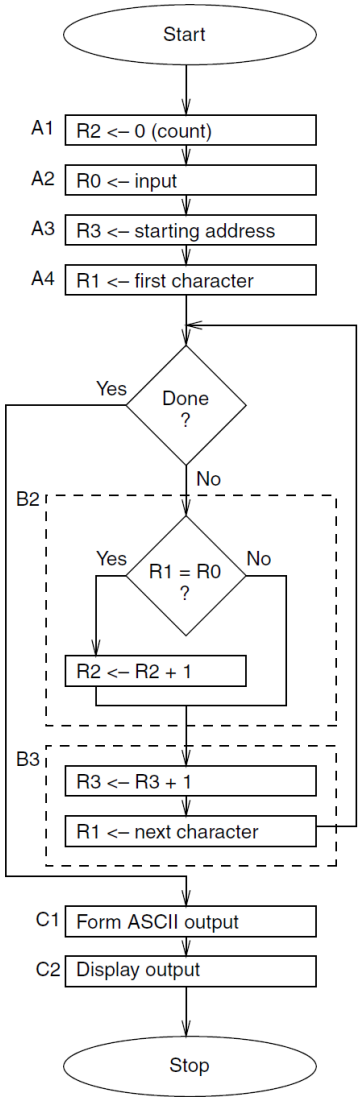




More Stepwise Refinement

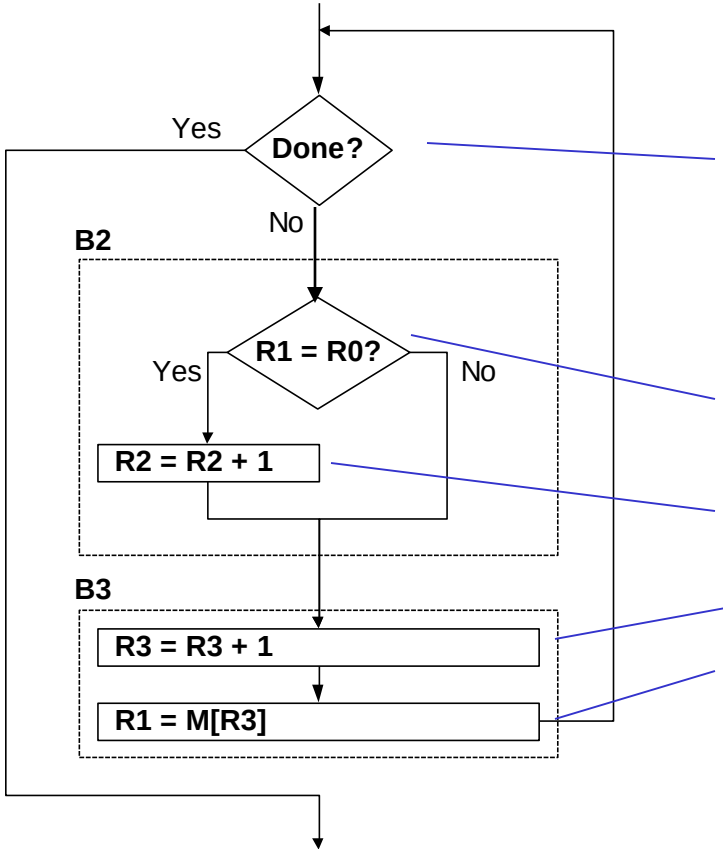






The Last Step: LC-3 Instructions

Use comments to separate into sections 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.
1001001001111111  ; R1 = -char
0001001001100001
0001001000000001  ; R1 = R0 - char
0000101xxxxxxxxx  ; no match, skip incr
0001010010100001  ; R2 = R2 + 1
; Incr file ptr and get next char
0001011011100001  ; R3 = R3 + 1
0110001011000000  ; R1 = M[R3]
```

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

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.**

Debugging Operations

Any debugging environment should provide ways 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**
  - **in-circuit emulators (ICE)**
    - **plug-in hardware replacements that give instruction-level control**

Types of Errors

Syntax Errors

- You made a typing error that resulted in an illegal operation.
- Not usually an issue with machine language, because almost any bit pattern corresponds to some legal instruction.
- In high-level languages, these are often caught during the translation from language to machine code.

Logic Errors

- Your program is legal, but does not perform the intended function, so the results don't match the problem statement.
- Trace the program to see what's really happening and determine how to get the proper behavior.

Data Errors

- Input data is different from what you expected.
- Make sure the assumptions about input data and test the program with a wide variety of inputs.

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.**
  - **To quickly execute instruction sequences to get a high-level overview of the execution behavior.**
  - **To skip instruction sequences that you believe are correct.**

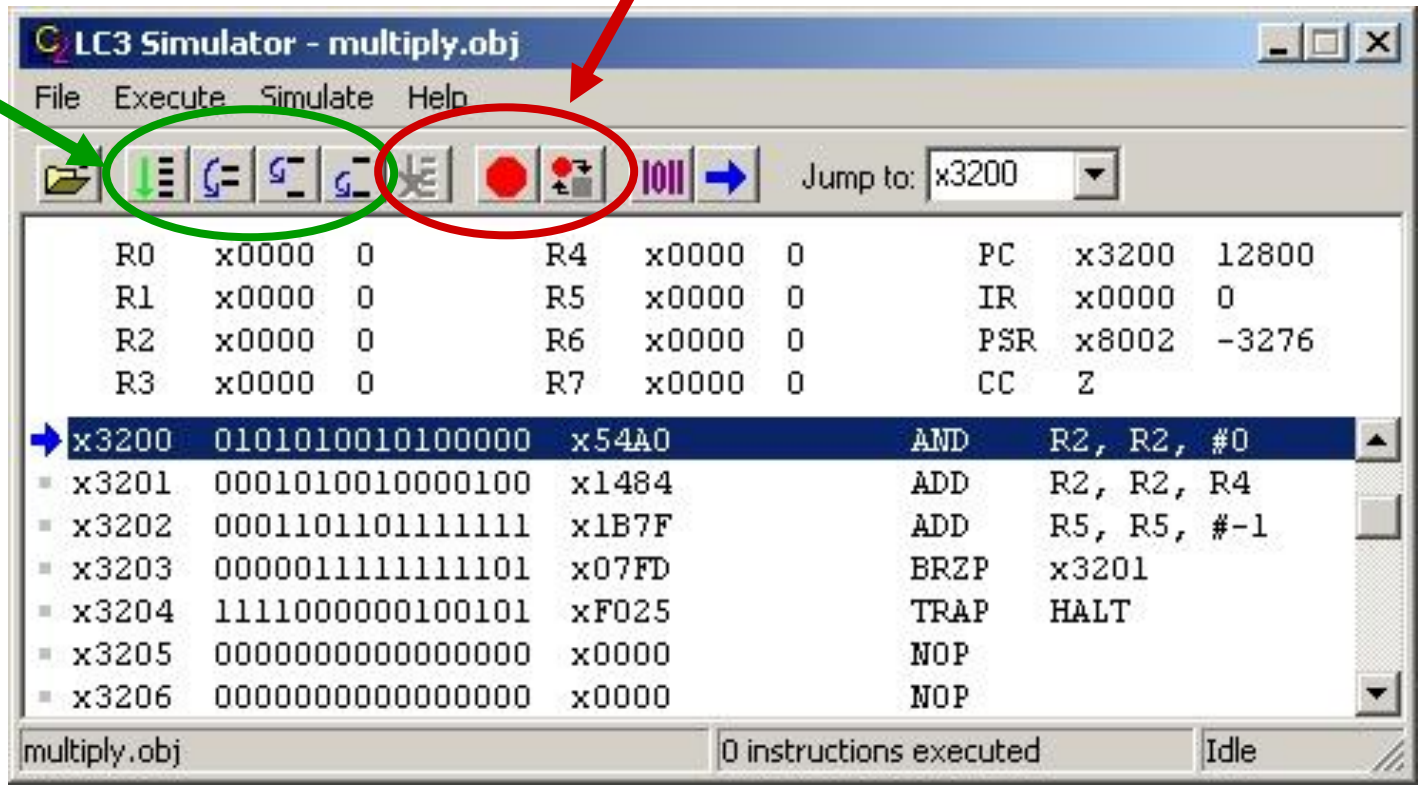
Watchpoints

- **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 the register or the memory location is changed.**

LC-3 Simulator

stop execution,
set breakpoints

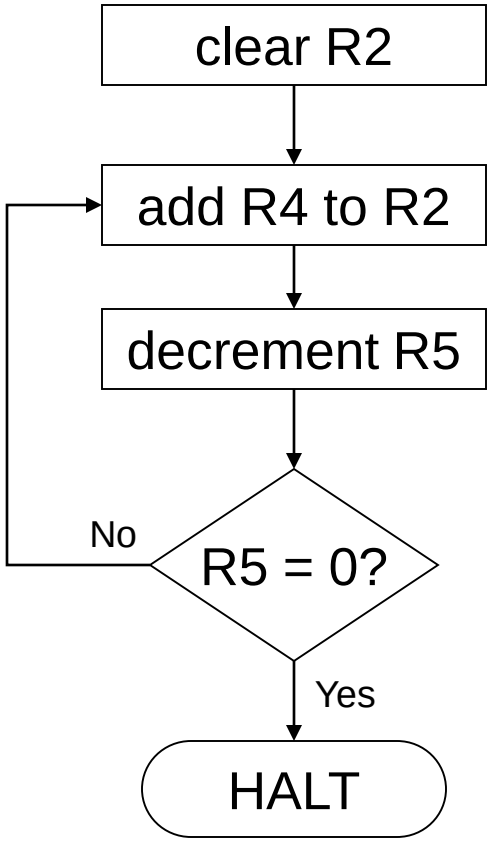
execute
instruction
sequences



set/display
registers
and memory

Example 1: Multiply

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




```
x3200 0101010010100000 R2 <- 0
x3201 0001010010000100 R2 <- R2 + R4
x3202 0001101101111111 R5 <- R5 - 1
x3203 0000011111111101 BRzp x3201
x3204 1111000000100101 HALT
```

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

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)

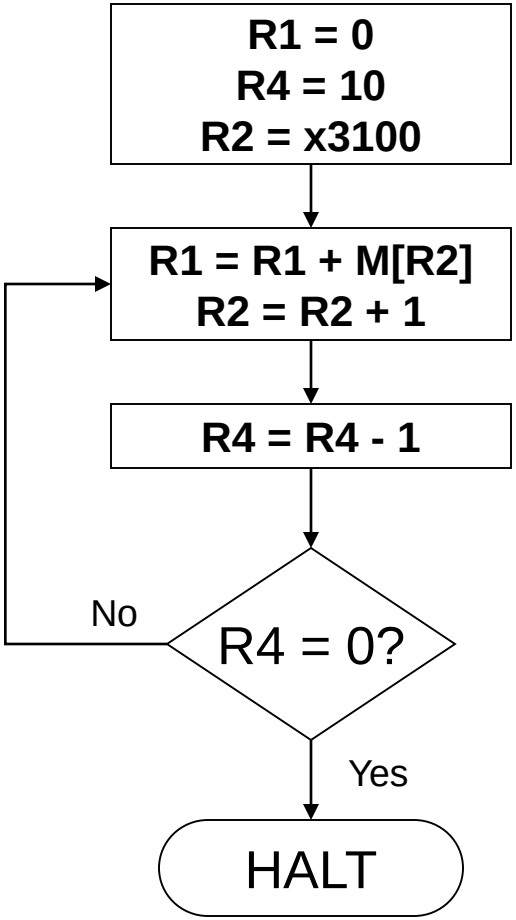
PC	R2	R4	R5
x3203	10	10	2
x3203	20	10	1
x3203	30	10	0
x3203	40	10	-1
	40	10	-1

Should stop looping here!

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

Example 2: Summing an Array of Numbers

This program is supposed to sum the numbers stored in 10 locations beginning with x3100, leaving the result in R1.




```
x3000 0101001001100000 R1 <- 0
x3001 0101100100100000 R4 <- 0
x3002 0001100100101010 R4 <- R4 + 10
x3003 0010010011111100 R2 <- M[x3100]
x3004 0110011010000000 R3 <- M[R2]
x3005 0001010010100001 R2 <- R2 + 1
x3006 0001001001000011 R1 <- R1 + R3
x3007 0001100100111111 R4 <- R4 - 1
x3008 0000001111111011 BRp x3004
x3009 1111000000100101 HALT
```

Debugging the Summing Program

Running the the data below yields R1 = x0024, but the sum should be x8135. What happened?

Address	*Contents*
x3100	x3107
x3101	x2819
x3102	x0110
x3103	x0310
x3104	x0110
x3105	x1110
x3106	x11B1
x3107	x0019
x3108	x0007
x3109	x0004

Start single-stepping program...

PC	R1	R2	R4
x3000	--	--	--
x3001	0	--	--
x3002	0	--	0
x3003	0	--	10
x3004	0	x3107	10

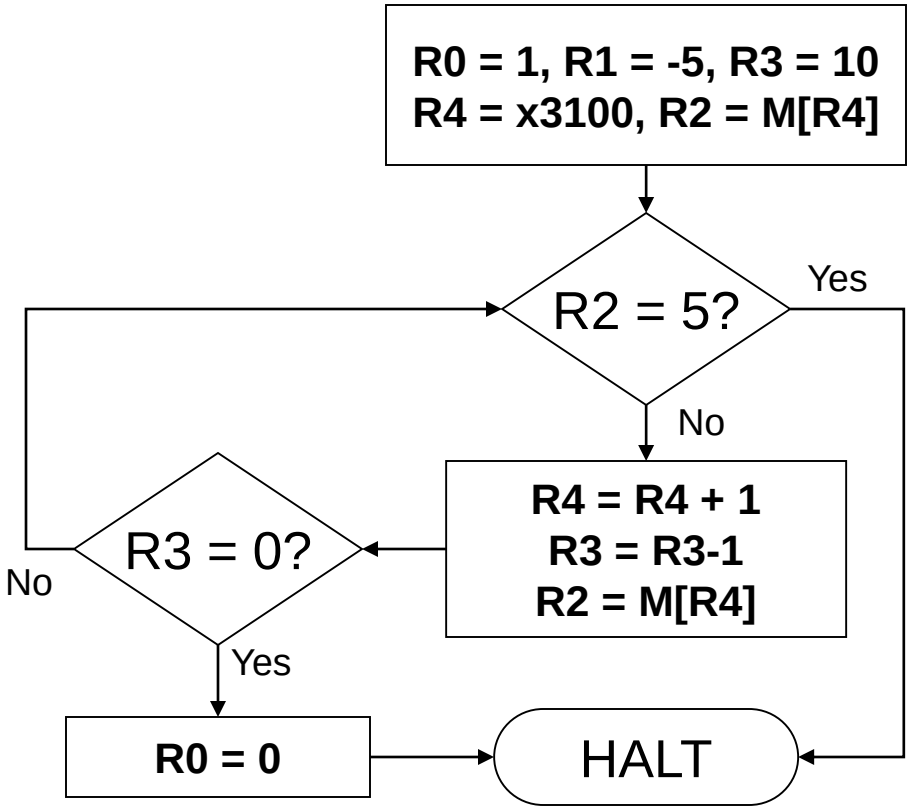
Should be x3100!

Program loads contents of M[x3100], not address.
Change the opcode of the instruction at x3003 from 0010 (LD) to 1110 (LEA).

Example 3: Looking for a 5

This program is supposed to set R0=1 if there's a 5 in one of the ten memory locations, starting at x3100.

Else, it should set R0 to 0.




```
x3000 0101000000100000 R0 <- 0
x3001 0001000000100001 R0 <- R0 + 1
x3002 0101001001100000 R1 <- 0
x3003 0001001001111011 R1 <- R1 - 5
x3004 0101011011100000 R3 <- 0
x3005 0001011011101010 R3 <- R3 + 10
x3006 0010100000001001 R4 <- M[x3010]
x3007 0110010100000000 R2 <- M[R4]
x3008 0001010010000001 R2 <- R2 + R1
x3009 0000010000000101 BRz x300F
x300A 0001100100100001 R4 <- R4 + 1
x300B 0001011011111111 R3 <- R3 - 1
x300C 0110010100000000 R2 <- M[R4]
x300D 0000001111111010 BRp x3008
x300E 0101000000100000 R0 <- 0
x300F 1111000000100101 HALT
x3010 0011000100000000 x3100
```

Debugging the Fives Program

Running the program with a 5 in location x3108 results in R0 = 0, not R0 = 1. What happened?

Address	*Contents*
x3100	9
x3101	7
x3102	32
x3103	0
x3104	-8
x3105	19
x3106	6
x3107	13
x3108	5
x3109	61

Perhaps we didn't look at all the data?
Put a breakpoint at x300D to see how many times we branch back.

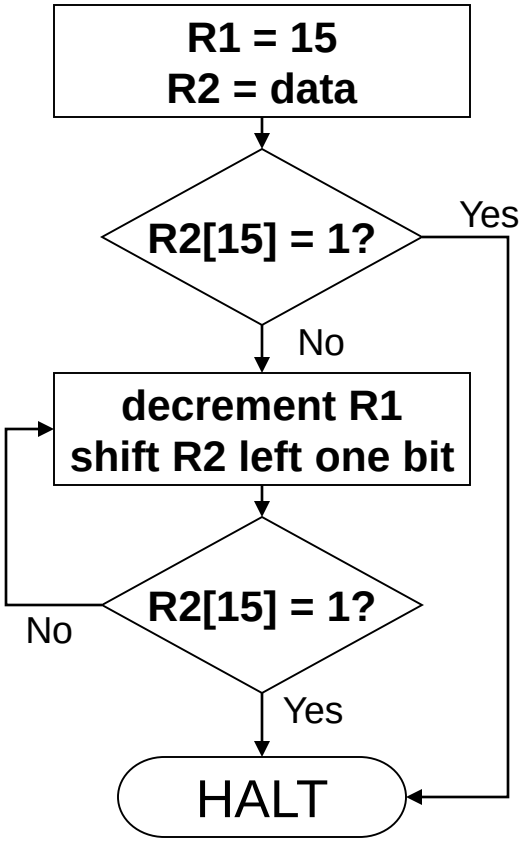
PC	R0	R2	R3	R4
x300D	1	7	9	x3101
x300D	1	32	8	x3102
x300D	1	0	7	x3103
	0	0	7	x3103

← Didn't branch back, even though R3 > 0?

Branch incorrectly uses condition code set by loading R2 with M[R4], not by decrementing R3.
Swap x300B and x300C, or remove x300C and branch back to x3007.

Example 4: Finding First 1 in a Word

This program is supposed to return (in R1) the bit position of the first 1 in a word. The address of the word is in location x3009 (just past the end of the program). If there are no ones, R1 should be set to –1.




```
x3000 0101001001100000 R1 <- 0
x3001 0001001001101111 R1 <- R1 + 15
x3002 1010010000000110 R2 <- M[M[x3009]]
x3003 0000100000000100 BRn x3008
x3004 0001001001111111 R1 <- R1 - 1
x3005 0001010010000010 R2 <- R2 + R2
x3006 0000100000000001 BRn x3008
x3007 0000111111111100 BRnzp x3004
x3008 1111000000100101 HALT
x3009 0011000100000000 x3100
```

Debugging the First-One Program

Program works most of the time, but if data is zero, it never seems to HALT.

Breakpoint at backwards branch (x3007)

PC	R1
x3007	14
x3007	13
x3007	12
x3007	11
x3007	10
x3007	9
x3007	8
x3007	7
x3007	6
x3007	5

PC	R1
x3007	4
x3007	3
x3007	2
x3007	1
x3007	0
x3007	-1
x3007	-2
x3007	-3
x3007	-4
x3007	-5

If no ones, then branch to HALT never occurs!

This is called an “infinite loop.”

Must change algorithm to either

(a) check for special case (R2=0), or

(b) exit loop if R1 < 0.

Debugging: Lessons Learned

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.

- **In summing program, it would be easy not to notice that the value at x3100 was loaded instead of x3100.**

Test your program using a variety of input data.

- **In Examples 3 and 4, the program works for many data sets.**
- **Be sure to test extreme cases (all ones, no ones, ...).**

꼭 기억해야 할 것

- Stepwise refinements by repetitive/nested/recursive applications of three control structures
  - Sequential
  - Conditional
  - Iterative
- Mechanical Translation of the control structures to (LC-3) instructions
- Debugging – Art rather than Science or Engineering but there are many useful tools