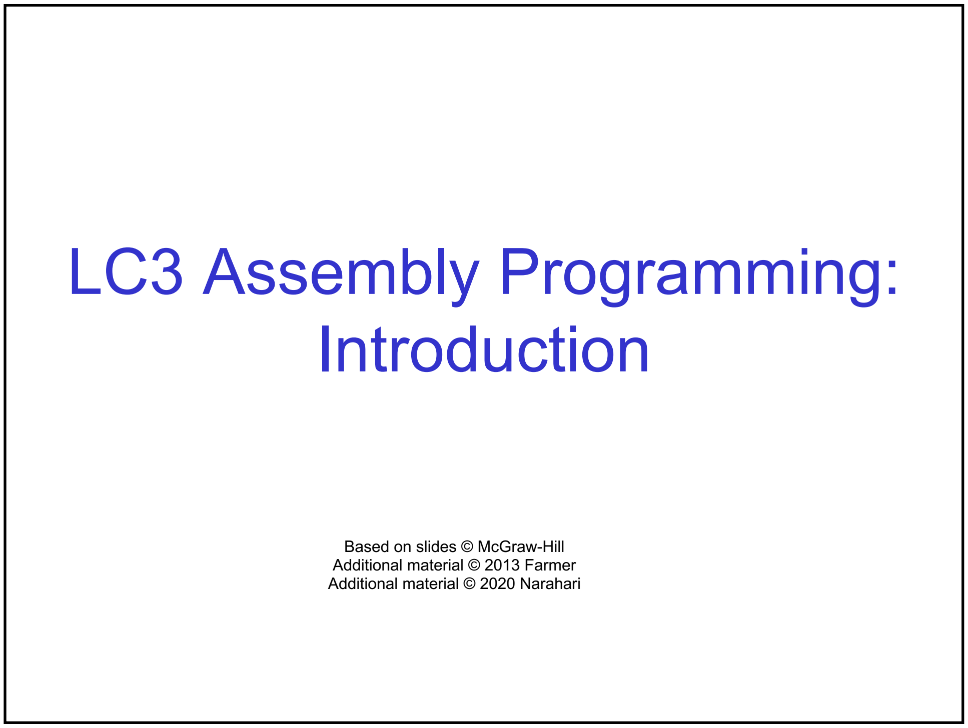
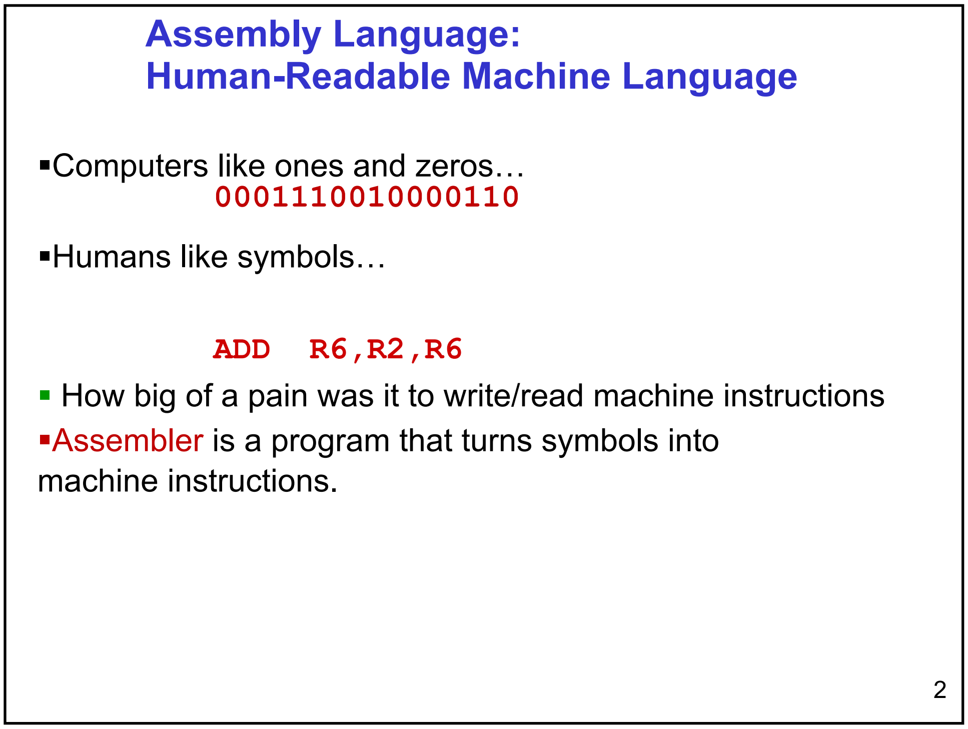


LC3 Assembly Programming: Introduction

1



Assembly Language: Human-Readable Machine Language

- Computers like ones and zeros…

```
0001110010000110
```

- Humans like symbols…

```
ADD  R6,R2,R6
```

- How big of a pain was it to write/read machine instructions
- Assembler is a program that turns symbols into machine instructions.

2

2

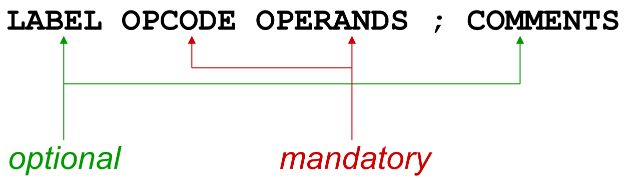
Programming in Assembly

- Assembly language level is one-step up from machine
  - All instructions used in Assembly are actual machine instructions….*somewhat!*
  - Use mnemonics and address labels to make it easier to understand the program
    - Labels converted to addresses and offsets by assembler
  - “macros” and utilities to make it easier
- Assembler directives
  - Tell assembler what to do without the programmer explicitly writing out the machine code to do the task
  - Allocating storage
  - Initializing data

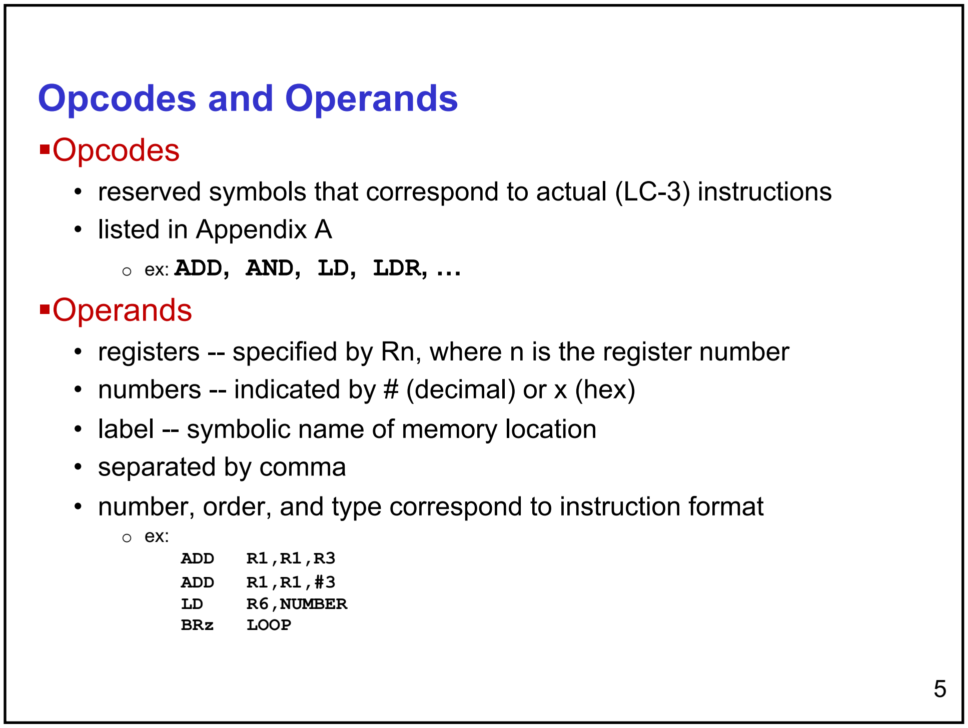
LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  - an instruction
  - an assember directive (or pseudo-op)
  - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with “;”) one one line are also ignored.

- An instruction has the following format:



```
LABEL OPCODE OPERANDS ; COMMENTS
```



Opcodes and Operands

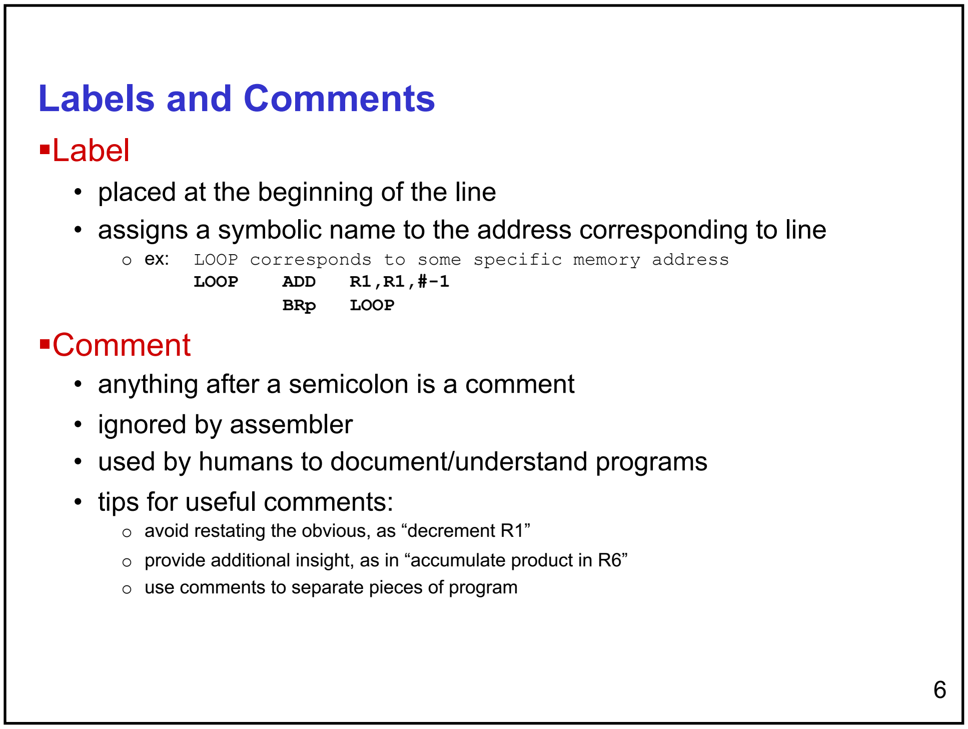
▪Opcodes

- reserved symbols that correspond to actual (LC-3) instructions
- listed in Appendix A
  - ex: **ADD, AND, LD, LDR, ...**

▪Operands

- registers -- specified by Rn, where n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - ex:

```
ADD   R1,R1,R3
ADD   R1,R1,#3
LD    R6,NUMBER
BRz   LOOP
```



Labels and Comments

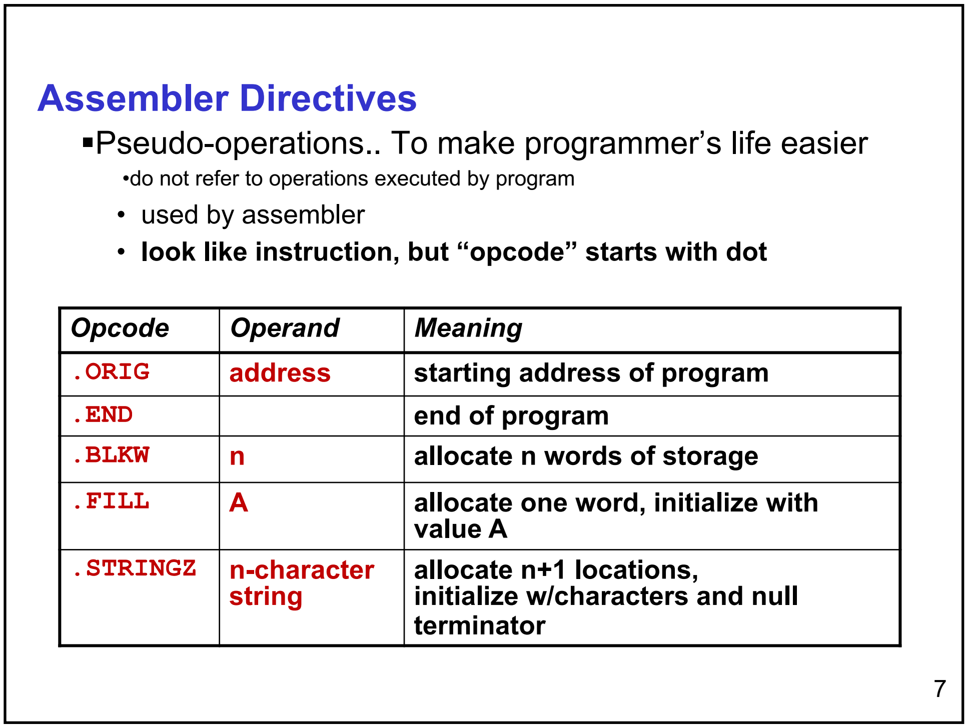
▪Label

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  - ex:

```
LOOP corresponds to some specific memory address
LOOP    ADD   R1,R1,#-1
        BRp   LOOP
```

▪Comment

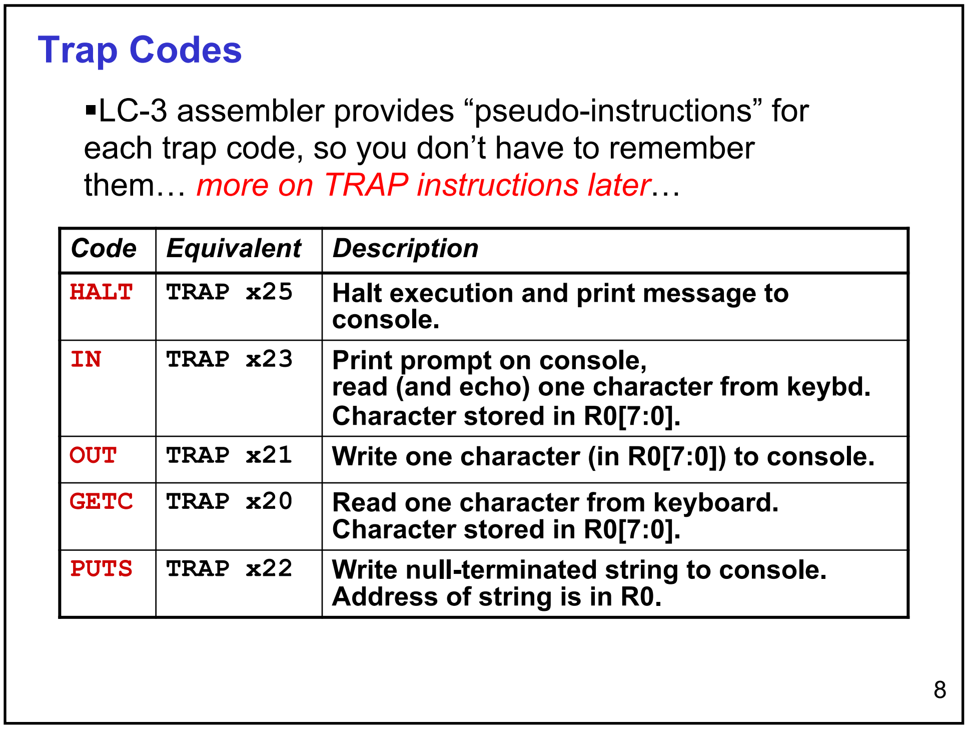
- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs
- tips for useful comments:
  - avoid restating the obvious, as “decrement R1”
  - provide additional insight, as in “accumulate product in R6”
  - use comments to separate pieces of program



Assembler Directives

- Pseudo-operations.. To make programmer’s life easier
  - do not refer to operations executed by program
  - used by assembler
  - **look like instruction, but “opcode” starts with dot**

Opcode	*Operand*	*Meaning*
.ORIG	address	starting address of program
.END		end of program
.BLKW	n	allocate n words of storage
.FILL	A	allocate one word, initialize with value A
.STRINGZ	n-character string	allocate n+1 locations, initialize w/characters and null terminator



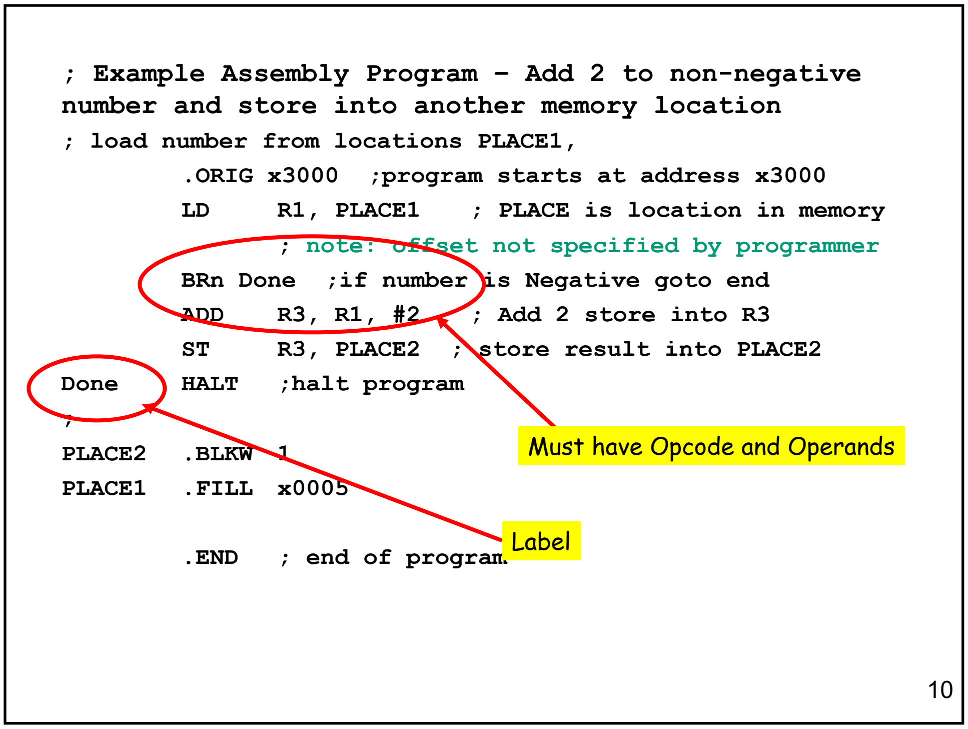
Trap Codes

- LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them… *more on TRAP instructions later*…

Code	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard. Character stored in R0[7:0].
PUTS	TRAP x22	Write null-terminated string to console. Address of string is in R0.

```
; Example Assembly Program - Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
        .ORIG x3000   ;program starts at address x3000
        LD     R1, PLACE1    ; PLACE is location in memory
               ; note: offset not specified by
                 programmer
               ; assembler calculates offset needed
        BRn Done  ;if number is Negative goto end
        ADD    R3, R1, #2    ; Add 2 store into R3
        ST     R3, PLACE2   ; store result into PLACE2
Done    HALT   ;halt program
;
PLACE2  .BLKW  1 ; reserve/set aside one word in memory
PLACE1  .FILL  x0005         ; initialize number to 5

        .END   ; end of program
```

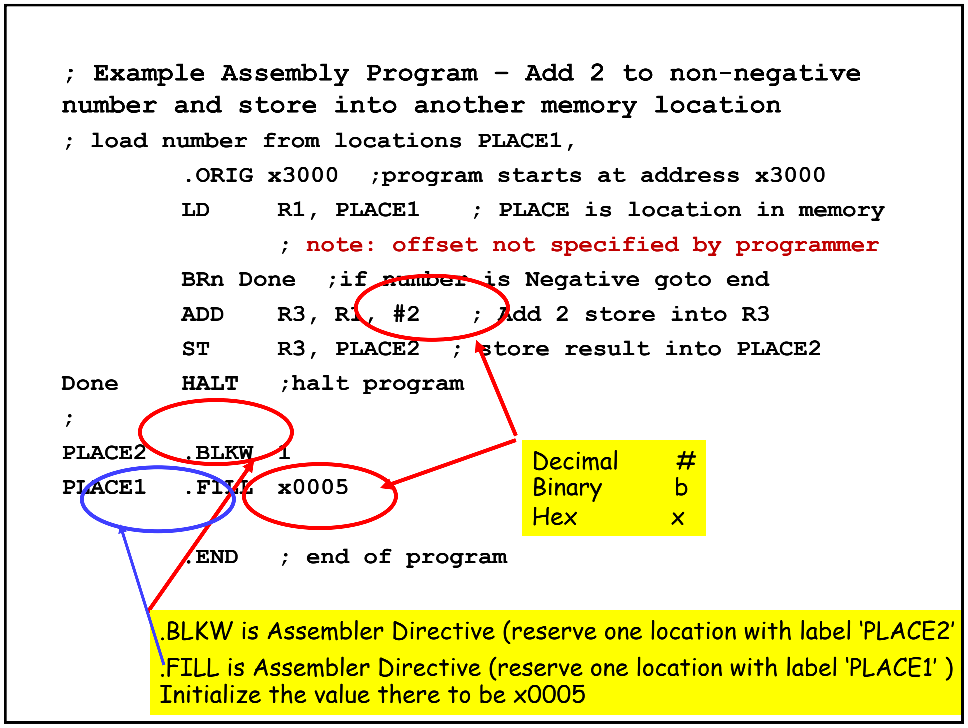


```
; Example Assembly Program - Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
        .ORIG x3000   ;program starts at address x3000
        LD     R1, PLACE1    ; PLACE is location in memory
               ; note: offset not specified by programmer
        BRn Done  ;if number is Negative goto end
        ADD    R3, R1, #2    ; Add 2 store into R3
        ST     R3, PLACE2   ; store result into PLACE2
Done    HALT   ;halt program
;
PLACE2  .BLKW  1
PLACE1  .FILL  x0005

        .END   ; end of program
```

Must have Opcode and Operands

Label



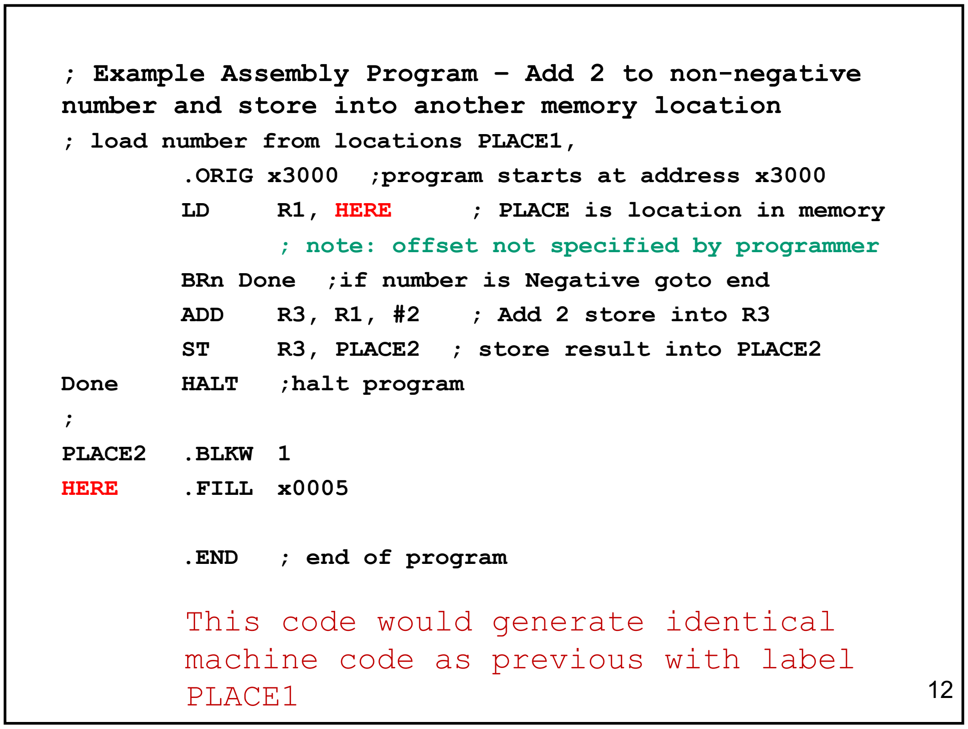
```
; Example Assembly Program - Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
         .ORIG x3000   ;program starts at address x3000
         LD     R1, PLACE1    ; PLACE is location in memory
                ; note: offset not specified by programmer
         BRn Done   ;if number is Negative goto end
         ADD    R3, R1, #2    ; Add 2 store into R3
         ST     R3, PLACE2   ; store result into PLACE2
Done     HALT   ;halt program
;
PLACE2   .BLKW  1
PLACE1   .FILL  x0005

         .END   ; end of program
```

Decimal	#
Binary	b
Hex	x

.BLKW is Assembler Directive (reserve one location with label 'PLACE2'

.FILL is Assembler Directive (reserve one location with label 'PLACE1')
Initialize the value there to be x0005



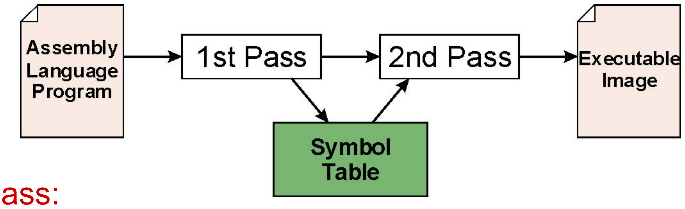
```
; Example Assembly Program - Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
         .ORIG x3000   ;program starts at address x3000
         LD     R1, HERE      ; PLACE is location in memory
                ; note: offset not specified by programmer
         BRn Done   ;if number is Negative goto end
         ADD    R3, R1, #2    ; Add 2 store into R3
         ST     R3, PLACE2   ; store result into PLACE2
Done     HALT   ;halt program
;
PLACE2   .BLKW  1
HERE     .FILL  x0005

         .END   ; end of program
```

This code would generate identical machine code as previous with label PLACE1

Assembly Process

- Assembler: Converts assembly language file (.asm) into an executable file (.obj) …*for the LC-3 simulator in our case*.



- First Pass:
  - scan program file
  - find all labels and calculate the corresponding addresses; this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table

First Pass: Constructing the Symbol Table

1. Find the `.ORIG` statement, which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      – **NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.**
3. Stop when `.END` statement is reached.

- NOTE: A line that contains only a comment is considered an empty line.

Pass 1

- Construct the symbol table for the program

Symbol	Address
Done	
PLACE2	
PLACE1	

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.

- Potential problems:
  - Improper number or type of arguments
    - ex:

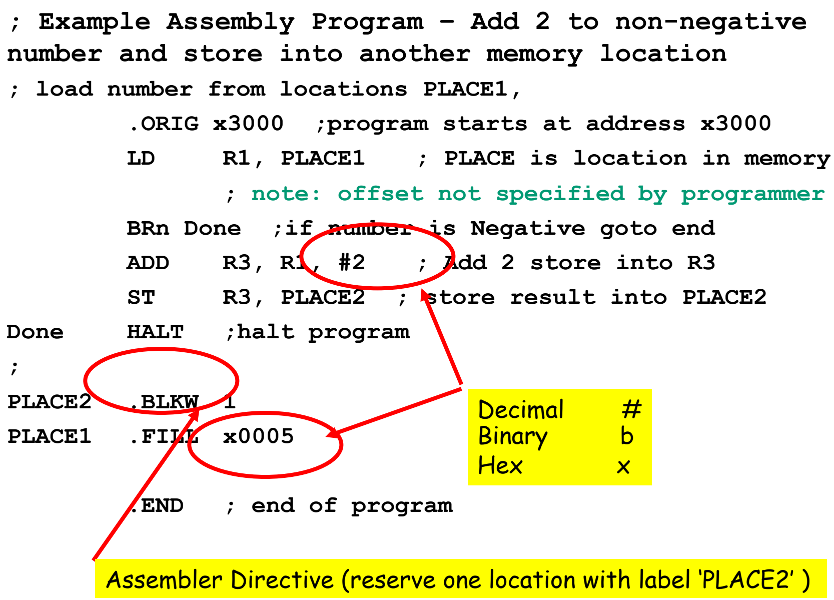
```
NOT    R1,#7
ADD    R1,R2
ADD    R3,R3,NUMBER
```

  - Immediate argument too large
    - ex: `ADD    R1,R2,#1023`
  - Address (associated with label) more than 256 from instruction
    - can't use PC-relative addressing mode

Pass 2

▪Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

Statement	Machine Language
`LD   R1, SIX`	
`BRp AGAIN`	
`LD  R2, NUMBER`	

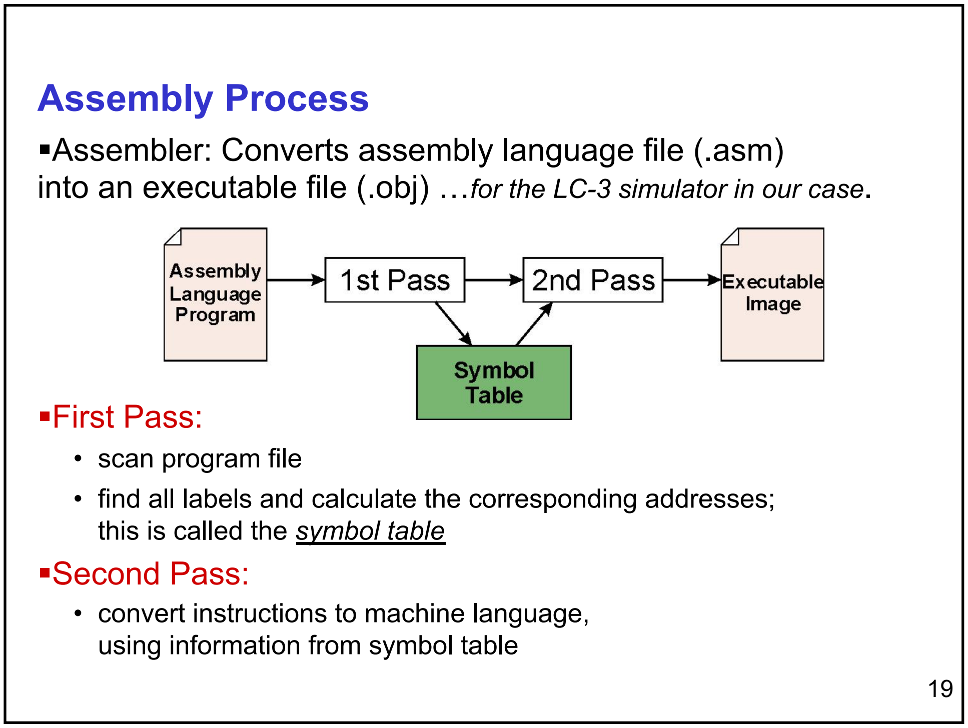


```
; Example Assembly Program - Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
        .ORIG x3000   ;program starts at address x3000
        LD     R1, PLACE1    ; PLACE is location in memory
                      ; note: offset not specified by programmer
        BRn Done   ;if number is Negative goto end
        ADD    R3, R1, #2    ; Add 2 store into R3
        ST     R3, PLACE2  ; store result into PLACE2
Done    HALT   ;halt program
;
PLACE2  .BLKW  1
PLACE1  .FILL  x0005

        .END   ; end of program
```

Decimal	#
Binary	b
Hex	x

Assembler Directive (reserve one location with label 'PLACE2')



Assembly Process

- Assembler: Converts assembly language file (.asm) into an executable file (.obj) …*for the LC-3 simulator in our case*.

Assembly Language Program

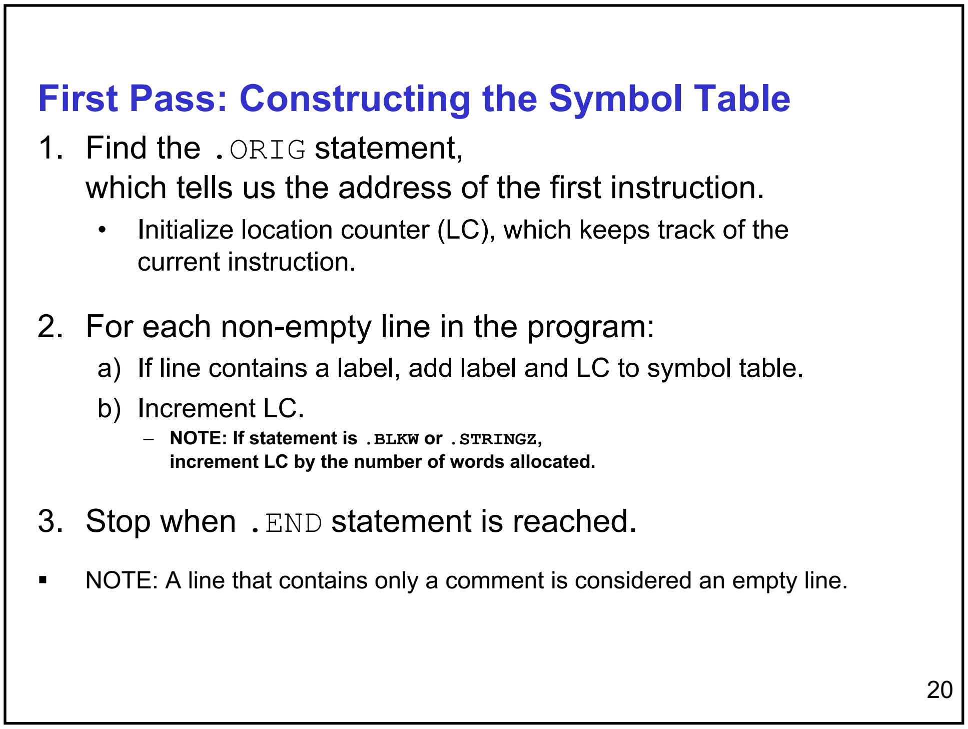
1st Pass

2nd Pass

Executable Image

Symbol Table

- First Pass:
  - scan program file
  - find all labels and calculate the corresponding addresses; this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table



First Pass: Constructing the Symbol Table

1. Find the `.ORIG` statement, which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      – **NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.**
3. Stop when `.END` statement is reached.

- NOTE: A line that contains only a comment is considered an empty line.

Pass 1

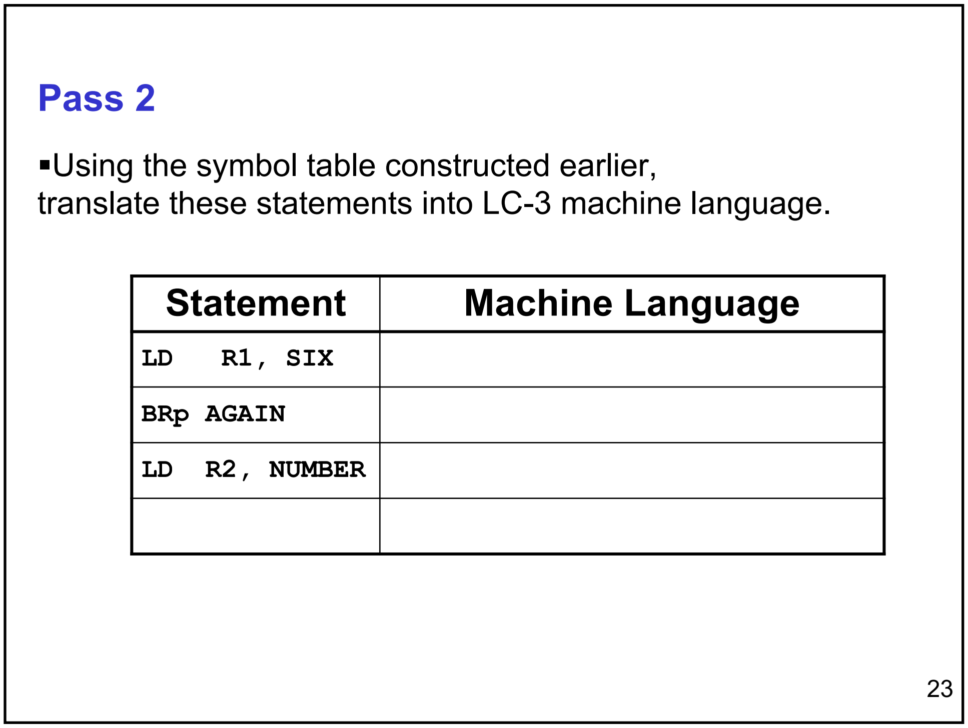
- Construct the symbol table for the program

Symbol	Address
Done	
PLACE2	
PLACE1	

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.

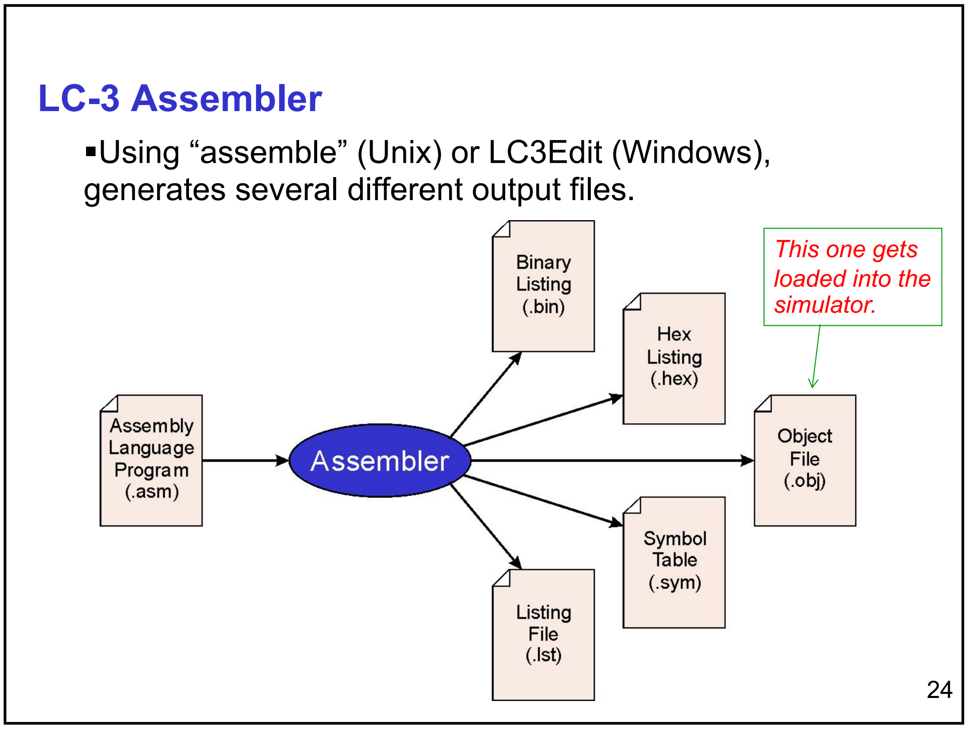
- Potential problems:
  - Improper number or type of arguments
    - ex:
      ```
      NOT    R1,#7
      ADD    R1,R2
      ADD    R3,R3,NUMBER
      ```
  - Immediate argument too large
    - ex:
      ```
      ADD    R1,R2,#1023
      ```
  - Address (associated with label) more than 256 from instruction
    - can't use PC-relative addressing mode



Pass 2

- Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

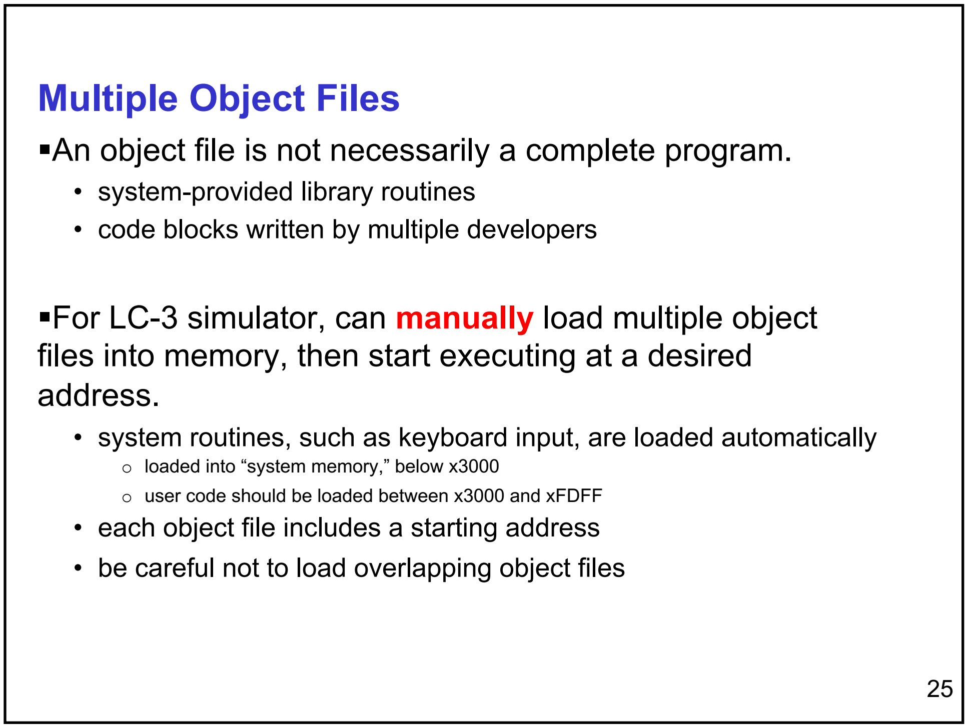
Statement	Machine Language
`LD   R1, SIX`	
`BRp AGAIN`	
`LD  R2, NUMBER`	



LC-3 Assembler

- Using “assemble” (Unix) or LC3Edit (Windows), generates several different output files.

This one gets loaded into the simulator.



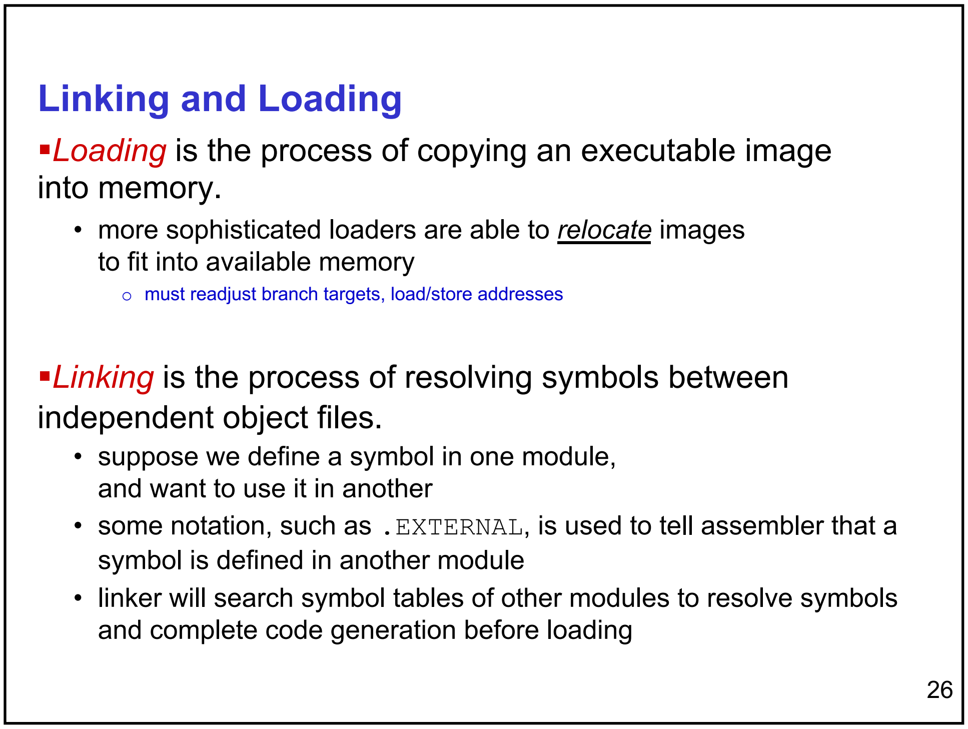
Multiple Object Files

▪An object file is not necessarily a complete program.

- system-provided library routines
- code blocks written by multiple developers

▪For LC-3 simulator, can **manually** load multiple object files into memory, then start executing at a desired address.

- system routines, such as keyboard input, are loaded automatically
  - loaded into “system memory,” below x3000
  - user code should be loaded between x3000 and xFDFF
- each object file includes a starting address
- be careful not to load overlapping object files



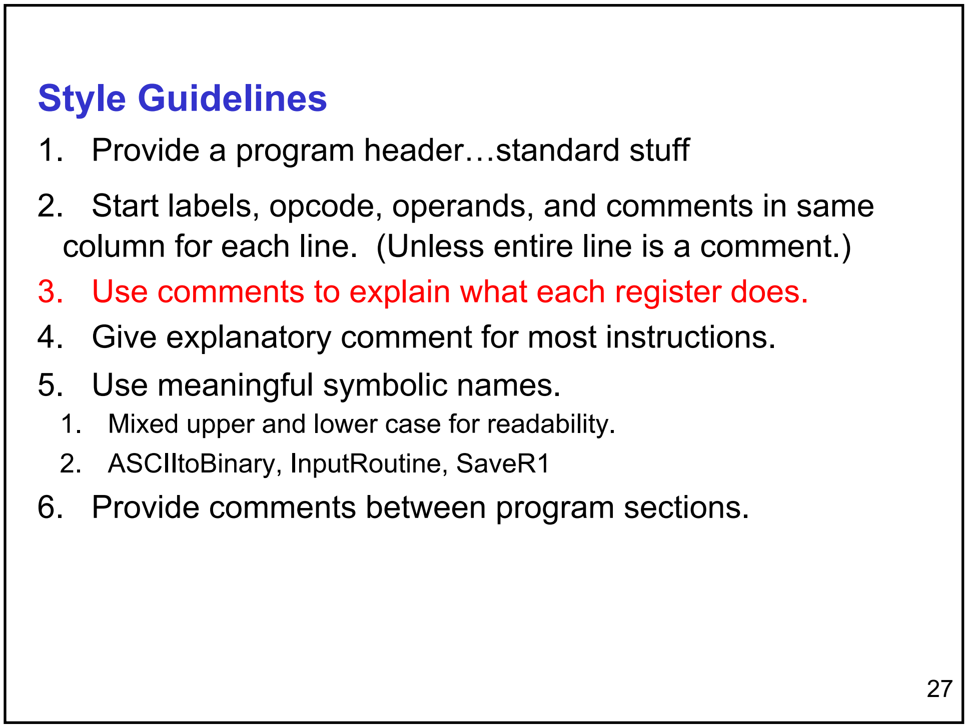
Linking and Loading

▪*Loading* is the process of copying an executable image into memory.

- more sophisticated loaders are able to *relocate* images to fit into available memory
  - must readjust branch targets, load/store addresses

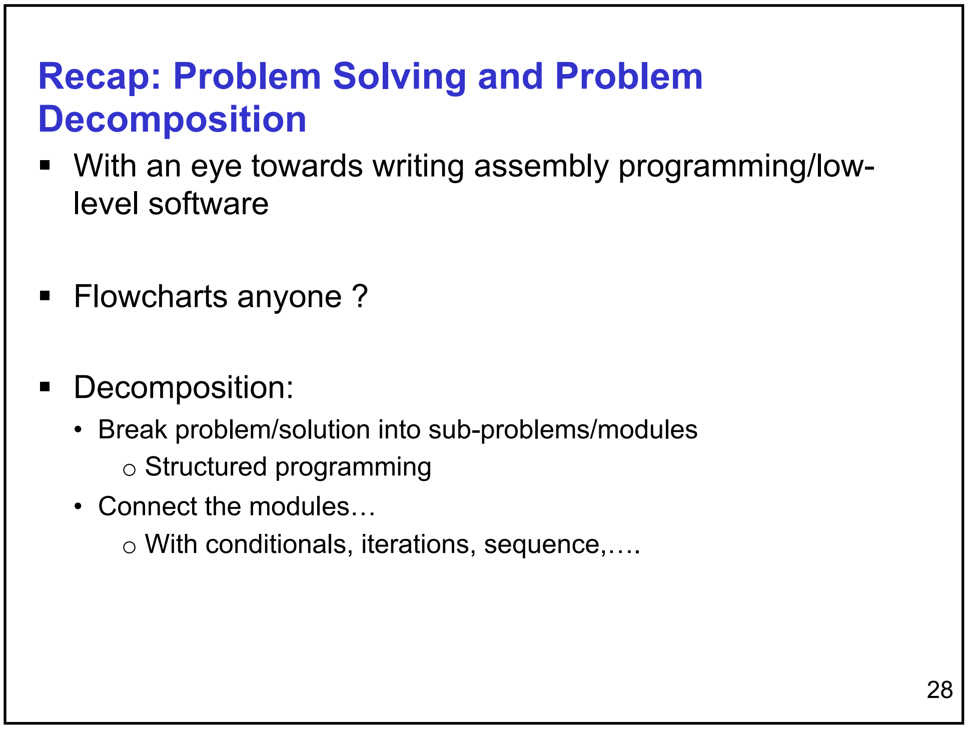
▪*Linking* is the process of resolving symbols between independent object files.

- suppose we define a symbol in one module, and want to use it in another
- some notation, such as `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module
- linker will search symbol tables of other modules to resolve symbols and complete code generation before loading



Style Guidelines

1. Provide a program header…standard stuff
2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
   1. Mixed upper and lower case for readability.
   2. ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.



Recap: Problem Solving and Problem Decomposition

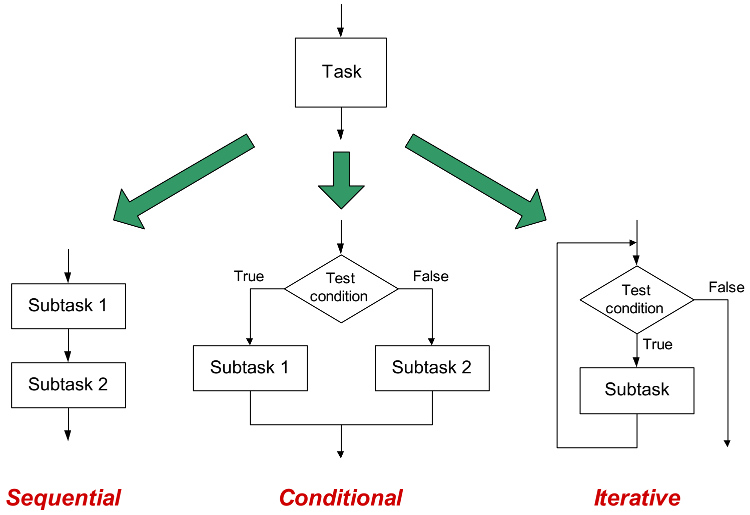
- With an eye towards writing assembly programming/low-level software
- Flowcharts anyone ?
- Decomposition:
  - Break problem/solution into sub-problems/modules
    - Structured programming
  - Connect the modules…
    - With conditionals, iterations, sequence,….

Example

- Array of N numbers
- Read length N of the array
- Replace negative numbers by 0
- Add all the (new) numbers
- Print the sum

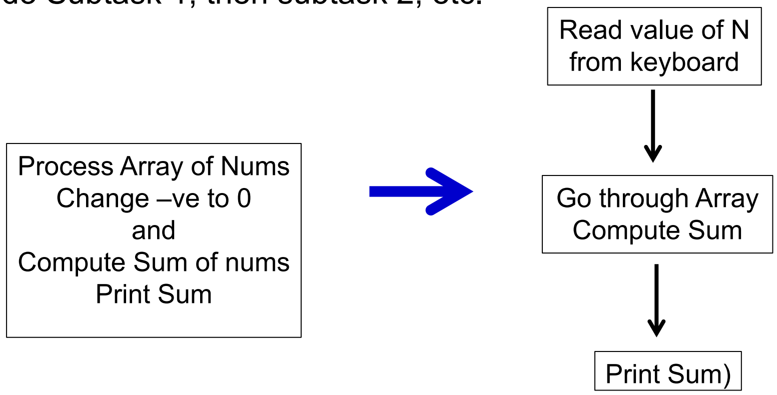
Three Basic Constructs

- There are three basic ways to decompose a task:



Sequential

- do Subtask 1, then subtask 2, etc.



Process Array of Nums
Change –ve to 0
and
Compute Sum of nums
Print Sum

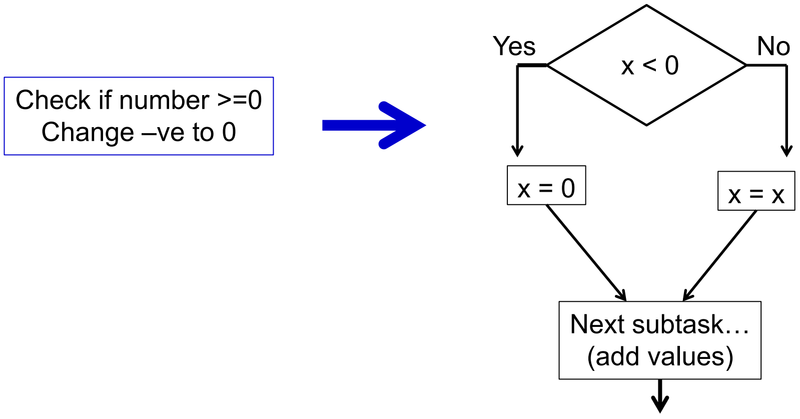
Read value of N
from keyboard

Go through Array
Compute Sum

Print Sum)

Conditional

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



Check if number >=0
Change –ve to 0

x < 0

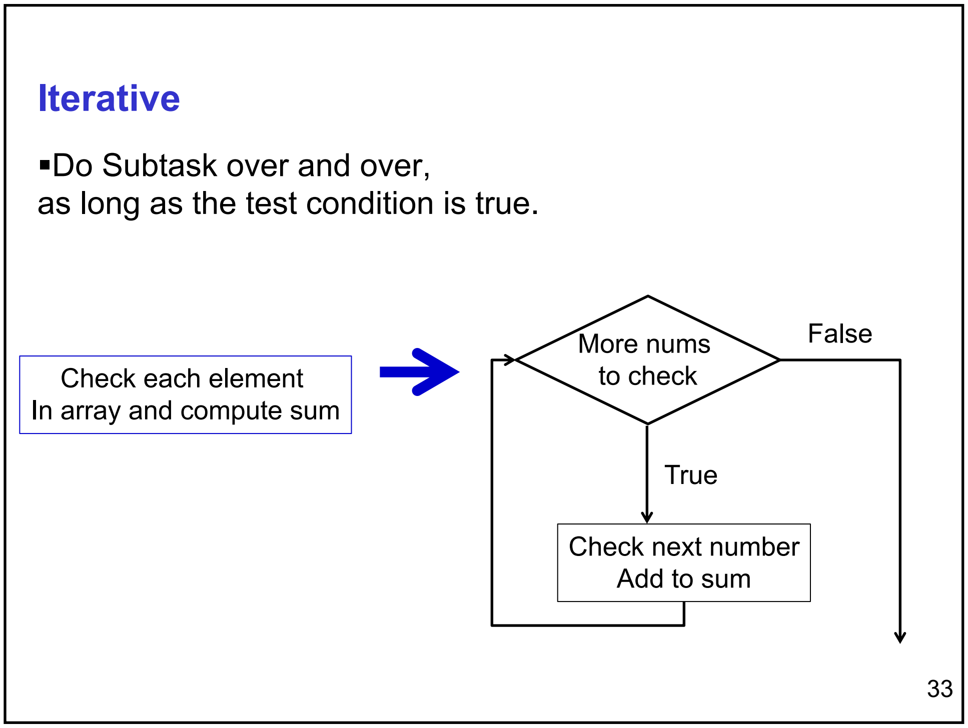
Yes

No

x = 0

x = x

Next subtask…
(add values)



Iterative

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

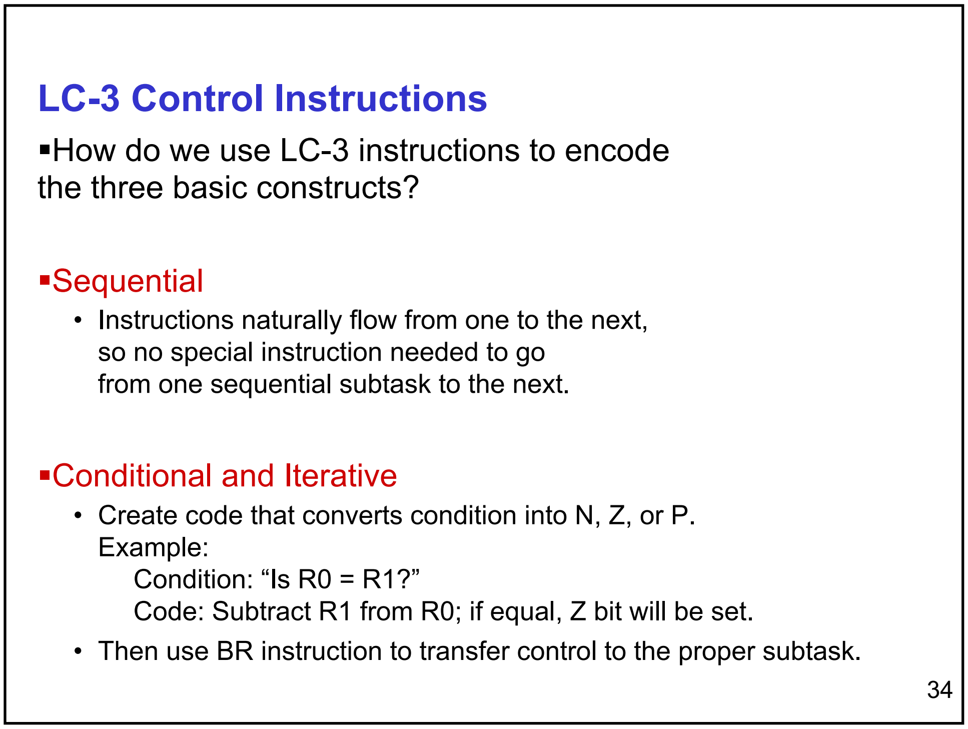
Check each element
In array and compute sum

More nums
to check

False

True

Check next number
Add to sum

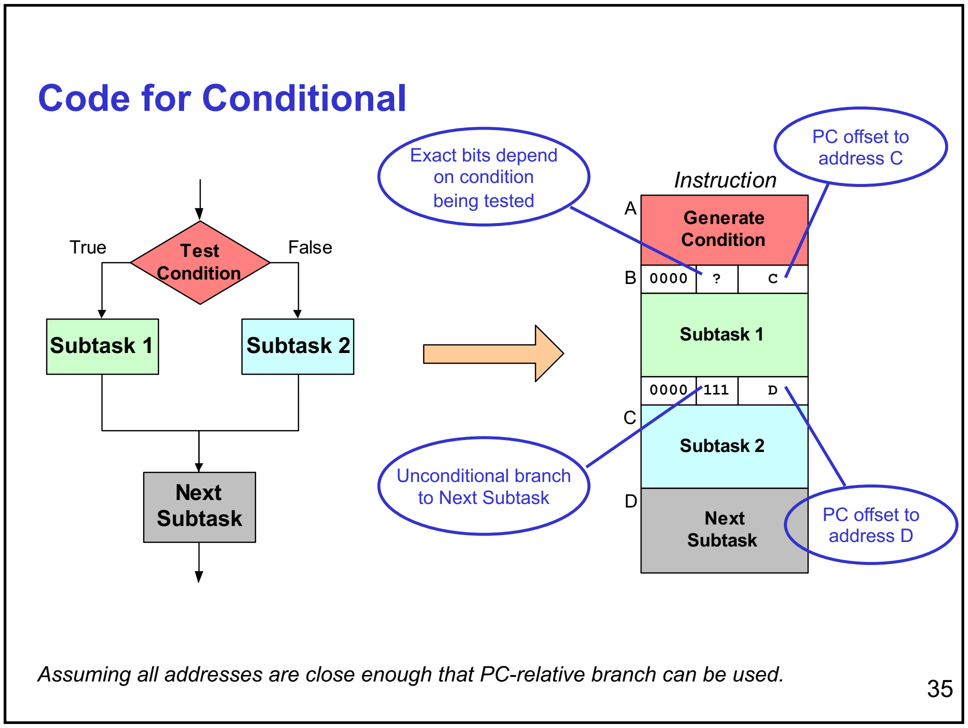


LC-3 Control Instructions

- 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

Test Condition

True

False

Subtask 1

Subtask 2

Next Subtask

Exact bits depend on condition being tested

PC offset to address C

Instruction

A

Generate Condition

B 0000 ? C

Subtask 1

0000 111 D

C

Subtask 2

Unconditional branch to Next Subtask

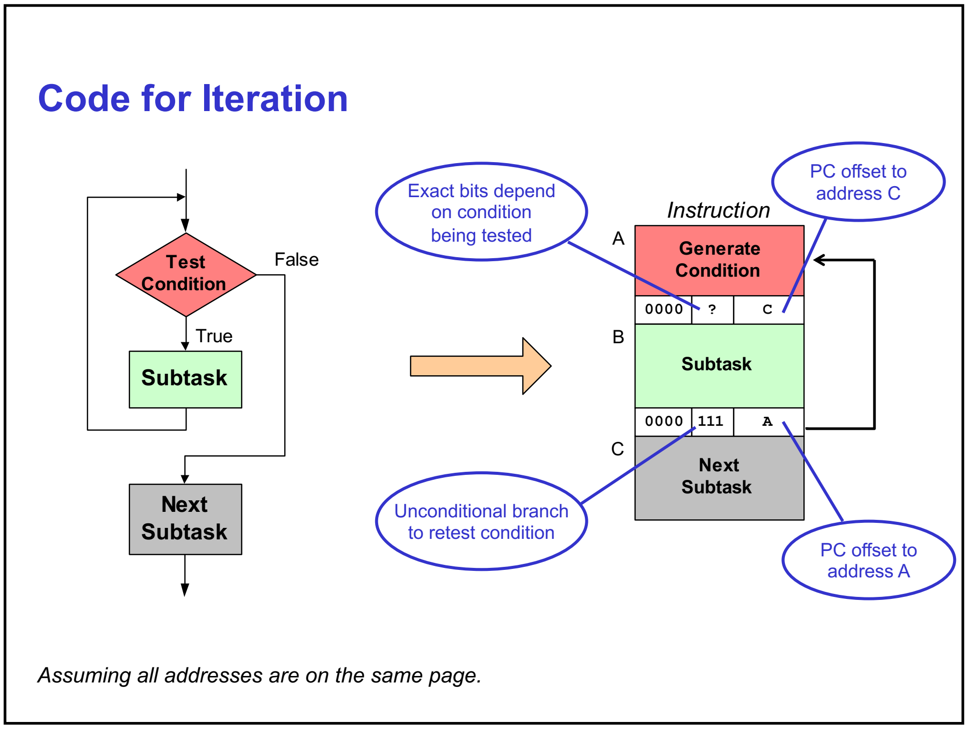
D

Next Subtask

PC offset to address D

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

35



Code for Iteration

Test Condition

False

True

Subtask

Next Subtask

Exact bits depend on condition being tested

PC offset to address C

Instruction

A

Generate Condition

0000 ? C

B

Subtask

0000 111 A

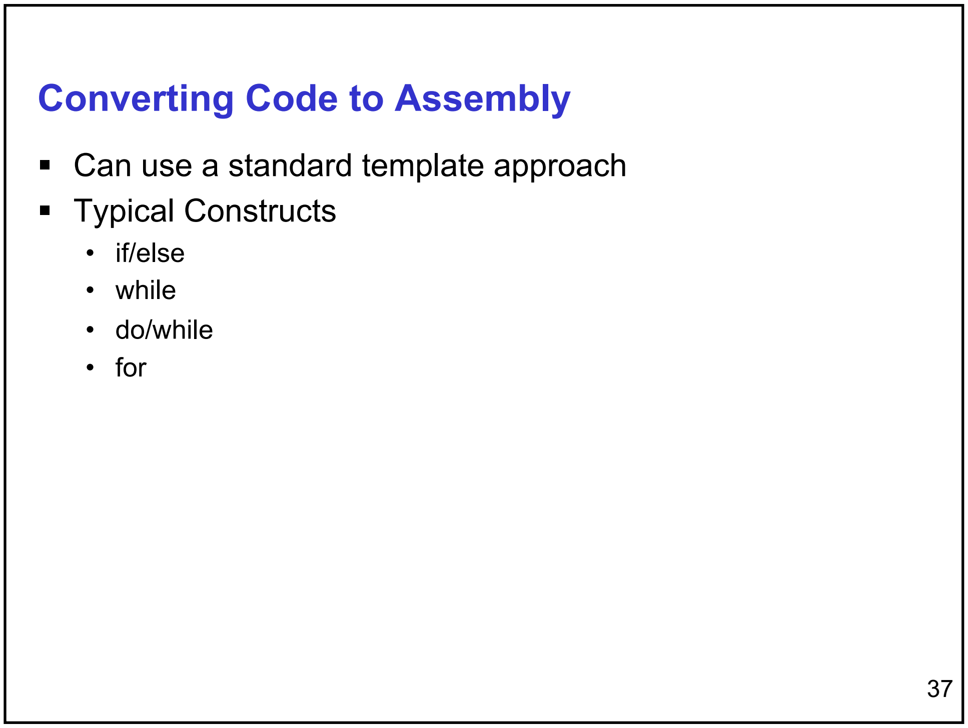
C

Next Subtask

Unconditional branch to retest condition

PC offset to address A

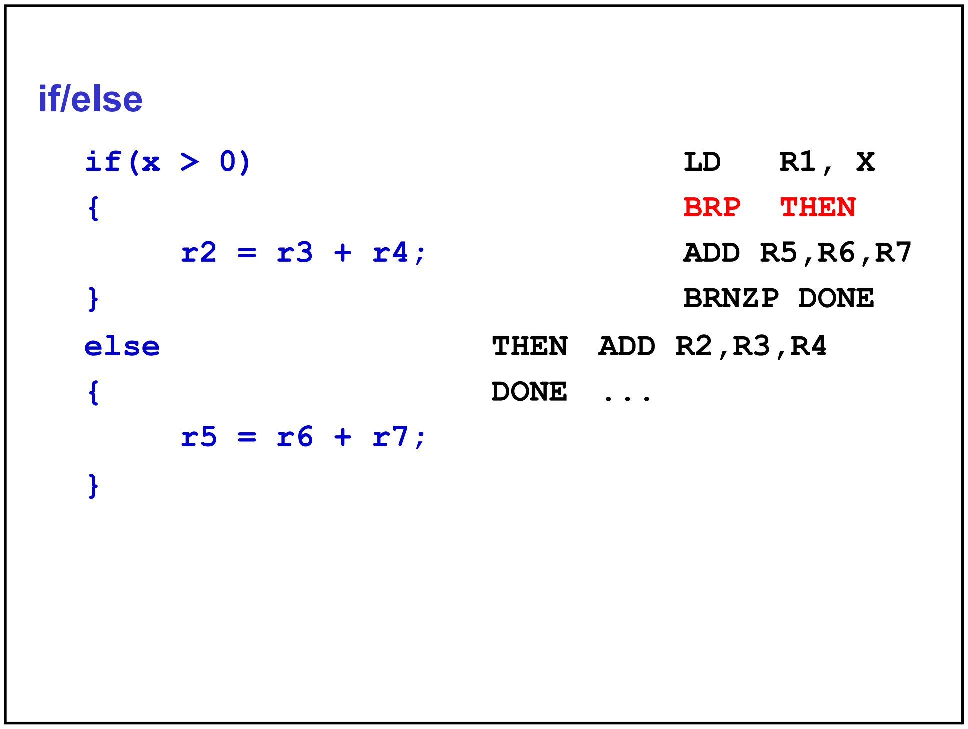
Assuming all addresses are on the same page.



Converting Code to Assembly

- Can use a standard template approach
- Typical Constructs
  - if/else
  - while
  - do/while
  - for

37



if/else

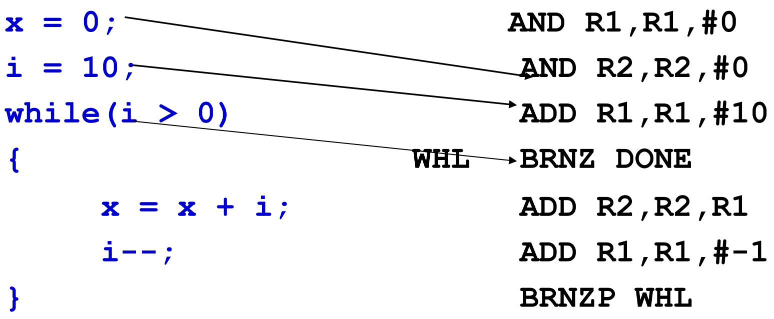
```
if(x > 0)                              LD    R1, X
{                                      BRP   THEN
     r2 = r3 + r4;                     ADD  R5,R6,R7
}                                      BRNZP DONE
else                         THEN  ADD R2,R3,R4
{                            DONE  ...
     r5 = r6 + r7;
}
```

38

if/else

```
if(x > 0)                      LD    R1,X
{                              BRNZ ELSE
     r2 = r3 + r4;             ADD R2,R3,R4
}                              BRNZP DONE
else                       ELSE  ADD R5,R6,R7
{                          DONE  ...
     r5 = r6 + r7;
}
```

while



```
x = 0;                          AND R1,R1,#0
i = 10;                          AND R2,R2,#0
while(i > 0)                     ADD R1,R1,#10
{                           WHL  BRNZ DONE
     x = x + i;                  ADD R2,R2,R1
     i--;                        ADD R1,R1,#-1
}                                BRNZP WHL
```