**Lab #5: Low-level Language Programming: assembly and machine language**

Due: Start of class, Tuesday, March 4th, 2014.

## Part 0: Hardware, software, and background

You will need your WFU-issued ThinkPad for this lab. This lab makes use of an application named *Pep8*. You will need to download and start this application using the instructions in the *Pep8\_downloading\_running* file.

The purpose of this lab is to reinforce what you learned about machine and assembly language programming in the text (Chapter 6 in Computer Science Illuminated) and from lectures. We will do this using the Pep/8 virtual computer described in the book. For our purposes, the idea of programming is to take an algorithm in some form and translate it into *code that a computer can understand*. The computer in this case in the “compute agent” for the algorithm, much in the same way you might think of a chef as the compute agent for a recipe. We have not discussed much in the way of algorithm design and representation, which we will do in the coming lectures, so for this lab you will rely on your innate ability to solve straightforward problems and to design a step-by-step procedure to implement them in machine or assembly language. Remember that there is a one-to-one mapping between an assembler instruction and a machine instruction. This does not include assembler directives (pg 167 of the text), which are different and which the assembler uses to aid the programmer and are not for computer instructions per se. (There are some exceptions to this one-to-one mapping. TRAP instructions actually branch to a segment of code but appear to be single instructions. The DECI and DECO and some others work in this way. For the interested student, you can investigate TRAP instructions. You can begin by selecting System -> Redefine Mnemonics in the PEP/8 Simulator.)

When learning to program, particularly at this level, it is informative to both analyze and synthesize code. That is, read and study what others have done then write some of your own. You will be asked to describe what existing code is doing, modify code, and write your own.

As we learned, the PC (program counter) points to the next instruction to be fetched while the current instruction being executed is in the IR (instruction register). Unless the current instruction changes the contents of the PC (an example is the branch instruction introduced in the code on page 169), the PC will be incremented by the size of the current instruction being executed when a new instruction is fetched. So for example, if the instruction in the IR is 3 bytes long, the value in the PC will be incremented by 3. If the instruction in the PC is a unary one (1 byte only), then the PC will only be incremented by 1. In this lab, you will deal with three types of programs:

1. Programs that run from beginning to end without modifying the PC (Part 1).
2. Programs with branching (page 170 of the text) (Part 2).
3. Programs with loops (page 173 of the text) (Part 3).

When asked to include code as part of your lab report, do the following:

For code segments (programs), copy and paste the assembler or object code (whichever is required) from the simulator into your lab report. Be sure that you are able to copy and paste back from your lab report to the simulator and that is assembles and runs. This is how your code will be tested and graded. Select Courier or Courier New font as these do not use relative spacing or adjusted character size. IMPORTANT: Including meaningful comments in programs, assembly or higher level, is an essential part of good programming. All code included in this assignment should be commented in meaningful way. Examples:

The following is not a good comment. It is in fact a tautology.

LDA 0x00FF, d ;Load the A register with the contents of location FF

The following is a meaningful comment.

LDA 0x00FF, d ;Get the next number in the list to add to the sum.

NOTE: When running code, it is a good idea to select System->Clear Memory to be sure there are no residual values in the data area. The Load the program again. Also, after executing code, you may have to select View -> Code Only then back to select View -> Code/CPU/Memory to get the update memory contents.

## Part 1: Programs without branching or loops

The text introduced machine language instructions by writing them in the following form (pg 155):

Instruction specifier

Operand specifier

An example is in the top row of the table on page 161. There we see the instruction

01010000

0000000001001000

This was purely for illustrative purposes. When writing our own machine language instructions to be input into the simulator, we will convert them to hex (base 16) and write each byte of the instruction separately. So for the above example, we would write

50 00 48

The above instruction is three bytes long. Remember that we have unary instructions (pg 157) that do not have an operand specifier and so are only one byte long. An example is the instruction for stop: 00000000

Which we would input into the assembler as: 00

**Question 1**

Consider the following object code (object code = machine language code):

51 00 07 51 00 08 00 48 69

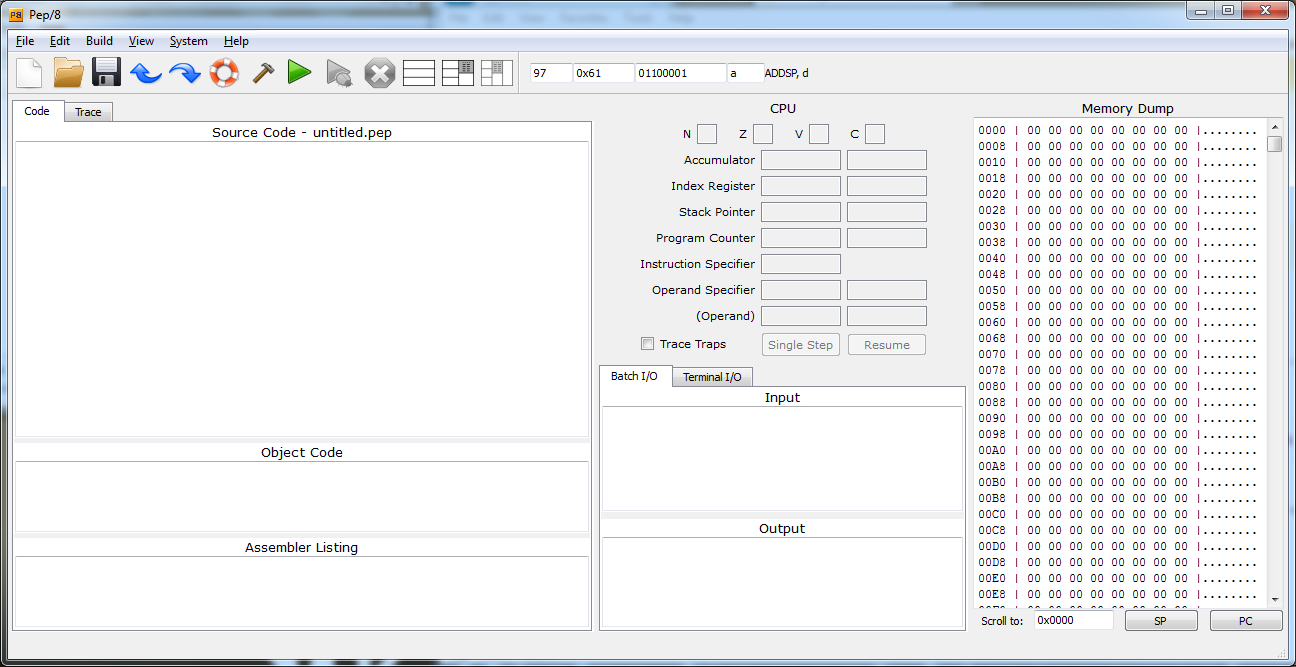
**1.a.)** Looking at the object code, how many distinct instructions are there \_\_\_\_\_3\_\_\_\_\_

**1.b.)** Write the assembly language code (assembler code) for the above object code.

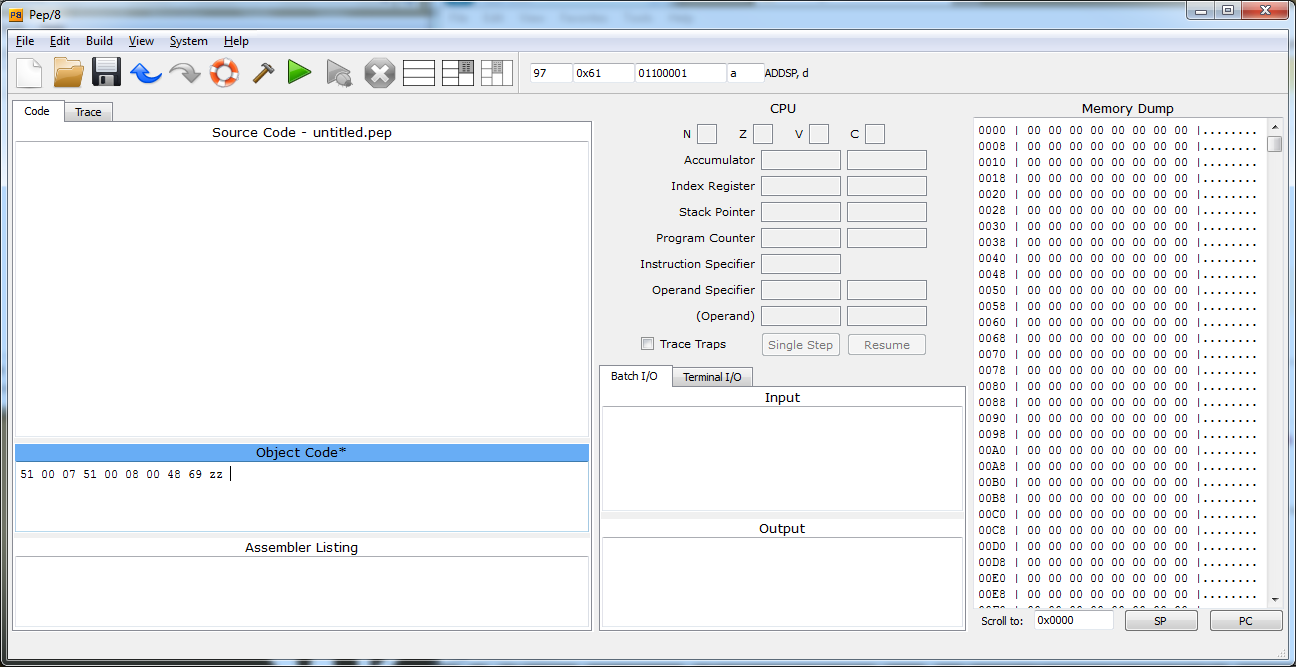
CHARO 0x0007, d

CHARO 0x0008, d

STOP

Step 1: Start the PEP/8 Simulator. Under View, select Code/CPU/Memory. Select the Code tab (the alternative is the Trace tab). The simulator should look like the following: 

Step 2: Copy the above machine language code to the Object Code window adding a ‘zz’ to the end (the ‘zz’ is an assembler directive that marks the end of the object code for the simulator).

So now the simulator should appear as follows: 

1.c.) First, clear memory by selecting System -> Clear Memory. This will reset memory so that it only contains 00’s as shown in the figure above (see the Memory Dump panel on the right). Under Build, select Load (This loads the code into memory. Be sure this happens.) Then select Build -> Execute. What is the output? \_\_\_\_Hi\_\_\_\_

1.d.) What is the contents of the PC after the code is run? 0x0007

1.e.) Select Build -> Start Debugging Object. Hit the “single step” button.

What is the contents of the Instruction Specifier (register)? \_\_\_\_01010001\_\_\_\_\_\_

What assembler instruction does this represent? \_\_\_\_\_CHARO\_\_\_\_\_\_\_

What is the contents of the PC? \_\_\_\_\_\_0x0003\_\_\_\_\_\_\_\_\_\_\_

Under Build, select Stop Debugging.

1.f.) Modify the object code to output “Yo.” Write your object code here (be sure to test it in the simulator):

51 00 07 51 00 08 59 6F

**Question 2**

2.a.) Copy the following object code to the Object Code window in the PEP/8 Simulator.

49 00 0D 49 00 0E 51 00 0E 51 00 0D 00 zz

As in Question 1, load this code into memory. Enter (without the quotes) “AZ” into the Batch I/O window. Select Build->Start Debugging Object. Single step through the code and record the contents of the PC and memory locations D­16 and E16 at each step. Note that the subscript 16 means that the digit is in base 16. A subscript of 10 would indicate that the digit is in base 10 (decimal). So F16 = 1510. In the comment column, describe what is going on at each step. NEXT TIME ASSIGNING THIS INCLUDE THE INSTRUCTION SPECIFIER.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PC16** | **PC10** | **D16** | **D10** | **E16** | **E10** | **Comment** |
| 0 | 0 | 41 | 65 | 0 | 0 |  |
| 3 | 3 | 41 | 65 | 5A | 90 |  |
| 6 | 6 | 41 | 65 | 5A | 90 |  |
| 9 | 9 | 41 | 65 | 5A | 90 |  |
| C | 12 | 41 | 65 | 5A | 90 |  |
| D | 13 | 41 | 65 | 5A | 90 |  |

2.b.) What does this program do?

Inputs two characters and outputs them in the opposite order.

2.c.) Modify the code to output the characters in the opposite order that is done by the object code entered above. Enter your object code here (be sure to test it in the simulator).

49 00 0E 49 00 0D 51 00 0E 51 00 0D 00 zz

**Question 3**

Select Help -> PEP/8 Reference. A separate window should open. Review the instruction set to familiarize yourself with what is available. You will not need all the instructions, addressing modes, or status bits so do not worry about this. But this will provide a valuable resource for your work.

3.a.) Find the “Bitwise OR to r” instruction.

What does this instruction do if using direct addressing mode?

Does a bit-by-bit logical or between the register and the memory location leaving the result in the register.

Write the object code in binary to Bitwise OR the A register (r = 0) with the contents of memory location 2016. \_\_\_\_\_1010 0001 0000 0000 0010 0000\_\_\_\_\_\_\_

Assume the following contents of the A register and memory location 2016:

B416 and 4116.

Write these values in binary and the result of the Bitwise OR instruction in hex and binary (in that order):

10110100 01000001 F5 11110101

3.b.) Load the following object code into the PEP/8 Simulator and run it.

C1 00 11 71 00 13 A1 00 15 F1 00 10 51 00 10 00 00 00 05 00 03 00 30 zz

What is the output? \_\_\_\_\_\_\_8\_\_\_\_\_\_\_\_\_

What does this program do?

Adds number in memory locations 11 and 13 (hex) then ORs the sum with hex 30. Outputs the result

What is the assembler instruction for the object code A1001516? \_\_ORA 0x0015, d\_

What is the purpose of this instruction? (Hint. Think numbers in ASCII.)

Converts the decimal sum to ASCII.

3.c.) Modify the code to subtract the decimal numbers 4 and 2, that is (4 – 2) and output the ASCII difference. Enter the object code here:

C1 00 11 81 00 13 A1 00 15 F1 00 10 51 00 10 00 00 00 04 00 02 00 30 zz

3.d.) Briefly describe your modifications.

Changed the data values in locations 11 and 13 to 4 and 2 and changed the add instruction, 71, to a subtract instruction, 81.

## Part 2: Programs with Branching

Branching allows for instructions to be conditionally executed based on data or results of other instructions. For example, you may want to have some instructions executed if a sum is negative and other instructions executed if that same sum is zero or positive. Various machine code branch instructions allow for this. You may want to review the branch instructions in the Reference under Help in the PEP/8 Simulator.

**Question 4: Branch Instructions.**

4.a.) How many branch instructions are there? 9

4.b.) Do you need all of these instructions or could you accomplish the same things with a subset? Explain your answer.

Do not need them all. For example, with the BRLT and BREQ, you could accomplish a BRLE.

4.c.) Copy the following assembler code to the source code panel in the simulator.

LDA 1,i

ADDA 1,i

STA VAL,d

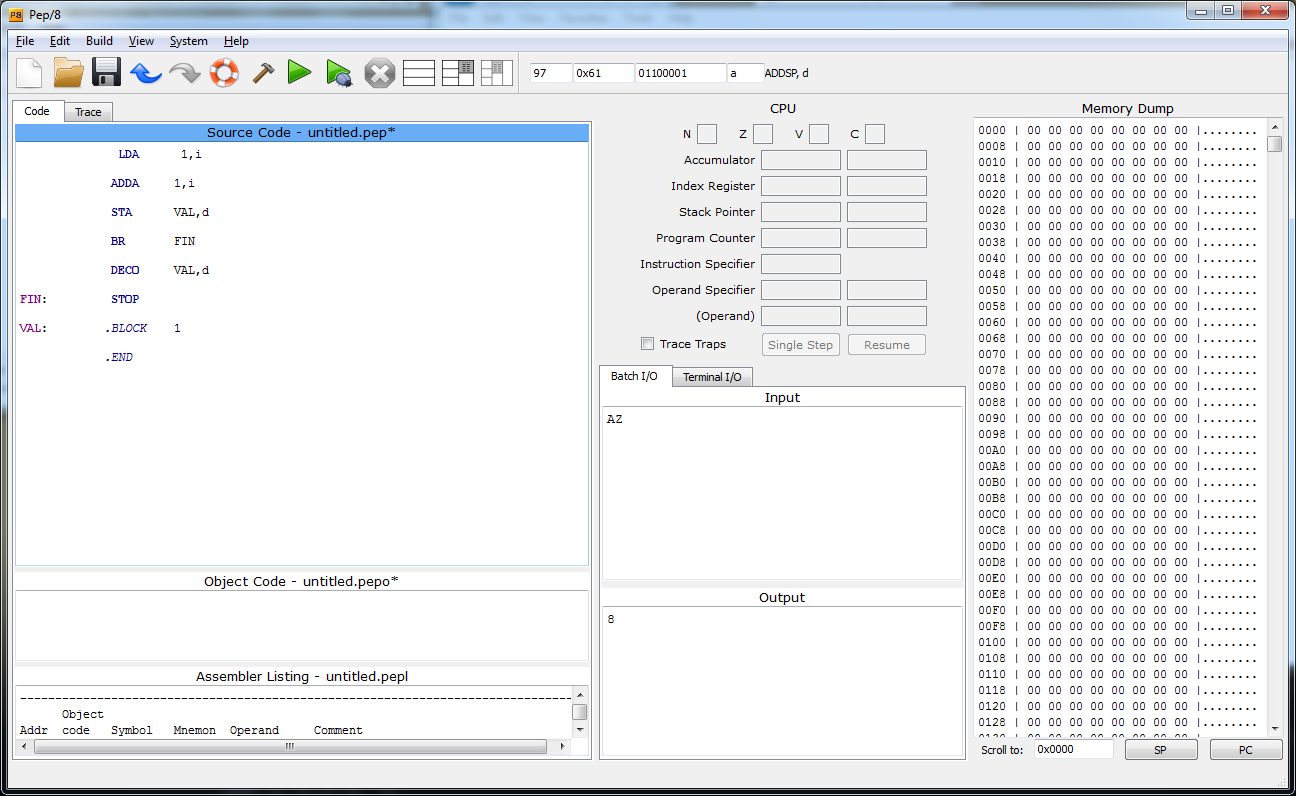
BR FIN

DECO VAL,d

FIN: STOP

VAL: .BLOCK 1

.END

It should now look something like. 

Select Build -> Assemble then select Build -> Load.

Select Build -> Start Debugging Source. Until it stops, Single Step through the program writing the PC contents below at each step. Explain the results of the PC contents.

0 LDA instruction is next

3 LDA took 3 bytes, next is the ADDA instruction

6 ADDA took 3 bytes, so next is the STA instruction

9 STA took 3 bytes, so next is the BR instruction

F BR will skip the DECO instruction, so next is the STOP instruction at location F

10 Program is done as the STOP instruction executed and PC set to next memory location.

4.d.) What would happen if you moved the FIN: label to the ADDA instruction line (only the label, not the entire Stop instruction)? As before, select Build -> Assemble, Build -> Load, then Build -> Start Debugging Source. Single step through the code. What happens?

Loops over and over. So will never end.

What is the PC contents? \_\_\_\_\_\_Depends on when you stop\_\_So any value is good here as long as it is in the program space somewhere.\_\_\_

Hit the Resume button then Interrupt Execution under Build. NEXT TIME STOP THE PROGRAM THEN JUST HIT RUN BEFORE DOING THIS.

What is the PC contents? 3

You may have to select Build -> Stop Debugging if the simulator has stopped responding.

**Question 5: Analyze and Understand a Program with Branch Instructions.**

;Lab #4 Branch Question Code

BR MAIN ;Branch around data

NUM: .BLOCK 2 ;Storage for input

VAL: .WORD 0x000A ;Value to add if input <0

ANS: .BLOCK 2 ;Storage for sum

SPACE: .BYTE 0x20 ;Space used for output

MAIN: DECI NUM,d ;Input a number

DECO NUM,d ;Output the number to verify

CHARO SPACE,d ;Output a blank to separate numbers

LDA NUM,d ;A <- the number

ADDA 1,i ;Add one to the number

STA ANS,d ;Store the sum

DECO ANS,d ;Output the sum

STOP

.END

5.a.) What does this code do?

Inputs a number. Outputs the number followed by a blank. Adds one the number, stores the sum in location ANS, then outputs the sum.

5.b.) Modify the code so it does the following: If the input number is >= zero, then add 1 to the number, store the sum to ANS and output ANS. Else, if the input number is < zero, add the value stored VAL to the number, store the sum to ANS, and output ANS. Enter your assembly language program here, neatly formatted as the code above.

BR MAIN ;Branch around data

NUM: .BLOCK 2 ;Storage for input

VAL: .WORD 0x000A ;Value to add if input <0

ANS: .BLOCK 2 ;Storage for sum

SPACE: .BYTE 0x20 ;Space used for output

MAIN: DECI NUM,d ;Input a number

DECO NUM,d ;Output the number to verify

CHARO SPACE,d ;Output a blank to separate numbers

LDA NUM, d

CPA 0x0000, i ;Compare to zero

BRLT SKIP, i ;Do not add 1 if less than zero

LDA NUM,d ;A <- the number

ADDA 1,i ;Add one to the number

STA ANS,d ;Store the sum

DECO ANS,d ;Output the sum

SKIP: STOP

.END

## Part 3: Programs with Loops

Loops (repetition) provide one the most fundamental and powerful capabilities of computer programs. The ability to cycle through processing of data and information millions, even billions of times, without tiring or error is the basis of many algorithms. Consider searching for terms in the vast information store in Google or trying to find a suspect SNP (single-nucleotide polymorphism) in the genome of tens of thousands of individuals where each genome may contain a billion SNPs. A computer chess player must be able to search through billions of possible board positions, and the solutions to a large set of linear equations may involve a search that is refined at each stage.

In higher-level languages, loops have special statements that make this operation simple. As these languages developed, programmers recognized the need for these constructs, and we will study some of them as we move forward. At the machine language level, loops are implemented in very fundamental ways, usually employing a compare of some sort and a conditional branch. The higher level language statements are translated into these machine instructions.

Review the program on page 173 of the text (Looping Program). Be sure you understand each statement.

**Question 6**

6.a.) Examine the various Branch instructions available for the PEP/8. Could the following two instructions from the Looping Program be replaced by a single Branch instruction? If your answer is Yes, write the instruction below. If your answer is No, explain your reasoning. NEXT TIME PUT THE QUIT AND LOOP LABELS IN. THIS IS AMBIGUOUS.

BREQ quit,i

BR loop, i

You could just use BRNE to loop, then if equal, it would simple fall through to the next instruction.

6.b.) In the Looping Program, the programmer decided to put the data at the beginning then branch around it. What would happen if the BR instruction at the beginning were left out?

The computer would assume the data were instructions and most likely fail.

**Question 7: Writing your own loop program. Introducing the index register and indexed addressing mode.**

As you have seen, the register specifier is a 0 for the A register. What if the specifier is a 1? This will refer to the index register. Together with indexed addressing mode, these allow for a powerful looping method. The index register, X, can be loaded, stored, added to, etc. in the same way as the A register. Run the following code:

LDX 0x00FF,i

STOP

.END

7.a.) The contents of the index register is: ­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_0x00FF\_\_\_\_\_\_\_\_

Run the following code:

LDX 0x00FF,i

ADDX 0x0001,i

STOP

.END

7.b.) The contents of the index register is: ­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_0x0100\_\_\_\_\_\_\_\_\_

Are you getting the idea?

Now you are ready to use Index Address Mode, specified by an “x” instead of a “d” or an “i” for direct or immediate mode. Find Indexed Addressing Mode in the PEP/8 Reference under Help in the simulator.

7.c.) Fill out the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Addressing Mode** | **aaa-field** | **Letters** | **Operand** |
| Direct | 001 | d | MEM[OprndSpec] |
| Indexed | 101 | x | MEM[OprndSpec + X] |

Indexed addressing mode works much like Direct; however, to form the address, you add the contents of the Index Register (X) to the contents of the operand field.

Run the following code:

BR main

num1: .WORD 0x0001

num2: .WORD 0x0002

sum: .WORD 0x0000

main: LDX 0x0002,I ;Load the index register with the value 2

LDA 0x0002,i ;Load the A register with the value 2

ADDA num1,x ;Add to A the value at the location specified but its address plus the value in x. Since num1 is at location 3 and the index register has the value two, then add the value to A that is in location 3+2 or 5, which is num2. num2 contains a 2, so the sum will be 4.

STA sum,d

DECO sum,d

STOP

.END

7.d.) The value output is: \_\_\_\_\_\_4\_\_\_\_\_\_\_\_\_. Explain. See comments on program

7.e.) Complete the program below. The comments specify the expected overall operation of the program. Replace the code below with your final version. Test your code. Be sure to vary the length of list (i.e., the number of values).

;Looping program to complete for lab

BR main

;There can be any number of .WORD values starting at

; location list, but the last one should be zero;

; (Obviously, you cannot exceed the memory available.)

list: .WORD +5

.WORD +12

.WORD -9

.WORD +0

sum: .WORD 0x0000

main: LDX 0x0000,i

;The following code reads through the values

; starting with the first value in list,

; sums the values, stops when it finds a zero,

; stores the sum in sum, and outputs the sum.

loop: LDA list,x

CPA 0x0000, i

BREQ finish, i

ADDA sum, d

STA sum, d

ADDX 0x0002, i

BR loop

finish: DECO sum,d

STOP

.END

**Question 8: Von Neuman machine illustration.**

Assemble and run the following code. Explain the output. Why does this illustrate the fact that code and data can be treated in the same way? Hint: In what memory locations are the opcode and the operand specifier? In what memory locations is the operand?

HERE: DECO HERE,d

STOP

.END

Output is 14592. It shows the data and instructions can be in the same place. I.e., both exist in locations 00 and 01. The instruction is in locations 0-2 and it outputs the value in 0-1.