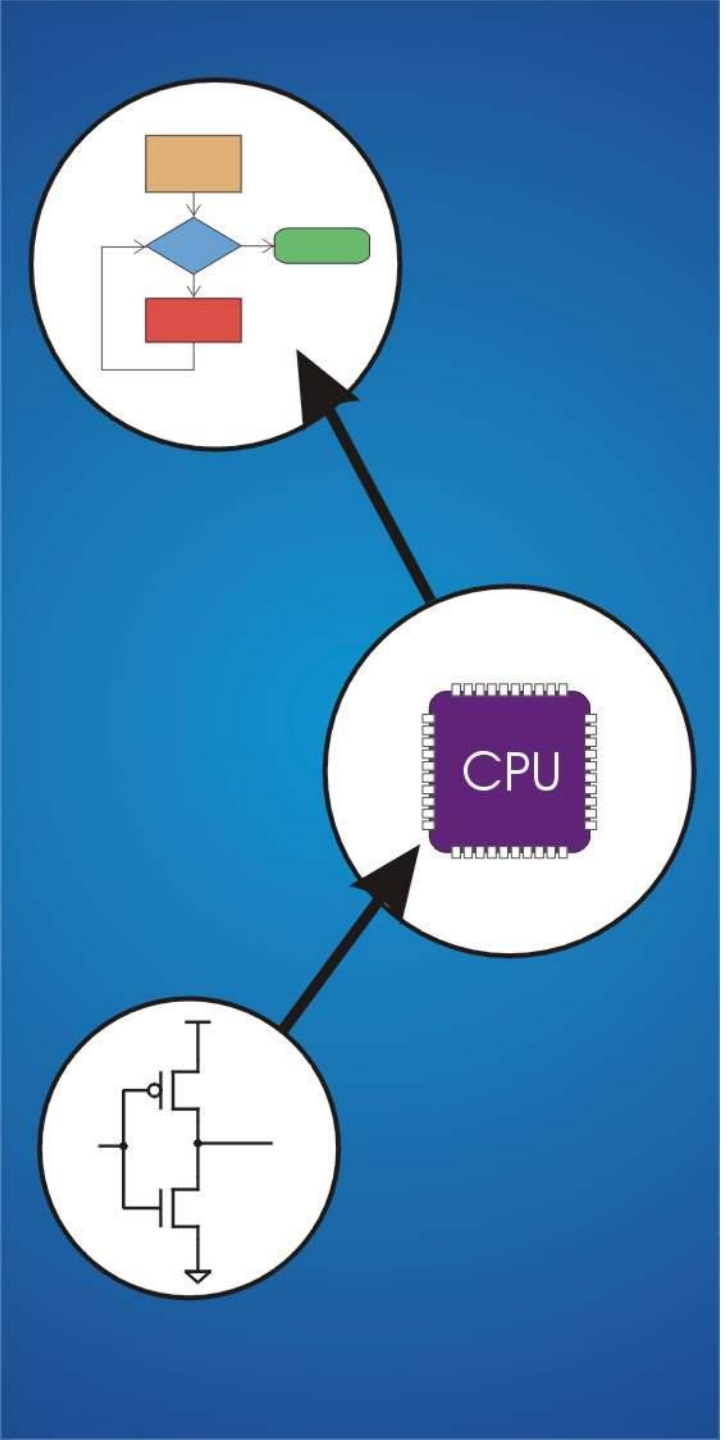


Chapter 6
Programming

Most of the material is left for your to study yourself.

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!

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.

Problem Statement

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

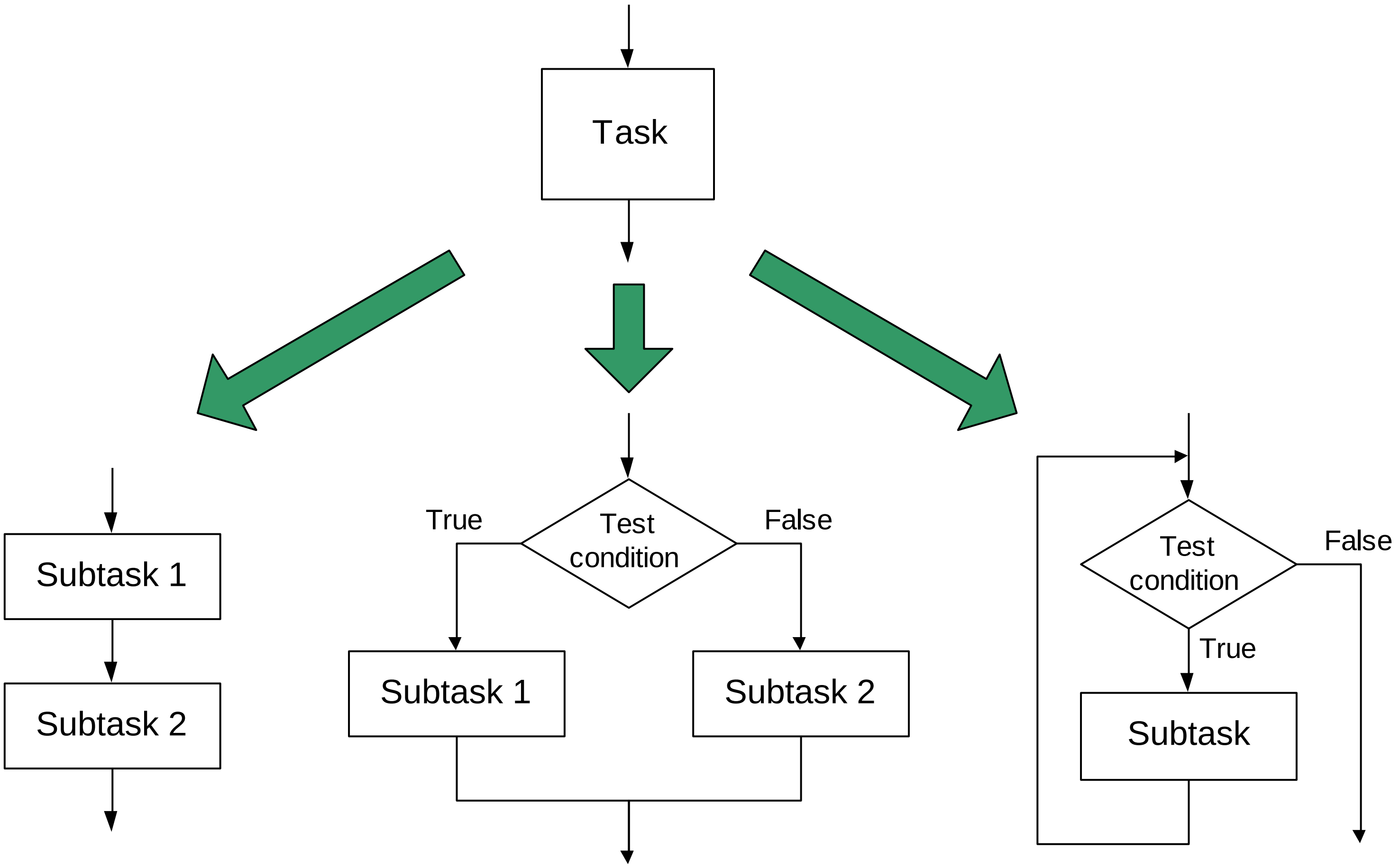
- **Where is “file” located? How big is it, or how do I know when I’ve reached the end?**
- **How should final count be printed? A decimal number?**
- **If the character is a letter, should I count both upper-case and lower-case occurrences?**

How do you resolve these issues?

- **Ask the person who wants the problem solved, or**
- **Make a decision and document it.**

Three Basic Constructs

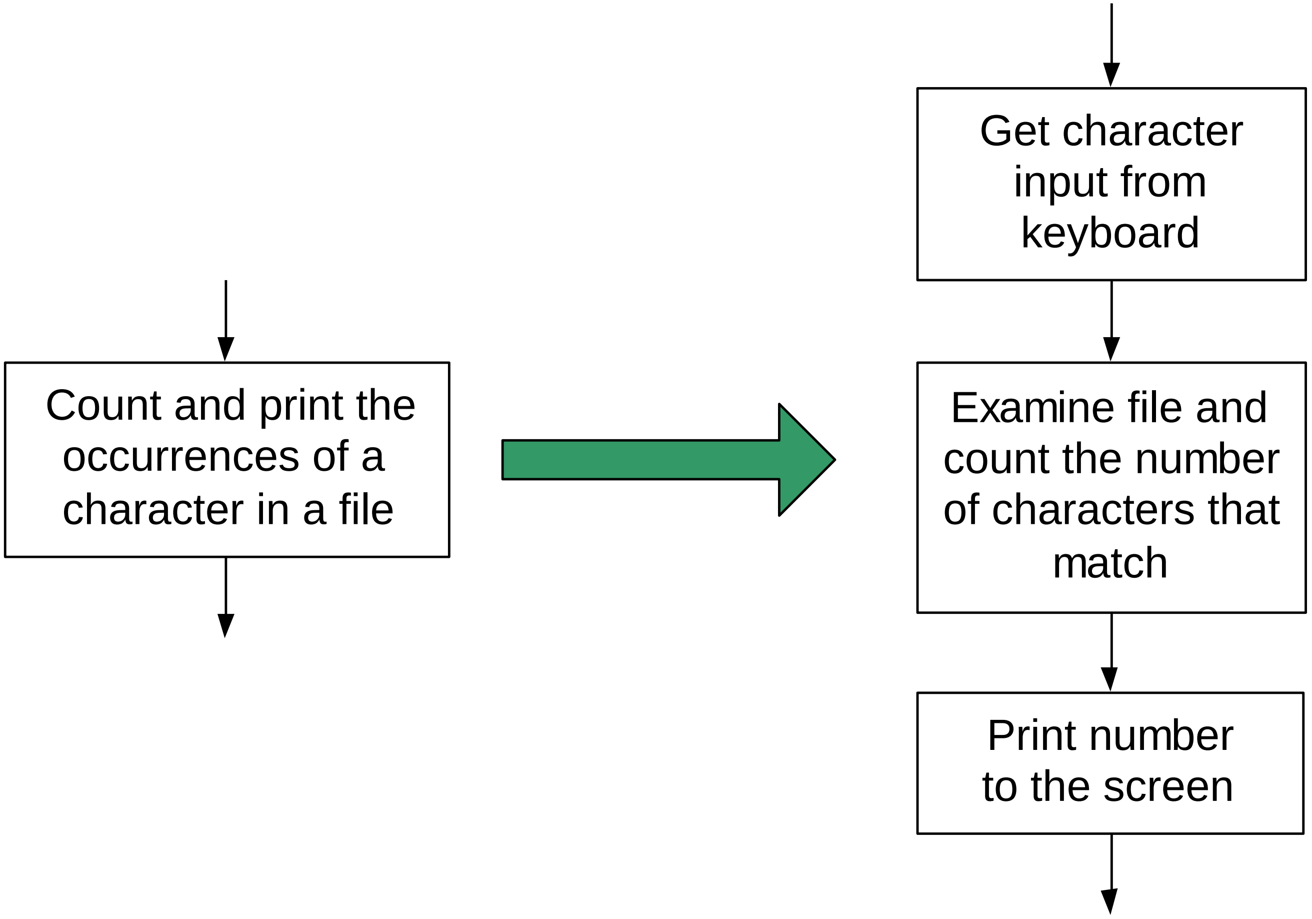
There are three basic ways to decompose a task:



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

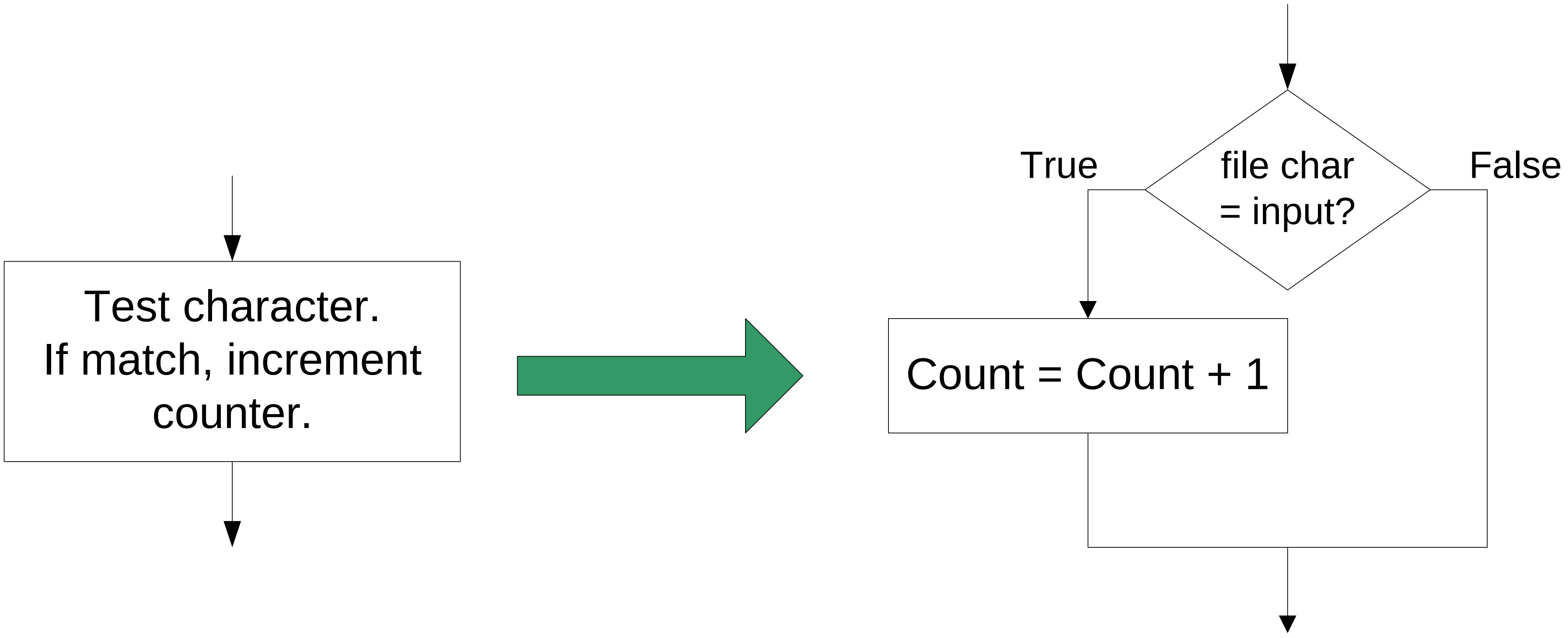
Sequential

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


Count and print the occurrences of a character in a file
Get character input from keyboard
Examine file and count the number of characters that match
Print number to the screen

Conditional

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



Test character.
If match, increment counter.

True

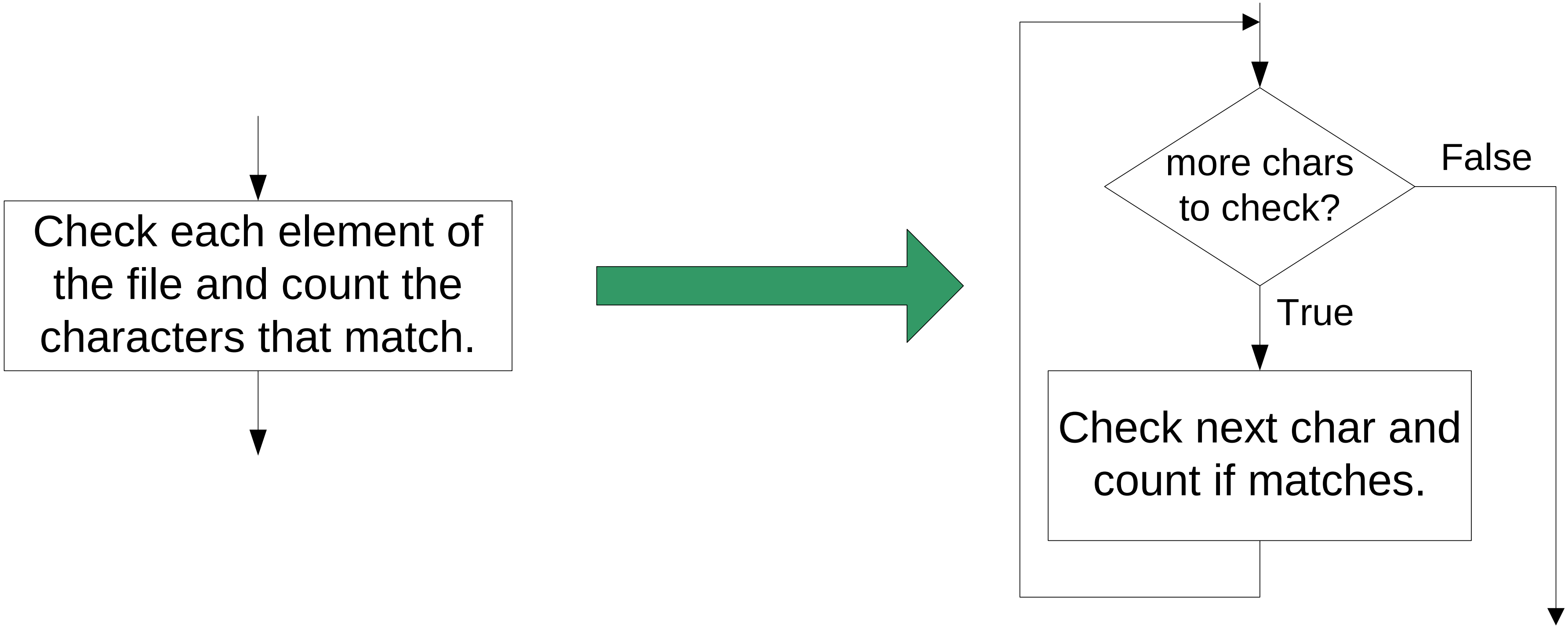
file char = input?

False

Count = Count + 1

Iterative

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


Check each element of the file and count the characters that match.
more chars to check?
False
True
Check next char and count if matches.

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**

LC-3 Control Instructions

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

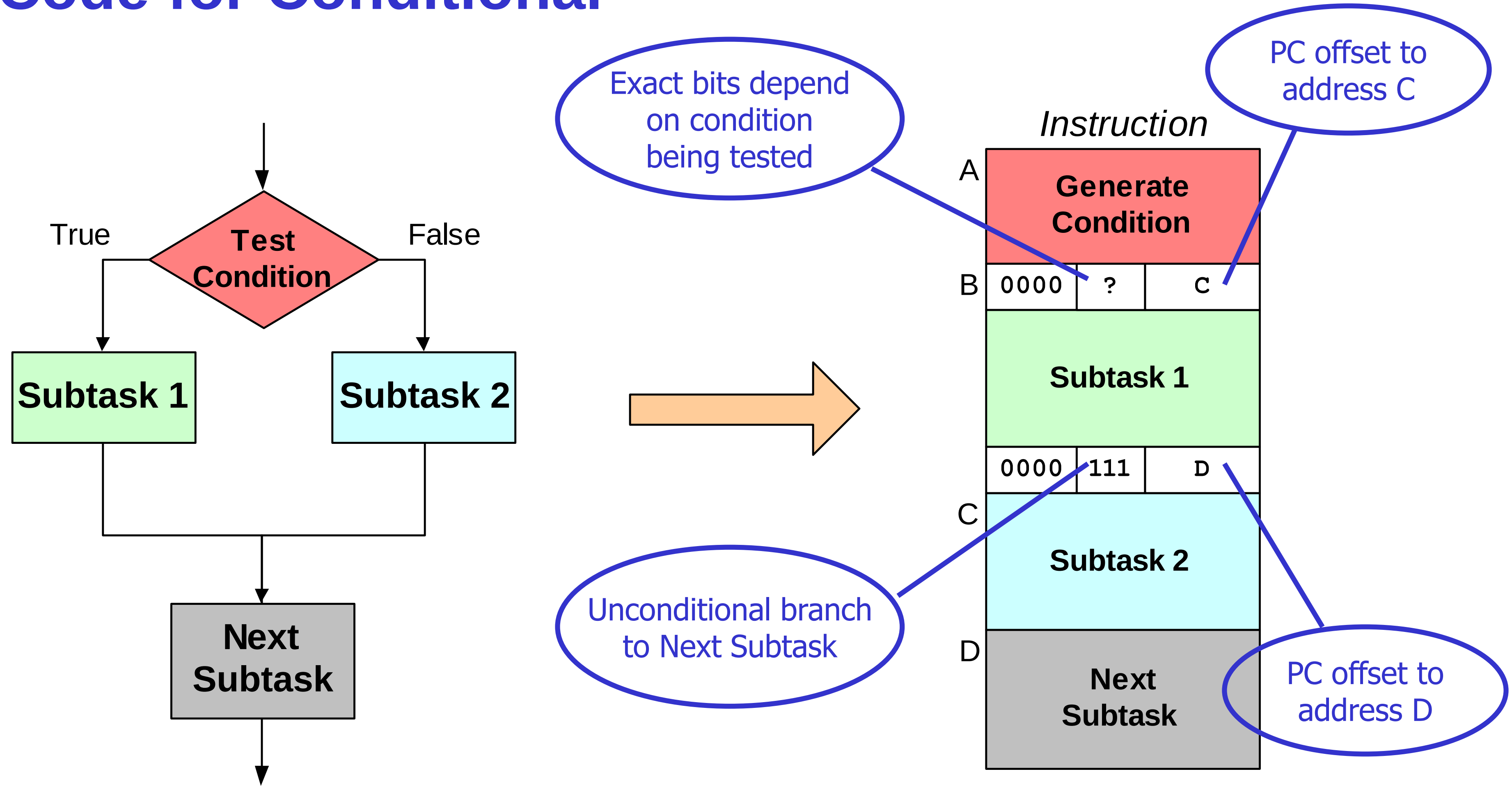
Sequential

- 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.

Code for Conditional



Exact bits depend on condition being tested

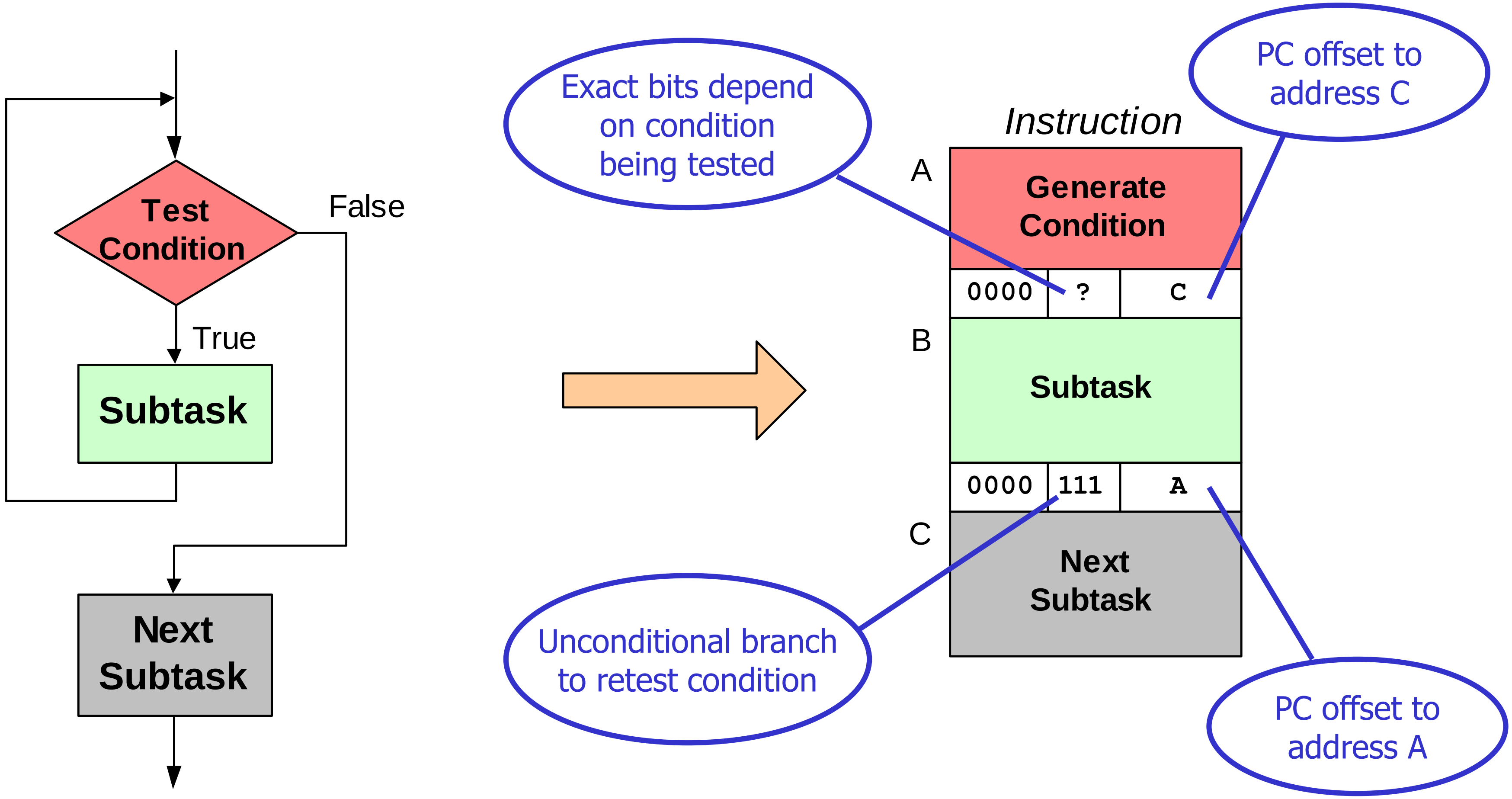
PC offset to address C

Unconditional branch to Next Subtask

PC offset to address D

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

Code for Iteration



Exact bits depend on condition being tested

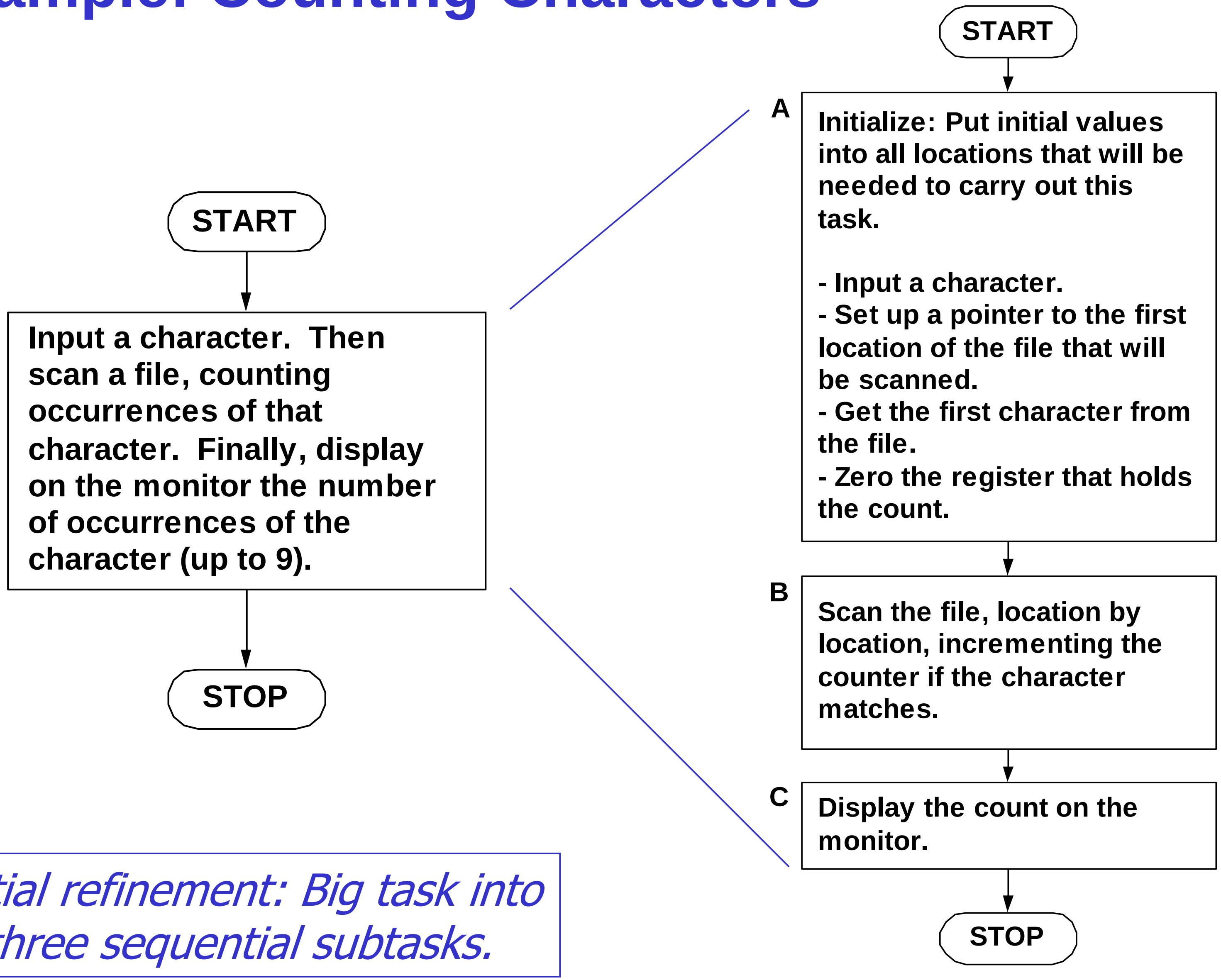
PC offset to address C

Unconditional branch to retest condition

PC offset to address A

Assuming all addresses are on the same page.

Example: Counting Characters



START

Input a character. Then scan a file, counting occurrences of that character. Finally, display on the monitor the number of occurrences of the character (up to 9).

STOP

START

A

Initialize: Put initial values into all locations that will be needed to carry out this task.

- Input a character.
- Set up a pointer to the first location of the file that will be scanned.
- Get the first character from the file.
- Zero the register that holds the count.

B

Scan the file, location by location, incrementing the counter if the character matches.

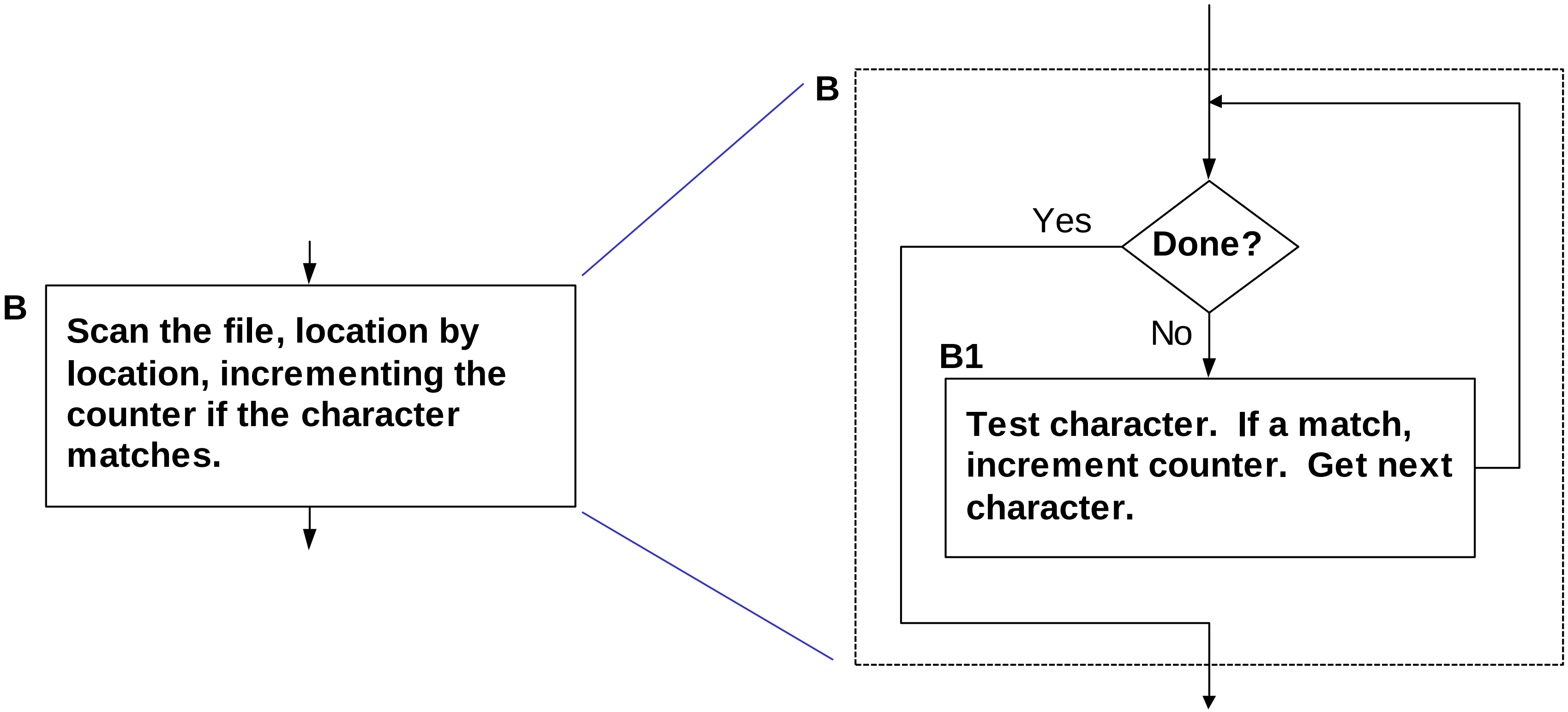
C

Display the count on the monitor.

STOP

Initial refinement: Big task into three sequential subtasks.

Refining B



B

Scan the file, location by location, incrementing the counter if the character matches.

B

Yes

Done?

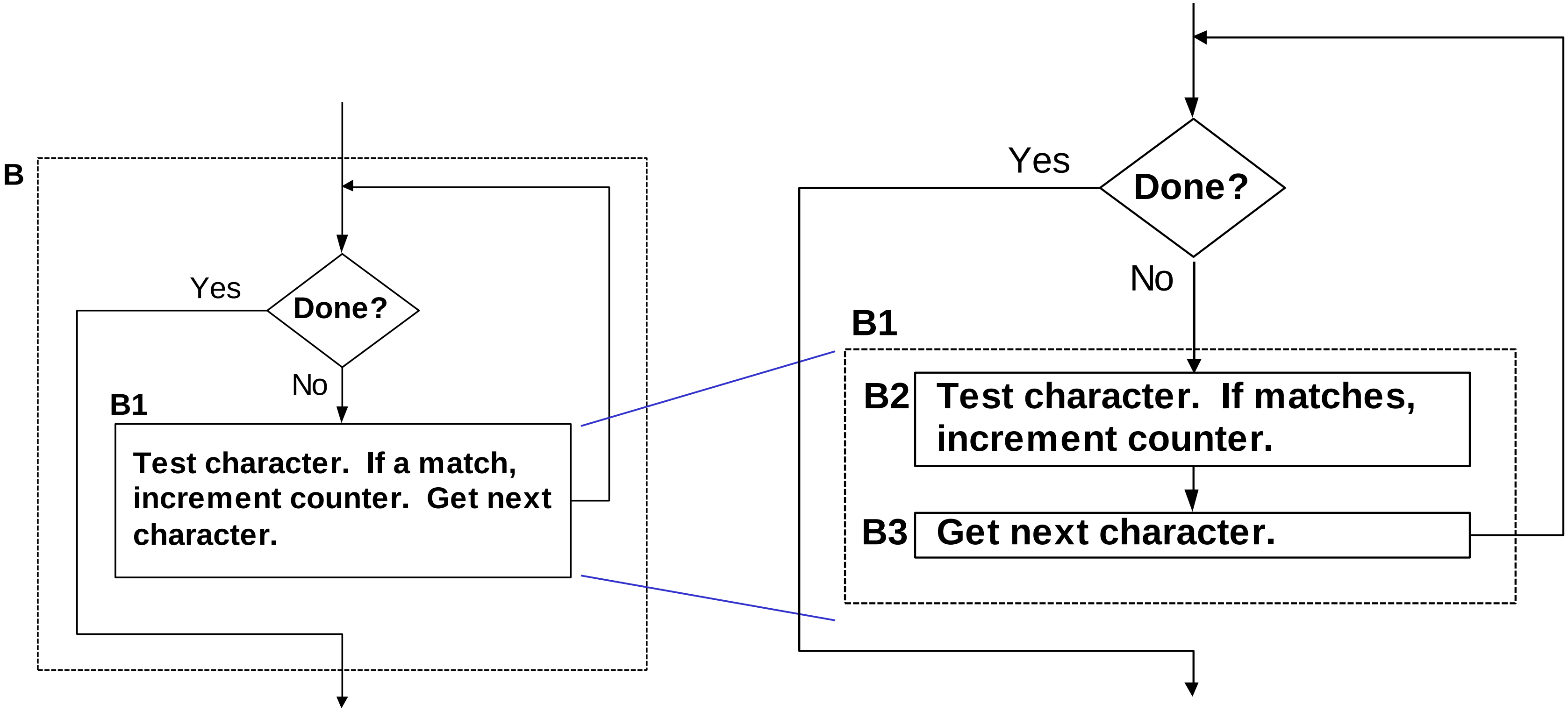
No

B1

Test character. If a match, increment counter. Get next character.

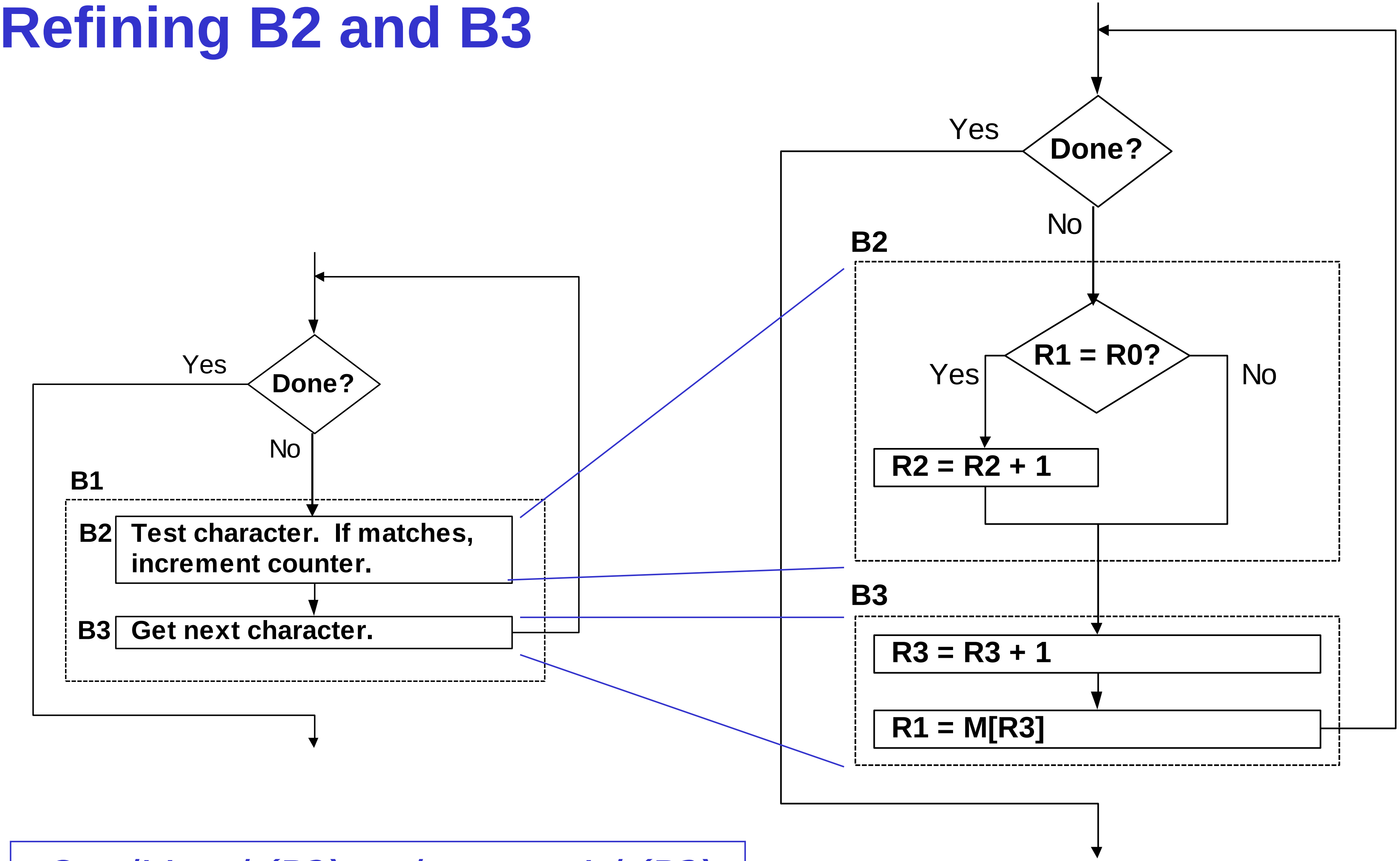
Refining B into iterative construct.

Refining B1


B
Yes
Done?
No
B1
Test character. If a match, increment counter. Get next character.
Yes
Done?
No
B1
B2
Test character. If matches, increment counter.
B3
Get next character.

Refining B1 into sequential subtasks.

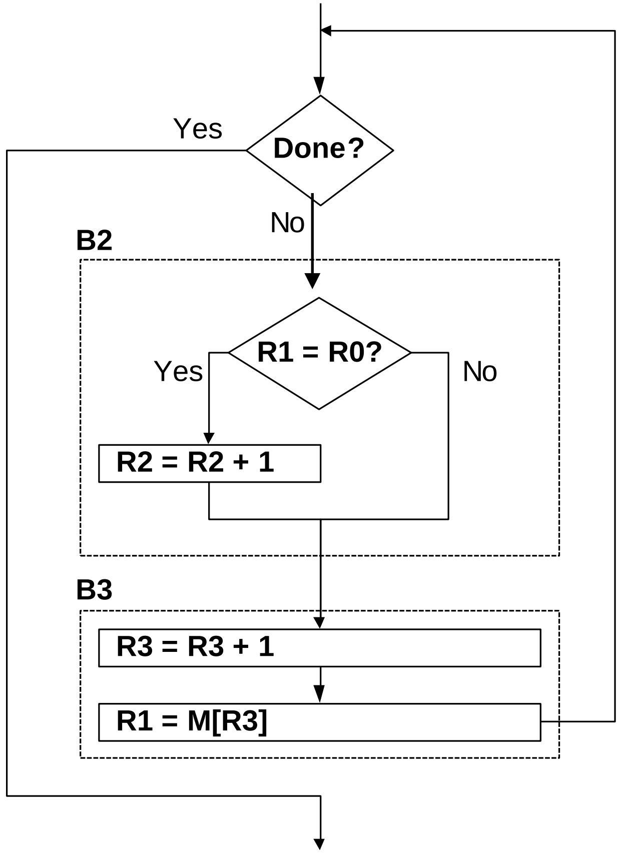
Refining B2 and B3



Conditional (B2) and sequential (B3).
Use of LC-2 registers and instructions.

The Last Step: LC-3 Instructions

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



```
;
; Test character for end of file
;
TEST        ADD         R4, R1, #-4 ; Test for EOT (ASCII x04)
            BRz         OUTPUT      ; If done, prepare the output

;
; 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             ;
               ADD          R2, R2, #1
;
; Get next character from file.
;
GETCHAR        ADD          R3, R3, #1          ; Point to next ch
               LDR          R1, R3, #0          ; R1 gets next char
               BRnzp        TEST
```

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

Will be skipped in the class.

The examples assumes that the programs are given only in binary.

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.

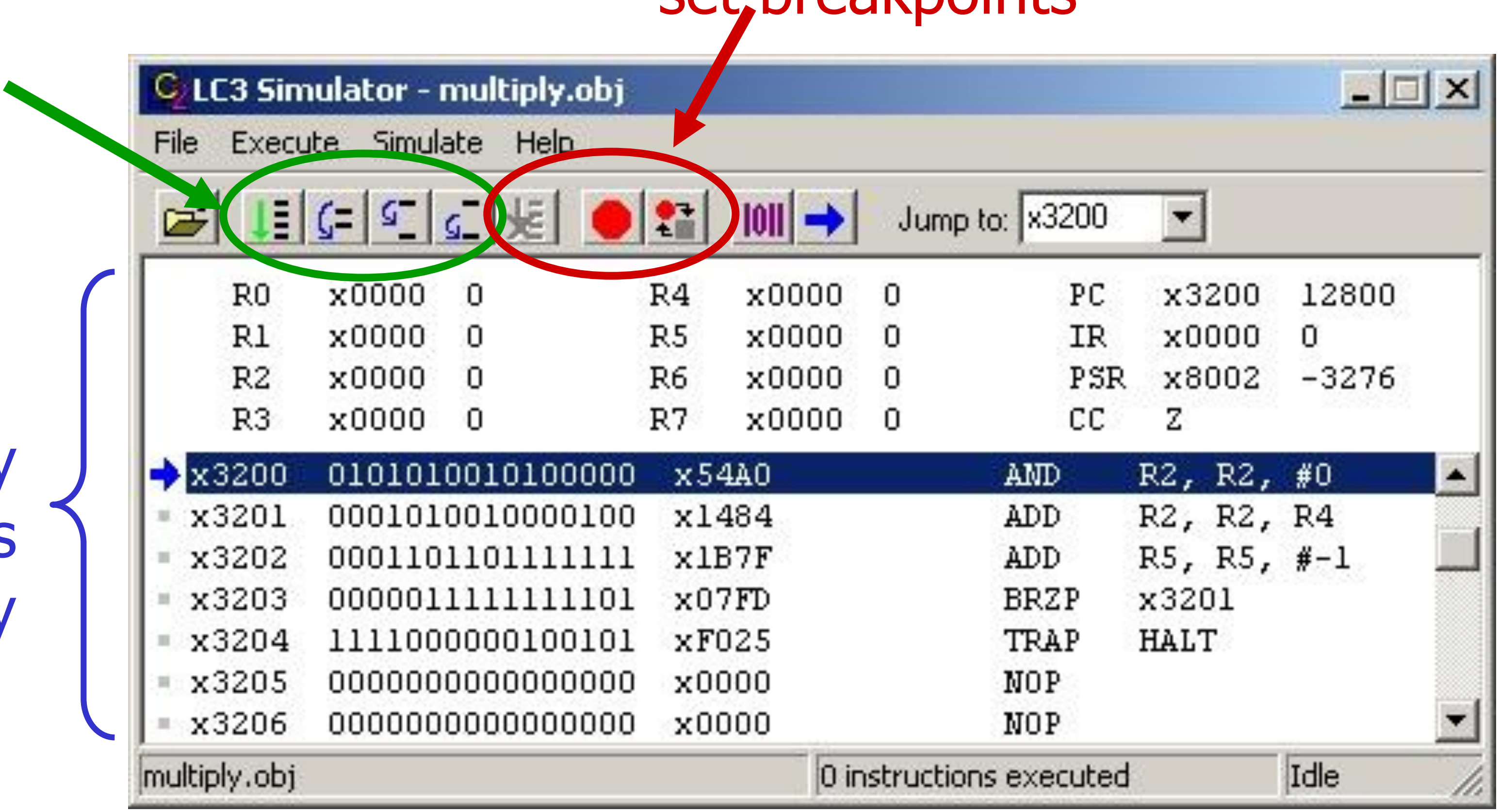
- **High-level languages (C, Java, ...)**
  usually have source-code debugging tools.
- **For debugging at the machine instruction level:**
  - ➢ **simulators**
  - ➢ **operating system “monitor” tools**
  - ➢ **in-circuit emulators (ICE)**
    - – **plug-in hardware replacements that give instruction-level control**

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        IR   x0000  0
R2   x0000  0        R6   x0000  0        PSR  x8002  -3276
R3   x0000  0        R7   x0000  0        CC   Z

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

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 wrong, 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 than what you expected.
- 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

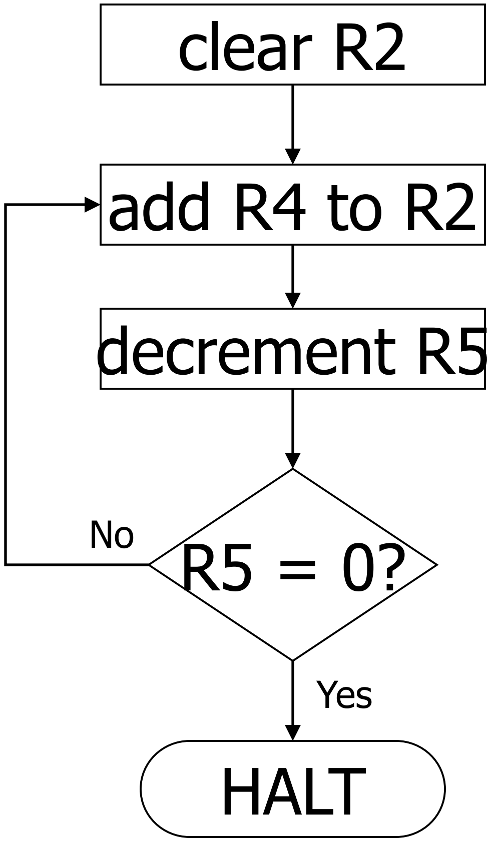
- **Tell the simulator to stop executing when it reaches a specific instruction.**
- **Check overall results at specific points in the program.**
  - **Lets you quickly execute sequences to get a high-level overview of the execution behavior.**
  - **Quickly execute sequences that your believe are correct.**

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 <u>where</u> or <u>when</u> a value is changed.**

Example 1: Multiply

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



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

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

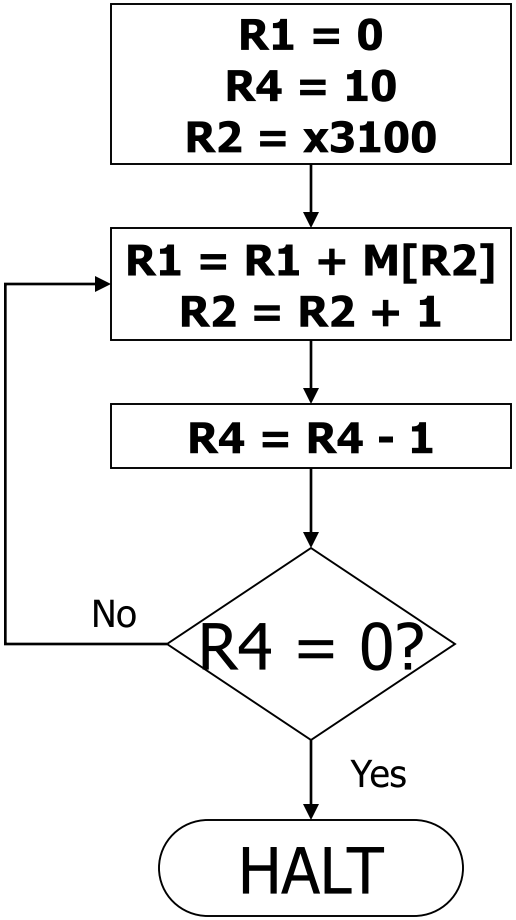
Should stop looping here!

Executing loop one time too many

Branch at x3203 should be based on Z 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.



R1 = 0
R4 = 10
R2 = x3100

R1 = R1 + M[R2]
R2 = R2 + 1

R4 = R4 - 1

R4 = 0?

No

Yes

HALT

```
x3000 0101001001100000
x3001 0101100100100000
x3002 0001100100101010
x3003 0010010011111100
x3004 0110011010000000
x3005 0001010010100001
x3006 0001001001000011
x3007 0001100100111111
x3008 0000001111111011
x3009 1111000000100101
```

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!

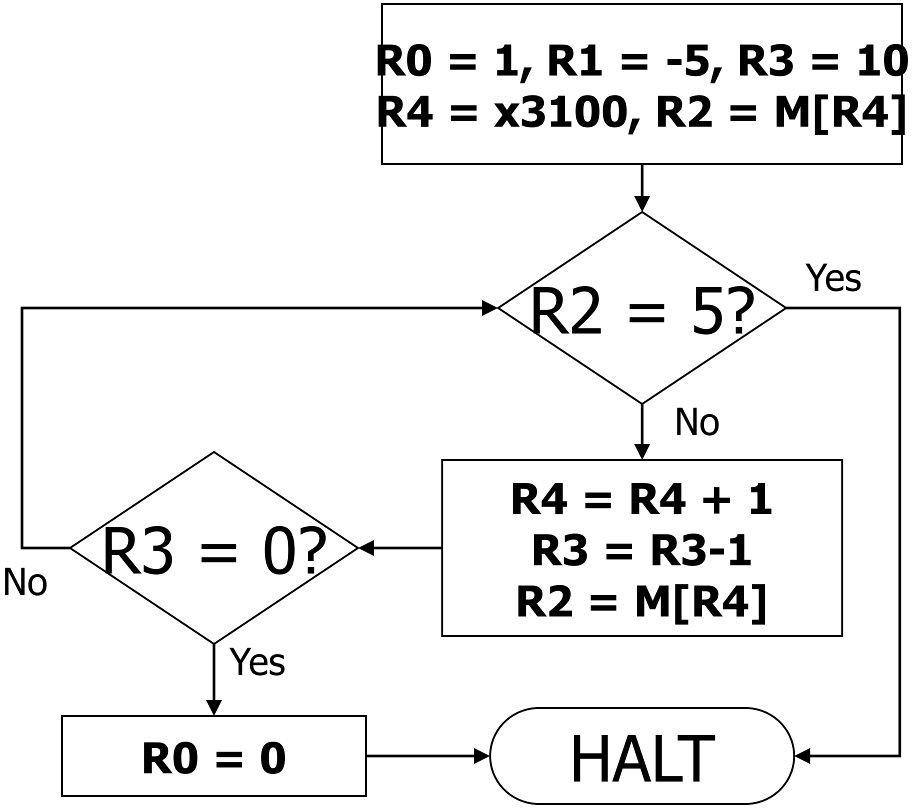
Loading contents of M[x3100], not address.

Change opcode of 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 ten memory locations, starting at x3100.

Else, it should set R0 to 0.



R0 = 1, R1 = -5, R3 = 10
R4 = x3100, R2 = M[R4]

R2 = 5?

Yes

No

R4 = R4 + 1
R3 = R3-1
R2 = M[R4]

R3 = 0?

No

Yes

R0 = 0

HALT

```
x3000  0101000000100000
x3001  0001000000100001
x3002  0101001001100000
x3003  0001001001111011
x3004  0101011011100000
x3005  0001011011101010
x3006  0010100000001001
x3007  0110010100000000
x3008  0001010010000001
x3009  0000010000000101
x300A  0001100100100001
x300B  0001011011111111
x300C  0110010100000000
x300D  0000001111111010
x300E  0101000000100000
x300F  1111000000100101
x3010  0011000100000000
```

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

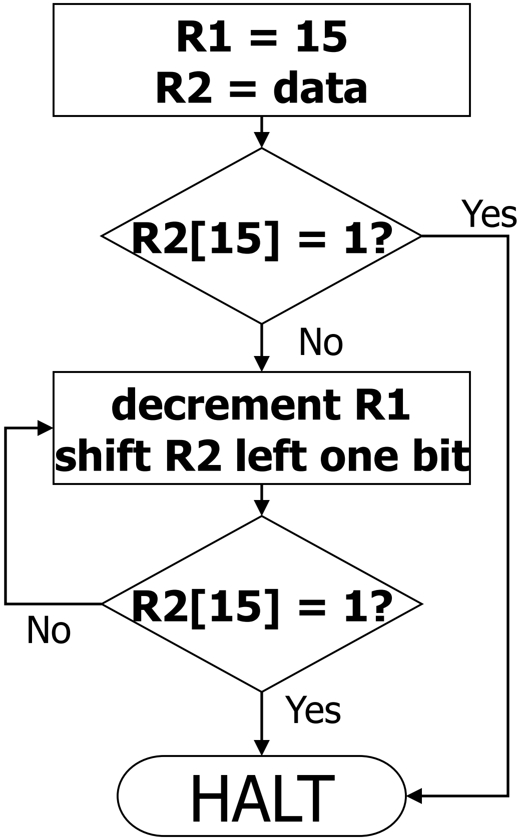
Didn't branch back, even though R3 > 0?

Branch 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
x3001 0001001001101111
x3002 1010010000000110
x3003 0000100000000100
x3004 0001001001111111
x3005 0001010010000010
x3006 0000100000000001
x3007 0000111111111100
x3008 1111000000100101
x3009 0011000100000000
```

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 to not notice that address x3107 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, ...).