Software Requirements Specifications (SRS)

Jorge Aguilar, RJ Alabado, Tiarnan Marsten, Dung Tran

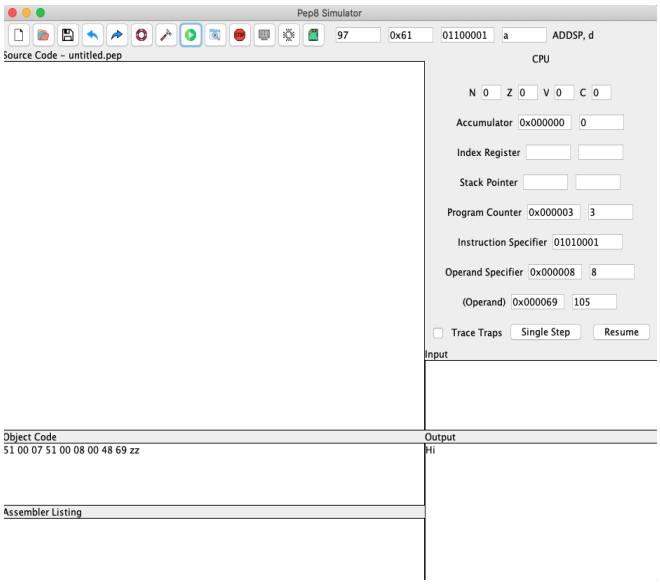
TCSS 360

November 8, 2020

Table of Contents

1.0 Program Preface

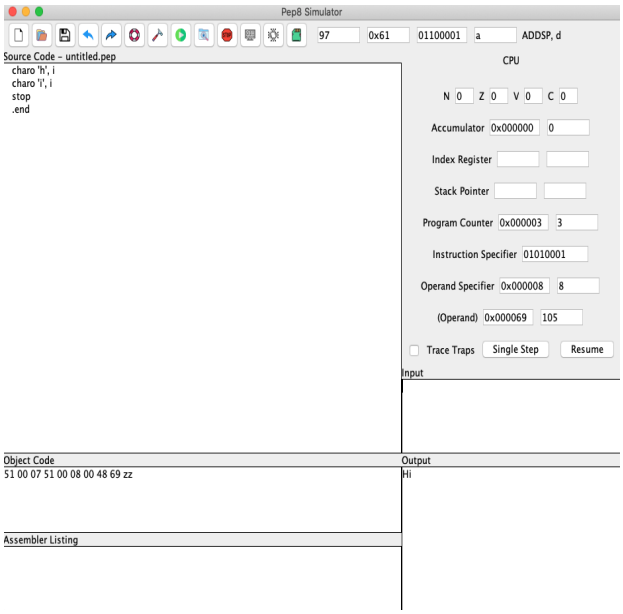
1.1 Previous Versions



The image on the right-hand side shows one of our earlier versions of this program. This earlier version had less functionality than the program that we are currently working on but does have some valuable features that we have chosen to build on for the updated version. For starters, the graphical user interface (GUI), developed using Java's Swing class, was designed in a way that almost accurately represents the Pep/8 program. The text areas, buttons, and panels were arranged to mirror the Pep/8 program. Another valuable feature of this older version is its ability to calculate and report the CPU and memory state of the program. Once a machine language instructions program is executed, the memory that has been used to allocate some of the instruction's commands is reported to the "Accumulator", "Program Counter", "Instruction Specifier", "Operand Specifier", and "(Operand)" text areas. Because memory is one of the more significant constraints of this program, we have chosen to use this feature for the updated version as well.

This previous version was only capable of handling three different types of machine language instruction (MLI) programs. The first was an "Output" program, which required that the user provide MLI to the object code text area in a hexadecimal format. Depending on the instructions that the user provided, the simulator would produce an ASCII representation of those instructions as two characters (e.g., "Hi", "No", "Up"). Another program that this simulator handled was a "Reverse" program that would reverse a two- or four-character string that the user supplied to the input text area. The final program that this simulator handled was a "Sum" program that found the sum of two numbers. To run this program, the user would have to supply those machine language instructions into the object code text area, which the simulator then used to calculate and output the correct sum. If you would like to learn more about the types of programs that this version could execute, please visit Appendix B and observe the test cases provided.

1.2 New Version



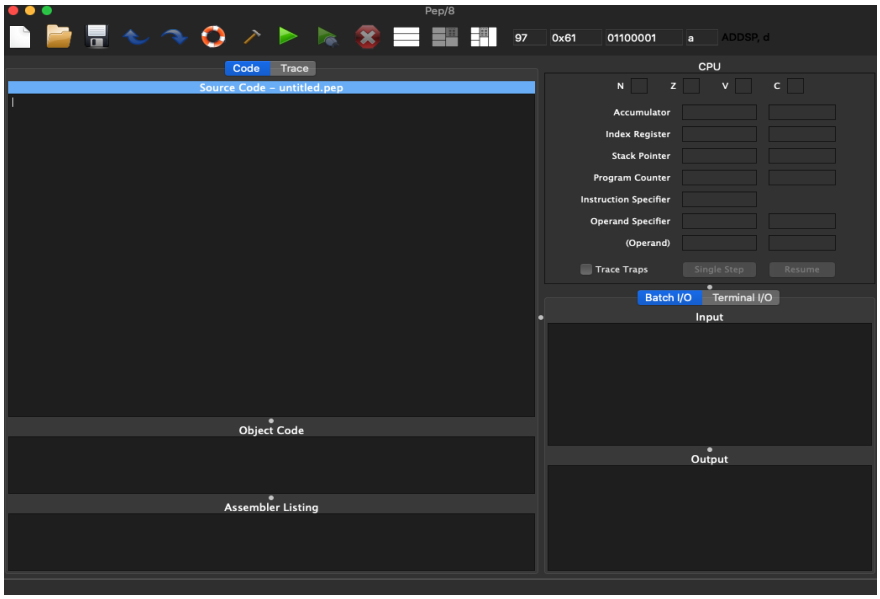
The image of the GUI on the right is a screenshot of the updated version of the program that I had shown above. As I had mentioned before, this new version builds on many of the features that the previous version provides. The key differences of this updated program can be found in its code and in the ways in which it can handle additional instructions.

Unlike the previous version, this program was developed using a Model, View, Controller (MVC) design. This method allows the programmers to organize this version's code into three packages. This makes the program's code more readable and maintainable. The View package contains most of the GUI code and design. This package is responsible for providing the user an interactive piece of software. The Model package keeps and manages the status of the program's numeric values. The Controller package communicates between the View and Model to ensure that the program functions properly.

The newest feature of this program is that it can accept and handle assembly language instructions. These are programs that are provided by the user to the source code text box. Once the simulator has that program, it is then assembled and converted into machine language instructions. These machine language instructions are then processed and converted into readable ASCII output. The image below shows some of the assembly language instructions that this program can process.

Opcode	Meaning of Instruction
0000	Stop execution
1100	Load the operand into the A register
1110	Store the contents of the A register into the operand
0111	Add the operand to the A register
1000	Subtract the operand from the A register
01001	Character input to the operand
01010	Character output from the operand

2.0 Introduction



This simulator was inspired by the Pep/8 program, which is a piece of software that can assemble assembly language instructions and run machine language instructions (Low-level languages). The Pep/8 program provides developers with an environment that is inspired by von Neuman's computer architecture to run these Low-level language programs and track how they are handled and stored in memory. This allows developers to create more efficient code and algorithms. von Neuman's computer architecture was designed and pioneered by John von Neuman in 1945. The von Neuman's computer architecture is composed of four parts which include an input device, a central processing unit (CPU), a memory unit, and an output device. The input device is used to accept low-level language programs, which are then sent to the CPU and memory. The CPU interprets these low-level programs and uses the memory unit to store the data of these low-level programs. Once the CPU has finished processing that data, it then translates it to readable ASCII output.

As noted earlier, our simulator was built to provide the user with the same look and feel of the Pep/8 program. When you compare our latest version to the Pep/8 program, you can find several similarities in the design of the user interface. However, our simulator is very limited and cannot perform all of the tasks of the Pep/8 program. For example, the only buttons that have functionality in our simulator are the build and run buttons, whereas the Pep/8 program has assigned functionality to all of its buttons. Another limitation of our program is that it does not have the same level of sophistication of the algorithms that are present in the Pep/8 program. As a result, our program is unable to handle large and complex instructions.

3.0 Description

3.1 Users of a Requirements Document

Types of Users	**Potential Uses**
System Customers/Users	The system customers/users can use the requirements document to see how the requirements that they have specified are reflected in the program. The requirements document is a detailed document of the requirements that the customer/user has requested.
Project Manager	The project manager may use the requirements document to design a project plan for the system development process and to create a schedule for the development team to complete the final product on time.
System Engineers/Developers	The system engineers/developers may use the requirements document to have a better understanding of the program so that they can design and develop its components.
System Test Engineers	The system test engineers may use the requirements document to understand how the different components and parts of the system interact. This information can guide their validation testing, thus ensuring that the system functions properly.

3.2 User Requirements

1. The Pep/8 simulator shall have a user-friendly design and interface, inspired by the original program, that is both easy to read and easy to use.
2. The Pep/8 simulator shall be able to process and assemble assembly language instructions.
3. The Pep/8 simulator shall be able to process the instructions of machine language instruction programs and produce a readable ASCII output.

3.3 System Requirements

1. The system shall efficiently execute machine language instructions using von Neuman's computer architecture to process the instructions, store important data in memory, and produce an ASCII output.
2. The system shall keep a database of opcodes for assembly language instructions in order to process user assembly language programs and convert them into machine language instructions.

3. The system shall report the state of the CPU and the memory locations that it has used in order to keep track of how the assembly language instructions and the machine language instructions are processed in the program.
4. The system shall provide useful error messages to the user when invalid or incomplete instructions have been supplied to the simulator. If possible, the system should specify which area this error was detected, and which type of error was detected.

3.4 Functional Requirements

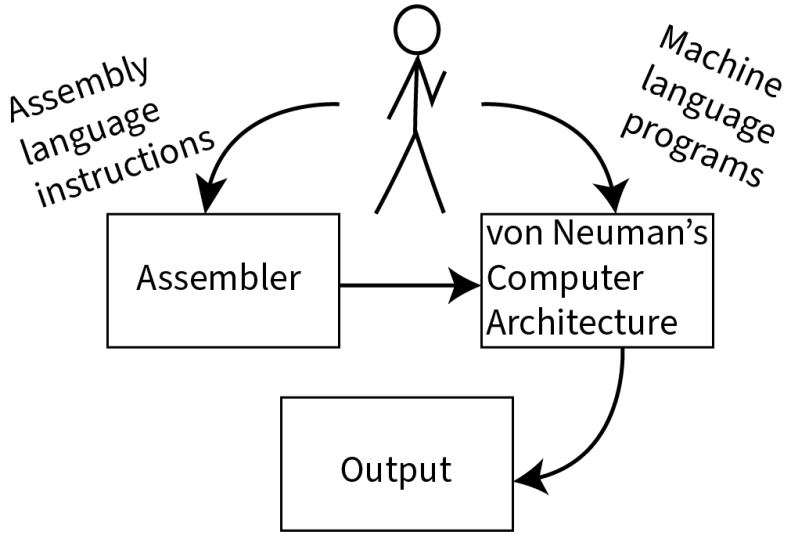
- The system shall provide a graphical user interface (GUI), designed by Java's Swift class, to replicate the same look and feel of the Pep/8 program.
- The system shall have text areas that are linked to buttons through an ActionListener that allows the simulator to process the user's input, assembly language instructions, and machine language programs.
- The system shall only accept complete assembly language instructions and machine language programs and provide the user with error messages if these items aren't provided.

3.5 Non-Functional Requirements

- The simulator shall only process short assembly language instructions and machine language programs.
- The simulator shall process machine language programs through the von Neuman's computer architecture in order to process and store important data more efficiently.
- The simulator shall use the least amount of memory registers as possible to process and convert machine language programs into readable ASCII output.

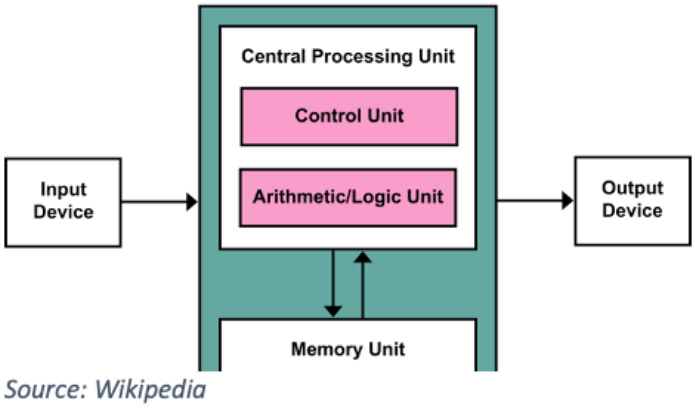
4.0 System Models

4.1 Processing User Input



The model on the right shows how our Pep/8 Simulator processes user input. A user may provide either assembly language instructions or machine language programs. When the user provides machine language programs, the simulator processes those programs using the von Neuman's computer architecture to produce a readable ASCII output of the data from that program. When the user provides Assembly language instructions, the simulator converts those instructions into machine language programs, which are processed in the von Neuman's computer architecture.

4.2 von Neuman's Computer Architecture

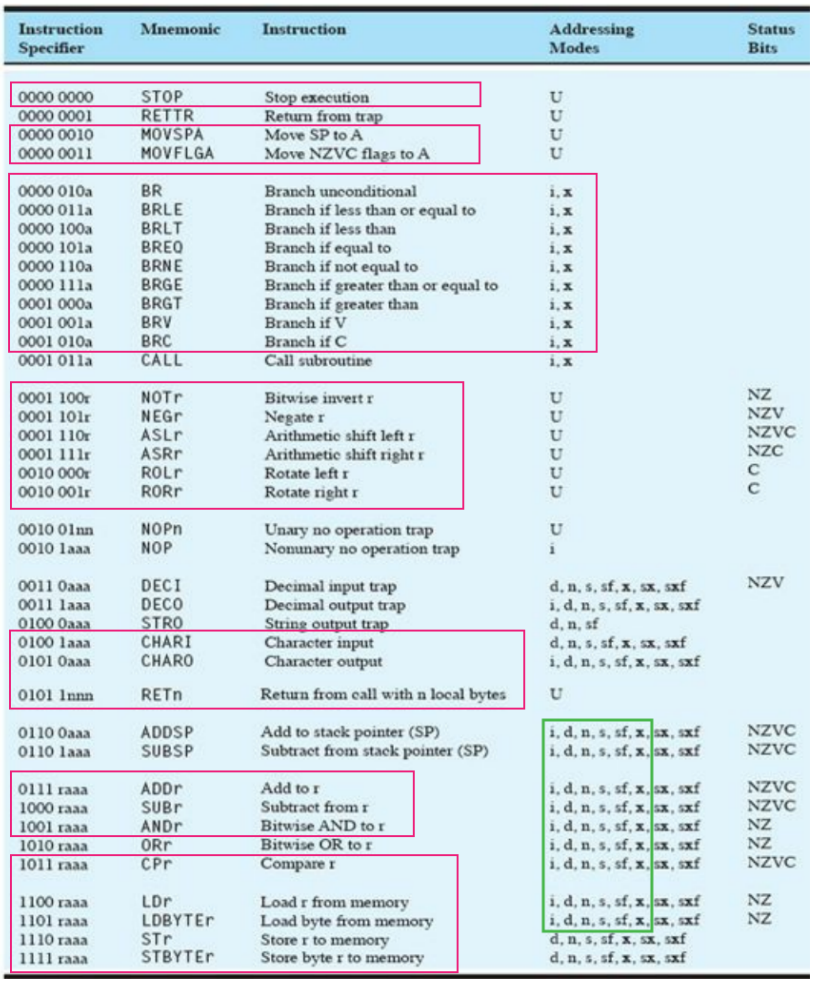


Source: Wikipedia

The von Neuman's computer architecture is an architecture that is developed to process low-level languages. The most commonly used low-level languages are assembly language instructions and machine language programs. The von Neuman's computer architecture breaks down these low-level languages using four different components. The first component is an input device that accepts user or system input. The second and third component work together to process that input through the central processing unit (CPU) and memory. The CPU stores key data and information in memory and retrieves that data when necessary. The final component is an output device that prints out readable and useable information, in our case this is an ASCII representation of the low-level languages provided.

5.0 Project Scope

The original Pep/8 program is able to handle 39 different instructions and operates through 8 different addressing modes. For our simulator, we hope to incorporate at least 30 different instructions and at least 6 different addressing modes.



Instruction Specifier	Mnemonic	Instruction	Addressing Modes	Status Bits
0000 0000	STOP	Stop execution	U	
0000 0001	RETTR	Return from trap	U	
0000 0010	MOVSPA	Move SP to A	U	
0000 0011	MOVFLGA	Move NZVC flags to A	U	
0000 010a	BR	Branch unconditional	i, x	
0000 011a	BRLE	Branch if less than or equal to	i, x	
0000 100a	BRLT	Branch if less than	i, x	
0000 101a	BREQ	Branch if equal to	i, x	
0000 110a	BRNE	Branch if not equal to	i, x	
0000 111a	BRGE	Branch if greater than or equal to	i, x	
0001 000a	BRGT	Branch if greater than	i, x	
0001 001a	BRV	Branch if V	i, x	
0001 010a	BRC	Branch if C	i, x	
0001 011a	CALL	Call subroutine	i, x	
0001 100r	NOTr	Bitwise invert r	U	NZ
0001 101r	NEGr	Negate r	U	NZV
0001 110r	ASLr	Arithmetic shift left r	U	NZVC
0001 111r	ASRr	Arithmetic shift right r	U	NZC
0010 000r	ROLr	Rotate left r	U	C
0010 001r	RORr	Rotate right r	U	C
0010 01nn	NOPn	Unary no operation trap	U	
0010 1aaa	NOP	Nonunary no operation trap	i	
0011 0aaa	DECI	Decimal input trap	d, n, s, sf, x, sx, sxf	NZV
0011 1aaa	DECO	Decimal output trap	i, d, n, s, sf, x, sx, sxf	
0100 0aaa	STRO	String output trap	d, n, sf	
0100 1aaa	CHARI	Character input	d, n, s, sf, x, sx, sxf	
0101 0aaa	CHARO	Character output	i, d, n, s, sf, x, sx, sxf	
0101 1nnn	RETn	Return from call with n local bytes	U	
0110 0aaa	ADDSP	Add to stack pointer (SP)	i, d, n, s, sf, x, sx, sxf	NZVC
0110 1aaa	SUBSP	Subtract from stack pointer (SP)	i, d, n, s, sf, x, sx, sxf	NZVC
0111 raaa	ADDr	Add to r	i, d, n, s, sf, x, sx, sxf	NZVC
1000 raaa	SUBr	Subtract from r	i, d, n, s, sf, x, sx, sxf	NZVC
1001 raaa	ANDr	Bitwise AND to r	i, d, n, s, sf, x, sx, sxf	NZ
1010 raaa	ORr	Bitwise OR to r	i, d, n, s, sf, x, sx, sxf	NZ
1011 raaa	CPr	Compare r	i, d, n, s, sf, x, sx, sxf	NZVC
1100 raaa	LDr	Load r from memory	i, d, n, s, sf, x, sx, sxf	NZ
1101 raaa	LDBYTEr	Load byte from memory	i, d, n, s, sf, x, sx, sxf	NZ
1110 raaa	STr	Store r to memory	d, n, s, sf, x, sx, sxf	
1111 raaa	STBYTEr	Store byte r to memory	d, n, s, sf, x, sx, sxf	

The instructions and addressing modes that we hope to use in our program are highlighted in the image to the right. The red squares cover the different instructions that we will be programming, and the green squares cover the addressing modes we will be programming.

To incorporate these different instructions and addressing modes in our program, we will be designing two Map data structures to store the mnemonics of the instruction specifiers and the identifiers of the addressing modes. Each line of assembly language instructions provided to the source code text box will be processed by our simulator and broken down into the parts that correspond to the mnemonics, addressing modes, and data. Once these parts are broken down, they will be assigned their appropriate binary string (containing 4 binary values) before being brought back together to create one long binary string of instructions. This long binary string of instructions will then be converted into hexadecimal, where it will become a machine language program, and will be provided to the object code text area to run the program that has been provided to the source code text area.

6.0 Appendices

6.1 Appendix A: Glossary

Term/Word	Definition
Pep/8	A software that provides developers with an environment that can process and execute low-level programs.
von Neuman's Computer Architecture	A computer architecture that is developed to process low-level languages. This architecture has four key components: an input device, a central processing unit, a memory unit, and an output device.
Machine Language Programs	Low-level programs that are written in hexadecimal.
Assembly Language Instructions	Low-level instructions that are assembled and converted into machine language programs.
Input Text Area	This is the text area in which the user shall provide a short string that may be reversed if the appropriate machine language program is executed.
Object Code Text Area	This is the text area in which the user shall provide complete machine language instructions so that the program can execute and output them as readable ASCII representations.
Source Code Text Area	This is the text area in which the user shall provide complete assembly language instructions so that the program can assemble and convert them to machine language programs.

6.2 Appendix B: Sample Test Cases (Machine Language Instructions)

Output a String:

Input for Object Code text area: 51 00 07 51 00 08 00 48 69 zz
Expected output: Hi

Reverse a String:

Two-letter Strings

Input for Object Code text area: 49 00 0D 49 00 0E 51 00 0E 51 00 0D 00 zz
Input for Input text area: up
Expected output: pu

Four-letter Strings

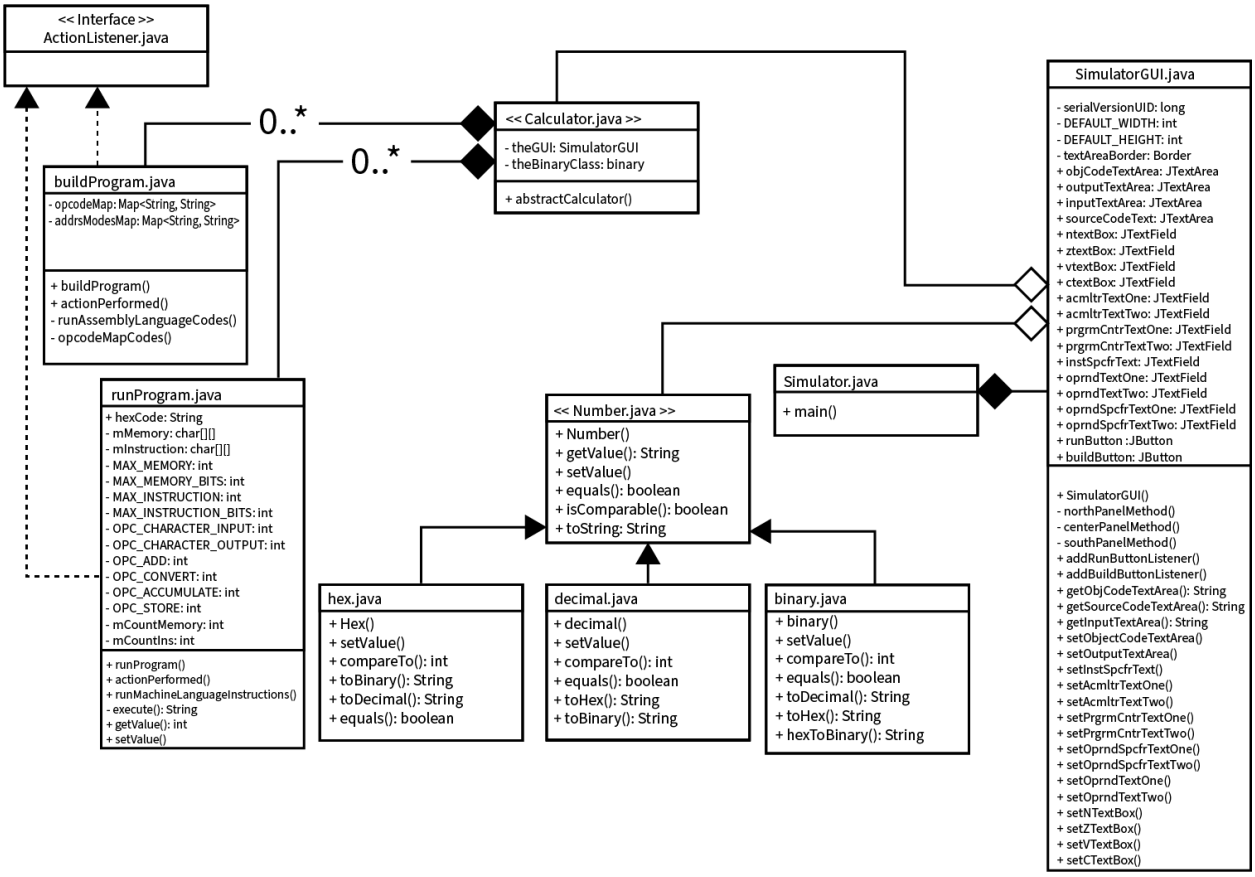
Input for Object Code text area: 49 00 15 49 00 16 49 00 17 49 00 18 51 00 18 51 00 17 51 00 15 51 00 16 00 zz
Input for Input text area: IOup
Expected output: puIO

Sum:

Input for Object Code text area: C1 00 11 71 00 13 A1 00 15 F1 00 10 51 00 10 00 00 02 00 04 00 30 zz
Expected output: 6

Input for Object Code text area: C1 00 11 71 00 13 A1 00 15 F1 00 10 51 00 10 00 00 05 00 03 00 30 zz
Expected output: 8

Activity 3) Revise your Design

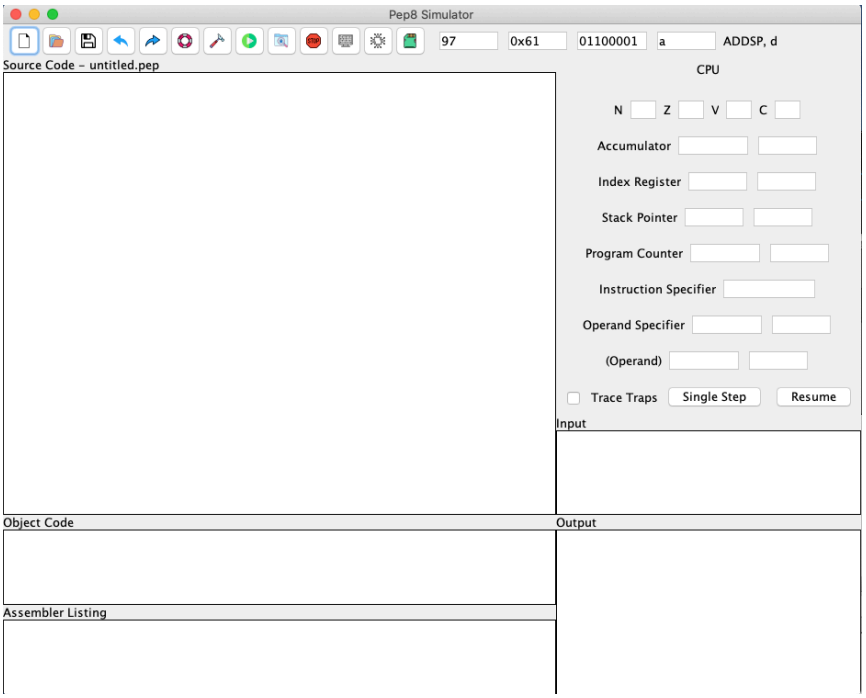


Activity 4) Revise your Mockup GUIs

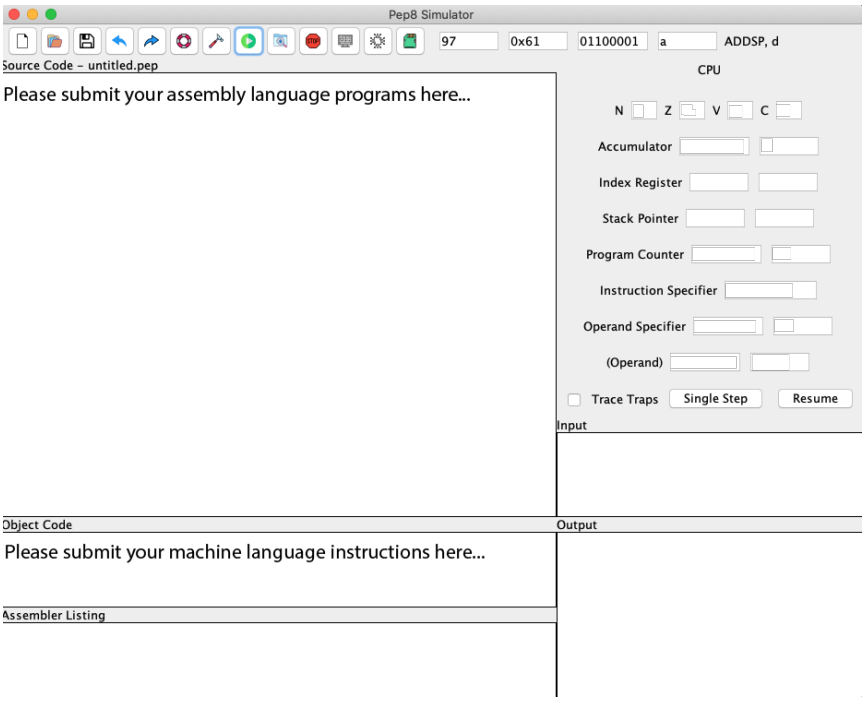
Pep/8 Simulator Mockup GUIs

GUI Design

1. GUI user interface:

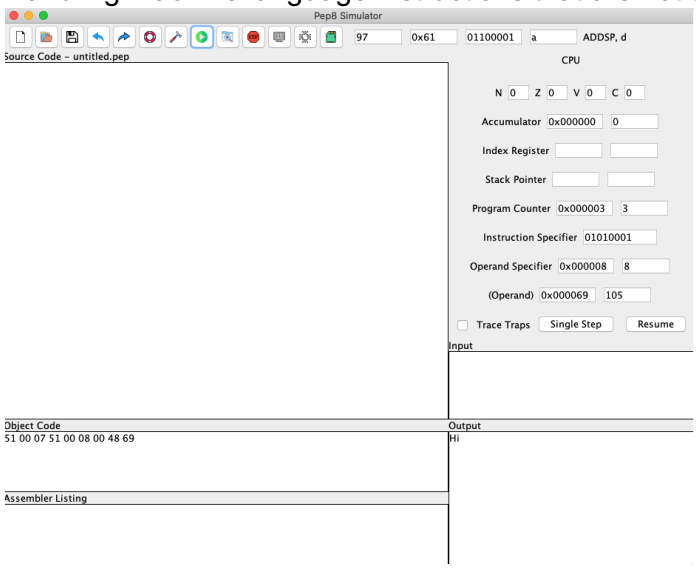


2. Prompting users to submit assembly language programs or machine language instructions:

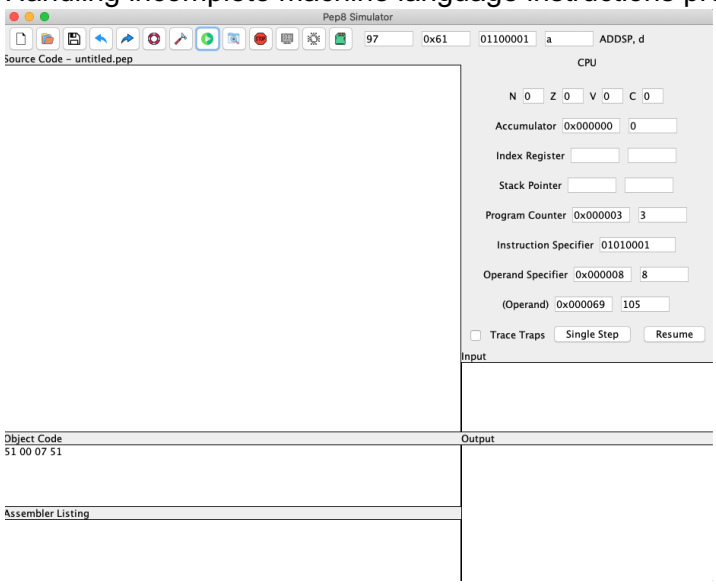


Handling Machine Language Instructions

1. Handling machine language instructions that are not terminated with “zz”:

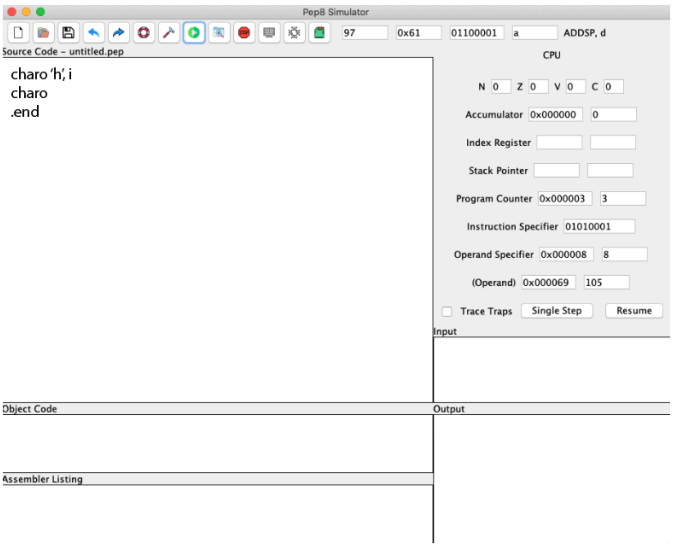


2. Handling incomplete machine language instructions programs:

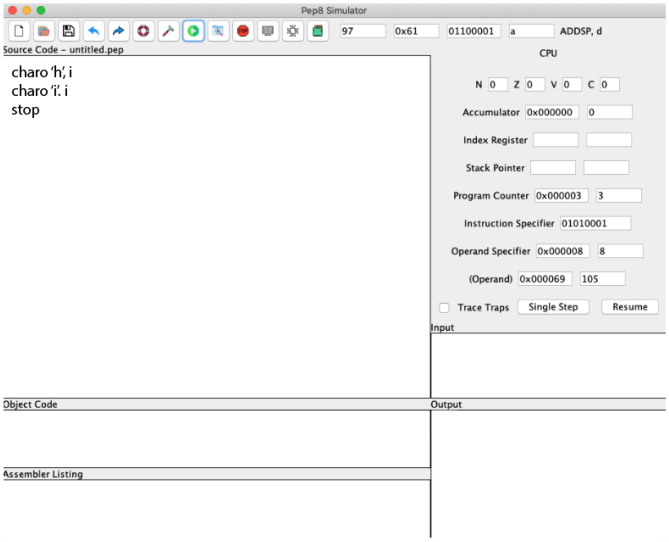


Handling Assembly Language Programs

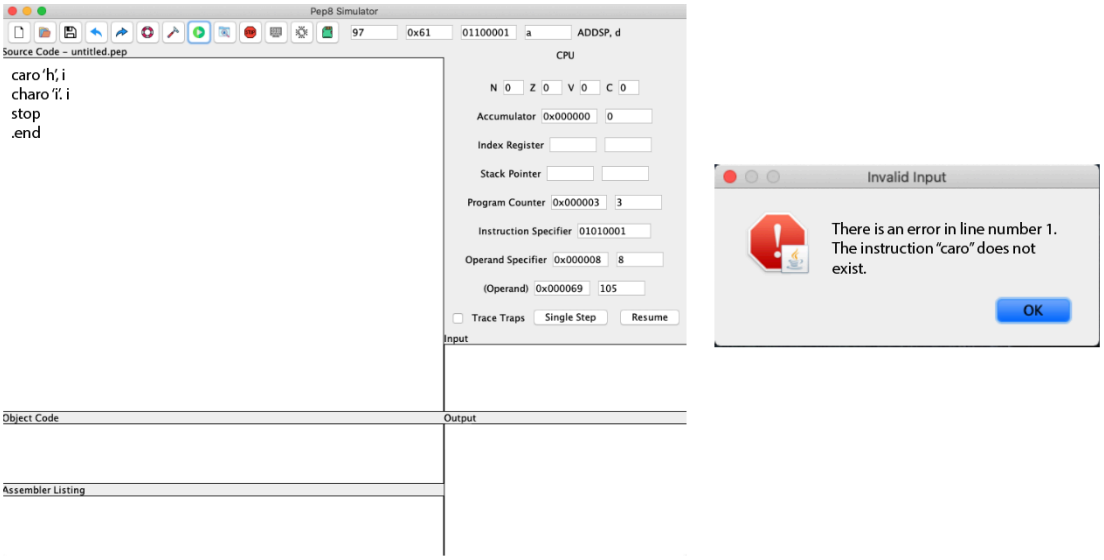
1. Handling assembly language programs that do not include “stop”:



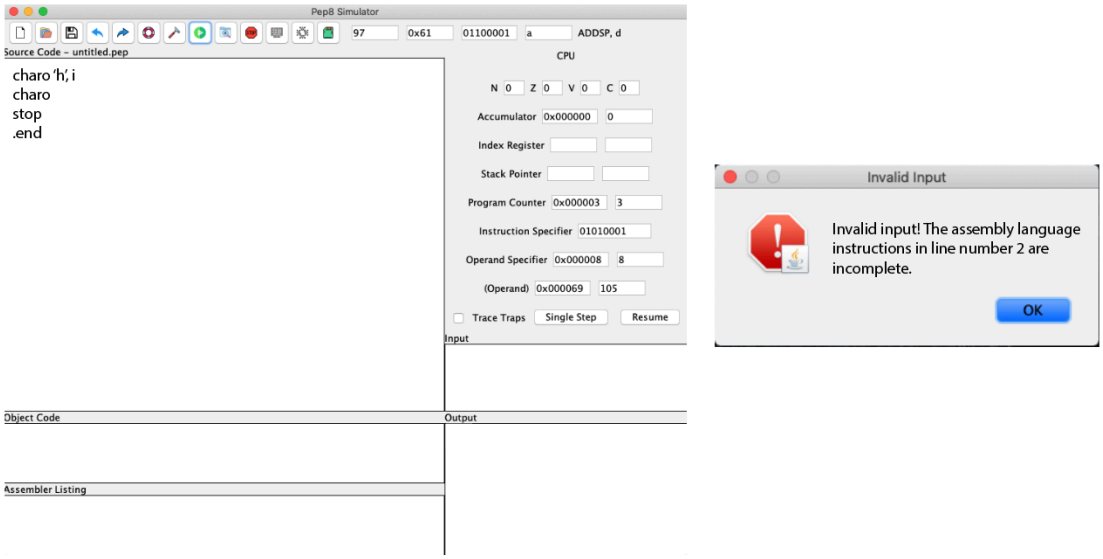
2. Handling assembly language programs that are not terminated with “.end”:



3. Handling errors for individual lines of assembly language programs:



4. Handling incomplete assembly language instructions:



New GUI Features (Memory Dump)

1. Adding a memory dump to the GUI to see how machine language programs and their corresponding data are stored in the simulator's memory unit.