310-2202 โครงสร้างของระบบคอมพิวเตอร์
(Computer Organization)

Topic 6: LC-3 Programming

Damrongrit Setsirichok

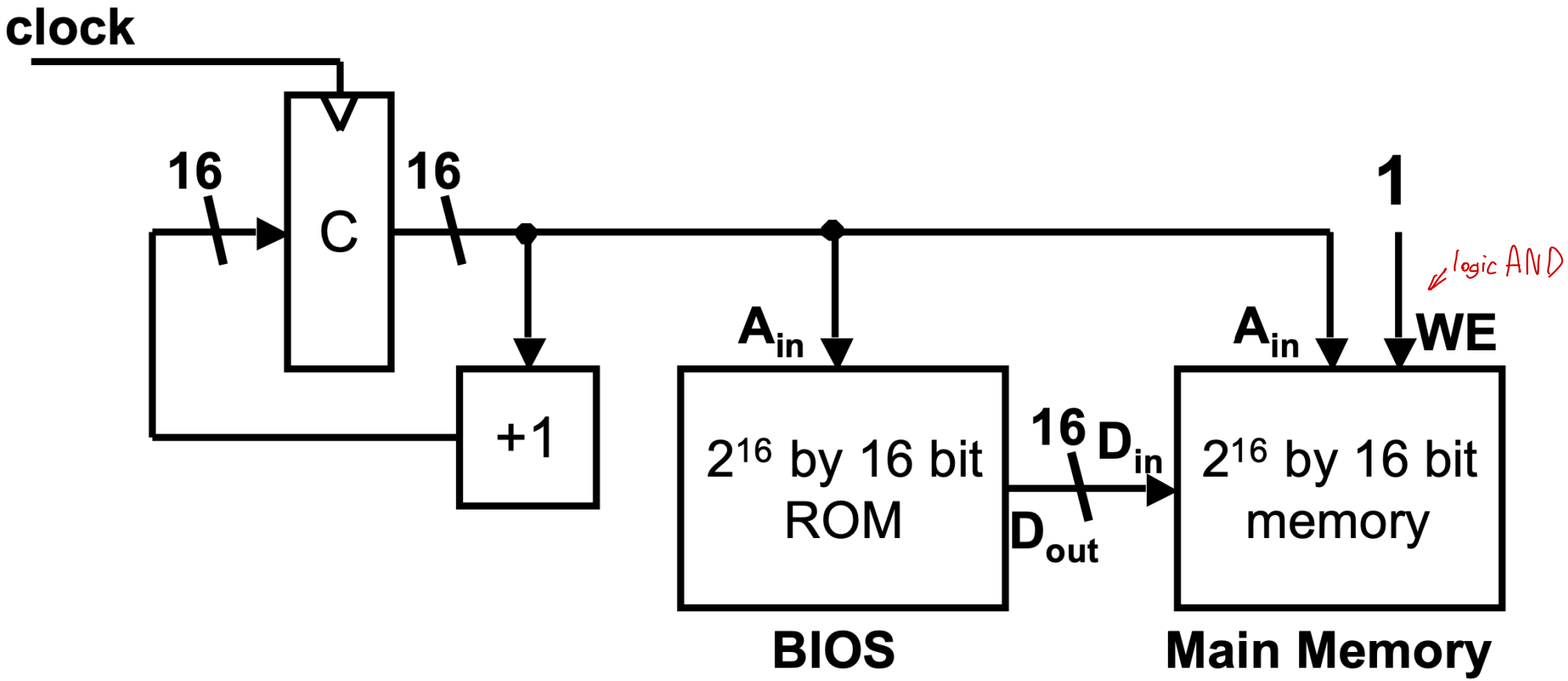
Topic

- Problem Solving
- Debugging
- Example

Booting the Computer

- 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 OS using TRAP instructions

Copying BIOS into Memory



Solving Problems using a Computer

- **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.

Problem Solving

Problem Solving

- **Systematic Decomposition**: The larger tasks are systematically broken down into smaller ones

- **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**

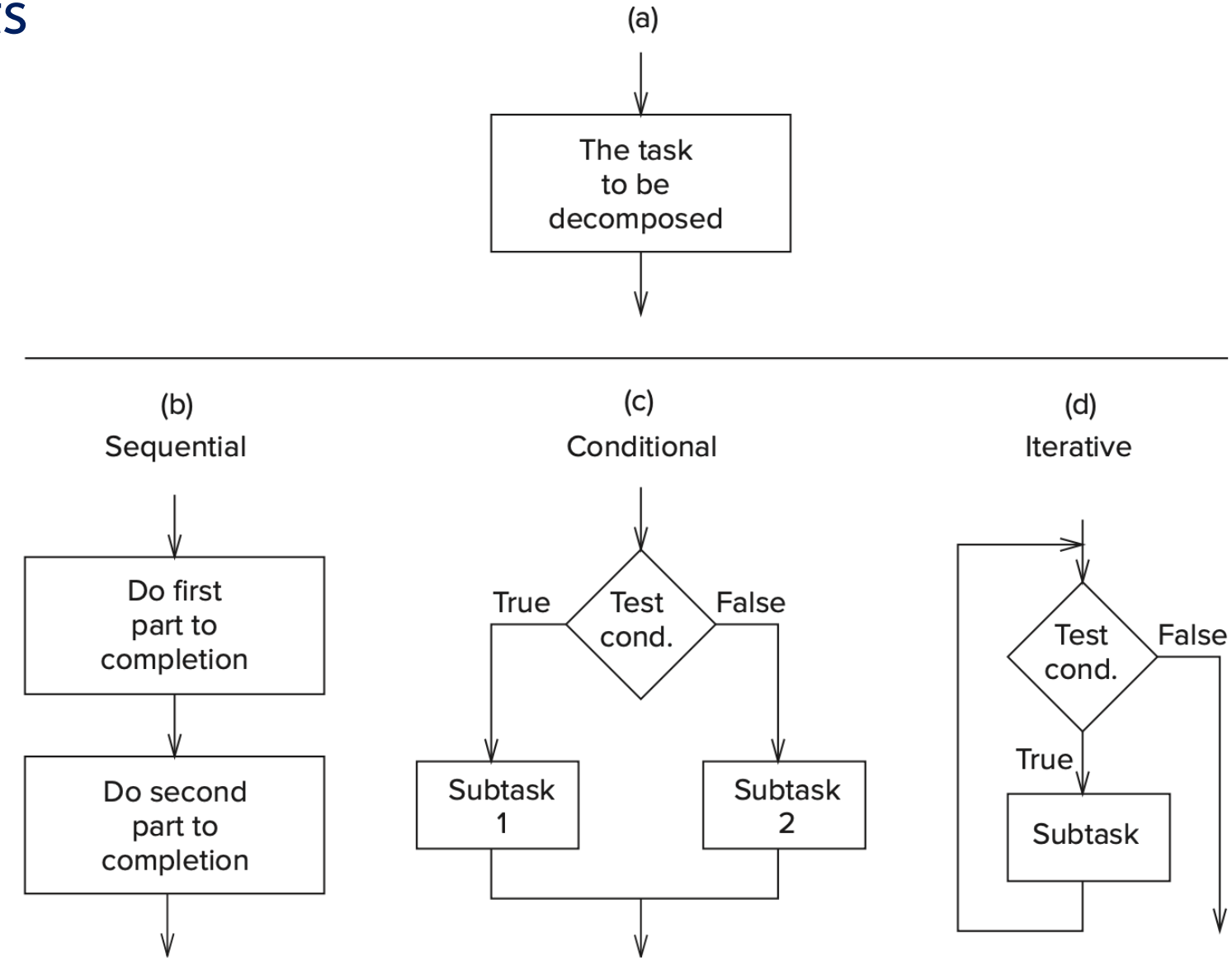
Three Basic Constructs

"do A then do B" ⇒ sequential

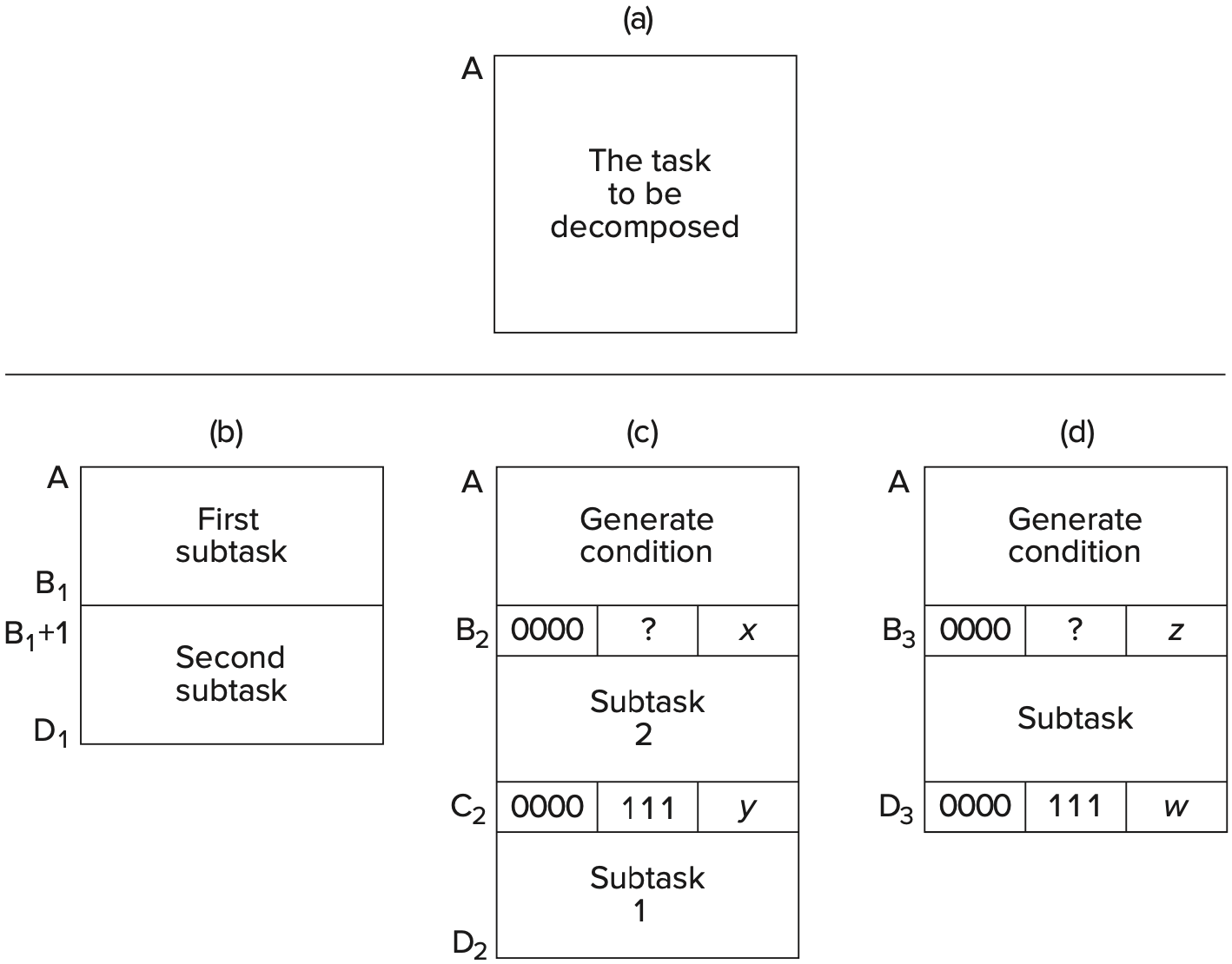
"if G, then do H" ⇒ conditional

"for each X, do Y" ⇒ iterative

"do Z until W" ⇒ iterative



LC-3 Control Instructions to Implement the Three Constructs

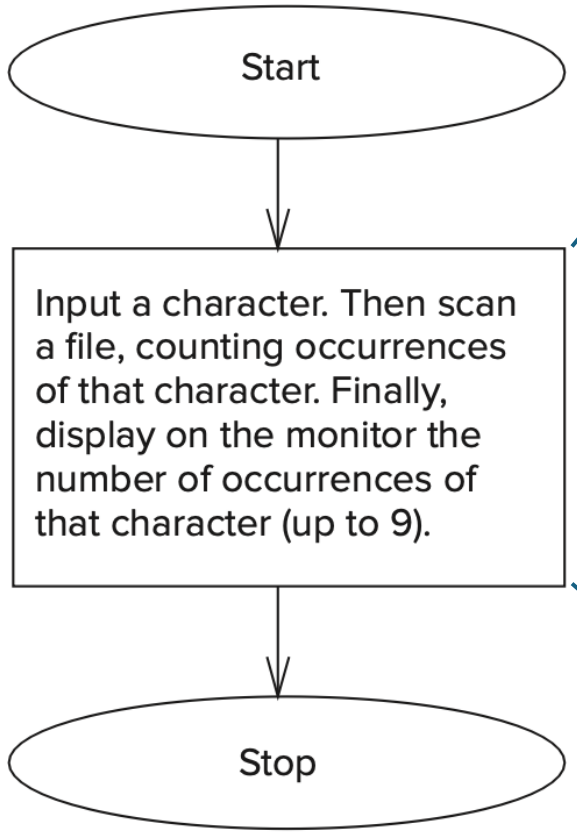


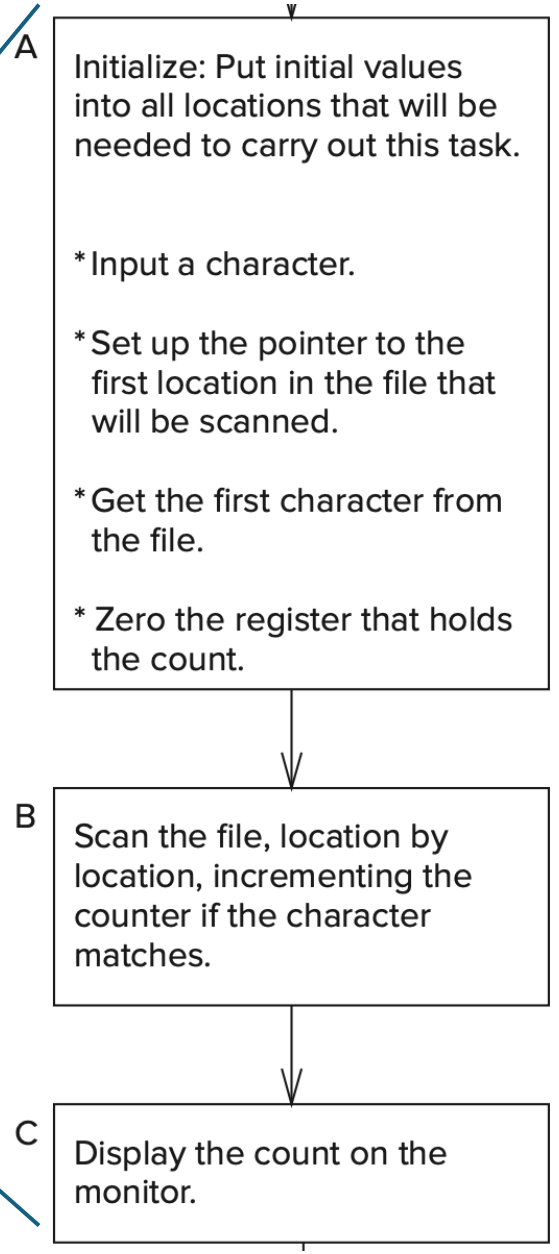
The Character Count Example from Chapter 5, Revisited

- **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), called a sentinel
- At the end, print the number of characters and halt

(assume there will be less than 10 occurrences of the character)

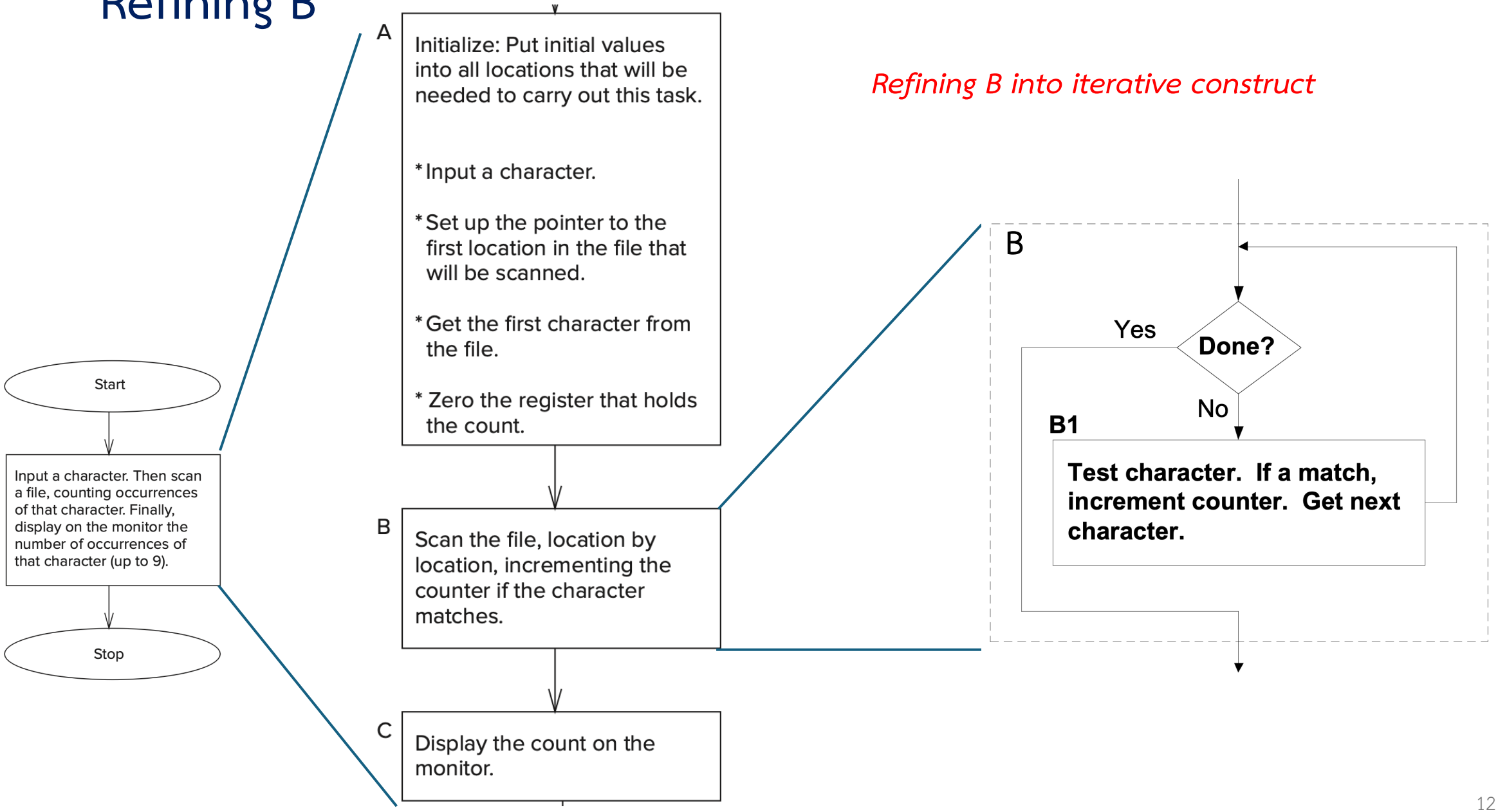
The Character Count Example from Chapter 5, Revisited





Initial refinement: Big task into three sequential subtasks

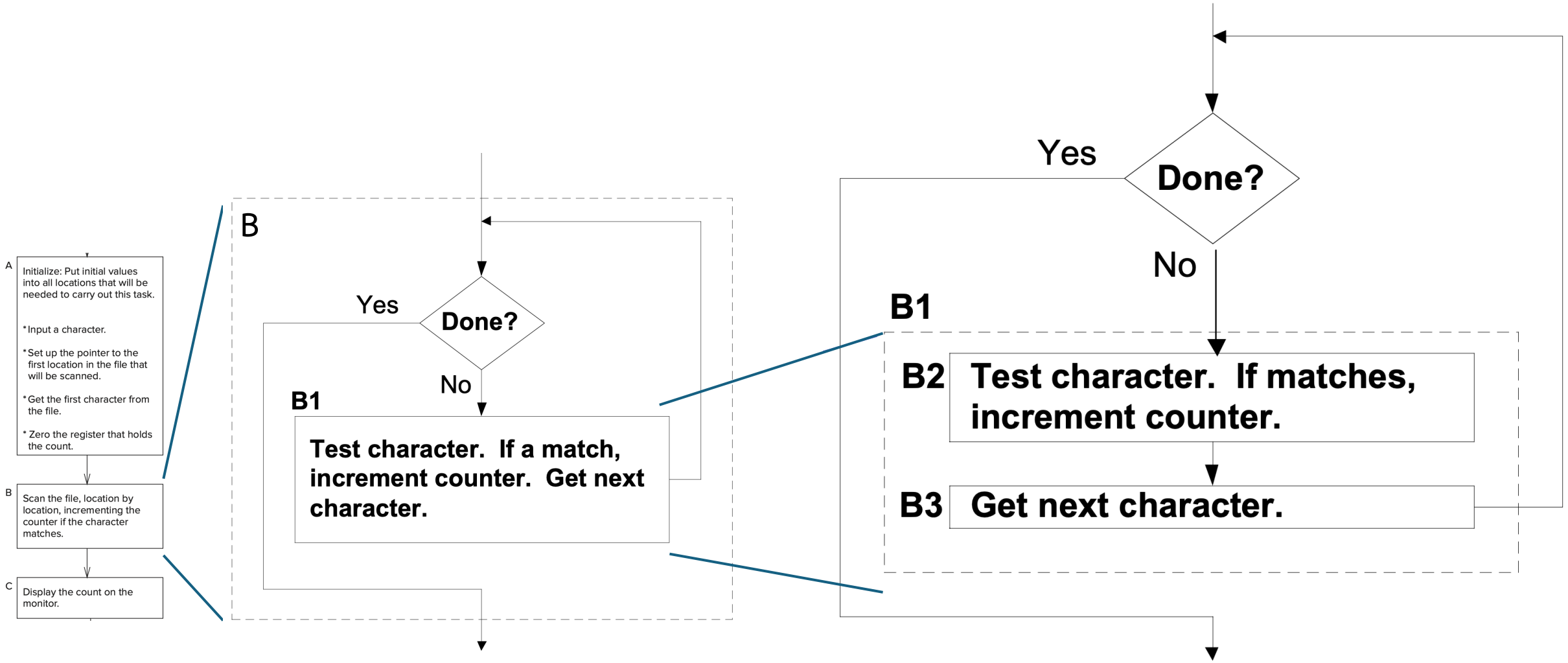
Refining B



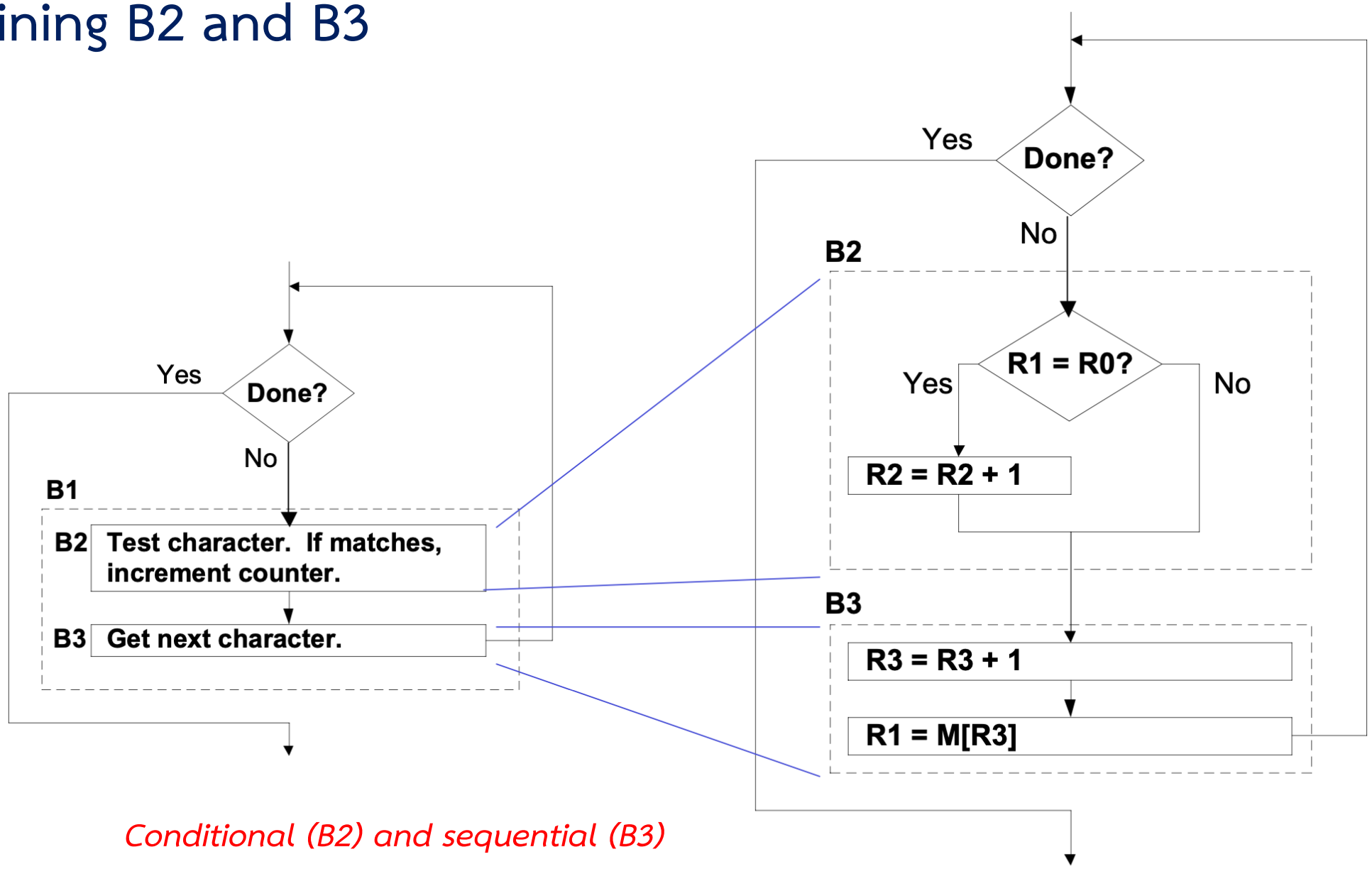
Refining B into iterative construct

Refining B1

Refining B1 into sequential subtasks



Refining B2 and B3

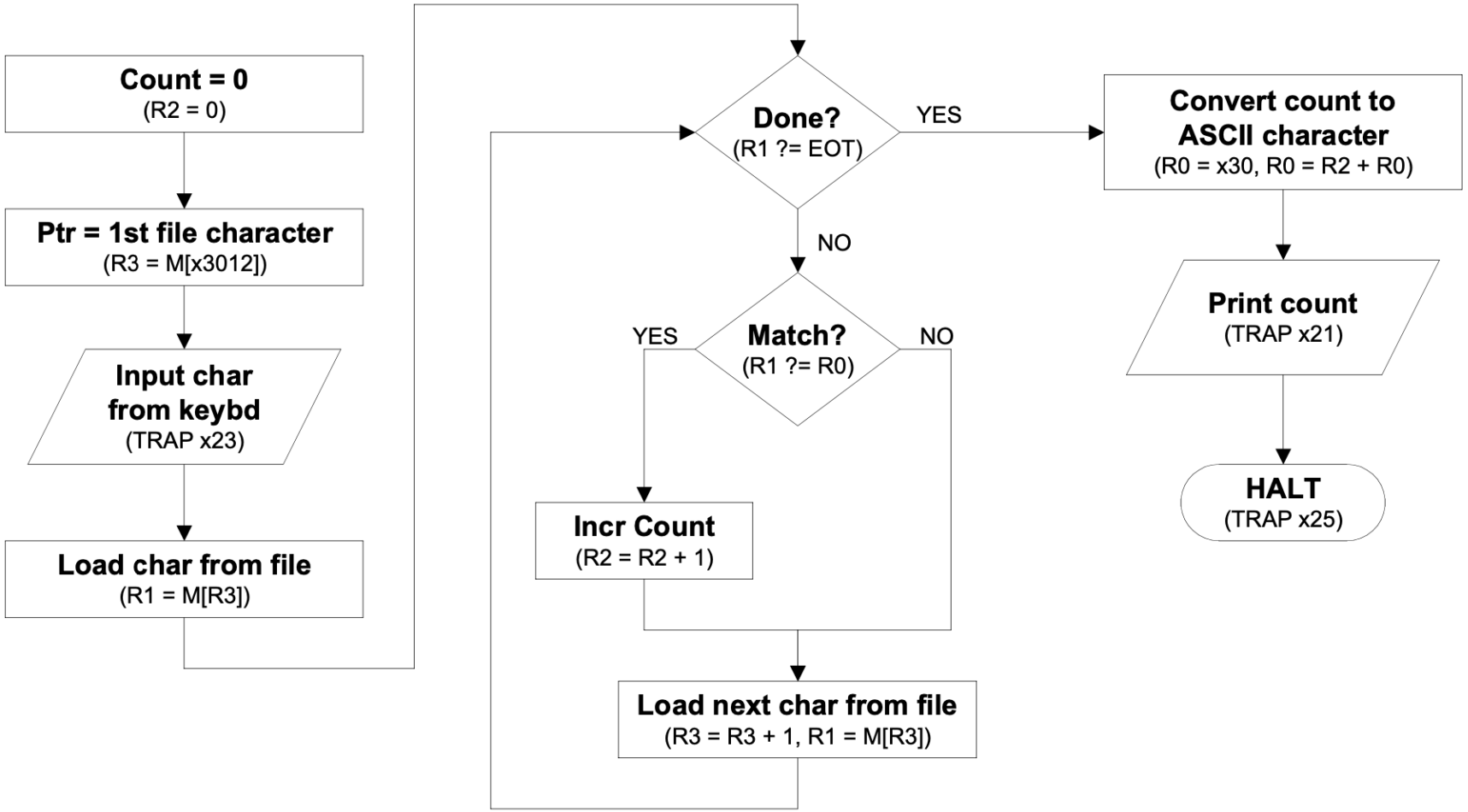


Conditional (B2) and sequential (B3)

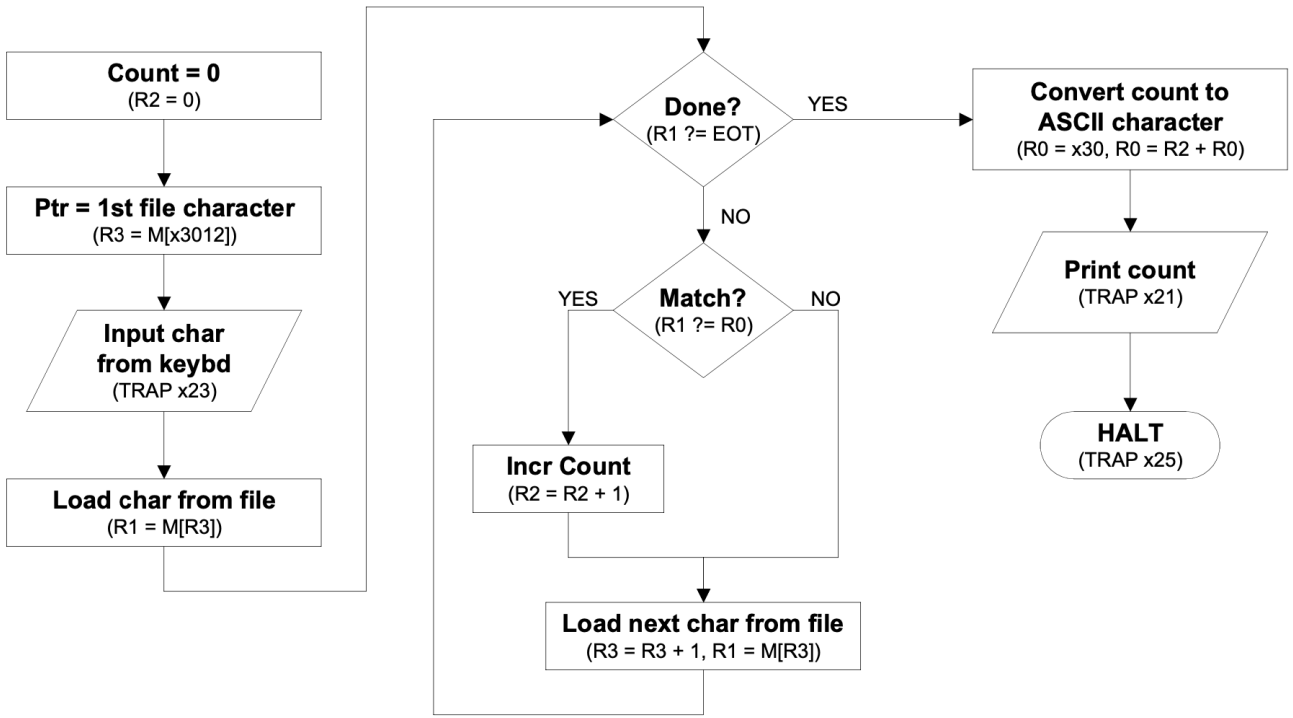
Use of LC-3 registers and instructions

Entire Flow Chart

- **Input: M[x3012]** (address of “file”)
- **Output:** Print count to display



Translate to Pseudocode

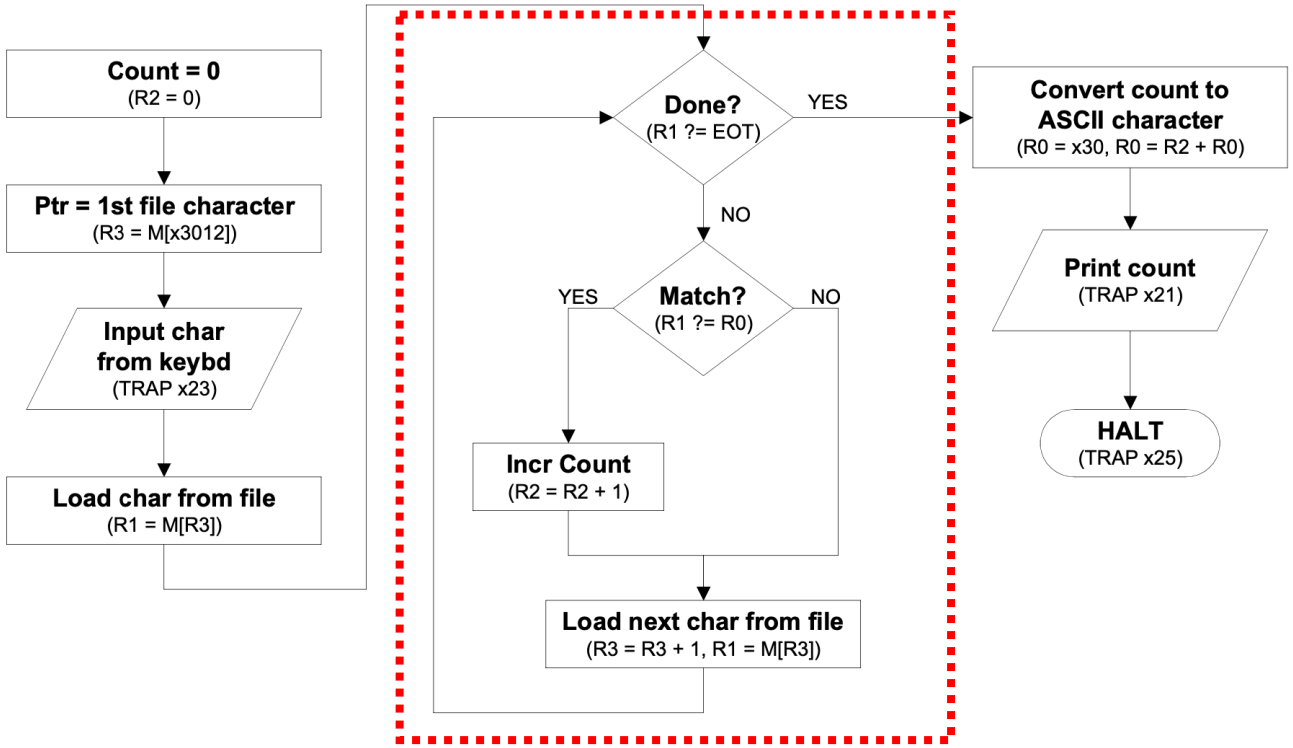


R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]

?

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

Iterative Construct in Pseudocode

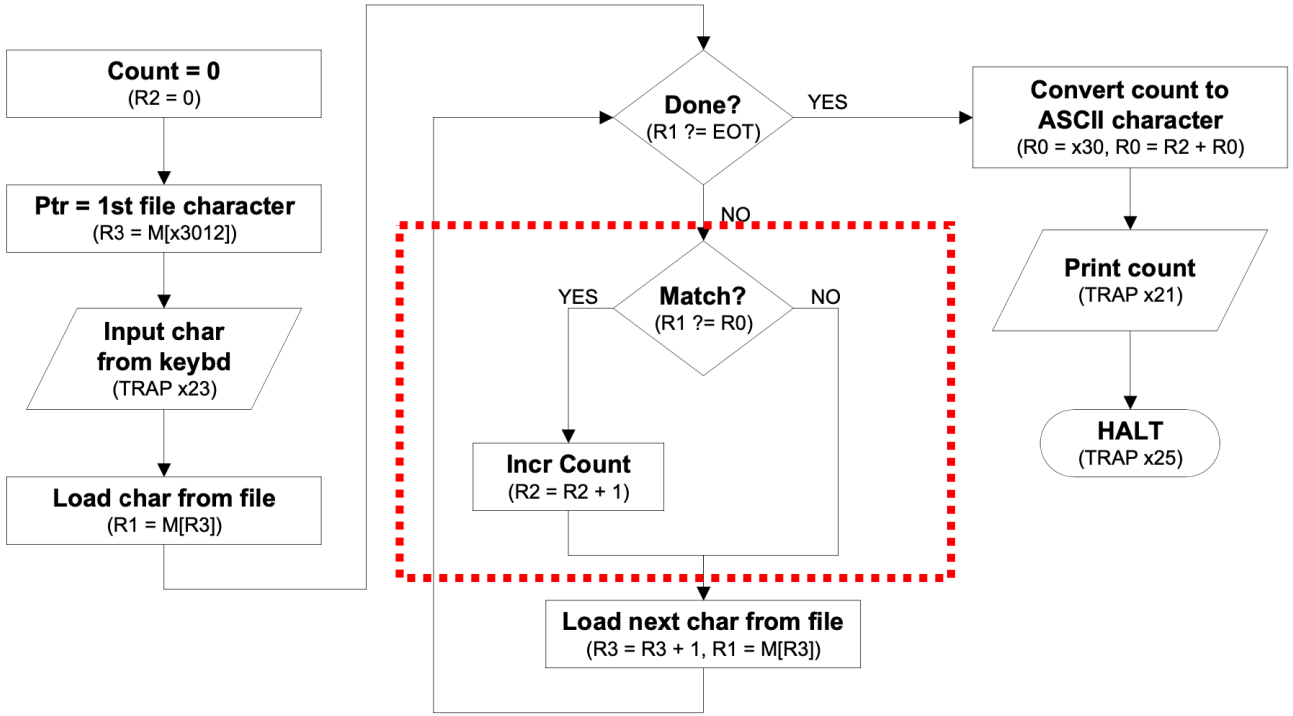


R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]
R4 ← R1 – 4 (EOT)
BRz x????

?

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

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????
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)
```

Machine Language Program

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 <- 0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 <- M[x3012]
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	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
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
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	TRAP x21
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	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 TEMPLATE

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

Program (1 of 2)

Address	Instruction	Comments
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0 (counter)* *AND R2,R2, #0*
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	*R3 ← M[x3012] (ptr)* *LD R3, x3012 (LD R3, PTR)*
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	*Input to R0 (TRAP x23)* *TRAP x23 (GETC)*
TEST x3003	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]* *LDR R1, R3, #0*
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	*R4 ← R1 – 4 (EOT)* *ADD R4,R1, #-4*
x3005	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0	*If Z, goto x300E* *BRz x300E (BRz OUTPUT)*
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	*R1 ← NOT R1* *NOT R1,R1*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1 + 1* *ADD R1,R1,#1*
X3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	*R1 ← R1 + R0* *ADD R1,R1,R0*
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	*If N or P, goto x300B* *BRnp x300B (BRnp GETCHAR)*

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

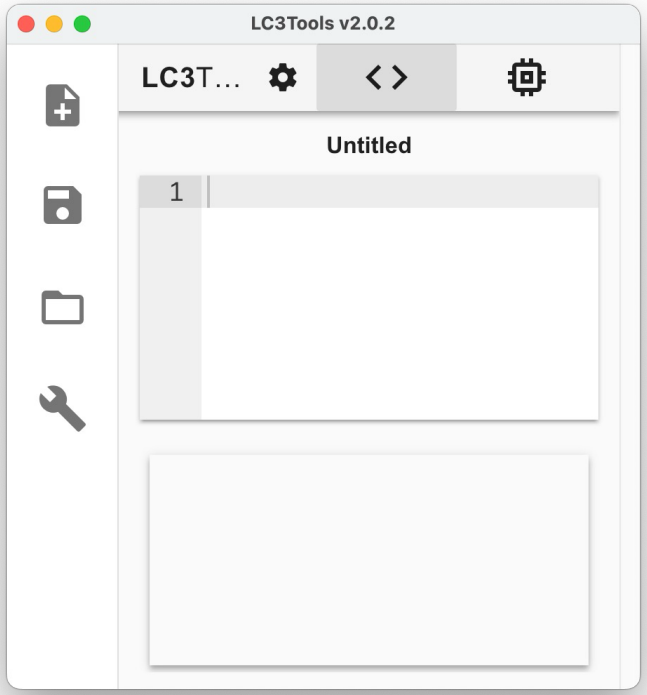
Program (2 of 2)

	Address	Instruction	Comments
	x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	*R2 ← R2 + 1* *ADD R2,R2,#1*
GETCHAR	x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	*R3 ← R3 + 1* *ADD R3,R3,#1*
	x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]* *LDR R1,R3,#0*
	x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	*Goto x3004* *BRnzp x3004 (BRnzp TEST)*
OUTPUT	x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	*R0 ← M[x3013]* *LD R0,x3013 (LD R0, ASCII)*
	x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0	*R0 ← R0 + R2* *ADD R0,R0,R2*
	x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	*Print R0* *TRAP x21 (OUT)*
	x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	*HALT* *TRAP x25 (HALT)*
	X3012	1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	Starting Address of File (X9000)
	x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	*ASCII x30 ('0')*

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

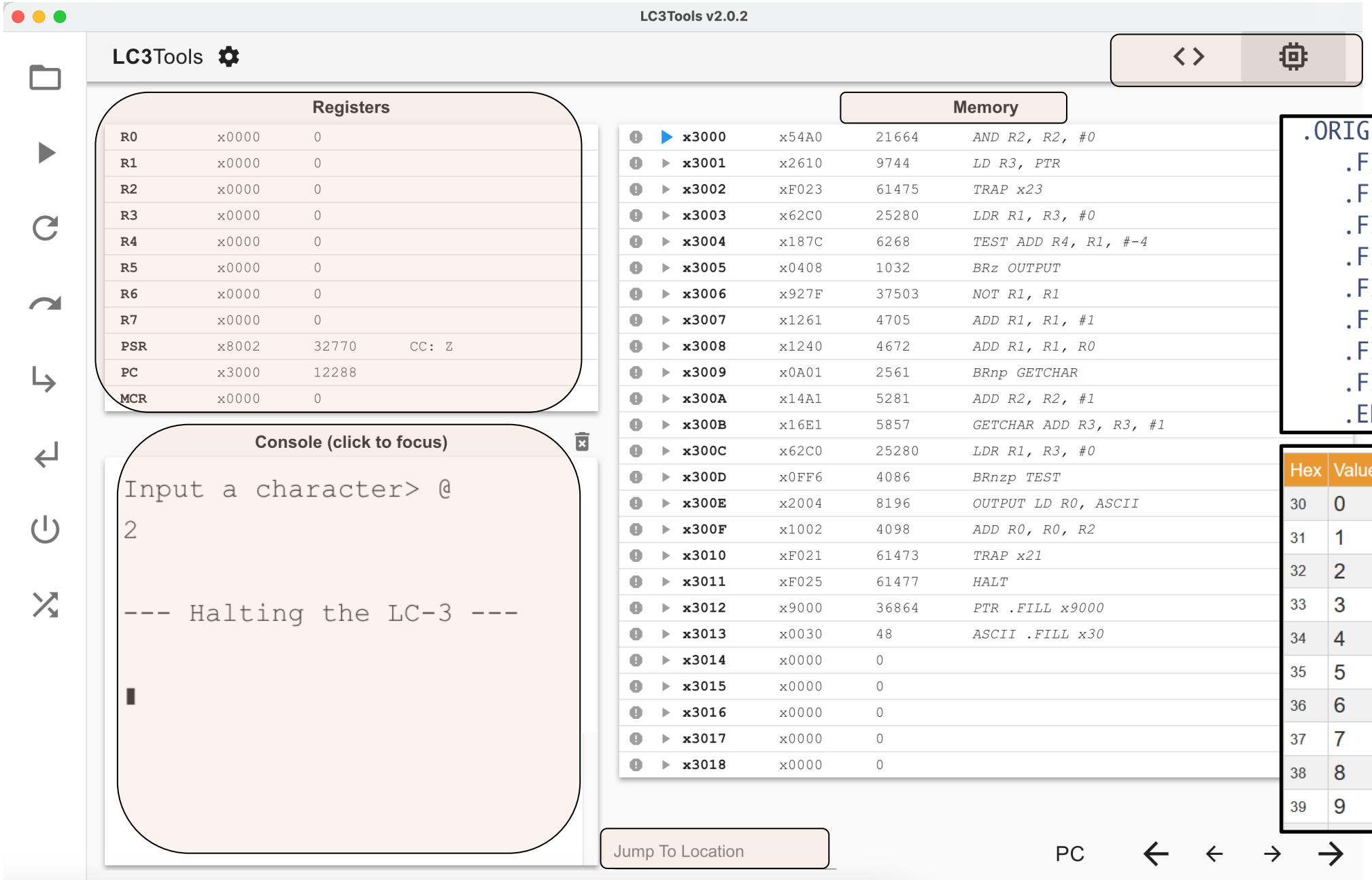
Assambly Program

Try simulator




```
1  .ORIG x3000
2          AND R2, R2, #0
3          LD R3, PTR
4          TRAP x23
5          LDR R1, R3, #0
6  TEST    ADD R4, R1, #-4
7          BRz OUTPUT
8          NOT R1, R1
9          ADD R1, R1, #1
10         ADD R1, R1, R0
11         BRnp GETCHAR
12         ADD R2, R2, #1
13 GETCHAR ADD R3, R3, #1
14         LDR R1, R3, #0
15         BRnzp TEST
16 OUTPUT  LD R0, ASCII
17         ADD R0, R0, R2
18         TRAP x21
19     ;
20         HALT
21 PTR     .FILL x9000
22 ASCII   .FILL x30
23     .END
24 ;
25  .ORIG X9000
26     .FILL x0031
27     .FILL x0032
28     .FILL x0031
29     .FILL x0040
30     .FILL x0033
31     .FILL x0043
32     .FILL x0040
33     .FILL x04
34     .END
```

```
 .ORIG x3000
        AND R2, R2, #0
        LD R3, PTR
        TRAP x23
        LDR R1, R3, #0
TEST    ADD R4, R1, #-4
        BRz OUTPUT
        NOT R1, R1
        ADD R1, R1, #1
        ADD R1, R1, R0
        BRnp GETCHAR
        ADD R2, R2, #1
GETCHAR ADD R3, R3, #1
        LDR R1, R3, #0
        BRnzp TEST
OUTPUT  LD R0, ASCII
        ADD R0, R0, R2
        TRAP x21
    ;
        HALT
PTR     .FILL x9000
ASCII   .FILL x30
    .END
;
    .ORIG X9000
    .FILL x0031
    .FILL x0032
    .FILL x0031
    .FILL x0040
    .FILL x0033
    .FILL x0043
    .FILL x0040
    .FILL x04
    .END
```


LC3Tools v2.0.2
LC3Tools
Registers
R0 x0000 0
R1 x0000 0
R2 x0000 0
R3 x0000 0
R4 x0000 0
R5 x0000 0
R6 x0000 0
R7 x0000 0
PSR x8002 32770 CC: Z
PC x3000 12288
MCR x0000 0
Console (click to focus)
Input a character> @
2
--- Halting the LC-3 ---
Memory
x3000 x54A0 21664 AND R2, R2, #0
x3001 x2610 9744 LD R3, PTR
x3002 xF023 61475 TRAP x23
x3003 x62C0 25280 LDR R1, R3, #0
x3004 x187C 6268 TEST ADD R4, R1, #-4
x3005 x0408 1032 BRz OUTPUT
x3006 x927F 37503 NOT R1, R1
x3007 x1261 4705 ADD R1, R1, #1
x3008 x1240 4672 ADD R1, R1, R0
x3009 x0A01 2561 BRnp GETCHAR
x300A x14A1 5281 ADD R2, R2, #1
x300B x16E1 5857 GETCHAR ADD R3, R3, #1
x300C x62C0 25280 LDR R1, R3, #0
x300D x0FF6 4086 BRnzp TEST
x300E x2004 8196 OUTPUT LD R0, ASCII
x300F x1002 4098 ADD R0, R0, R2
x3010 xF021 61473 TRAP x21
x3011 xF025 61477 HALT
x3012 x9000 36864 PTR .FILL x9000
x3013 x0030 48 ASCII .FILL x30
x3014 x0000 0
x3015 x0000 0
x3016 x0000 0
x3017 x0000 0
x3018 x0000 0
Jump To Location
PC
.ORIG X9000
.FILL x0031
.FILL x0032
.FILL x0031
.FILL x0040
.FILL x0033
.FILL x0043
.FILL x0040
.FILL x04
.END
Hex Value Hex Value Hex Value
30 0 40 @ 50 P
31 1 41 A 51 Q
32 2 42 B 52 R
33 3 43 C 53 S
34 4 44 D 54 T
35 5 45 E 55 U
36 6 46 F 56 V
37 7 47 G 57 W
38 8 48 H 58 X
39 9 49 I 59 Y

Debugging

Debugging

- You've written your program, and it **doesn't work → Then 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. 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, Python, …) have source-code debugging tools
  - For debugging at the machine instruction level:
    - Simulators
    - Operating system “monitor” tools
    - Special hardware

Types of Errors

- **Syntax Errors**
  - Typing error that resulted in an illegal operation
  - Machine language: not caught, because almost any bit pattern corresponds to some legal instruction
  - High-level language: caught during the translation from language to machine code
- **Logic Errors**
  - 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

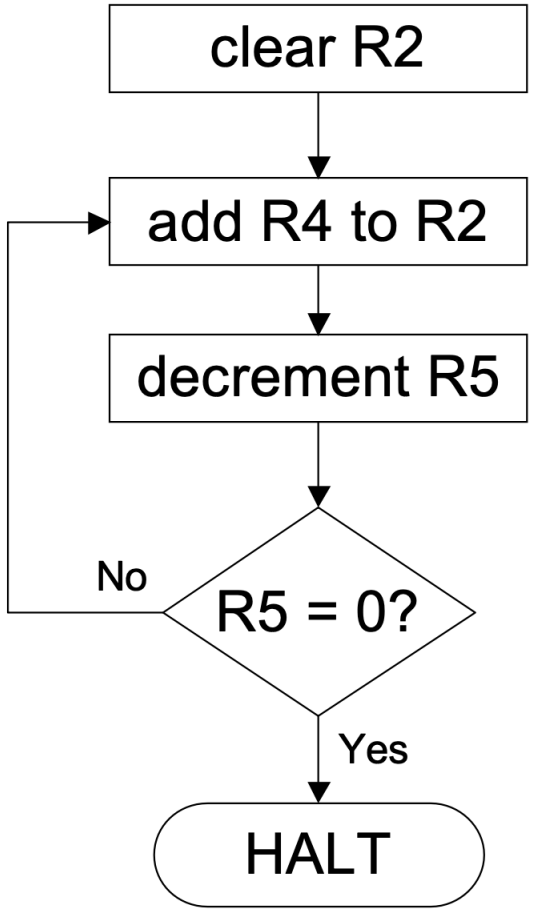
Tracing the Program

- 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
    - Let’s you quickly execute sequences to get a high-level overview of the execution behavior
    - Quickly execute sequences that you believe are correct

Example 1: Multiply

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

- Goal: Multiply the **two unsigned integers** in R4 and R5, and **place result in R2**




```
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)

***

```
1  .ORIG x3200
2        AND R2, R2, #0
3  TEST  ADD R2, R2, R4
4        ADD R5, R5, #-1
5        BRzp TEST
6     ;
7        HALT
8     .END
```

*** Try simulator ***

Debugging the Multiply Program in Single-Stepping

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

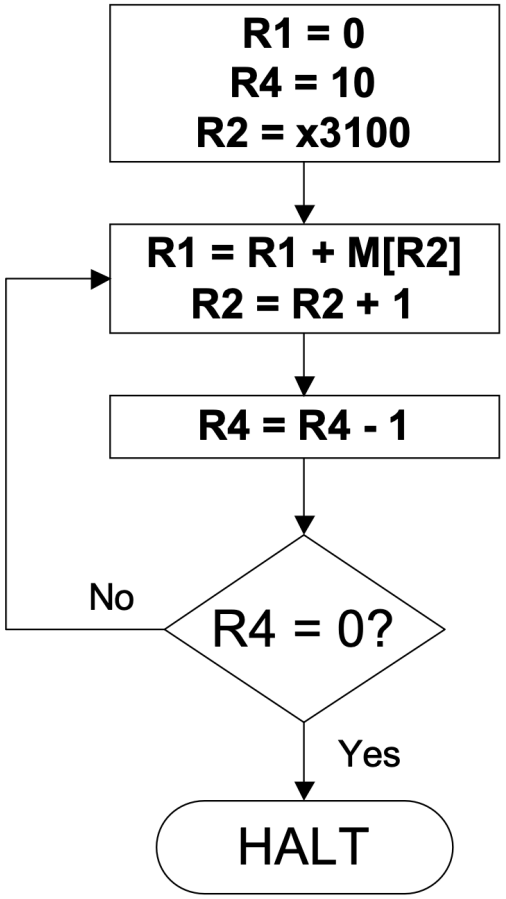
Executing loop one time too many

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




```
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
```

Address	Contents
x3100	x3107
x3101	x2819
x3102	x0110
x3103	x0310
x3104	x0110
x3105	x1110
x3106	x11B1
x3107	x0019
x3108	x0007
x3109	x0004
x310A	x0000
x310B	x0000
x310C	x0000
x310D	x0000
x310E	x0000
x310F	x0000
x3110	x0000
x3111	x0000
x3112	x0000
x3113	x0000

Debugging the Summing Program

- Check R1, the sum should be **x8135, What happened?**
- **Try single-stepping program**

PC	R1	R2	R4
x3001	0	x	x
x3002	0	x	0
x3003	0	x	#10
x3004	0	x3107	#10

Should be x3100

- R2 contains x3107, not x3100 as we had expected
- The opcode loaded the contents of M[x3100] (i.e., x3107) into R2, not the **address x3100**

Example 2: Summing an Array of Numbers

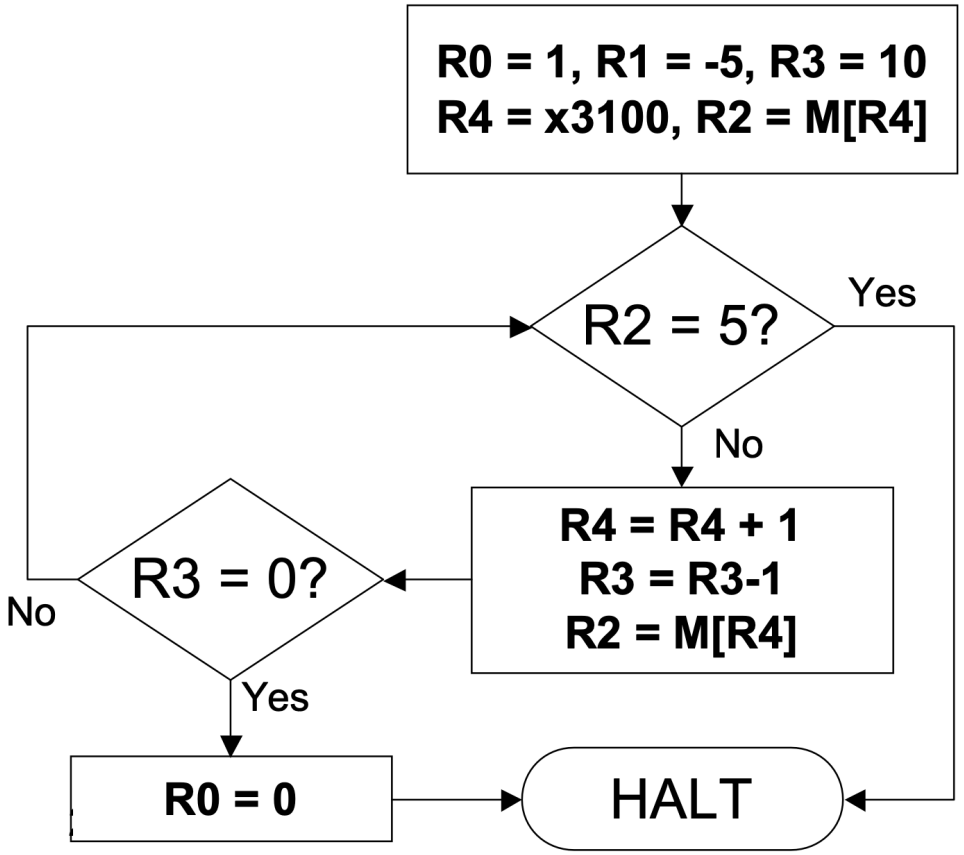
*** Try simulator ***

```
1   .ORIG x3000
2         AND R1, R1, #0
3         AND R4, R4, #0
4         ADD R4, R4, #10
5         LD  R2, PTR
6   TEST  LDR R3, R2, #0
7         ADD R2, R2, #1
8         ADD R1, R1, R3
9         ADD R4, R4, #-1
10        BRp TEST
11      ;
12  PTR .FILL x3100
13      .END
14  ;
```

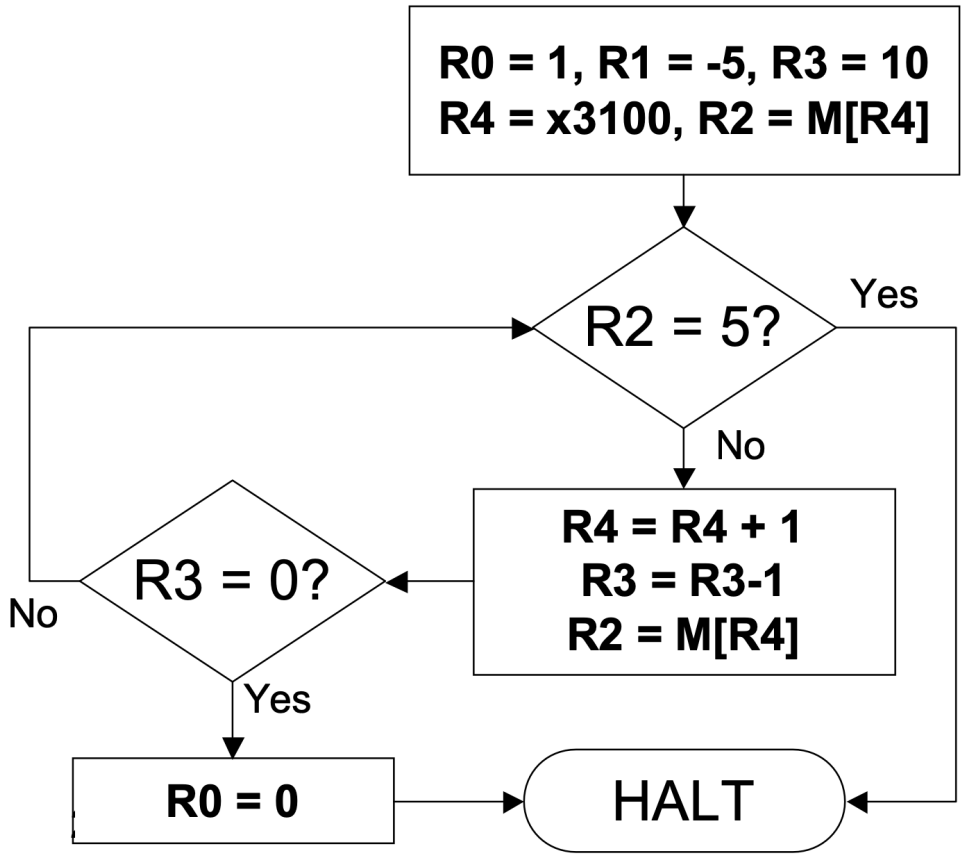
```
15  .ORIG X3100
16      .FILL x3107
17      .FILL x2819
18      .FILL x0110
19      .FILL x0310
20      .FILL x0110
21      .FILL x1110
22      .FILL x11B1
23      .FILL x0019
24      .FILL x0007
25      .FILL x0004
26      .END
```

Example 3: Does a Sequence of Memory Locations Contain a 5?

- 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



Example 3: Does a Sequence of Memory Locations Contain a 5?

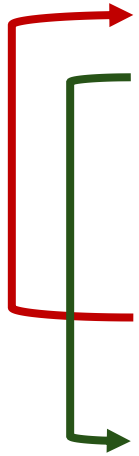



```
x3000 AND   R0,R0,#0
x3001 ADD   R0,R0,#1
x3002 AND   R1,R1,#0
x3003 ADD   R1,R1,#-5
x3004 AND   R3,R3,#0
x3005 ADD   R3,R3,#10
x3006 LD    R4,x3010
x3007 LDR   R2,R4,#0
x3008 ADD   R2,R2,R1
x3009 BRz   x300F
x300A ADD   R4,R4,#1
x300B ADD   R3,R3,#-1
x300C LDR   R2,R4,#0
x300D BRp   x3008
x300E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

Debugging Looking for the Fives Program

```
.ORIG X3100
    .FILL #9
    .FILL #7
    .FILL #32
    .FILL #0
    .FILL #-8
    .FILL #19
    .FILL #6
    .FILL #13
    .FILL #5
    .FILL #61
    .END
```

```
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
```



PC	R1	R2	R3	R4
x300D	−5	7	9	3101
x300D	−5	32	8	3102
x300D	−5	0	7	3013

Wrong Branch Back

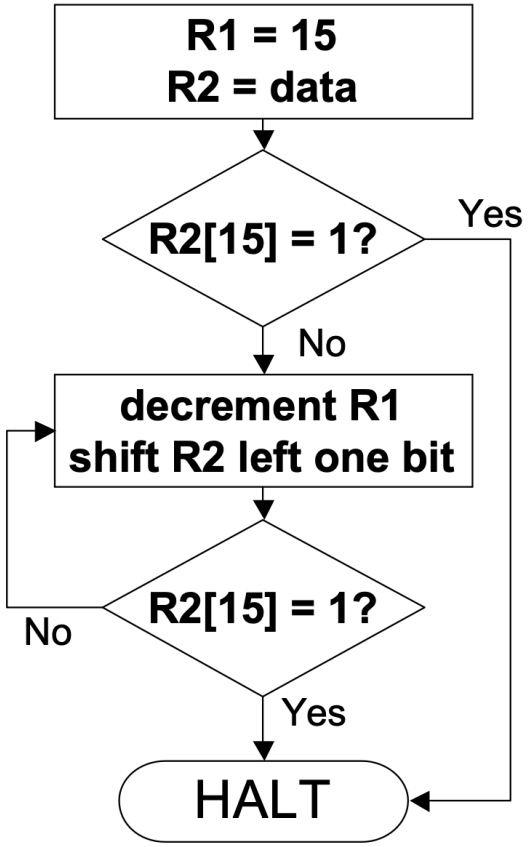
Must check R3, not R2

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 the 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

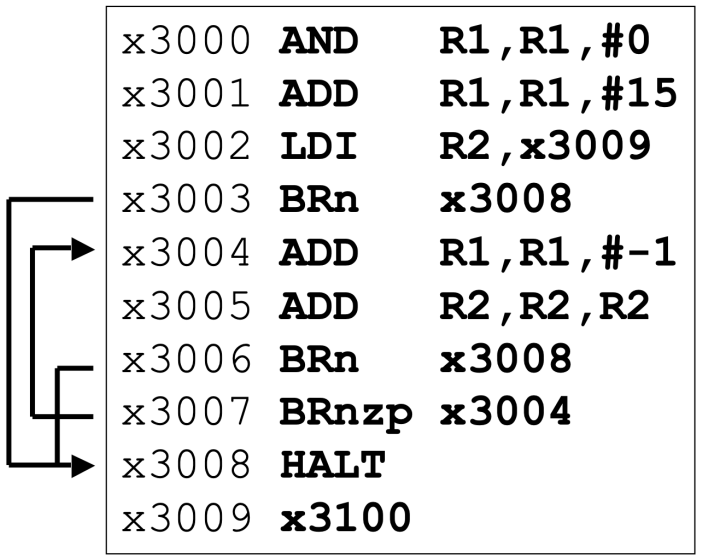
- We often want to manipulate individual bits
  - Example: is a number odd or even?
  - → Answer: R1 := R0 AND 0x1
    - If R1 is 0 -> R0 was even
    - If R1 is 1 -> R0 was odd
- LC-3 doesn't give us an instruction to "shift" bits
  - *Most ISAs include "shift left" and "shift right"*
  - Example: If 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 = 0100**

** 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

Try Breakpoint at backwards branch (x3007)



```
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
```

PC	R1
x3007	14
x3007	13
x3007	12
x3007	11
x3007	10
x3007	9
x3007	8
x3007	7
x3007	6
x3007	5
x3007	4
x3007	3
x3007	2
x3007	1
x3007	0
x3007	−1
x3007	−2
x3007	−3
x3007	−4

If no ones, then branch to HALT never occurs!

"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, …)

Homework

- 6.4, 6.5, 6.6, 6.7
- 6.9, 6.18