10. Example Program, Debugging (Chapters 5, 6.1, 6.2)

October 3, 2018

- **Review: LC3 data path and control state machine**
  - **Steps during instruction execution**
- **Example programs**
- **LC3 Tools (Edit, Simulate)**
- **Introduction to debugging**

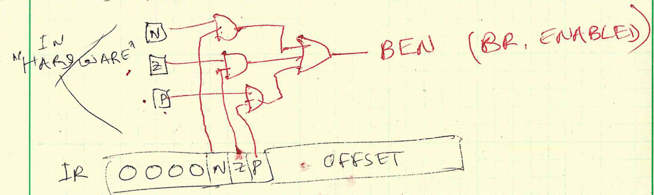
CONTROL INSTRUCTIONS

ALTERS SEQUENCE OF INSTRUCTIONS
⇒ CHANGES PC

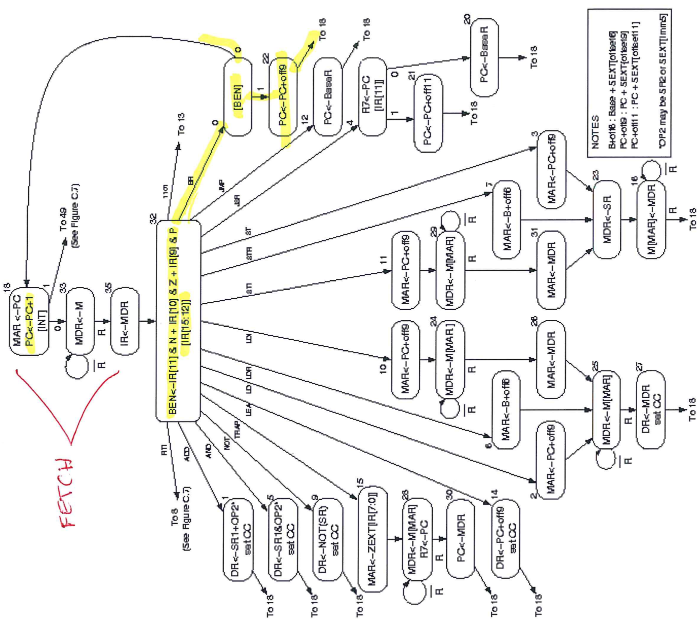
JUMP – UNCONDITIONAL
BRANCH – CONDITIONAL
LC3 : CONDITION CODE REGISTERS (1-BIT)
SET BY ANY INSTR. WHICH WRITES A VALUE INTO A REGISTER
(ADD, AND, NOT, LDR, LD, LDI, LEA)

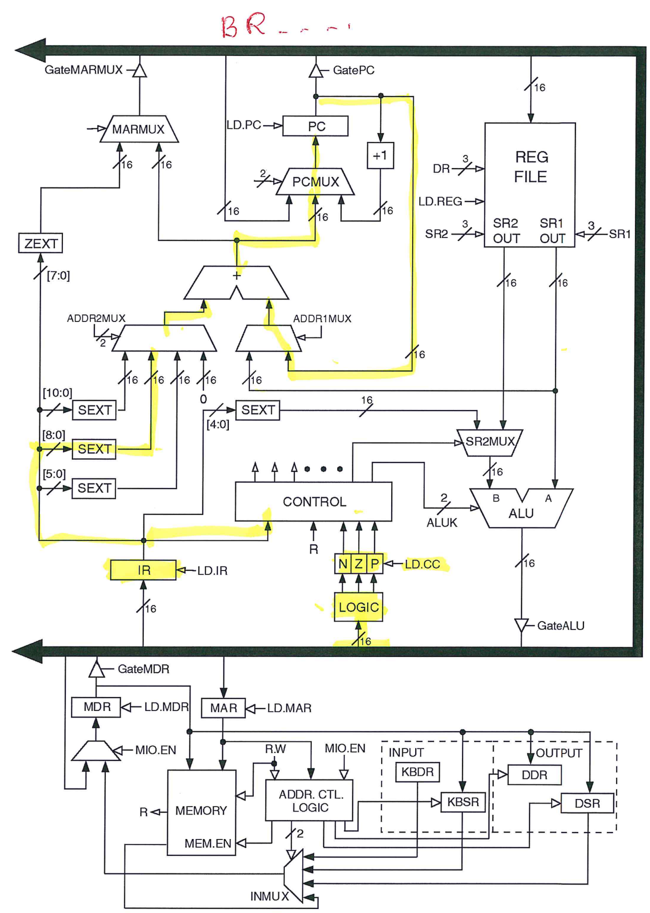
n – NEGATIVE
z – ZERO
p – POSITIVE

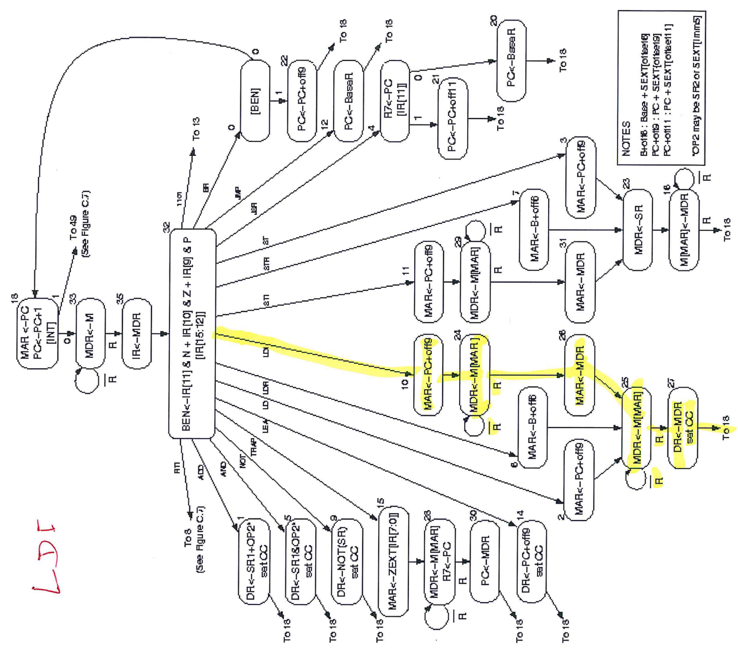
IN A PROGRAM, BR IS TAKEN, OR NOT TAKEN DEPENDING ON WHAT IS SPECIFIED

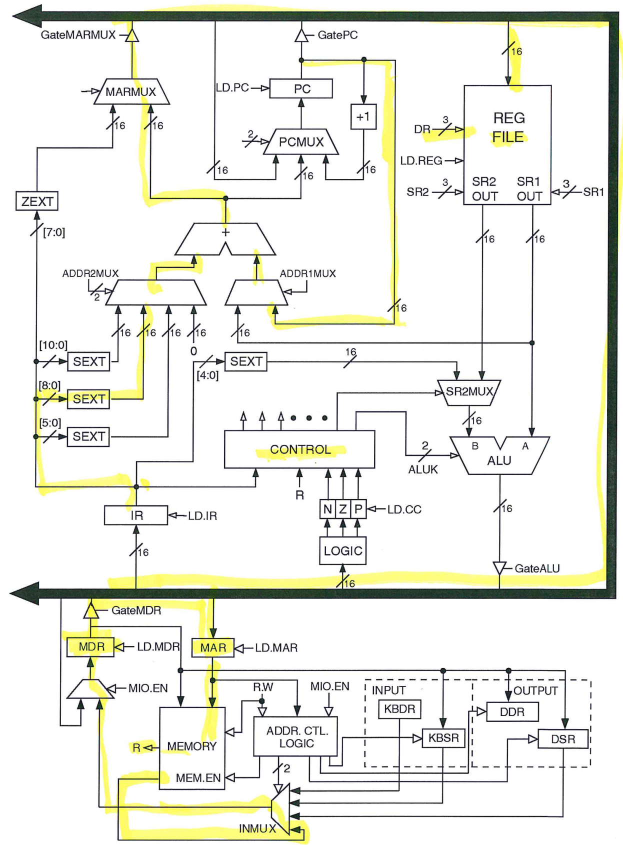


WHAT IS THE X0000 INSTRUCTION
= NOP (NO OPERATION)









TRAP (CONTROL)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

Calls a service routine, identified by 8-bit "trap vector."

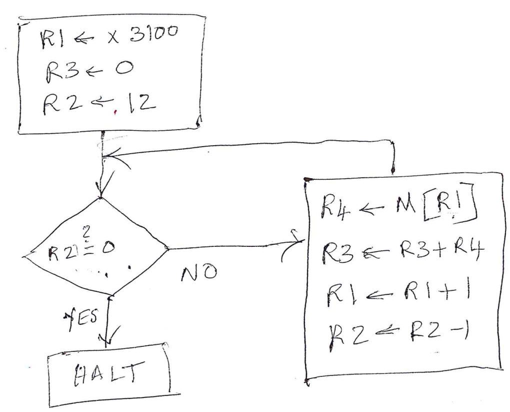
vector	*routine*
x23	input a character from the keyboard
x21	output a character to the monitor
x25	halt the program

When routine is done,
PC is set to the instruction following TRAP.
(We'll talk about how this works later.)

Using Branch Instructions

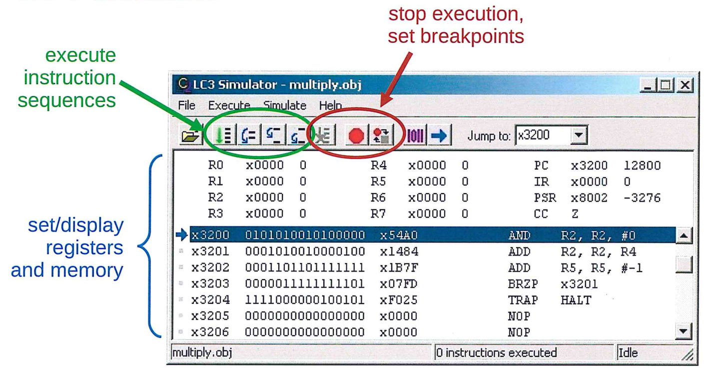
Compute sum of 12 integers.
Numbers start at location x3100. Program starts at location x3000.

R1: POINTS TO MEM. LOCATION (ADDRESS) WHERE NEXT NUMBER IS STORED
R2: NEXT INTEGER TO BE ADDED (OF 12)
⇒ R3: SUM
R4: LOADS CURRENT INTEGER (FROM MEMORY)



Address	Instruction	Comments
LEA x3000	1110 001 011111111	R1 ← x3100 (PC+OFFSET)
AND x3001	0101 011 011 100000	R3 ← 0
AND x3002	0101 010 010 100000	R2 ← 0
ADD x3003	0001 010 010 101100	R2 ← 12
BR x3004	0000 010 000 000101	IF Z, GOTO HALT (PC+5)
LDR x3005	0110 100 001 000000	LOAD NEXT VALUE INTO R4
ADD x3006	0001 011 011 000100	ADD TO R3 (R3←R3+R4)
ADD x3007	0001 001 001 100001	R1 ← R1+1
ADD x3008	0001 010 010 111111	R2 ← R2−1
BR x3009	0000 111 111 111010	GO TO x3004 (−6 OFFSET)
HALT x300A	1111 000 000 100101	HALT

LC-3 Simulator



Using "Sentinel"

Compute sum of 12 integers.

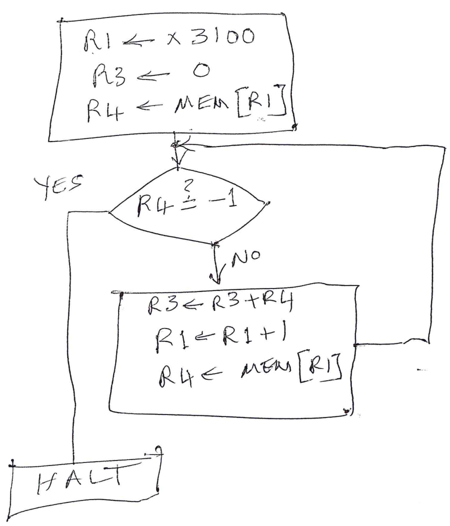
Numbers start at location x3100. Program starts at location x3000.

Sentinel stored in x310C is -1

R1 : LOCATIONS WHERE NUMBERS ARE STORED

R3 : SUM

R4 : CURRENT INTEGER



Program Using "Sentinel" for Loop Control

Address	Instruction	Comments
x3000	1110 001 011111111	R1 ← x3100 (PC+0xFF) LEA R1, 0x0FF
x3001	0101 011 011 100000	R3 ← 0 AND R3, R3, 0x00
x3002	0110 100 001 000000	R4 ← M[R1] LDR R4, R1 0x00
x3003	0000 100 000000100	BRn x3008 (0x04)
x3004	0001 011 011 000100	R3 ← R3 + R4 ADD R3, R3, R4
x3005	0001 001 001 100001	R1 ← R1 + 1 ADD R1. R1. 0x01
x3006	0110 100 001 000000	R4 ← M[R1] LDR R4, R1 0x00
x3007	0000 111 111111010	BRnzp (goto) x3003 (#-6)
x3008	1111 0000 00100101	HALT

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.

Text: ASCII Characters

ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, …) characters

<table>
<tr><td>00</td><td>nul</td><td>10</td><td>dle</td><td>20</td><td>sp</td><td>30</td><td>0</td><td>40</td><td>@</td><td>50</td><td>P</td><td>60</td><td>`</td><td>70</td><td>p</td></tr>
<tr><td>01</td><td>soh</td><td>11</td><td>dc1</td><td>21</td><td>!</td><td>31</td><td>1</td><td>41</td><td>A</td><td>51</td><td>Q</td><td>61</td><td>a</td><td>71</td><td>q</td></tr>
<tr><td>02</td><td>stx</td><td>12</td><td>dc2</td><td>22</td><td>"</td><td>32</td><td>2</td><td>42</td><td>B</td><td>52</td><td>R</td><td>62</td><td>b</td><td>72</td><td>r</td></tr>
<tr><td>03</td><td>etx</td><td>13</td><td>dc3</td><td>23</td><td>#</td><td>33</td><td>3</td><td>43</td><td>C</td><td>53</td><td>S</td><td>63</td><td>c</td><td>73</td><td>s</td></tr>
<tr><td>04</td><td>eot</td><td>14</td><td>dc4</td><td>24</td><td>$</td><td>34</td><td>4</td><td>44</td><td>D</td><td>54</td><td>T</td><td>64</td><td>d</td><td>74</td><td>t</td></tr>
<tr><td>05</td><td>enq</td><td>15</td><td>nak</td><td>25</td><td>%</td><td>35</td><td>5</td><td>45</td><td>E</td><td>55</td><td>U</td><td>65</td><td>e</td><td>75</td><td>u</td></tr>
<tr><td>06</td><td>ack</td><td>16</td><td>syn</td><td>26</td><td>&</td><td>36</td><td>6</td><td>46</td><td>F</td><td>56</td><td>V</td><td>66</td><td>f</td><td>76</td><td>v</td></tr>
<tr><td>07</td><td>bel</td><td>17</td><td>etb</td><td>27</td><td>'</td><td>37</td><td>7</td><td>47</td><td>G</td><td>57</td><td>W</td><td>67</td><td>g</td><td>77</td><td>w</td></tr>
<tr><td>08</td><td>bs</td><td>18</td><td>can</td><td>28</td><td>(</td><td>38</td><td>8</td><td>48</td><td>H</td><td>58</td><td>X</td><td>68</td><td>h</td><td>78</td><td>x</td></tr>
<tr><td>09</td><td>ht</td><td>19</td><td>em</td><td>29</td><td>)</td><td>39</td><td>9</td><td>49</td><td>I</td><td>59</td><td>Y</td><td>69</td><td>i</td><td>79</td><td>y</td></tr>
<tr><td>0a</td><td>nl</td><td>1a</td><td>sub</td><td>2a</td><td>*</td><td>3a</td><td>:</td><td>4a</td><td>J</td><td>5a</td><td>Z</td><td>6a</td><td>j</td><td>7a</td><td>z</td></tr>
<tr><td>0b</td><td>vt</td><td>1b</td><td>esc</td><td>2b</td><td>+</td><td>3b</td><td>;</td><td>4b</td><td>K</td><td>5b</td><td>[</td><td>6b</td><td>k</td><td>7b</td><td>{</td></tr>
<tr><td>0c</td><td>np</td><td>1c</td><td>fs</td><td>2c</td><td>,</td><td>3c</td><td><</td><td>4c</td><td>L</td><td>5c</td><td>\</td><td>6c</td><td>l</td><td>7c</td><td>|</td></tr>
<tr><td>0d</td><td>cr</td><td>1d</td><td>gs</td><td>2d</td><td>-</td><td>3d</td><td>=</td><td>4d</td><td>M</td><td>5d</td><td>]</td><td>6d</td><td>m</td><td>7d</td><td>}</td></tr>
<tr><td>0e</td><td>so</td><td>1e</td><td>rs</td><td>2e</td><td>.</td><td>3e</td><td>></td><td>4e</td><td>N</td><td>5e</td><td>^</td><td>6e</td><td>n</td><td>7e</td><td>~</td></tr>
<tr><td>0f</td><td>si</td><td>1f</td><td>us</td><td>2f</td><td>/</td><td>3f</td><td>?</td><td>4f</td><td>O</td><td>5f</td><td>_</td><td>6f</td><td>o</td><td>7f</td><td>del</td></tr>
</table>

Problem Statement

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

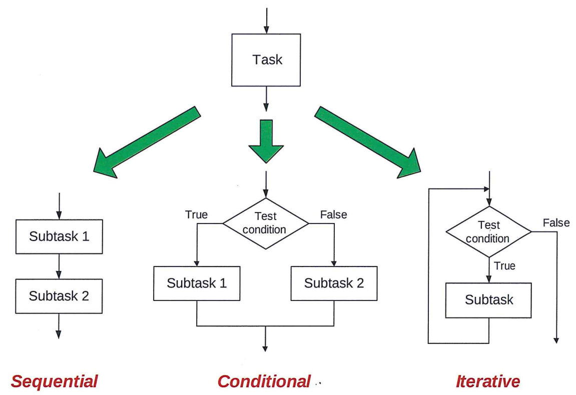
- Where is "file" located? How big is it, or how do I know when I've reached the end?
- How should final count be printed? A decimal number?
- If the character is a letter, should I count both upper-case and lower-case occurrences?

How do you resolve these issues?

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

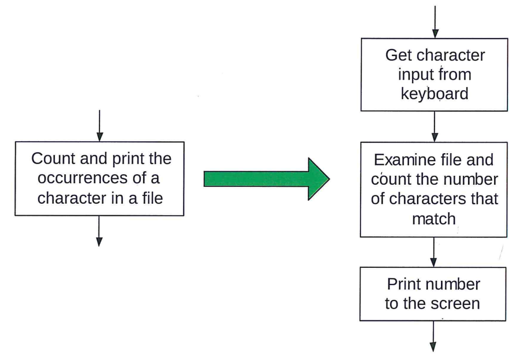
Three Basic Constructs

There are three basic ways to decompose a task:



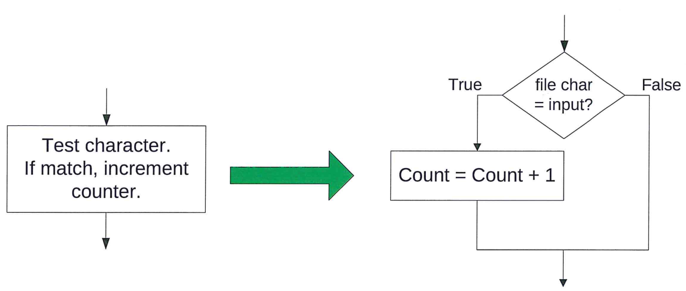
Sequential

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



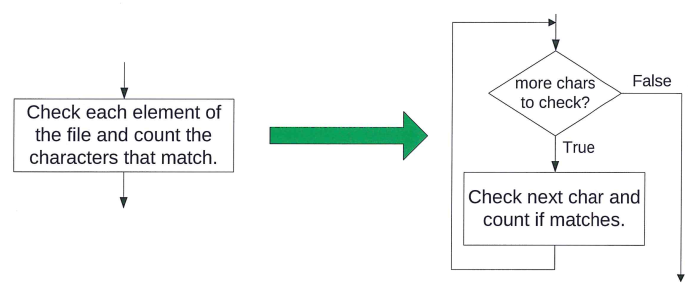
Conditional

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



Iterative

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

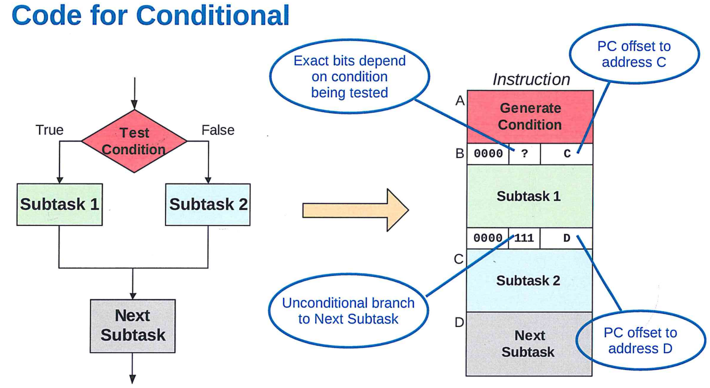


Problem Solving Skills

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

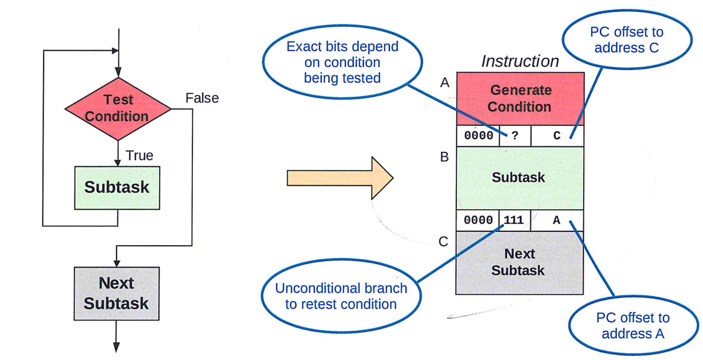
- **Like a puzzle, or a "word problem" from grammar school math.**
  - **What is the starting state of the system?**
  - **What is the desired ending state?**
  - **How do we move from one state to another?**
- **Recognize English words that correlate to three basic constructs:**
  - **"do A then do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

Code for Conditional



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

Code for Iteration



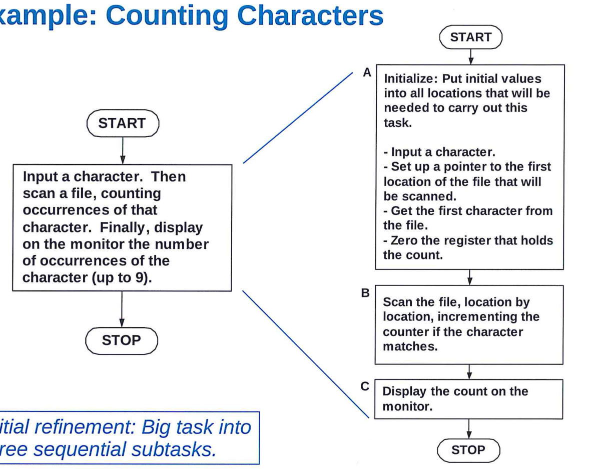
Assuming all addresses are on the same page.

Detailed Example

Count the occurrences of a character in a file

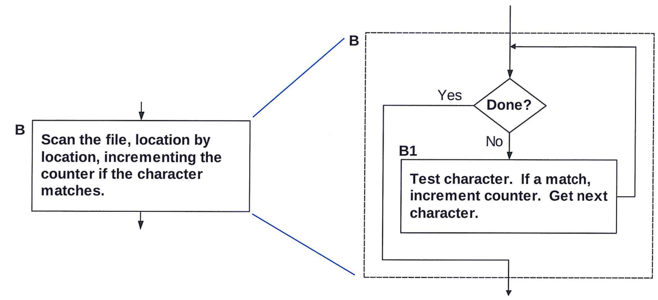
- **Program begins at location x3000**
- **Read character from keyboard**
- **Load each character from a "file"**
  - File is a sequence of memory locations
  - Starting address of file is stored in the memory location immediately after the program
- **If file character equals input character, increment counter**
- **End of file is indicated by a special ASCII value: EOT (x04)**
  - **Sentinal**
- **At the end, print the number of characters and halt**
  (assume there will be less than 10 occurrences of the character)

Example: Counting Characters



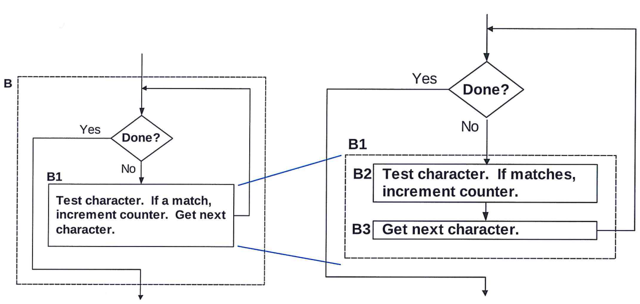
Initial refinement: Big task into three sequential subtasks.

Refining B



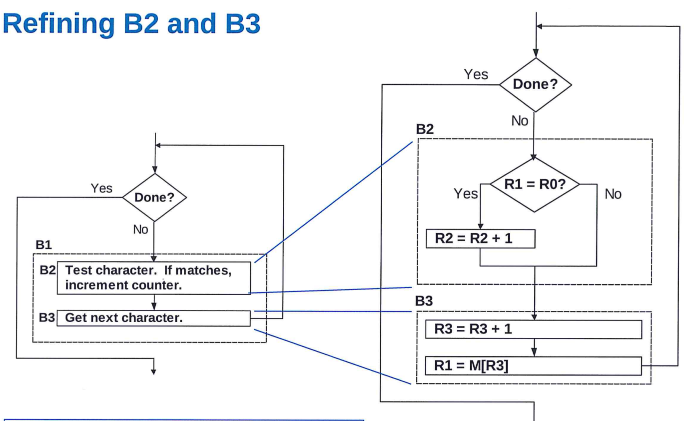
Refining B into iterative construct.

Refining B1



Refining B1 into sequential subtasks.

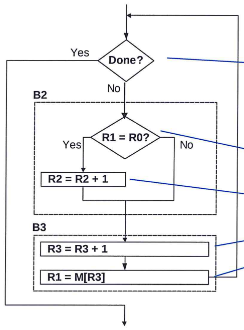
Refining B2 and B3



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.




```
; 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.**