Chapter 6
Programming the LC-3

Aside: Booting the Computer

How does it all begin?

- We have LC-3 hardware and a program, but what next?

Initial state of computer

- All zeros (registers, memory, condition codes)
- Only *mostly* true

Boot process

- Load boot code held in ROM (read-only memory)
  - BIOS (*basic input/output system)*
- Loads operating system from disk (or other input device)
- Operating systems loads other programs
  - Uses memory operations (loads, stores)
  - Sets PC to beginning of program to run it
  - Programs invoke O.S. using TRAP instructions

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 *(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

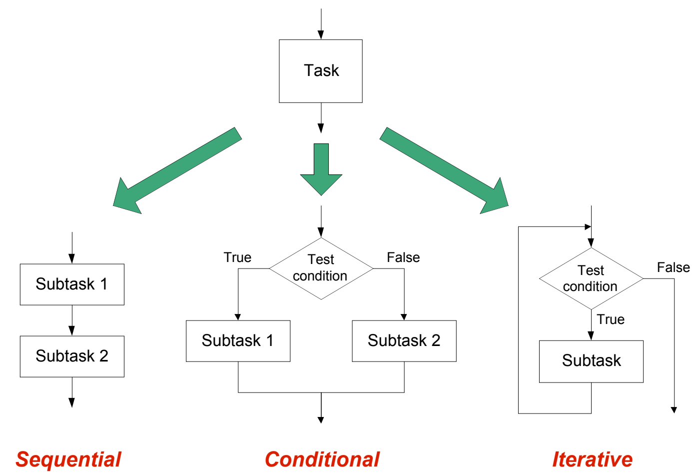
- **Where is the data located? How big is it, or how do I know when I've reached the end?**
- **How should final count be printed? A decimal number?**
- **If the character is a letter, should I count both upper-case and lower-case occurrences?**

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:



Programming at the Instruction Level

Advantage: can do anything

- **General, powerful**

Disadvantage: can do anything

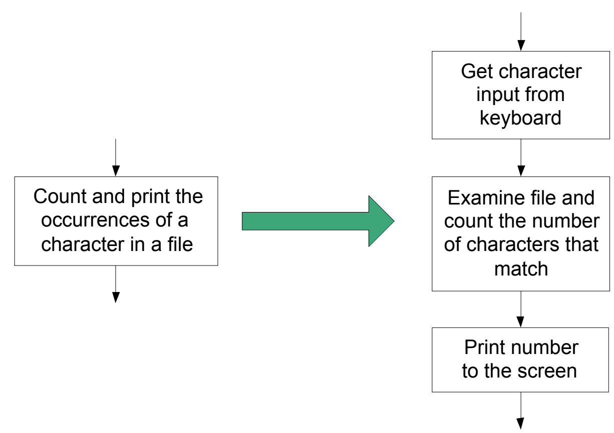
- **Difficult to structure, modify, understand**

Mitigate disadvantages using structured programming

- **Use familiar constructs (even at the instruction level)**
  - **From Java/C/Pascal/Fortran/Basic**
- **Iteration (while loop, for loop)**
- **Conditional (if statement, switch/case statement)**

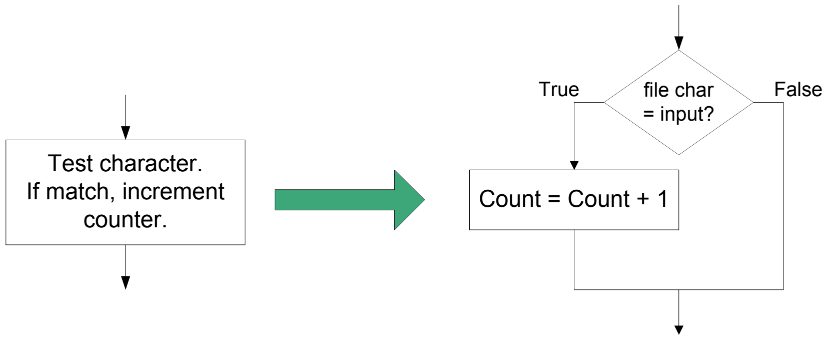
Sequential

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



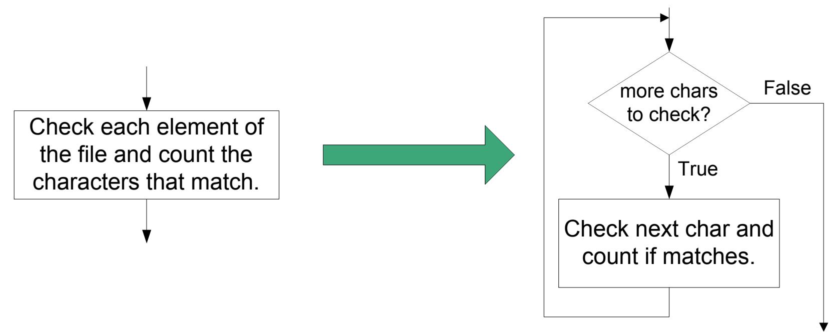
Conditional

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



LC-3 Control Instructions

How can instructions encode these basic constructs?

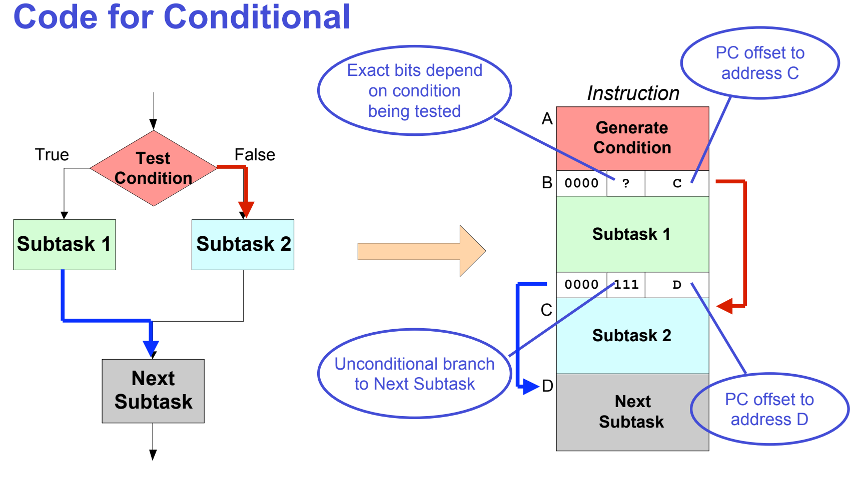
Sequential

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

Conditional and Iterative

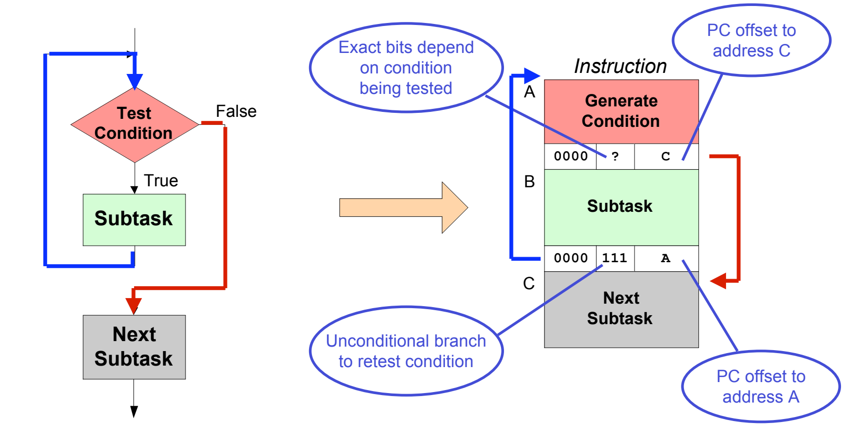
- **Create code that converts condition into N, Z, or P**
  - **Condition: "Is R0 = R1?"**
  - **Code: Subtract R1 from R0; if equal, Z bit will be set**
- **Use BR instruction to transfer control**
- **What about R0 < R1?**
  - **Code: Subtract R1 from R0 (R0-R1), if less, N bit will be set**

Code for Conditional



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

Code for Iteration



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

Example (from both Ch 5 and 6)

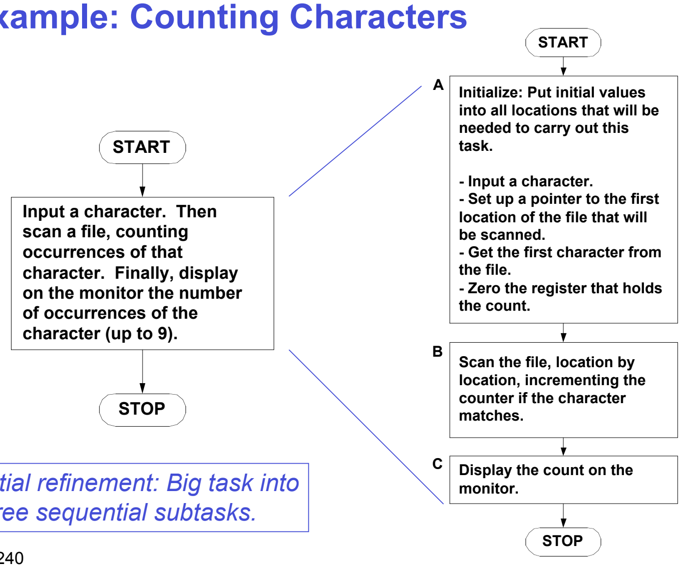
Count the occurrences of a character in a file

- **Program begins at location x3000**
- **Read character from keyboard**
- **Load each character from a "file"**
  - In this example the "file" is already in 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)**
- **At the end, print the number of characters and halt**
  (assume there will be fewer than 10 occurrences of the character)

A special character used to indicate the end of a sequence is often called a sentinel

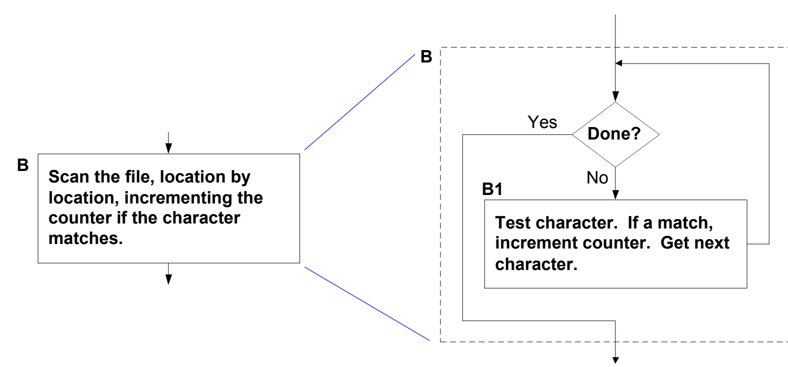
- Useful when you don't know ahead of time how many times to execute a loop

Example: Counting Characters



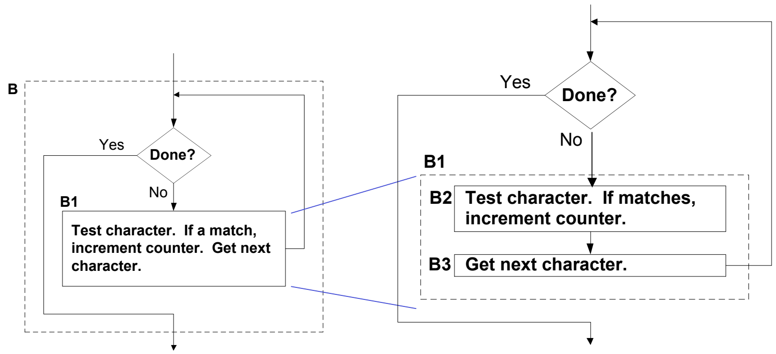
Initial refinement: Big task into three sequential subtasks.

Refining B



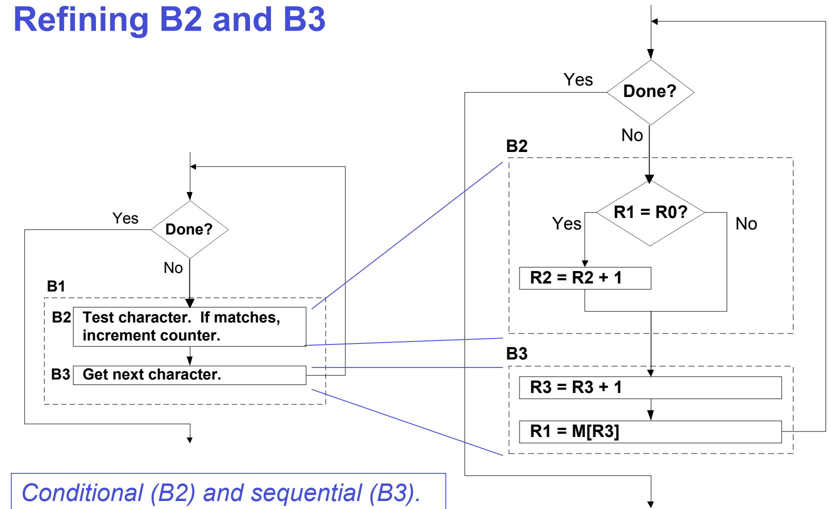
Refining B into iterative construct.

Refining B1



Refining B1 into sequential subtasks.

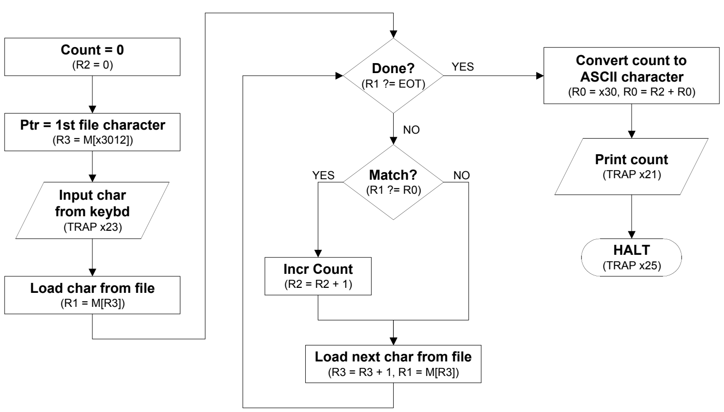
Refining B2 and B3



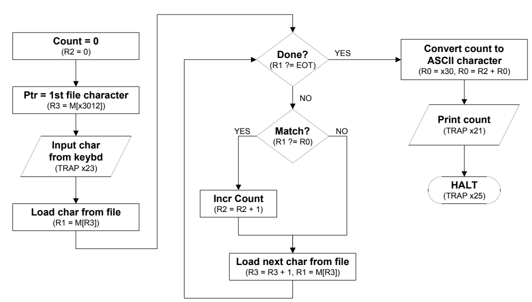
Conditional (B2) and sequential (B3).
Use of LC-3 registers and instructions.

Entire Flow Chart

Input: M[x3012] (address of "file")
Output: Print count to display



Translate to Pseudocode

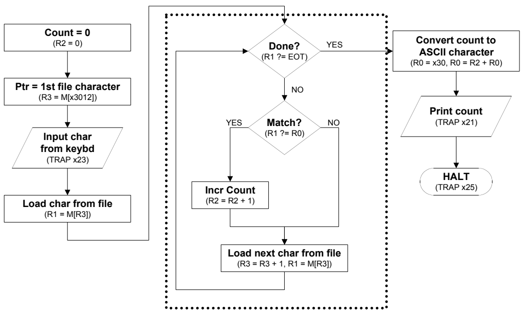


R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]

?

R0 ← M[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)

Iterative Construct in Pseudocode

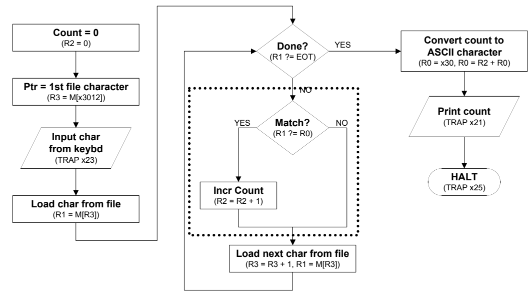


R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]
R4 ← R1 – 4 (EOT)
BRz x????

?

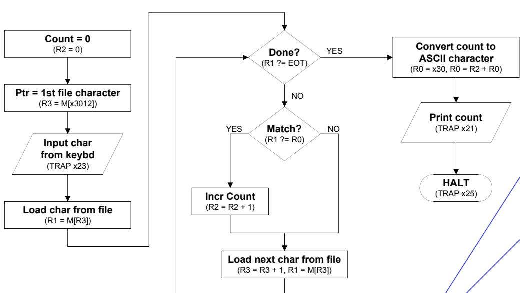
BRnzp x????
R0 ← M[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)

Conditional in Pseudocode



R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]
R4 ← R1 – 4 (EOT)
BRz x????
R1 ← NOT R1
R1 ← R1 + 1
R1 ← R1 + R0
BRnp x????
R2 ← R2 + 1
R3 ← R3 + 1
R1 ← M[R3]
BRnzp x????
R0 ← M[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)

Final Pseudocode



R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]
R4 ← R1 – 4 (EOT)
BRz x????
R1 ← NOT R1
R1 ← R1 + 1
R1 ← R1 + R0
BRnp x????
R2 ← R2 + 1
R3 ← R3 + 1
R1 ← M[R3]
BRnzp x????
R0 ← M[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)

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

Translate Pseudocode (1 of 2)

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

Address	*Instruction*		*Comments*
x3000	0 1 0 1	0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0 (counter)*
x3001	0 0 1 0	0 1 1 0 0 0 0 1 0 0 0 0	*R3 ← M[x3012] (ptr)*
x3002	1 1 1 1	0 0 0 0 0 0 1 0 0 0 1 1	*Input to R0 (TRAP x23)*
x3003	0 1 1 0	0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]*
x3004	0 0 0 1	1 0 0 0 0 1 1 1 1 1 0 0	*R4 ← R1 – 4 (EOT)*
x3005	0 0 0 0	0 1 0 0 0 0 0 0 1 0 0 0	*BRz x300E*
x3006	1 0 0 1	0 0 1 0 0 1 1 1 1 1 1 1	*R1 ← NOT R1*
x3007	0 0 0 1	0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1 + 1*
X3008	0 0 0 1	0 0 1 0 0 1 0 0 0 0 0 0	*R1 ← R1 + R0*
x3009	0 0 0 0	1 0 1 0 0 0 0 0 0 0 0 1	*BRnp x300B*

Translate Pseudocode (2 of 2)

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

Address	*Instruction*	*Comments*
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	*R2 ← R2 + 1*
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	*R3 ← R3 + 1*
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]*
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	*BRnzp x3004*
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	*R0 ← M[x3013]*
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0	*R0 ← R0 + R2*
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	*Print R0 (TRAP x21)*
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	*HALT (TRAP x25)*
X3012	**Starting Address of File**	
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	*ASCII x30 ('0')*

Structured Programming of LC-3 Summary

Decompose task

- **Top-down**
- **Specification often ambiguous**
- **Continual refinement of details**

Write code

- **Focus on one bite-sized part at a time**
- **Use structured programming (even at the instruction level)**
- **Translate flowchart to pseudo code then to machine code**

Continual testing and debugging of code

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 might provide means to:

1. **Display values in memory and registers**
2. **Change values in memory and registers**
3. **Execute instructions in a program**
4. **Stop execution when desired**

Different programming levels offer different tools

- **High-level languages (C, Java, ...) have source-code debugging tools**
- **For debugging at the machine instruction level:**
  - **Simulators**
  - **Operating system "monitor" tools**
  - **Special hardware**

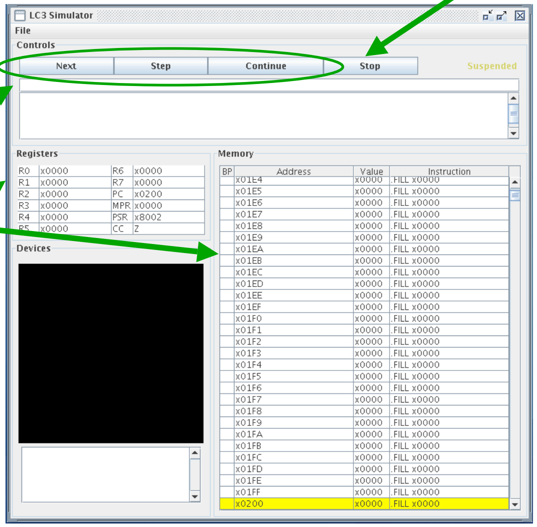
LC-3 Simulator

Stop Execution

Start Execution

Command Line

Mem/Reg Values



Tracing the Program

Execute the program one piece at a time, examining register and memory to see results at each step

Single-Stepping

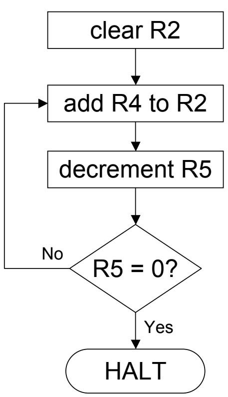
- Execute one instruction at a time
- Tedious, but useful to help you verify each step of your program

Breakpoints

- Tell simulator to stop exec. 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

Example 1: Multiply

Goal: Multiply the two positive integers in R4 and R5, and place result in R2 (does not handle multiple by zero case)

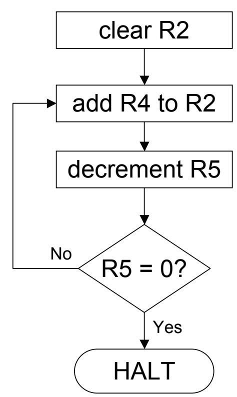



```
x3200  0101010010100000
x3201  0001010010000100
x3202  0001101101111111
x3203  0000011111111101
x3204  1111000000100101
```

Set R4 = 10, R5 =3
Run program
Result: R2 = 40, not 30
(R2 = x0028, not x001E)

Example 1: Multiply

Goal: Multiply the two positive integers in R4 and R5, and place result in R2 (does not handle multiple by zero case)




```
x3000 AND   R2,R2,#0
x3001 ADD   R2,R2,R4
x3002 ADD   R5,R5,#-1
x3003 BRzp  x3001
x3004 HALT
```

Set R4 = 10, R5 =3
Run program
Result: R2 = 40, not 30
(R2 = x0028, not x001E)

Debugging the Multiply Program

PC and registers at the beginning of each instruction

Single-stepping

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

Breakpoint at branch (x3203)

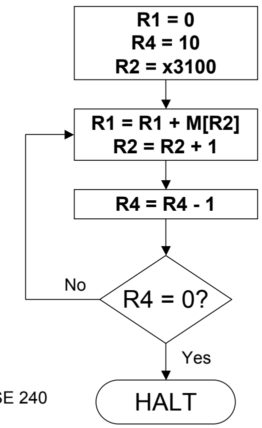
PC	R2	R4	R5
x3203	10	10	2
x3203	20	10	1
x3203	30	10	0
x3203	40	10	-1
	40	10	-1

Should stop looping here!

Executing loop one time too many
Branch at x3203 should be based on Z bit only, not Z and P
(change x07FD to x03FD)

Example 2: Summing an Array of Numbers

Goal: Sum the numbers stored in 10 memory locations beginning with x3100, leaving the result in R1




```
x3000 AND   R1,R1,#0
x3001 AND   R4,R4,#0
x3002 ADD   R4,R4,#10
x3003 LD    R2,x3100
x3004 LDR   R3,R2,#0
x3005 ADD   R2,R2,#1
x3006 ADD   R1,R1,R3
x3007 ADD   R4,R4,#-1
x3008 BRp   x3004
x3009 HALT
```


Debugging the Summing Program

Running 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

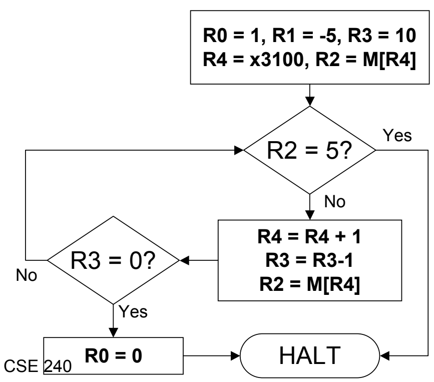
Should be x3100!

Loading contents of M[x3100], not address
Change opcode of x3003
from 0010 (LD) to xE or 1110 (LEA)

Example 3: Looking for a 5

Scan ten memory locations
- **starting at x3100**

If a "5" is found
- **set R0 to 1, otherwise set R0 to 0**




```
x3000 AND  R0,R0,#0
x3001 ADD  R0,R0,#1
x3002 AND  R1,R1,#0
x3003 ADD  R1,R1,#-5
x3004 AND  R3,R3,#0
x3005 ADD  R3,R3,#10
x3006 LD   R4,x3010
x3007 LDR  R2,R4,#0
x3008 ADD  R2,R2,R1
x3009 BRz  x300F
x300A ADD  R4,R4,#1
x300B ADD  R3,R3,#-1
x300C LDR  R2,R4,#0
x300D BRp  x3008
x300E AND  R0,R0,#0
x300F HALT
X3010 x3100
```

Debugging the Fives Program

Running the program with a 5 in location x3108 results is 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

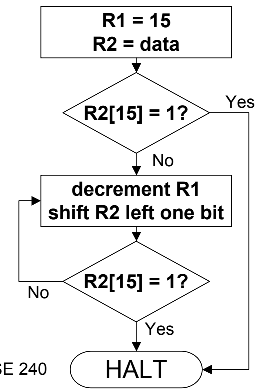
PC	R0	R2	R3	R4
x300D	1	7	9	x3101
x300D	1	32	8	x3102
x300D	1	0	7	x3103
	0	0	7	x3103

← 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

Goal: Return (in R1) the bit position of the first 1 in a word; address of word is in location x3009 (just past the end of the program); if there are no ones, R1 should be set to –1




```
x3000 AND   R1,R1,#0
x3001 ADD   R1,R1,#15
x3002 LDI   R2,x3009
x3003 BRn   x3008
x3004 ADD   R1,R1,#-1
x3005 ADD   R2,R2,R2
x3006 BRn   x3008
x3007 BRnzp x3004
x3008 HALT
x3009 x3100
```


Shifting Left

We often want to manipulate individual bits

- **Example: is a number odd or even?**
- **Answer: R1 := R0 AND 0x1**
  - **If R1 is 0 -> R0 was even**
  - **If R1 is 1 -> R0 was odd**

LC-3 doesn't give us an instruction to "shift" bits

- **Most ISAs include "shift left" and "shift right"**
- **Example: If you shift 0010 left one place, 0100 results**

How do we shift left in LC-3?

- **Multiple value by 2 (why?)**
- **Same as R1 := R0 + R0**
- **Example: 0010 + 0010 = 010**

Adding a value to itself shifts the bits left one place

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, ...)