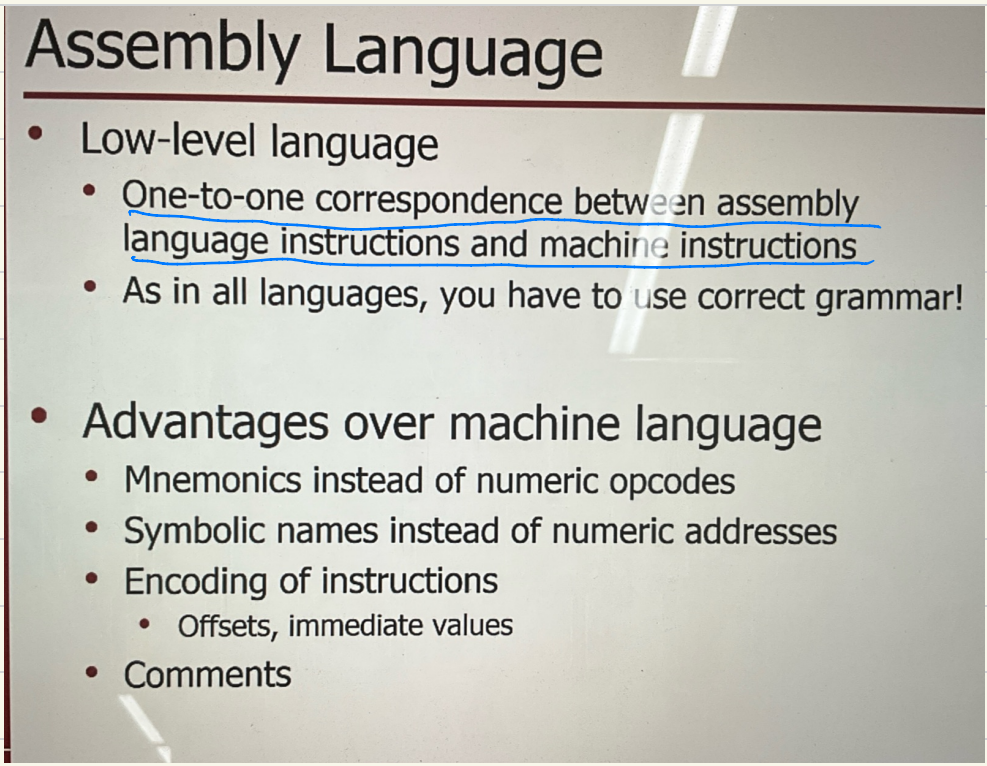
LC-3 Assembly Language



(☆It is higher than the machine language)

Basic Parts of Assembly Program

- Source code includes:
  - Instructions
  - Comments
  - Labels
  - Assembler directives

Example Program

```
; sums the values in a 4-entry array
        .ORIG x0200
START   AND R0, R0, #0          ; initialize sum to zero
        ADD R1, R0, #4          ; count of values in array
        LEA R2, ARRAY4          ; address of start of array

LOOP    LDR R3, R2, #0          ; get current array value,
        ADD R0, R0, R3          ;    and add to sum
        ADD R2, R2, #1          ; go to next value in array
        ADD R1, R1, #-1         ; loop if not done with array
        BRp LOOP

DONE    ST R0, RESULT           ; store sum into memory
        BR START                ; repeat the program

RESULT  .BLKW  1                ; allocate space for result
ARRAY4  .FILL  #17              ; start of the array
        .FILL  #-2
        .FILL  #3
        .FILL  xB
        .END
```

memory allocation directives, reserve space in memory for data

(1) Comments explain the purpose of the code

(2) Instructions include an opcode mnemonic and operands

.ORIG indicates starting memory address for program

.END indicates the end of the source code

Running a Program

- **Assemble** the source code (to machine language)
  - Produce a program image (code and initialized data)
- **Load** the program into the memory
  - A real processor includes special hardware to write the assembled program binary into memory
  - In a simulator, the binary program file is read and copied into the simulator's model of the memory
- **Execute** the program
  - Often want to go through a program one instruction at a time, or run to a certain instruction and pause
  - Simulators and hardware debug tools both provide this capability

Does It Work?

- You need to understand what a program is supposed to do in order to know if it works!
  - How can you write a program to do something if you do not know how to do it yourself?
- Test the program by running it!
  - Test for a variety of input sets, particularly those most likely to be a problem
  - Do not assume that the "answer" your program computes is correct – check it!

Debugging

- Non-trivial programs rarely work the first time
  - Debugging is an important part of programming!
  - Depending on who you ask, test, debugging, and verification are generally anywhere from 70% to 90% of the total software development time

- How can we figure out what went wrong?
  - Observe the program as it executes to see if does what it is supposed to while it computes the "answer"
  - Once you find where in the program something goes wrong, you can figure out what went wrong and fix it

Debugging Capabilities

- You have more options than just running the entire program and seeing what it did!
- Debuggers usually provide two types of single-step execution that you can use:
  - **Step into:** executes one instruction and then pauses (step in Pemsim)
  - **Step over:** like step into, but if the instruction is a subroutine call, executes the entire subroutine (including any subroutines it calls) before pausing (Next in Pemsim)
- **Breakpoints** are another important tool that let you run the program up to a certain instruction
  - You can then check the register and memory values when execution reaches that point, and if needed, single-step from there

LC-3 Memory Allocation

Terminology for Using Memory

- **Allocate:** reserve space in memory
  - No other variable or code will be placed there
- **Initialize:** set the value in a memory location when program is loaded
  - In other words, before the program executes

eg: SOME .FILL xEFb2

↓

① Reserve space at SOME

② Initialize the value to EFb2 (Hex)

Memory Allocation

- There are three assembler directives that we will use for memory allocation:
  - `<label> .FILL value`
    Allocates one location and initializes it to ***value***
  - `<label> .BLKW count` (Allocate some space in memory)
    Allocates ***count*** words of memory (not initialized)
  - `<label> .STRINGZ "string"`
    Allocates and initializes memory to hold an ASCIIZ string (null-terminated sequence of ASCII characters)

- The optional label is the symbolic address of the first word of memory allocated

`<label> .FILL value`

- Allocates one word of memory, initialized to the indicated ***value*** when the program is loaded
  - The value in that location may be changed later by executing a store instruction
  - Optional label can be used to refer to that location

- **Examples:**

```
; allocate one word initialized to -1 (xFFFF)
NEG1   .FILL #-1
; allocate and initialize two words where
; the label ARRAY is address of first word
ARRAY .FILL #42
      .FILL x17
```

<label> .BLKW *count*

- Allocates ***count*** words of memory
  - BLKW = "**BL**oc**K** of **W**ords"
- The allocated memory locations are not initialized
  - They may happen to be 0... or they may not...
- **Examples:**

```
; allocates one uninitialized memory word at the
; memory address corresponding to the ONEWORD label
ONEWORD   .BLKW #1
; allocates 42 uninitialized memory words,
; where the first word is at the BIGBLOCK label
BIGBLOCK .BLKW #42
```

→ 42 spaces

☆ This line indicates the first space, the second space is BIGBLOCK+1

<label> .STRINGZ "string"

- Allocates and initializes memory to hold a null-terminated character string
  - One word per character, where each character is represented by its ASCII code
  - Last location holds the value 0 to indicate end of string
    - This is the '\Z' in STRINGZ
    - Also called the "null terminator"
  - Total allocated words = number of characters plus 1
- **Example:**

```
; allocates 4 memory words, initialized to
; 'a', 'b', 'c', null  (0x61, 0x62, 0x63, 0x00),
; where the 'a' is at the ABCSTR label
ABCSTR .STRINGZ "abc"
```

→ 4 locations, a is at the ABCSTR, b is next one.

Memory Allocation Example

mem.asm

```
; demonstrates several ways to
; allocate memory
          .ORIG x0200

NEG1      .FILL #-1
TWOWORDS  .BLKW #2
ABCSTR    .STRINGZ "abc"
ARRAY     .FILL #42
          .FILL x17

          .END
```

allocate & initialize

allocate

allocate & initialize

allocate & initialize

allocate & initialize

Memory

ADDRESS	CONTENTS
⋮	
0x01FF	???
0x0200	0xFFFF
0x0201	???
0x0202	???
0x0203	0x0061
0x0204	0x0062
0x0205	0x0063
0x0206	0x0000
0x0207	0x002A
0x0208	0x0017
0x0209	???
0x020A	???
⋮	

Remember: memory is **initialized** when a program is loaded into memory

LC-3 Assembler

What is an assembler?

→ It is a software

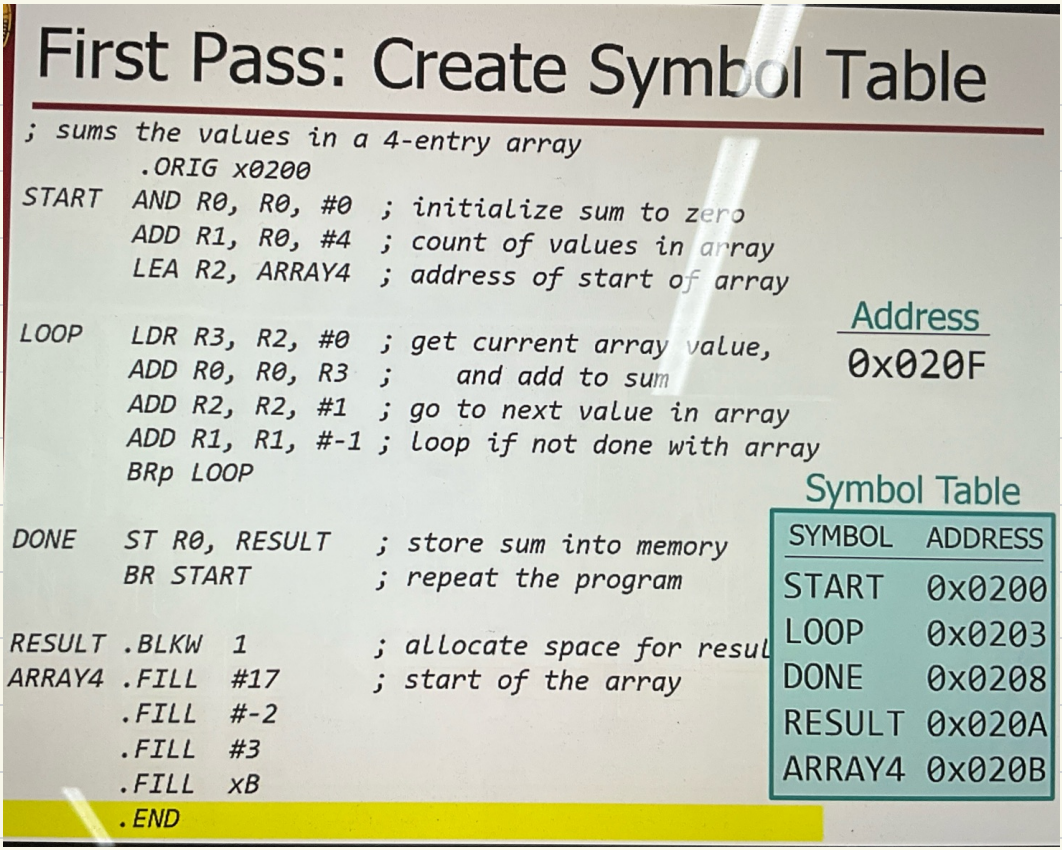
- Input: assembly-language source code
- Output: binary program image (Machine Language)
- Types of assemblers
  - Absolute assembler
    - Produces a program image – the machine code that needs to be loaded into memory
    - Generally only allows a single assembly source code file
  - Relocatable assembler
    - Produces an object file for each assembly language file
      - The actual address of the code in each object file is not known
    - Compilers may produce other object files
    - A linker then joins all the object files into a one program image

Two-Pass Assembler

- Processes each source file twice
- First pass: check syntax and build **symbol table**
  - Table of labels and their addresses in the program
- Second pass: create **program image**
  - Sequence of address:value pairs that indicate what information needs be written into memory when the program is loaded

First Pass: Create Symbol Table

- Determine the address of each label in the file
  - Assume the address is 0 at the start
- For each line of the source file:
  - Ignore blank lines and comments
  - Verify that it is syntactically valid
  - If it is an ORIG directive, update the address
  - If it is an END directive, stop processing the file
  - If it has a label, make an entry in the symbol table
  - If it is an instruction, increment the address
  - If it is a data allocation directive, update the address based on the number of allocated words



① Treat the "address" as x0000

② Go to .ORIG, update the address to x0200

③ Go to START Label, put the symbol and address in the table

④ Repeat the above step until reach to .END

(☆ when encountering the instruction line, simply +1 to the address, no need to put it into the symbol table)

Second Pass: Assemble Program

- Encode all instructions and data directives
  - Assume the starting address is 0
- For each line of the source file:
  - Ignore blank lines and comments
  - If it is an ORIG directive, just update address
  - If it is an END directive, stop processing the file
  - If it is an instruction:
    - Encode the instruction and write the address:instruction pair to the output file
    - Then increment address
  - If it is a data allocation directive:
    - Write address:data pair for each initialized word to output file
    - Then, update address based on # of allocated words

Second Pass: Assemble Program

```
; sums the values in a 4-entry array
        .ORIG x0200
START   AND R0, R0, #0
        ADD R1, R0, #4
        LEA R2, ARRAY4

LOOP    LDR R3, R2, #0
        ADD R0, R0, R3
        ADD R2, R2, #1
        ADD R1, R1, #-1
        BRp LOOP

DONE    ST R0, RESULT
        BR START

RESULT .BLKW  1
ARRAY4 .FILL  #17
       .FILL  #-2
       .FILL  #3
       .FILL  xB
       .END
```

(Not initialized, x020A)

Symbol Table

SYMBOL	ADDRESS
START	0x0200
LOOP	0x0203
DONE	0x0208
RESULT	0x020A
ARRAY4	0x020B

Program Image

ADDRESS	VALUE
0x0200	0x5020
0x0201	0x1224
0x0202	0xE408
0x0203	0x6680
0x0204	0x1003
0x0205	0x14A1
0x0206	0x127F
0x0207	0x03FB
0x0208	0x3001
0x0209	0x0FF6
0x020B	0x0011
0x020C	0xFFFE
0x020D	0x0003
0x020E	0x000B

Creating the Object File

Program Image

ADDRESS	VALUE
0x0200	0x5020
0x0201	0x1224
0x0202	0xE408
0x0203	0x6680
0x0204	0x1003
0x0205	0x14A1
0x0206	0x127F
0x0207	0x03FB
0x0208	0x3001
0x0209	0x0FF6
0x020B	0x0011
0x020C	0xFFFE
0x020D	0x0003
0x020E	0x000B

Assembled program could be written to an object file as a sequence of address:data pairs

14 address:data pairs
28 16-bit words

Half of this file contains addresses, most of which are sequential!

0x0200
0x5020
0x0201
0x1224
0x0202
0xE408
⋮
0x0209
0x0FF6
0x020B
0x0011
0x020C
0xFFFE
0x020D
0x0003
0x020E
0x000B

Pairs

Compressing the Object File

- Store the program image information in a (slightly) more complex data structure
  - A sequence of **sections** of information
- For each section, specify:
  - Starting address of the section
  - Number of words in the section
  - The words in the section

starting address
number of words
data
data
data
data
⋮

Compressing the Object File