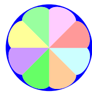
ISSN: 1545-679X
Information Systems Education Journal
Volume 4, Number 61 http://isedj.org/4/61/ August 23, 2006

In this issue:

Simulated Assembler-Objects and a Glass Bottom Computer (a Polytechnic Approach)

William G. Verbrugge
California State Polytechnic University, Pomona
Pomona, CA 91768 USA

Abstract: Integrated Development Environments are excellent production tools for intermediate and advanced programming students and even beginners after they have learned the core concepts (stored data, stored programs, computer instructions, and the anatomy of the computer). Most authors of introduction to programming books recognize this by their inclusion of one to twenty pages on this topic. This paper presents how using a simulated assembler (a tool for learning) with a simple assembly language can introduce the beginning student to the core concepts without having to be concerned with all the exceptions and rigor of a full assembler language. The Simulated Assembler with a full viewable Computer Machine (Glass Bottom Computer) and the easy procedures for using it in a first programming course are illustrated. Using the assembler tool described here should provide an increase in learning via a polytechnic (learn by doing) approach. A comparative analysis of using the assembler in an introduction to object programming course is provided.

Keywords: assembler, simple machine, software tools, language, programming, object oriented, machine language

Recommended Citation: Verbrugge (2006). Simulated Assembler-Objects and a Glass Bottom Computer (a Polytechnic Approach) *Information Systems Education Journal, 4* (61). http://isedj.org/4/61/. ISSN: 1545-679X. (A preliminary version appears in *The Proceedings of ISECON 2005:* §2524. ISSN: 1542-7382.)

This issue is on the Internet at **http://isedj.org/4/61/**

The **Information Systems Education Journal** (ISEDJ) is a peer-reviewed academic journal published by the Education Special Interest Group (EDSIG) of the Association of Information Technology Professionals (AITP, Chicago, Illinois). • ISSN: 1545-679X. • First issue: 8 Sep 2003. • Title: Information Systems Education Journal. Variants: IS Education Journal; ISEDJ. • Physical format: online. • Publishing frequency: irregular; as each article is approved, it is published immediately and constitutes a complete separate issue of the current volume. • Single issue price: free. • Subscription address: subscribe@isedj.org. • Subscription price: free. • Electronic access: http://isedj.org/ • Contact person: Don Colton (editor@isedj.org)

2006 AITP Education Special Interest Group Board of Directors

Stuart A. Varden
Pace University
EDSIG President 2004

Paul M. Leidig
Grand Valley State University
EDSIG President 2005-2006

Don Colton
Brigham Young Univ Hawaii
Vice President 2005-2006

Wendy Ceccucci
Quinnipiac Univ
Director 2006-07

Ronald I. Frank
Pace University
Secretary 2005-06

Kenneth A. Grant
Ryerson University
Director 2005-06

Albert L. Harris
Appalachian St
JISE Editor

Thomas N. Janicki
Univ NC Wilmington
Director 2006-07

Jens O. Liegle
Georgia State Univ
Member Svcs 2006

Patricia Sendall
Merrimack College
Director 2006

Marcos Sivitanides
Texas St San Marcos
Chair ISECON 2006

Robert B. Sweeney
U South Alabama
Treasurer 2004-06

Gary Ury
NW Missouri St
Director 2006-07

Information Systems Education Journal 2005-2006 Editorial and Review Board

Don Colton
Brigham Young Univ Hawaii
Editor

Thomas N. Janicki
Univ of North Carolina Wilmington
Associate Editor

Samuel Abraham
Siena Heights U

Tonda Bone
Tarleton State U

Alan T. Burns
DePaul University

Lucia Dettori
DePaul University

Kenneth A. Grant
Ryerson Univ

Robert Grenier
Saint Ambrose Univ

Owen P. Hall, Jr
Pepperdine Univ

Jason B. Huett
Univ W Georgia

James Lawler
Pace University

Terri L. Lenox
Westminster Coll

Jens O. Liegle
Georgia State U

Denise R. McGinnis
Mesa State College

Therese D. O'Neil
Indiana Univ PA

Alan R. Peslak
Penn State Univ

Jack P. Russell
Northwestern St U

Jason H. Sharp
Tarleton State U

Charles Woratschek
Robert Morris Univ

EDSIG activities include the publication of ISEDJ, the organization and execution of the annual ISECON conference held each fall, the publication of the Journal of Information Systems Education (JISE), and the designation and honoring of an IS Educator of the Year. • The Foundation for Information Technology Education has been the key sponsor of ISECON over the years. • The Association for Information Technology Professionals (AITP) provides the corporate umbrella under which EDSIG operates.

© **Copyright 2006 EDSIG.** In the spirit of academic freedom, permission is granted to make and distribute unlimited copies of this issue in its PDF or printed form, so long as the entire document is presented, and it is not modified in any substantial way.

Simulated Assembler-Objects and a Glass Bottom Computer (a Polytechnic Approach)

William G. Verbrugge
wgverbrugge@csupomona.edu
California State Polytechnic University, Pomona
3801 West Temple Avenue
Pomona, CA 91768 USA

Abstract

Integrated Development Environments are excellent production tools for intermediate and advanced programming students and even beginners after they have learned the core concepts (stored data, stored programs, computer instructions, and the anatomy of the computer). Most authors of introduction to programming books recognize this by their inclusion of one to twenty pages on this topic. This paper presents how using a simulated assembler (a tool for learning) with a simple assembly language can introduce the beginning student to the core concepts without having to be concerned with all the exceptions and rigor of a full assembler language. The Simulated Assembler with a full viewable Computer Machine (Glass Bottom Computer) and the easy procedures for using it in a first programming course are illustrated. Using the assembler tool described here should provide an increase in learning via a polytechnic (learn by doing) approach. A comparative analysis of using the assembler in an introduction to object programming course is provided.

Keywords: assembler, simple machine, software tools, language, programming, object oriented, machine language

1. INTRODUCTION

The growth in hardware technology has allowed the theories of modern programming languages to become a reality. In the beginning, developers of computer languages were hindered by the lack of processing speed and memory to implement their vision. Variable names and data were restricted in size and thus not very descriptive of their meaning. Most languages then followed a close representation of the function of the hardware in order to conserve on memory and be resourceful. Still, researchers continued to work on natural languages. One of the good outcomes from learning our industry's first languages is that the concept of how the computer worked was inherited in the language. Thus the logic of the application and how the computer actually ran the program was a natural outcome of learning the language.

Most computer languages taught today are object oriented programming languages. In these languages one builds objects (black boxes that have attributes and behavior and identity (a name)) that can be used by other objects. To aid in covering all the essential topics, production tools are used so more time can be spent on logic and object concepts. Many instructors use Integrated Development Environments (IDE) to aid in writing the source code. Results of student tests in introduction to object oriented programming languages using an IDE, has indicated that many students had a weak understanding of the concepts of stored programs, memory, the difference between instructions and data, the compilation process, and simple execution logic. This poor outcome was in spite of the Instructors clearly covering these topics and providing diagrams of how a created object would be ref-

erenced in memory. What seemed to be missing were the hands on writing and viewing of a logical process in the core of a computer.

This paper presents how using a simulated assembler (a tool for learning) with a simple assembly language can introduce the beginning student to the basic concepts of how programming languages will run on the hardware. Although the Simulated Machine (SM) can solve complicated procedures (sorting, simulations, etc.), it is best used in an introduction programming course to show simple comparisons, arithmetic operations, transfers, etc. The author has found that thirty to forty minutes of class time and a simple assignment provides an excellent reference when introducing an object oriented programming language. Most authors of introduction to programming books recognize this by their inclusion of one to twenty pages on this topic (Gittleman, 2002;Koffman, 2002). The Simulated Assembler is available at www.csupomona.edu/~wgverbrugge and the easy procedures for using it in a first programming course are provided.

2. LEARN THE ANATOMY

The Simulated Machine (SM) illustrates the anatomy of the computer. Its view allows the student to see all phases of the programming cycle (writing the source, compiling the source to object code, and running the object code) in one view. And with the ability to execute one instruction at a time, the student can see the program move data (instructions, variable, or constants) from memory to registrars to memory. A common practice in introduction programming courses is for the instructor to display a small set of numbers and ask the class to tell them the average (answers come quickly). Then the class is asked to explain how they obtained the answer step by step (answers come slowly). This leads to a flow chart or pseudo code of the procedure and then how one needs to tell the computer box to perform the task (Malik, 2005). One can then describe the anatomy of the computer (Central Processing Unit (CPU) - machine instructions and registers, memory, input/output, etc.). Next with a simple assembler language, described below, the pseudo code can be translated into a computer language that represents the instructions of the CPU. Subsequently the same program could be illustrated in the language being taught. Figure 1 (in the appendix) shows a more general-purpose program that uses a loop and test to accept a sequence of numbers. Figure 1a illustrates the flow chart that set the logic for writing the assembler source code. The Java equivalent is also shown.

A sequence of machine instructions is a program. Each instruction command is represented by a binary pattern (0001 0100 0011 0100). If the first 6 bits (= 5 in decimal) of this pattern represents the operation code (opCode) and the remaining 10 bits (the operand) represents the memory location (= 100 in decimal), then in the SM this instruction would mean - clear the accumulator registrar and add the value at location 100 in memory. The SM illustrates the binary using decimal so that the instructions and memory locations are easy to read. Many instructors teach binary to decimal conversion. The SM provides an answer to the question "Why are we doing this?" Programming in the machines language would be a real test of ones personal memory. Thus a programming language called an assembler was created that used a mnemonic code for the instruction operation and used numbers or variable names that represented the operand. In Figure 1 the first assembler instruction (Start: CLA 0) is shown in the source code section. The compiler (translator), which is called when one presses the **Compile Code** button, translates this instruction to its machine instruction equivalent. "Start:" would have the value 0, since the first instruction is in location zero of memory. The operation code "CLA" is translated to a binary 5. The "0" gets stored in location 100 of memory (the first location that data is stored in this SM). The object code section of Figure 1 shows the result "00 05100" of the translation to machine code. This is the instruction CLA (5) - clear the accumulator and add to it the value at memory location 100.

Thus a source program is entered into the Source Code panel. It is then submitted by pressing the **Compile Code** button. Each line is read sequentially, interpreted, and stored into the Simulated Machine's memory, which is fully viewable via the Glass Bottom Computer concept. The instructions are stored in addresses 000-099. All vari-

ables and constants are stored in addresses 100-999. This is shown in the “Object Code from Source” panel. After writing a program it is easy to execute the code using the Simple Machine - "**Run Object Code" or “Run One Inst”** button. The third panel shows the execution results and the values in the registrars as the program is executed in the machine.

The Simulated Machine is an Applet and thus can be run in most browsers. This makes it very deployable. However, because of security two very useful request buttons “Save_Source”, which allows one to save their source code to disk; and “Load_Source”, which allows one to load a saved source code back into the Simulated Machine are not allowed when running in the browser. “Why this is the case?” is a topic for another paper. Programs that require over fifteen lines of code probably need this feature. To accommodate the use of these two features, the web site that runs the Simulated Machine allows for the download of an executable Windows jar version. The event buttons shown in figure 1 perform the following actions:

- The **"Insert Row"** button allows one to insert an instruction - click on the row to insert and then click the **"Insert Row"** button. A blank row is inserted above the selected row. Once focus is set in a table cell by clicking in it, **the left or right arrow key** will move the cursor.
- The **"Clear Code"** button will clear the source code panel.
- The **"Compile Code"** button will translate the source code into object code and display it in the Object Code from Source panel. The instructions are shown starting in memory location zero (0) and are displayed in decimal. The last three digits are the memory location of the operand and the beginning digits represent the operation code. Operands are assigned to memory locations starting at memory location one hundred (100) and their values are displayed.
- The **"Save_Source"** button will copy the source code as a comma delimitated file to the clipboard. To save the source code so it may be reloaded at another time, open Notepad and Save (Ctrl-V) it. Then save the Notepad file with proper name to a computer disk directory.

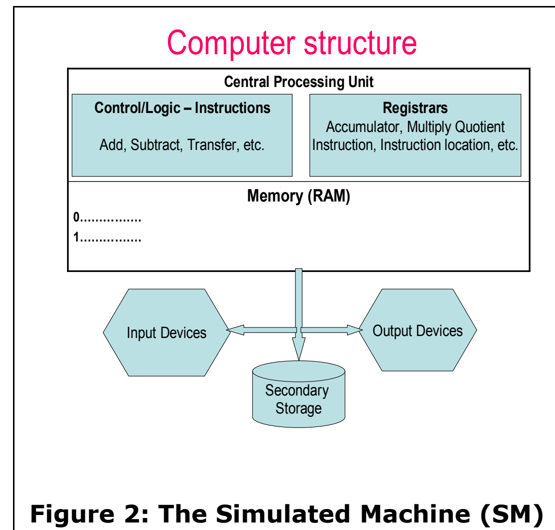

Figure 2: The Simulated Machine (SM)

- The **"Load_Source"** button presents a dialog box to paste the source copied from the backup Notepad file and puts it in the Source table.
- The **"Print"** button will print the output of the execution and the source code.
- The **"Run Object Code"** button will run the object code starting at memory location zero. When the STP (01) operation code is executed, clicking OK will print the results in the Execution Results panel.
- The **"Run One Inst"** button will run the object code one instruction at a time - thus letting you see the contents of the registers and memory locations change as the program runs.

3. THE HARDWARE

The simulated machine, like most computers, consists of three major elements: core memory, instruction control unit, and registers (see figure 2). All these elements store data in a five digit numerical format. In a real computer these decimal digits are binary numbers (i.e. memory location 030 is 11110 in binary. We could make the machine all binary, but using decimal digits does not lose any concepts and it makes it much easier to visualize the internal hardware.

The SM has:

- 1000 Memory Locations (Addressed 0 - 999)
    - 000 - 099 Reserved for Instructions
    - 100 - 999 Reserved for Variables and Constants
- AC -- Accumulator Register
- MQ -- Multiplier-Quotient Register
    - Both registers can hold any positive or negative value greater than or equal to –2,147,483,648 and less than or equal to 2,147,483,647.
- Two controlling Registers
    - Instruction Register -- Holds the binary instruction - viewed in Decimal
    - Instruction Location Register -- Holds the binary value of the memory location where the instruction was located - viewed in Decimal

The simple machine will execute instruction sequentially beginning with address 000, unless altered by a transfer statement.

4. THE ASSEMBLER LANGUAGE

The assembler language presents mnemonic codes that represent the machine hard wired bit code instructions. An instruction consists of an operation code and an operand. The operation code determines the action the computer should perform and the operand is the location in memory that the action is performed on. An assembler program consists of a list of instructions. Assembler instructions have the following format:

LABEL: OpCode Operand # Comment

The assembler is not case sensitive. Thus cla, CLA, and Cla are the same. Some different forms of an instruction are the following:

```
start:  CLA 1      # 1 is a constant
                   # start: is a label
        STO one    #one is a variable
                   # and holds 1
        TRA Next:
        ADD one    # this instruction
                   #will be skipped
Next:   STP
```

Notice that labels and comments are not essential for an instruction. However, all operation codes except "STP" (Stop the program) require an operand.Each instruction may be comprised of the following four major elements.

1. Labels
    - Labels are used as a reference to a specific memory location.
    - Labels follow the following format.
      Name:
        - "Name:" is an identifier, which refers to the current line. It can be any word followed by a colon, which is left to the programmer's discretion.
        - The colon, (:), is used to signify that it is a label. It follows directly after the name.
2. Operation Code
    - An operation code is a special three-character command, which informs the computer to perform a specific function, such as add or subtract.
    - At run time, the operation code has been translated into a two-digit code, which the Machine simulator can understand and manipulate.
    - See Table 1 for a list of the operation codes and their function.
3. Operands
    - Operands can consist of labels, variables, and constants.
    - Using a label as an operand would allow one to modify an instruction.
    - Variables refer to memory locations, which store binary data.
    - Variables are formatted as follows.
      variable or VARIABLE;
      number1 or NUMBER1
        - Any sequence of characters - the compiler is not case sensitive.
    - Constants are positive or negative numbers, which can range from negative 2,147,483,648 to positive 2,147,483,647. These do not have any distinctive characters attached.
        - To use a constant, simply use the positive or negative number after an operation code.
    - At run time, the simulator will translate the operand into its numerical code and store it in the proper memory address. For example, if you had only one variable, the simulator would store that variable at address 100. Anytime it is referenced in an instruction, the variable is replaced with its address.
4. Comments
    - Comments are non-essential parts of a program. They are there for the sole purpose of readability of a pro-

gram. The format of a comment is as follows:
 # this is a comment
- Notice that a comment may begin with a pound sign (#).
- The simulator will ignore anything following the pound sign.

- At run time, the simulator will strip all comments from the instructions.

5. THE MACHINE INSTRUCTIONS

The machine has fourteen instructions as listed in Table 1. When reading the table, note that **(x)** should be read as the contents of x (e.g. (MQ) means the contents of the Multiplier-Quotient Register). The "->" symbol should be read as "is placed into". The letters **"bbb"** refer to the memory address of the operand. For example, the instruction "**ADD one"** would be interpreted as operation code (OpCode) = ADD and operand = one (a variable which is a reference to a memory location). The effect ((bbb) + (AC) -> (AC)) is read as "the contents of the memory location of the variable (one) plus the contents of AC are placed into the contents of AC." Also, all operations except STP need a memory location (represented by bbb), which can be a constant, label, or a variable. The letter sequence "iff" is read as "if and only if."

Table 2: The results of an unpaired t-test

t = -1.96 ; Standard Deviation = 3.24 ; degrees of freedom = 7

The probability of this result, assuming the null hypothesis, is 0.090

Data:
Group A: Final class average without using the Simulated Machine. Number of items= 6
76.0, 76.0, 77.0, 78.0, 78.0, 82.0 - Mean = 77.8
95% confidence interval for Mean: 74.71 thru 80.96
Standard Deviation = 2.23
Hi = 82.0 Low = 76.0
Median = 77.5
Average Absolute Deviation from Median = 1.50

Group B: Final class average using the Simulated Machine. Number of items= 3
79.0, 80.0, 88.0 - Mean = 82.3
95% confidence interval for Mean: 77.91 thru 86.76
Standard Deviation = 4.93
Hi = 88.0 Low = 79.0
Median = 80.0
Average Absolute Deviation from Median = 3.00

6. PRELIMINARY BENEFIT RESULTS

There is not enough data to perform a t-test assessing whether the means of the two groups (classes using the assembler and classes not using the assembler) are statistically different from each other (Newcombe, 1998). The t-test gives the probability that the difference between the two means is caused by chance. It is customary to say that if this probability is less than 0.05, that the difference is 'significant' (the difference is not caused by chance). The current data is from the author's experience and is illustrated in Table 2. Other professors at CSU-Pomona are now using the Assembler, thus the data analysis will become more stable. Another outcome experienced by the author was that the average percent of gain of material offered using the Simulated Assembler was 9 percent.

7. CONCLUSION

Experience has established that an understanding of how a stored program is executed by a computer is one of the main learning concepts to understanding how to write a program in a procedural oriented language. As one moves to object oriented languages, where the running program creates objects and stores them in memory, the understanding of the concept becomes even more important (McKeachie, 2002; Bransford,2000). The Simulated Assembler presented here should provide the student with the fundamental concepts of developing and running a computer program. Thus, the learning progression of defining global and local variables, operations, and objects will have a foundation to build on. The Simulated Assembler can be used as a root to many courses – providing a time saving reference as new topics are presented. The author has found that using the Simulated Assembler in an introduction to programming course using an object oriented language provided at least a full class period extra for introducing new topics. Open access to the Simulated Assembler via an applet or win-

dows version (which can be down loaded) is on the University's web server located at www.csupomona.edu/~wgverbrugge. Some courses of study require learning a full assembler language (IBM, 2001) as the root to their discipline. For those that do not have this requirement, the SM can be illustrated in one lecture. If a course requires more profound study, the SM can be used to illustrate topics like setting up arrays, modifying instructions, etc. The SM is easy to operate and operating instructions are provided on the web, since these will change as enhancements are added. The current version (Version 2) provides more readability with changing memory views, save/load source code to provide more productivity in larger programs, and a print capability for hard copy results.

8. REFERENCES

Bransford, J.D., A.L. Brown, and R.R. Cocking, eds. "How People Learn: Brain, Mind, Experience, and School Committee on Developments in the Science of Learning. Commission on Behavioral and Social Sciences and Education of the National Research Council", National Academy Press, 2000, ISBN: 0-309-07036-8.

Gittleman, Art (2002). "Computing with JAVA Alternate second Edition". Scott/Jones. pp 2-5.

IBM - International Business Machines Corporation (2001). "AIX 5L for POWER-based Systems Assembler Language Reference 2nd Edition."

Kirk, J.J. "An Unofficial Guide to Web-based Instructional Gaming and Simulation Resources," ERIC Document Reproduction Service ED472675, 2001.

Koffman, Elliot and Wolz, Ursala (2002). "Problem Solving with Java - 2nd Edition". Addison Wesley. pp 1-16.

Kohn, A. "Students Don't 'Work' – They Learn," Education week, September 3, 1977.

Larman, Craig (2002). "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Ed)". Upper Saddle River, NJ: Prentice Hall PTR.

Malik, D.S. (2005). "Java Programming – From Problem Analysis to Program Design 2nd Edition". Thomson Course Technology. pp 1-21.

McKeachie, W McKeachie's "Teaching Tips: Strategies, Research and Theory for College and University Teachers, 11th ed" Boston: Houghton Mifflin, 2002.

Newcombe RG. "Two sided confidence intervals for the single proportion: Comparison of seven methods" *Statistics in Medicine* 1998;17:857-872.

Null, Linda and Lobur, Julia (2003). "MarieSim: The MARIE computer simulator" Journal on Educational Resources in Computing (JERIC),v.3 n.2, p'1-29, June 2003.

Sim, Edward R, and Wright, George "'The Difficulties of Learning Object-Oriented Analysis and Design: An Exploratory" Journal of Computer Information Systems, XXXXII, 2, 95.

Yehezkel, Cecile (2002). "A taxonomy of computer architecture visualizations" ACM SIGCSE Bulletion, v.34 n.3, September 2002.

Appendix

Table 1 – Instructions (Operation Codes and their Effect)

Machine Instructions

The structure of an instruction is [OpCode Operand]

Operation Code	Numerical Value	(bbb) means contents of the memory location as specified by the Operand; (AC) means contents of the Accumulator
STP	01	Stops the program
TRA	02	Transfer to next instruction at (bbb)
TLE	03	Transfer to next instruction at memory location specified by the operand iff (AC) <= 0, else next instruction
TNZ	04	Transfer to next instruction at location bbb iff (AC) != 0, else next instruction
TEZ	15	Transfer to next instruction at location bbb iff (AC) = 0, else next instruction
CLA	05	(bbb) -> (AC)
STO	06	(AC) -> (bbb)
LDQ	07	(bbb) -> (MQ) ; MQ = Multiplier-Quotient Register
STQ	08	(MQ) -> (bbb)
ADD	09	(bbb) + (AC) -> (AC)
SUB	10	(AC) - (bbb) -> (AC)
MPY	11	(MQ) * (bbb) -> (MQ)
DIV	12	(MQ) / (bbb) -> (MQ), remainder -> (AC)
RD	13	Contents of input -> (bbb)
WRT	14	(bbb) -> Printed to output frame

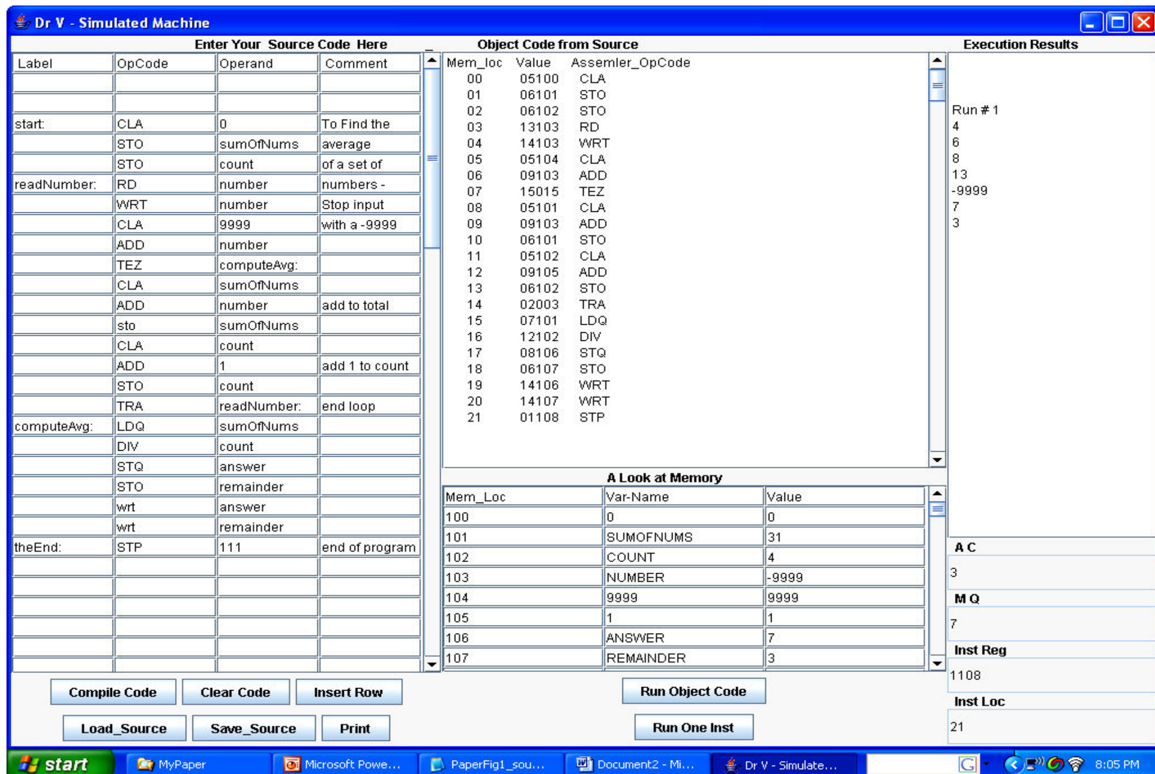

Figure 1 -- This program finds the average of a set of numbers input until a –9999 is entered. See corresponding Java program below.

```
//  The Java model class for the assembler find average program
public class Avg
{          public void computeAvg()
           {  double num = Double.parseDouble( JOptionPane.showInputDialog("Enter A Number"));
              double count = 0;
              double sum = 0;
               while (!(num == -9999))
                {    sum = sum + num;
                     count = count + 1;
                     num = Double.parseDouble( JOptionPane.showInputDialog("Enter A Number"));
                 }
                JOptionPane.showMessageDialog(null, " Avg = " + sum / count);
           } // end computeAvg()
 } // end class
```

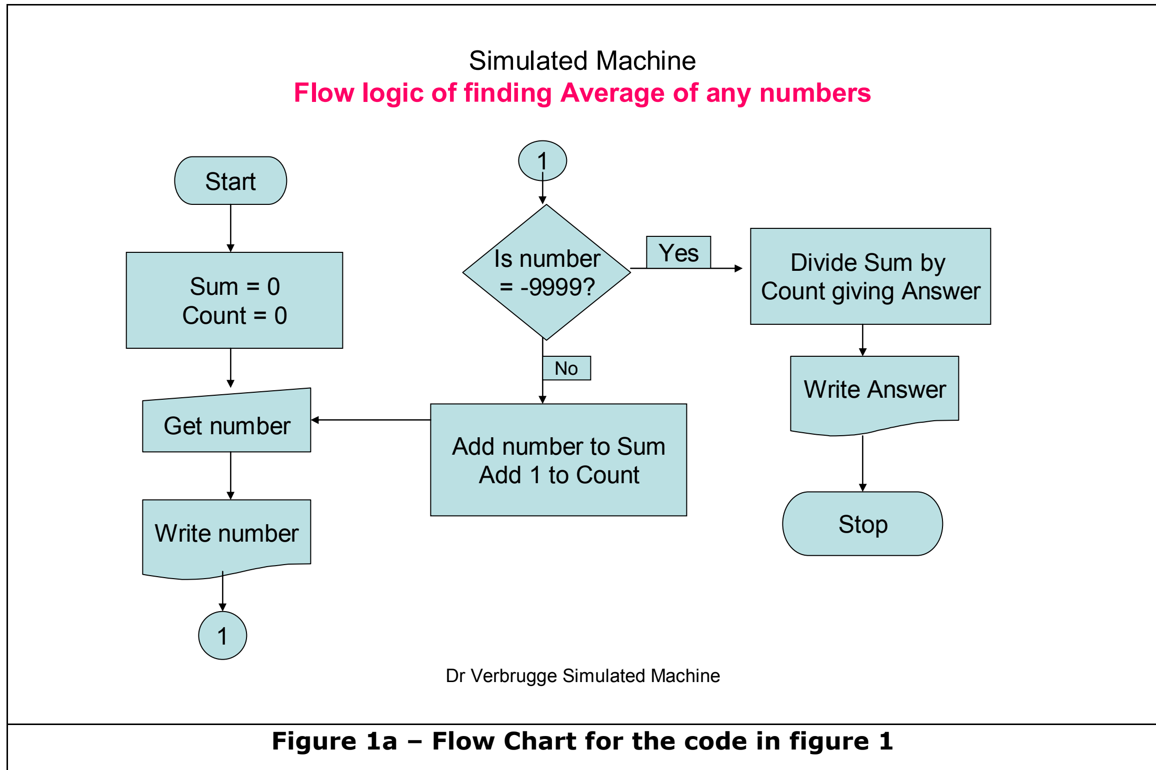

Figure 1a – Flow Chart for the code in figure 1