VISUALIZATION OF MICROPROCESSOR EXECUTION IN COMPUTER ARCHITECTURE COURSES: A CASE STUDY AT KABUL UNIVERSITY

By

Mohammad Hadi Hedayati

A thesis submitted

In fulfilment of the requirements

For the degree of Master of Science

in the

Department of Computer Science

Faculty of Natural Sciences

University of the Western Cape

October 2010

Supervisor: Prof. H.O. Nyongesa

ABSTRACT

Computer architecture and assembly language programming microprocessor execution are basic courses taught in every computer science department. Generally, however, students have difficulties in mastering many of the concepts in the courses, particularly students whose first language is not English. In addition to their difficulties in understanding the purpose of given instructions, students struggle to mentally visualize the data movement, control and processing operations. To address this problem, this research proposed a graphical visualization approach and investigated the visual illustrations of such concepts and instruction execution by implementing a graphical visualization simulator as a teaching aid.

The graphical simulator developed during the course of this research was applied in a computer architecture course at Kabul University, Afghanistan. Results obtained from student evaluation of the simulator show significant levels of success using the visual simulation teaching aid. The results showed that improved learning was achieved, suggesting that this approach could be useful in other computer science departments in Afghanistan, and elsewhere where similar challenges are experienced.

KEYWORDS: Computer Architecture, Assembly Language Programming, Microprocessor Operations, Instruction Set Architecture, Microprocessor Visualization, Computer Visualization and Simulation, Computer Assisted Learning, Human Computer Interface.

DECLARATION

I, Hadi Hedayati, declare that this thesis titled *Visualization of Microprocessor Execution in Computer Architecture Courses: a Case Study at Kabul University* is my own work, that it has not been submitted before for any degree or examination in any other university, and that all the sources I have used have been indicated and acknowledged as complete references.

Mohammad Hadi Hedayati

October 2010

ACKNOWLEDGMENTS

I would sincerely like to thank my supervisor Prof. Henry Nyongesa for all the help during the conduct of this research. I would also like to thank Ms Jandelyn Plane, my co-supervisor at the University of Maryland who has helped me during the writing of this dissertation.

TABLE OF CONTENTS

Contents **Pages**

List of Tables

Tables **Pages**

List of Figures

Figure **Pages**

Chapter 1

STATEMENT AND ANALYSIS OF THE PROBLEM

1.1 Introduction

During their 40 years in existence, the functionality of microprocessors has advanced at an exponential rate, yet their basic architecture remains the same. Consequently, the teaching of the internal operation of the central processing unit (CPU) of a microprocessor has remained relatively stable. Students' first exposure to CPU operation concepts is usually on simple microprocessor architectures. But even grasping the basic architecture can be daunting at first and many students find it difficult to understand microprocessor operations.

Currently, students at Kabul University (KU) in Afghanistan are introduced to the Computer Architecture course using uncomplicated Intel-based microprocessor architecture. Yet, although the architecture is uncomplicated, the CPU instructions are in English and the students find it difficult to comprehend the material concepts presented in the course. This research investigated how these challenges can be addressed. It was proposed that a graphical approach would be beneficial and would aid in the understanding and comprehension of microprocessor architecture and instruction execution. The objective of the research was to develop a new software tool to help students at KU understand microprocessor operations in a manner not alien to them.

1.2 Research Objectives

The proposed graphical simulation of CPU operations is designed to illustrate data movement, processing and control at the register level. Students may enter a piece of program and observe the effect of its execution visually (Martins, 2002). A flexible graphical user interface (GUI) was developed for this visualization. The specific objectives of the research were as follows.

- To create a portable application that could be used on different operating system platforms. This is necessary as the equipment available for teaching may differ between institutions.
- To develop an interactive teaching system that guides students through the learning process.
- To provide, within the software, error detection and debugging help, including program listing and highlighting of errors in a program.

For the above objectives, it was necessary to investigate and understand three technical areas imperative for the creation of the software (Yehezkel, 2003):

1. Microprocessor architecture
2. The assembly programming language
3. Programming languages – Visual Basic.NET was the programming language selected, for reasons discussed in Chapter 2.

1.3 Research Questions

In order to achieve the research objectives it was necessary to formulate formal research questions for the research. These were:

1. What would be the best way to visualize microprocessor operations in the context of KU students?
2. What would be the optimal subset of the Intel-based microprocessor instruction set to implement in order to visualize the microprocessor?
3. How could the effectiveness and usability of the visualization tool be evaluated?

1.4 Outline of the Dissertation

This dissertation is divided into seven chapters, each of which describes a stage in the research development. Chapter 2 gives general background information, first presenting detailed assembly language programming principles, and secondly an overview of the Intel-based microprocessor instructions, where justification is also given for the selected subset. The background material in Chapter 2 concludes with a brief overview of Visual Basic.NET. Chapter 3 presents a literature review of microprocessor simulation concepts and tools. It reviews computer-aided learning where it has been used in institutions world-wide and across different disciplines, and describes reported levels of success. Chapter 4 presents a design specification for the visualization tool. The chapter outlines the specific goals for each internal component part of the microprocessor, and describes their implementation. Chapter 5 describes the stages involved in the implementation of the simulator as specified in Chapter 4. Chapter 6 discusses the tests carried out in the development process to ensure the software operates correctly and that design specifications are met. The student experience survey that was used to evaluate the software's suitability for its intended purpose is then presented. Chapter 7 concludes the report, providing a summary of what was achieved, and suggests possible improvements and extensions to the tool.

Chapter 2

BACKGROUND

2.1 Computer Organization and Architecture

The organization of most modern computers and almost all microprocessors is that described by John von Neumann in a 1945 draught report describing the EDVAC (Electric Discrete Variable Automatic Computer) (Von Neumann, 1945). The simplified layout of this computer is shown in Figure 1**.**

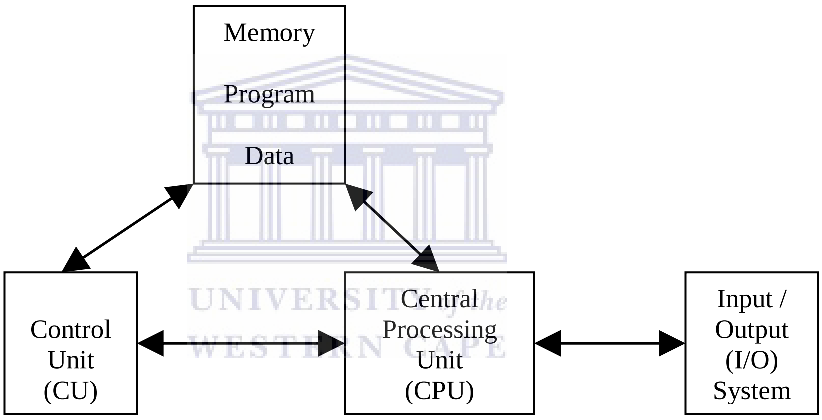

Figure 1: The von Neumann architecture

The computer consists of four basic components: a Memory, a Control Unit (CU), a Central Processing Unit (CPU) and an Input/Output (IO) system.

The CPU consists of an arithmetic logic unit (ALU), which carries out all logical and arithmetic computations, and a set of registers for high-speed storage of temporary results. This part of the computer is sometimes referred to as the *register arithmetic logic unit*.

The memory is used to store the program and all the data that may be required for the execution of the program, or generated by it. Memory can be thought of as a set of *n* numbered registers indexed from *0* to *n-1*. Each index is referred to as an address. Usually the program to be executed and the data are stored in different areas of memory. The program is loaded in contiguous memory registers.

The CU coordinates the CPU and the data flow to and from memory. To give an illustration of the CU's typical function: in running a program the CU indexes memory and fetches the next instruction to be executed. It then identifies what the instruction must do and sets up the CPU to perform this function. For example, an instruction indicates that the number at an address indexed by address `30AAH` must be added to the number in a register within the CPU. The CU would set up the CPU's ALU so that the sources of the two arguments needed for addition are the CPU register and the output of memory. Then the CU sends the memory the address of the required byte at (`30AAH`) and waits for it to appear at the memory's output. The CU then lets the ALU perform the addition on the two arguments and places the result in the desired output location.

The IO is the section of the architecture which allows user interaction with the computer. The user can enter data or operations and receive the results of computations through devices such as a keyboard, monitor and local disk. Since the computer must know which devices it is to get information from and send information to, each device must have an associated address, just as each register in memory must have a separate address. There is, usually, no reason why the IO must be thought of as separate from memory, because it looks like memory, to the processor, which can be selectively read from and written to. Conceptually this is referred to as memory mapped IO. The von Neumann architecture modified to the memory-mapped version is shown in Figure 2.

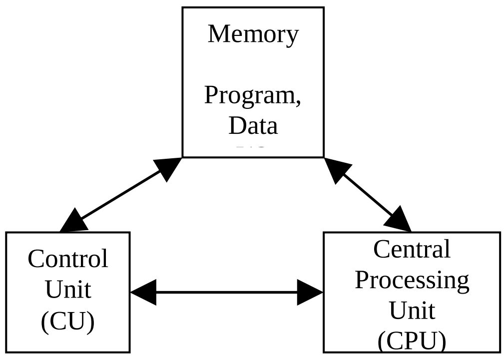

Figure 2: Memory mapped I/O version of the von Neumann architecture

2.2 8086 Microprocessor

The 8086 microprocessor is modelled on the von Neumann architecture. Far from the commonplace 64-bit 4GHz+ microprocessors of today, the 8086 is a simple and basic implementation of the von Neumann architecture. It presents all the basic concepts of microprocessor architecture without the technical overhead (such as read-ahead caches) associated with today's processors. As such, it provides an ideal device for introductory level teaching courses because students can see the basic ideas actually operating in a hardware environment.

The 8086 microprocessor can be understood easily by breaking it down into its components (Abel, 2001):

1. High and Low Accumulators (AX)

These are actually two separate registers used to store or manipulate data, either one 8-bit (one-byte), or can be combined to make one 16-bit register.

2. Index Register (IR)

This is a 16-bit register that holds a memory address when using indexed addressing modes. The register can be either loaded, its contents manipulated, or it may be stored using the appropriate instructions.

3. Program Counter (PC)

The program counter is a 16-bit register that contains the address of the next byte of the instruction to be fetched from memory. When the current value of the program counter is placed on the address bus the program counter is incremented to point at the next instruction.

4. Stack Pointer (SP)

The stack pointer is a 16-bit register that holds the starting address of sequential memory locations in the random access memory (RAM) where the contents of the microprocessor's registers may be stored or retrieved. This area of RAM is referred to as the 'stack'. After the content of any register is stored on the stack, the SP is decremented. When the stack is unloaded, the last register to go onto the stack will be the first to leave (last in, first out). This function is commonly used for passing values to subroutines, and receiving return values.

5. Code Condition Register (CCR)

This is an 8-bit register in which individual bits are set to 1 or reset to 0 as the result of executing an instruction, as explained below.

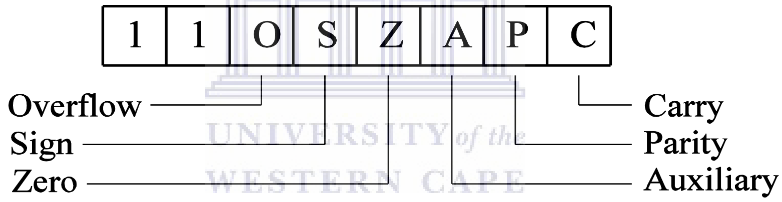

Figure 3: Code condition register layout

Bit 0 – Carry Flag (CF)

This flag is set to 1 when there is an unsigned overflow. For example, adding 1 to the byte valued at 255 yields 256 which overflows because it requires 9 bits. The result of the addition is a zero byte with the carry bit set to 1. When there is no overflow this flag is set to 0.

Bit 1 – Parity Flag (PF)

This flag is set to 1 when there is an even number of one bits in the result, and to 0 when there is an odd number of one bits.

Bit 2 - Auxiliary Flag (AF)

This flag is set to 1 when there is an unsigned overflow for the low nibble (4 bits).

Bit 3 – Zero Flag (ZF)

This flag is set to 1 when the result is zero. For a non-zero result this flag is set to 0.

Bit 4 – Sign Flag (SF)

This flag is set to 1 when the result is negative. When the result is positive it is set to 0. This flag takes the value of the most significant bit.

Bit 5 – Overflow Flag (OF)

This flag is set to 1 when there is a signed overflow. For example, when adding the bytes 100 and 50 the result is not in range (-128, 127).

The 8086 addresses the system devices, RAM, read-only memory (ROM), and I/O via its 16-bit address bus, and is therefore capable of addressing 2^{16} unique memory locations in the range 0000H to FFFFH, i.e. from 0 to 65535. The data bus in the 8086 is 8 bits wide.

2.3 Instruction Set

The instruction set of a microprocessor is all the commands that the microprocessor can execute. There are many instructions in the 8086 instruction set from which all the software programs are produced. Simple operations such as addition, subtraction, and comparisons can be implemented using a single instruction. More complex operations such as exponential mathematics can be calculated using standard algorithms consisting of many simple operations.

2.4 Visual Basic.NET Programming Language

Visual Basic.NET is a simple object-oriented language that is distributed, interpreted, robust, secure, architecturally neutral, portable, high-performance, multithreaded, and dynamic (Balena, 2002). It is easy to implement a GUI form using Visual Basic.NET. A Visual Basic.NET program runs an application as a form. The application is a compiled, stand-alone program that will run on a local machine, or a machine accessing a server on the opposite side of the world. Furthermore, either of the formats can be executed on any operating system with Visual Basic.NET support.

2.4.1 Visual Basic.net Features

The design of Visual Basic.NET is highly structured, engineered to meet a firmly fixed set of goals which amalgamates the best features of existing languages such as Lisp, Smalltalk, Pascal, Objective-C, Self, and Beta, as well as adding several features unique to Visual Basic.NET. The design specification for the final product suggested that:

The language should be familiar: The program flow control structures and data types look like some of those provided in C, and its object-oriented facilities resemble those found in C++. This helps shorten the learning curve, allowing more people access to the language, and for those people to learn it quickly.

The language should be object-oriented: (No code is accessible from outside an object.) A language is said to be object-oriented if it offers facilities to define and manipulate objects, which are self-contained entities having a state, and to which messages can be sent. There are two major advantages to an object-oriented programming language. First, by adhering to a small set of programming principles it is possible to write systems that are relatively easy to modify. This is important because it allows for changes in requirements. Secondly, an object-oriented architecture encourages a high level of code reuse. One object can be reused many times within a program where it is possible to have many entities of the same kind. The other kind of code reuse is where a new object can 'inherit' the properties of an existing object and add any extra properties required to make the object functional, without rewriting the code common to both objects.

The language should have high performance: Visual Basic.NET supports threads, which are multiple simultaneous executions of code that provides a high level implementation of concurrent processing. This means that, for example, when a time consuming computation is executing in one thread the user can still interact with the program in a different thread. User interaction does not have to stop while the user waits for the computation to finish.

The language should be portable: One major aim of Visual Basic.NET language developers was to create an executable format that when compiled could run on any supporting platform, regardless of the platform used to develop the software. This has been achieved using a compiler that generates "byte-code" rather than native machine code. This architecture neutral object file format can be executed by any machine.

2.4.2 The Abstract Windowing Tool (AWT)

The AWT is a GUI toolkit designed to work across multiple platforms. As such, it doesn't include all the features of any particular platform, but it has a common set of features that can be supported on most platforms. These features can be broken down into groups of related classes:

Components: The component is the parent of most of the AWT classes and it provides the ability to represent something that paints itself on the screen, has a size and position, and can receive input events.

Containers: Two types of containers are provided: `window` and `panel`, both of which are subclasses of the Visual Basic.NET `container` class. Containers, as the name suggests, are used to hold other components that are placed inside them. A useful helper class is provided for a container called the layout manager, which lays out the contents of the container in a predefined alignment and spacing, for example in a grid.

Control Elements: This group of elements provides the means by which the user will interact with the program. Buttons, menus, text boxes, and scrollbars are among the controls that provide the building blocks required to generate a GUI.

2.5 Summary

This chapter has presented details about the 8086 microprocessor and its instruction set, and so was a justification for the choice of Visual Basic.NET as the development language for the project. In the next chapter the field of computer-assisted learning will be reviewed, as a precursor to the design specification for the visualization tool.

Chapter 3

COMPUTER-ASSISTED LEARNING: AN OVERVIEW

3. Introduction

This chapter is an overview of the literature covering computer-assisted learning (CAL) and its application in computer science education in general and the teaching of computer architecture in particular. The chapter also contains a review of existing CAL tools across different disciplines.

3.1 Computer-Assisted Learning (CAL)

The applications of computers are continually growing, and application expectations of the technology are growing at the same rate. CAL's history began in the early 1960s, when the third generation of digital computers were built and more widely used. These systems were also cheaper and more reliable than earlier models. Digital computers thus became more common facilities in universities and research centres. Consequently, researchers started to find new fields of application for computers, and CAL was one of those. Certainly at the beginning, as with other technological products, CAL systems, which are a combination of computer hardware, added special purpose peripherals, and CAL software had only scientific and academic applications and were experimental. At that time, before any other specialists, psychologists used the computer as an ideal tool for conveying programmed instructions. These early applications were called computer-assisted instruction (CAI). CAL is a more recent development of computer applications in learning (Yushau, 2004).

Computing technology allows us to create simulated systems for real environments. In real-world applications, some of the most common applications of such systems are flying or sailing training systems (Everingham, 1998). The UNESCO International Centre for Engineering Education (UICEE, 2004) presented an article about a group of universities that had conducted research in engineering education. The UICEE aims to provide a focus for academic and research activities in the field of teaching methodologies in developing countries and, in particular, in the development of teaching methodologies for education in the establishment of small and medium-sized enterprises that are so vital to economies in developing countries. The purpose of the reported research was to provide assistance to those willing to conduct research in engineering and technology education. Particular emphasis was placed on research into human aspects of engineering, engineering pedagogy, training methodologies in engineering, educational technology, multimedia and computer-aided engineering education.

Cerrato (2002) investigated CAL simulators in the context of learning Swedish both as a second language and learning Swedish from a second language perspective in secondary school education. The use of language tools for writers in the context of learning Swedish as a second language was a one-year project funded by the Swedish Research Council. It aims at formulating a research agenda for investigating the use of computer-based language tools for writers in the context of learning a second language, from a pedagogical and human-computer interaction perspective (Cerratto, 2002).

Navarro (2004) designed educational software called “Let’s Play With” to teach basic concepts involving shapes and body postures to preschool students. The software structure follows a behavioural design and uses a stimulus control procedure. The study was carried out with 64 preschool students in the Cadiz school district in Spain. Statistically significant differences were found between the experimental group and control group. The study showed that CAL is an efficient learning/teaching procedure (Navarro, 2004).

Imhanlahimi (2008) assessed the effectiveness of a CAL strategy and the expository or traditional method of teaching biology using a high school in Uromi, Nigeria for the study. The study had an experimental design: randomized two groups with a pre-test and a post-test control group involved sixty senior secondary class one (SSC 1) students of the high school. The instrument for the study consisted of six essay questions based on three selected topics from the SSC 1 curriculum. Their results show that the computer-assisted learning strategy method of instruction was superior to the expository method in teaching biology (Imhanlahimi, 2008).

Medical Sciences use visualization emulating tools to teach the identification of diseases. The AIDS pandemic is one of the toughest challenges facing human society, and AIDS Information Modification and Simulation (AIMS) is a tool created at Agder University in Norway. AIMS focuses on those features that make the AIDS pandemic such a tough challenge (Gonzalez, 1995).

3.2 Computer Architecture Simulators

One of the major challenges in the teaching of computer architecture courses is to demonstrate how things tie together in various layers of computers to make them work the way they do. The difficulty of teaching computer architecture courses is widely acknowledged (e.g. Yurcik, 2002; Fienup, 2002), and a number of recent articles on teaching computer design and architecture suggest that hands-on simulation and learning tools are essential for effective instruction of the subject material (e.g. Herath, 2002; Yushau, 2004).

Existing processor simulation tools help explain how various parts of computers function and illustrate the operations of computers at various levels of specificity (Dickerson, 2000). Nonetheless, these simulation tools are more suitable for generating statistical information and validating architectural innovations than for classroom instruction. An alternative approach that is also used in computer engineering courses is to assign design projects with the desired effect of involving students in the process of developing a contemporary processor in a simulated environment if not one in silicon (Phillips, 2007) and (Yurcik, 2001).

There are many different simulators designed for educational purposes and for research purposes (Yehezkel, 2003; Wainer, 2001). Some simulators are targeted at students who have no background in computer architecture and need a simple introduction. The simulators try to show the basic ideas of computer organization with relatively few details and complexity. All these simulation environments allow us to execute code, either step-by-step or continuously. All of them present the architecture's state (registers and memory) in a graphical form. Finally, some of them include visualization of the core components with their interconnections and interactions.

3.2.1 CAL at Kabul University

Students at the University of Kabul (KU) find it difficult to comprehend computer architecture concepts, especially the execution of microprocessor instruction sets. In this regard, a visual simulator tool would ease the problem of comprehension. In order to understand the framework of computer-aided learning and the design of the visualization simulation tool, a background to the teaching of computer architecture at KU will next be described.

There are three courses concerned with teaching computer architecture. The first is *Computer Fundamentals* where first year students learn basic concepts of computer architecture with a simple CPU. The second is *Computer Architecture I*, where in the first semester of the second year the students learn about hardware, numerical systems, logic gates, and more advanced aspects of computer architecture. In *Computer Architecture II*, which is offered during the second semester of the second year, the students learn the operation of microprocessors.

No simulation tool is available for the computer architecture courses, but a simulation tool has been in use for computer networks courses. PacketTracer is a simulation tool that enables students to work with Cisco routers, switches, and cabling. The experience has shown that when students used PacketTracer for their assignments they learned much more and, said they enjoyed learning. Our observations and student feedback over many years have lead us to believe that the learning benefits of PacketTracer in the Computer Networks course has exceeded expectations. It provides exciting new opportunities for creativity and interactivity inside the classroom and outside the classroom. PacketTracer helps students and teachers collaborate, solve problems, and learn concepts in an engaging and dynamic cooperative environment. We had similar expectations for a visualization simulator of a microprocessor when this study was commenced.

Summary

This chapter has reviewed the use of CAL across different disciplines. In particular, the use of CAL in the teaching of computer architecture courses was highlighted. CAL is currently being used at KU for the teaching of computer networks courses with great success and was the inspiration and motivation behind this research, to determine whether CAL could also be successfully applied to computer architecture courses.

Chapter 4

SYSTEM DESIGN AND DEVELOPMENT

4. Introduction

This chapter specifies the design and development of a computer architecture and assembly language simulator. The objectives of the Computer Architecture course at KU are discussed, from which the requirements specification for a simulator tool to aid the teaching of a basic computer architecture course are derived. This chapter then describes the specific functions of each internal component of the selected micro-processor, and also shows its implementation. The interaction of the different components is then demonstrated through the execution of the instructions of an assembly language program. The overall design of the simulator is also given using pseudo-code.

4.1 Learning Objectives for Computer Architecture Courses

There are many computer science courses for the study of computer architecture at undergraduate level, including computer organization, micro-processors, and computer architecture (Yehezkel, 2002). In the bachelor's curriculum at KU there are two computer architecture courses. Computer Architecture I is taught during the first semester of the second year. There are four lecture hours per week. This course is theoretically oriented. Students have substantial readings on the topics of computer organization and computer hardware. Computer Architecture II, for which the tool developed in our research was proposed, is taught during the second semester of the second year. There are again four hours of lectures per week. This course is both theoretically and practically oriented.

Computer Architecture II at KU covers the following topics:

- Introduction to the 8086 CPU internal architecture

        - Internal Storage in the CPU–Stack versus Registers
        - Data path and control
- Memory
    - How data is stored in computer memory
- Microprocessor operation Execution
- Assembly language and assembler
    - Language and the machine
    - The instruction set architecture
    - Instruction types
        - Data movement operations
        - Arithmetic operations
        - Control operations
        - Logical operations

The main goal of teaching computer architecture courses is to provide students with a complete overview of microprocessor operation and assembly programming language (Abel, 2001; Alpert, 1993). The teaching of computer architecture at KU is based on the Intel 8086 microprocessor. In these courses, the student learns to understand the internal parts of the 8086 microprocessor and the process of execution of its instructions. Students should be able to comprehend the processor status under different programming conditions. This is exemplified by understanding the status and control within an instruction cycle (Stallings, 2000). Execution of an instruction on a microprocessor follows a set of steps known as the instruction or machine cycle, which comprises the following sequence:

1. Fetch the instruction from the code area,
2. Read its operands form the register(s), memory, or from the code area,
3. Execute the operation on the operand(s),
4. Access the memory, if needed, and
5. Write back the result into the register(s) or into the memory.

The difficulty faced by most of the students is how to imagine the instruction groups involved in microprocessor operations. The aim of this study was to design a system to aid the understanding of these operations. The objective is to help students understand more thoroughly how computers work, as well as helping the instructor to demonstrate his ideas easily.

4.2 Requirements Specification

The aim of the tool was to help students to understand the following topics that are taught in computer architecture courses:

- *Registers*
- *8086 Instruction set*
- *Memory organization and structure*
- *Assembly programming language*
- *Register-stack, register-memory, register-register and stack-memory relationships.*

Since this tool would be used for the specific purpose of education, the essential requirements of the tool will first be described, before discussing its design and development. Our main goal was to develop a computer-aided learning tool which had to cover the mentioned syllabus taught at KU in the computer architecture course. It was therefore necessary to specify the following topics in order to develop this tool.

4.2.1 Register Set

Registers are a special, high-speed storage area within the CPU. All data must be represented in a register before it can be processed. For example, if two numbers are to be multiplied, both numbers must be in registers, and the result is also placed in a register. The number of registers that a CPU has and the size (number of bits) of each determine the power and speed of a CPU. For example, a 16-bit CPU is one in which each register is 16 bits wide. This means that, each CPU instruction can manipulate 16 bits of data. Usually, the movement of data in and out of registers is transparent to assembly language programmers. Assembly language programs are adept at manipulating registers. In high-level languages, the compiler is responsible for translating high-level operations into low-level operations that access registers. The first step that could be taken to achieve a level of simplicity was to specify the set of registers available to the user. The simplest analytical method to achieve this was to start with no registers, and add registers until a useful set had been accumulated. The following figure presents general purpose registers, segment registers and flags (Yu, 1992).

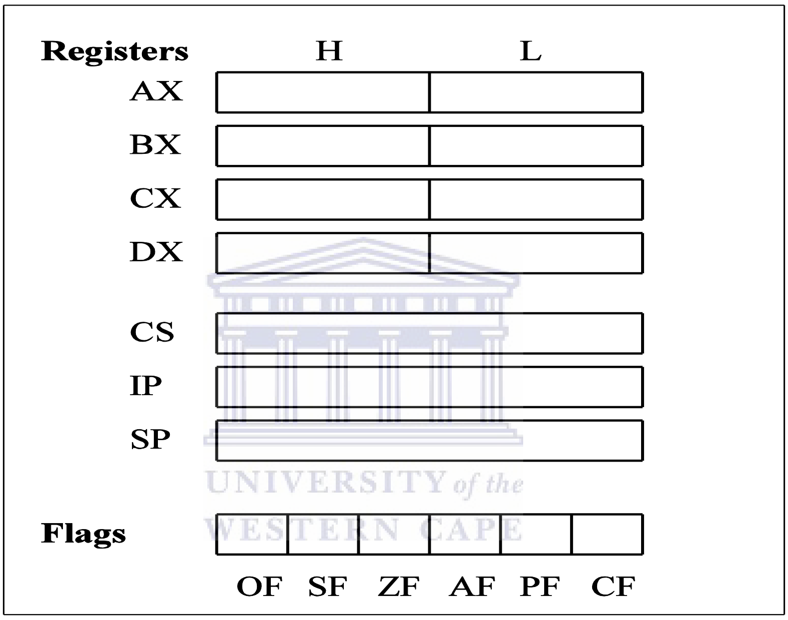

Figure 4: Register set

An accumulator register (AX) is a register that can be used for arithmetic, logical, shift, rotate, and other similar operations. The first computers typically had only one accumulator. Often special purpose registers are used to hold the source data for an accumulator. Accumulators were replaced with data registers and general purpose registers. Accumulators reappeared in the first microprocessors. An AX is divided into accumulators. A single accumulator is required to display the temporary internal storage of data, so Accumulator L must appear in the design. Accumulator H, however, is not necessary until the value exceeds 255, and it is possible to envisage a teaching tool that provides sufficient example programs by using only one accumulator. In order to demonstrate the fetch/execute cycle it is necessary to show the program counter as it increments through memory locations, pointing to the operation code (opcode), and the operand(s). Each instruction as it is executed needs to be stored internally in the instruction register, so the latter must also be used.

The remaining registers, the stack pointer and the index register are all also necessary for simulation of other types of instructions. The stack pointer is mostly used in programming to pass values to and from subroutines. The index register would be highly desirable to implement, as it would greatly increase the teaching of more complex instructions. However, it was not included in the initial specification, although all design will be completed with consideration given to the addition of this register.

4.2.2 Instruction Set

The next stage in defining a target instruction set was to look at the mnemonic level. At the mnemonic level, the implemented simulator instructions was to represent 8086 assembly code mnemonics. Only a sufficient collection/number of instructions were implemented to permit realistic programming; since for the intended purposes it was not necessary to implement the full 8086 instruction set. Only declarative, data movement, control, arithmetic, and logical instructions were implemented. In addition, the simulated instructions apply only to the lower 8 bits of the 8086 CPU. For example, with typical 8086 machine code, the mnemonic MOV AX, 15 is encoded in two bytes. MOV AX is encoded into one byte and the 15 goes into another. The proposed simulator requires three bytes. MOV is encoded as a byte-size opcode. AX is encoded as another byte, and 15 goes into the third byte. This means that different instructions will be used for many of the usual 8086 instructions. Although this is not very efficient it is very simple.

The instruction set is shown in Table 1. Group 1 instructions are the declarative instructions, for declaring bytes (DB), words (DW) and procedures (PROC). These are essential and therefore part of the optimal set. The next group are the data movement instructions, which are also necessary for moving data in or out of registers and memory. The third group set are the branching instructions that implement program flow control. Of these it is sufficient to implement only the jump (JMP), jump equal (JE) and the loop (LOOP) instructions. The next group are the arithmetic instructions, of which only the essential ADD, SUB, MUL and DIV were implemented. The final group is used for bit-wise manipulation of data that is useful to experienced programmers. These were consequently not implemented.

Table 1: The full proposed instruction set (Yu & Marut, 1992).

No.	Instructions	Declarative	Data Movement	Control	Arithmetic	Logical
1	DB	X				
2	DW	X				
3	PROC	X				
4	LEA		X			
5	MOV		X			
6	PUSH		X			
7	POP		X			
8	JB			X		
9	JE			X		
10	JL			X		
11	JG			X		
12	JMP			X		
13	JNC			X		
14	JZ			X		
15	LODSB			X		
16	LOOP			X		
17	CALL			X		
18	RET			X		
19	END			X		
20	ENDP			X		
21	INC				X	
22	DEC				X	
23	CMP				X	
24	ADD				X	
25	MUL				X	
26	SUB				X	
27	DIV				X	

28	MOD				X	
29	SHR					X
30	SHL					X
31	OR					X
32	XOR					X

4.2.2.1 Active Components for the Instruction Set

Table 2 summarizes the components that are active for specific instructions (Yu & Marut, 1992).

Table 2: Active components for subset instruction

Instructions	Components	Examples
ADD	register, memory; register, register	Algorithm: operand1 = operand1 + operand2 Example: ADD AL, 2;

CALL	procedure name
	label
		Example:
		MOV AX, 9
		CALL PROC1
		MOV BX, 8
		ADD AX, BX
		PROC1 PROC
		MOV DX, 7
		MOV CX, 2
		END PROC1
		ENDP

CMP	register, register	Compare.
		Algorithm:
		operand1 - operand2
		result is not stored anywhere, flags are set (SF, ZF) according to result.
		Example:
		MOV AL, 5
		MOV BL, 5
		CMP AL, BL ; AL = 5, ZF = 1 (so equal!)

DB	No component	Define a byte length variable VAR1 DB 7h

DEC	register	Decrement.
		Algorithm:
		operand = operand - 1
		Example 1:
		MOV AX, 4
		DEC AX ;AX=AX-1
		Example 2:
		MOV AL, 255 ; AL = 0FFh (255 or -1)
		DEC AL ; AL = 0FEh (254 or -2)

DIV	register
	
	memory
		Algorithm:
		
		when operand is a **byte**:
		AL = AX / operand
		AH = remainder (modulus)
		when operand is a **word**:
		AX = (DX AX) / operand
		DX = remainder (modulus)
		Example:
		MOV AX, 203 ; AX = 00CBh
		MOV BL, 4
		DIV BL ; AL = 50 (32h), AH = 3

DW	No component	Define one word length variable VAR1 DB 7h
END	No Component	End of procedure
ENDP	No Component	End of program
INC	register	Increment. Algorithm: operand = operand + 1 Example: MOV AL, 4 INC AL ; AL = 5

JB	Label	Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 then jump
		Example:
		
		MOV AL, 1
		MOV BL, 4
		CMP AL, BL
		JB label1
		VAR1 DW 12h
		JMP END
		label1:
		VAR2 DW 10h
		END

JE	Label	Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned.
		
		Algorithm:
		
		if ZF = 1 then jump
		Example:
		MOV AL, 5
		CMP AL, 5
		JE label1
		VAR1 DB 8
		JMP END
		label1:
		VAR2 DW 45

		END

JG	Label	Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed.
		Algorithm:
		
		if (ZF = 0) and (SF = OF) then jump
		Example:
		MOV AL, 5
		CMP AL, -5
		JG label1
		MOV DX, 9
		JMP END
		label1:
		MOV DX, 7
		END

JL	Label	Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed.
		Algorithm:
		
		if SF <> OF then jump
		Example:
		MOV AL, -2
		CMP AL, 5
		JL label1
		VAR1 DW 99
		JMP END
		label1:
		VAR2 DW 100
		END

JMP	Label	Unconditional Jump. Transfers control to another part of the program.
		
		Algorithm:
		
		always jump
		Example:
		MOV AL, 5
		JMP label1 ; jump over 2 lines!
		MOV DX, 8
		MOV AL, 0
		label1:
		MOV DX, 9
		END

JNC	Label	Short Jump if Carry flag is set to 0.
		
		Algorithm:
		
		if CF = 0 then jump
		Example:
		
		MOV AL, 2
		ADD AL, 3
		JNC label1
		MOV DX, 7
		JMP END
		label1:
		MOV DX, 9
		END

JZ	Label	Short Jump if Zero (equal). Set by CMP, SUB, ADD, OR, XOR instructions.
		
		Algorithm:
		
		if ZF = 1 then jump
		Example:
		
		MOV AL, 5
		CMP AL, 5
		JZ label1
		MOV DX, 8
		JMP END
		label1:
		MOV DX, 4
		END

LEA	Register, memory	Load Effective Address. Algorithm: • register = address of memory (offset) Example: Note: The integrated 8086 assembler automatically replaces **LEA** with a more efficient **MOV** where possible. example: LEA AX, m ; AX = offset of m RET m dw 1234h END
LODSB	No component	

LOOP	Label	Decrease CX, jump to label if CX not zero.
		
		Algorithm:
		
		• CX = CX - 1
		• if CX <> 0 then
		  ○ jump
		
		else
		
		  ○ no jump, continue
		
		Example:
		
		MOV CX, 5
		label1:
		VAR1 DW 99
		LOOP label1

MOD	No component	this instruction is executed with DIV

MOV	register, memory
	memory, register
	register, register
		• Copy operand2 to operand1.
		Algorithm:
		operand1 = operand2
		Example:
		MOV AX, 0B800h ; set AX = B800h (VGA memory).
		MOV DS, AX ; copy value of AX to DS.

MUL	register
	memory
		Algorithm:
		when operand is a **byte**:
		AX = AL * operand.
		when operand is a **word**:
		(DX AX) = AX * operand.
		Example:
		MOV AL, 200 ; AL = 0C8h
		MOV BL, 4
		MUL BL ; AX = 0320h (800)

OR	register, register	Logical OR between all bits of two operands. Result is stored in first operand.
		These rules apply:
		1 OR 1 = 1
		1 OR 0 = 1
		0 OR 1 = 1
		0 OR 0 = 0
		Example:
		MOV AL, 8 ; AL = 00001000b
		MOV BL, 2 ; BL = 00000010
		OR AL, BL ; AL = 00001010b ('a')

POP	register memory	Get 16 bit value from the stack. Algorithm: • SP = SP + 2 Example: MOV AX, 1234h PUSH AX POP DX ; DX = 1234h
PROC	No component	Specify the name of procedure
PUSH	register memory	Store 16 bit value in the stack. Algorithm: • SP = SP - 2 Example: MOV AX, 1234h PUSH AX POP DX ; DX = 1234h

RET	No component	Return from near procedure.
		
		Example:
		
		CALL p1
		
		ADD AX, 1
		
		RET ; return to OS.
		
		p1 PROC ; procedure declaration.
		MOV AX, 1234h
		RET ; return to caller.
		p1 ENDP

SHL	register, CL	Shift operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 192 ; AL = 11000000b SHL AL ; AL = 10000000b = 128, CF=1.
SHR	register, CL	Shift operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • Zero bit is inserted to the left-most position. Example: MOV AL, 7 ; AL = 00000111b SHR AL ; AL = 00000011b = 6, CF=1.
SUB	register, memory	Subtract. Algorithm: operand1 = operand1 - operand2

	register, register	Example:
		MOV AL, 5
		SUB AL, 1 ; AL = 4

XOR	register, register	Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand.
		
		These rules apply:
		
		1 XOR 1 = 0
		1 XOR 0 = 1
		0 XOR 1 = 1
		0 XOR 0 = 0
		
		Example:
		MOV AL, 7 ; AL = 00000111b
		MOV BL, 12 ; BL = 00001100b
		XOR AL, BL ; AL = 00001011b

4.2.3 Instruction Execution

The tool executes the instructions like a real microprocessor. An instruction is separated into two parts: an opcode and operands. The opcode specifies the type of instruction operation, and the operands are often given as a memory address to the data to be operated on. Our tool goes through the following steps to execute an instruction, in what is called the fetch-execute cycle:

Fetch

1. It fetches an instruction from the code area.
2. It checks the instruction correctness and type and then determines the operation.
3. It checks the operands and fetches the data from memory, from registers or from the stack if necessary.

Execute

1. It performs the operation on the data.
2. It stores the result in memory registers or the stack if needed.

To see what this entails, one can follow the execution of a typical instruction in our tool, for example an instruction that adds the contents of register AX to the contents of the memory word at address 00. The tool actually adds the two numbers and stores the result to memory cell 00. The fetch-execute processing is illustrated in Figure 5.

```
executeVisualizeClass
    Declare Context run_context
    Declare variable Integer temp_ip
    Declare Form1 form

  NewSub(Form1 frm , Context local_s)
        IP (local_s) = 0
        run_context = local_s
        form = frm
         execute_inst()
        show_after(0)
```

Figure 5: Execute function pseudo-code

4.3. Assembler Design

In order to design a satisfactory assembler simulator for KU, it was necessary to think of all the possible instructions, as specified in Table 1. The assembler simulator had to demonstrate all the instructions and the components of the 8086 active during their execution. The following descriptions clarify the 8086 assembly instructions (Stanley, 2005):

Line: is the assembly code written by the user in the code area. We need to store all lines somewhere.

Word: We need to extract words from the line, which could be an instruction, variable name, label, procedure name, or comment by tokenizing the line into blank separated words.

Instruction: The extracted word could be an instruction; we need to compare the extracted word in the instruction table. If the extracted word matches an opcode in the list of instructions then it is an instruction.

Label: If the first extracted word ends with ":" our tool knows that this word is a label. Labels are used for jump instructions where the instruction immediately after the label is referred to by the label, e.g.

```
MAIN:
```

Variable name: If the first extracted word is not an instruction and is not a label it could be a variable name. A variable name is only identifiable when the word following it has been processed as "DB", or "DW", then the first extracted word is a variable name.

Comment: Characters in lines that follow ";" are comments and are completely ignored by the assembler, e.g.


```
;loading effective address
```

The following examples make the above definitions clearer:

Program Line: A line that contains instructions for assembly into executable code, e.g.

```
Call main:
```

Mixed Line: A line that may contain any combination of line types, e.g.

```
main: lea num1  ;loading effective address
```

As each line is processed, it is separated into its constituent fields. Directive and comment lines are simply ignored, meaning that the assembler has to process only executable lines. Executable lines can be classified using a simple algorithm as follows:

- All text to the left of a colon will be regarded as a label.
- All text to the right of a semi-colon will be regarded as a comment.
- Any remaining text has to be an instruction, variable name, or procedure name.

An instruction can be divided into opcode and operand by checking if there is a space in the text. After this final division, the line has been parsed into four discrete sections: label, opcode, operand, and comment.

4.3.1. Instruction Checking

In the process of assembly it is necessary to check that the mnemonic entered by the user actually exists. The simplest method to do this is to have a look-up table containing all the instructions and their types in the preview tables. As each instruction has its type determined, a simple check through this table determines its existence. Note that it is necessary to check both type and instruction name against the table, as instructions may not be available for certain types.

4.3.2. Important Instruction Exercises

All examples for each instruction mentioned in Table 2 can be executed. Some examples follow.

4.3.2.1. Declarative Instruction Example

Variables play the same role in assembly language, in high-level languages and in our developed tool. In our tool we have used DB and DW to define byte variables and word variables.

The directive that defines a byte variable takes the following form:

```
Variable name  DB   initial value
```

4.3.2.2. Data Movement Instruction Examples

The MOV, LEA, PUSH, and POP are used to transfer data between registers, between a register and a memory location, or to move a number directly into a register, or from register to stack and vice versa (Yu & Marut, 1992).

The syntax for each instruction is implemented as follows:

MOV destination, source
LEA destination, source
PUSH source, to stack
POP from stack, to destination

e.g.

MOV AL, BL
AL, gets what was previously in BL. BL is unchanged.

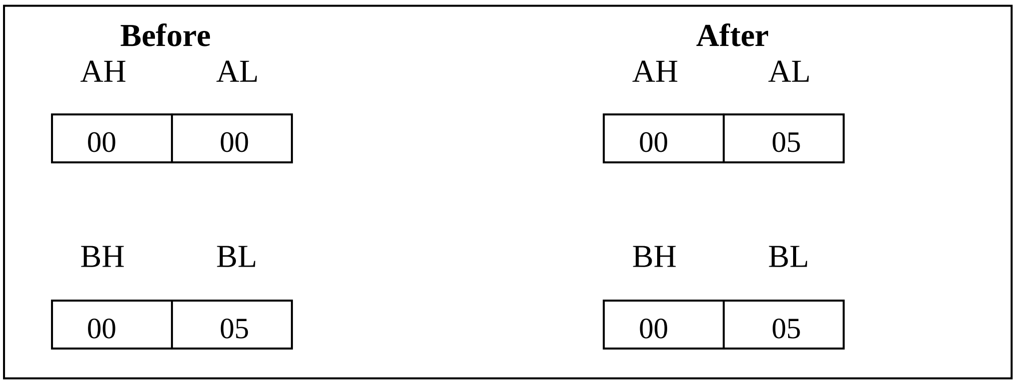

Figure 6: Example of the register

LEA AX, m : AX gets the offset of m from memory.

PUSH m : Value of m goes to top location of the stack.

POP AX :value from the top location of the stack comes to AX.

4.3.2.3. Flow Control Instructions

Any assembly language program needs to make decisions and repeat sections of code. This can be accomplished with the jump/loop and test instructions. We have implemented simple jump and conditional jump instructions such as JB, JL, JNZ, JG and JL. Conditional jump instructions are often preceded by the CMP (compare) instruction. When a CMP instruction is executed the status register (flags) gets set. Conditional jump instructions will be executed according to the flags (Yu & Marut, 1992). For example, suppose a program contains these lines:


```
CMP AX, BX
JG LABEL1
```

Where `AX = 7FFFH`, and `BX = 0001h`. The result of `CMP AX, BX` is

`7FFFH - 0001H = 7FFEH.`

Jump conditional is satisfied, because ZF = SF = 0, i.e. the ZF (zero flag) is set to 1 when two compared values are equal. Since these two values are not equal, it is zero.

The SF (sign flag) is set to 1 when the first value of the two compared values is less than the second value. When the first value is not smaller, SF becomes zero.

The following pseudo-code algorithm is an example of a conditional jump:

```
IF AX < 0 THEN
INCREASE AX by 1
END IF
```

It can be coded as follows:

```
;if  AX  < 0
        CMP AX, 0 ;    AX  < 0
        JG END_IF
; then
        INC AX  ; yes, increase AX
END_IF:
```

The condition AX < 0 is expressed by `CMP AX, 0`. If AX is not less than 0, there is nothing to do, so we use a JG (jump if it is greater) to jump around the INC. It means if condition AX < 0 is true, the program goes on to execute INC AX.

4.4. The User Interface

Now it is necessary to decide what the best options are for displaying the information to the user. A good starting point for assessing elements that are needed in the user interface is to visualize those components that are specified in Table 2.

The user must type in the assembly code to be executed, so an input area for this will be required. During the assembly process, error information will be generated, and for this to be useful to the user in debugging their code it will need to be displayed. Finally, a control area will need to be supplied to allow the user to control the simulation and set other user-definable aspects of the program.

In the next section the suggested and developed screen layout is given.

4.4.1. Screen Layout

The screen layout has been decided on in detail in previous sections. The computers in our laboratory operate at a screen resolution of 800x600 pixels, which it is reasonable to consider as the minimum for any modern computer. The completed software had to run in a form window maximized inside this space, allowing about 1024x768 pixels for the user interface. Figure 7 shows our initial proposed screen layout.

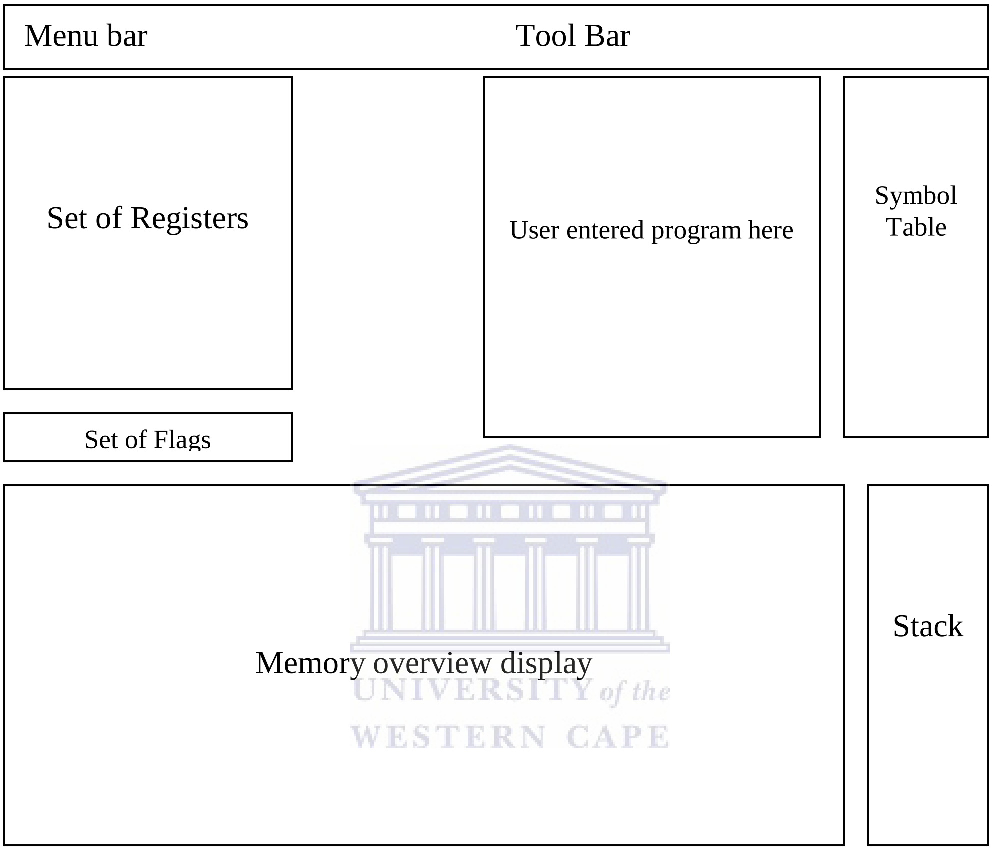

Figure 7: Proposed layout for the tool

Within the available space each component should have given a size relative to its importance. Each component area is described and specified in the following section.

4.4.2. Memory View

As the target user is new to the idea of microprocessor-based systems, it is necessary to try to relate the concept of a list of data in memory locations to a simple list of instructions. In many textbooks on microprocessor operation, memory is displayed as a row of blocks, all numbered and containing two representations of their contents (Abel & Peter, 2001; Yu & Marut, 1992; and Sayed Razi, 2004[1]). The user's program variables and values are shown in text as the user would have typed it in before assembly. Clearly, this does not actually exist in any memory location, but it helps the student to relate the program listing to what actually appears in memory.

Next we specify some of the cells to store the data in. There is no simple way to display more than 64 cells in a normal window. If we scroll we can have as many cells as necessary. We also have to specify the length of each cell and its address. It is best to present the address of cells in hexadecimal because four digits in hexadecimals can present `65536`, which is `FFFFH`. Therefore the four-digit address with two digits for real data was used in this tool. Furthermore, if we wanted to have 8KB of memory, we needed to specify the start address and end address of this number of cells.

`8KB => 8 * 1024 = 8192D = 2000H`. A cell's address ranges from `0000H` and ends `1FFFH`.

Using this format a student can easily see the correlation between the source code and what appears in memory. However, the drawback in using this method for memory display is that the GUI cannot display very many locations simultaneously.

[1] The book is written in Farsi and its date according to the Hire Shams calendar is 1383.

The memory cell length is one byte of data. Each cell has an individual address. Data occupies two digits, and addresses use four digits. Figure 8 shows how memory is displayed. The initial value for a cell is 00.

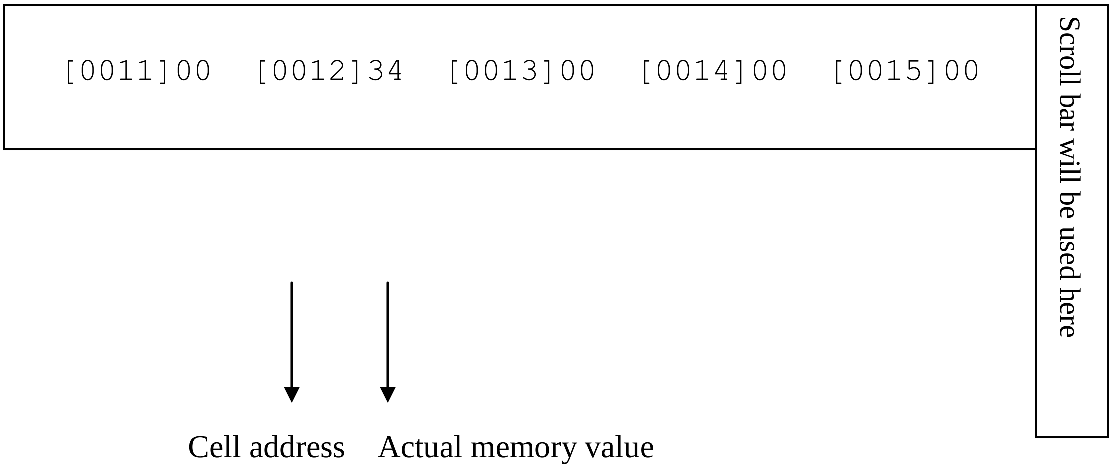

Figure 8: One-line memory view display

Figure 9 gives the class and functions implement the memory display.


```
memoryClass
    Shared variable Integer maxmemory = 8192 ;2000 hexadicimal
    Public variable location(maxmemory)  Byte
    Public Shared Char usedlist(maxmemory)
    Public Shared Integer current = -1
```

```
New Sub()
      Declare i As Integer
        For i = 0 To maxmemory
            usedlist(i) = "n"
        Next i
    End Sub
```

```
Integerset_memory_location(integer typ)
        Declare variable Integer temp
        Declare variable Integer i

        If current = maxmemory Then Show message ("Memory full. Quitting...")
        Else
            If typ = 1 Then 'character type or byte type
                current = current + 1
                usedlist(current) = "y"
                Return current
            ElseIf typ = 2 Then 'integer type or word type
                current = current + 1
                usedlist(current) = "y"
                temp = current
                current = current + 1
                usedlist(current) = "y"
                Return temp
```

```
            ElseIf typ = 3 Then 'Float type
                current = current + 1
                usedlist(current) = "y"
                temp = current
                For i = 1 To 3
                    current = current + i
                    usedlist(current) = "y"
                Next i
                Return temp
            ElseIf typ = 10 Then 'number of bytes to be allocated is specified in typ
                current = current + 1
                usedlist(current) = "y"
                temp = current
                For i = 1 To typ
                    current = current + i
                    usedlist(current) = "y"
                Next i
                Return temp
            End If
        End If
```

Integer read_memory(Integer typ, Integer address)

```
      If typ = 1 Then 'character
            Return location(address)
        ElseIf typ = 2 Then 'integer
            Return location(address + 1) * 256 + location(address)
        ElseIf typ = 3 Then 'float
            Return -9999
        End If
```

Integer write_memory(Integer typ, Integer address, Integer value)

```
        If typ = 1 Then 'character
            If value < 256 Then
                location(address) = value
            Else
                Show message ("Cannot Assign Data to Byte Variable")
            End If
        Else typ = 2 Then 'integer
            location(address + 1) = value \ 256
            location(address) = value Mod 256
        End If
```

Figure 9: Pseudo-code for the memory display

The code for the symbol table to display the variable names and their locations in memory follows in Figure 10.

SymtableClass

Structure item()

```
Declare variable String token
Declare variable Integer type
Declare variable Integer address
Declare variable Integer value
End Structure
```

```
Declare variable integer maxsize = 100
Public item symbol(maxsize)
Public variable Integer count
Public variable novalue = -9999
```

NewSub ()

```
        Declare variable integer i
        count = -1
        For i = 0 To 100
            token (symbol(i)) = ""
            type (symbol(i)) = -1
            address (symbol(i)) = -1
            value (symbol(i)) = novalue
        Next i
```

Integeraddtoken(String s1, integer typ, integer val)

```
      If (count < maxsize - 1) Then
            count = count + 1
            token (symbol(count)) = s1
            type (symbol(count)) = typ
            value (symbol(count)) = val 'here val is the type of data
            If (typ = 3) Then
                address (symbol(count)) = set_memory_location (memory(val))
            ElseIf typ = 5 Then
                address (symbol(count)) = val
                'at the moment label is not stored in memory, keep in symbol table
itself
            ElseIf typ = 7 Then
                address (symbol(count)) = set_memory_location (memory (10))
            End If
        Else
            Show meesage("Symbol Table Overflow: Too many tokens in the Program.
Quitting")
        End If
```

Integersearchtoken(string key)

```
      Declare i As Integer
        For i = 0 To count
            If token (symbol(i)) = key Then
                Return i
            End If
        Next i
        Return -1
```


Integerget_address_of_token(string s1)

```
        Declare variable integerr place
        place = searchtoken(s1)
        If place <> -1 Then
            Return address (symbol(place))
        Else
            Return -1
        End If
```

Integerfind_value_of_token(string s1)

```
      Declare variable Integer place
        place = searchtoken(s1)
        If place <> -1 Then
            Return value (symbol(place))
        Else
            Return novalue
        End If
```

Integerupdatetoken(string name, Integer newval)

```
        Declare variable Integer x
        x = searchtoken(name)
        value (symbol(x)) = newval
```

Itemget_token(string str)

```
        Declare variable Integer place
        place = searchtoken(str)
        Return symbol(place)
```

Figure 10: ***Pseudo-code for the symbol table***

As the program execution progresses, this area can be scrolled. The view of memory gives the user a clear idea of exactly what is happening and the option to inspect any specific location in memory.

4.5. Overall Design

After each component of the simulator software had been examined in detail, the next step was to suggest how they all related to each other. This was best achieved using pseudo-code of the anticipated program. The detail of how the user will enter code and begin simulation execution is irrelevant here as the objective of this section is to summarize the process of the assembly visualization.

Since there was no real memory in the tool to store all assembly lines written by the user in the code area, it was necessary to generate a place to store all the code. All lines of the code area are stored in real memory as an array of strings. Words are extracted from the line as needed by the simulator. Words extracted from the line are easily tokenized into opcode, operand, address, etc. (See Section 4.3.) The following declaration creates an array to store all the lines of a program.

```
Declare public string array instru for 1000 elements

;copy the instructions line by line here
```

The array *instru* has string type, its length is 1000, i.e. it can store up to 1000 lines, and this array is global. The assembler stores all its lines in *instru*. The last line of the code stored contains “ENDP”, the end of the program. If the user forgot to type

"ENDP", the assembler gives a warning. The following code in Figure 11 searches for "ENDP" and stores all lines to *instru*.

```
Public Sub storepgm()'this should be written in the main class
Declare Integer variable i = 0  'to start a loop
Declare string of array instru '(instru is holding the lines of codes)


        Try
            While (Lines(i)in code are not "ENDP")
                       'check last line value should be "ENDP"

                instru(i) = Lines(i) from code area
                              'hold lines to instru
                add 1 to I 'each to of loop shows total number of lines

            End While


        Catch index of the instru 'to show whick line is executed, or to specify error
wich will be coded further

            Show message("End marker of code not in sight or missing") 'if "ENDP" is
not found in the code area
            Exit
        End Try
```

Figure 11: Pseudo-code for the storage program

The instructions are represented by their own class. The pseudo-code in Figure 12 declares an array of all the instructions. Operands are done similarly. Figure 12 also implements the registers in an obvious manner.

```
Declare publec class of Lex (for lexical analyze)

Declare integer variable num_opcode = 32 'specifies number of instructions

Declare string array of opcode = {"DB", "DW", "JB", "JE", "JG", "JL", "JMP", "JNC",
"JZ", "LODSB", "LOOP", "RET", "CALL", "DEC", "INC", "MUL","MOD", "POP", "PUSH", "DIV",
"ADD", "CMP", "LEA", "MOV", "OR", "SHR", "SHL", "END","ENDP", "SUB", "PROC", "XOR"}
'introduced instructions

Declare integer num_reg = 16 'specifies number registers
Declare string array of Register = {"AX", "BX", "CX", "DX", "AH", "AL", "BH", "BL",
"CH", "CL", "DH", "DL", "SP", "BP", "SI", "DI"} 'introdcued regiser
```

Figure 12: Pseudo- code for introductory instruction

Figure 13 illustrates the scanning of lines to extract words that represent instructions, registers, labels, or comments.

CodegenClass

```
    Declare New Hashtable()opcodes
    Declare New Hashtable()regs
    Declare New Context ccon
    Declare array of String words(10)
    Declare array of Integer type(10)
    Declare variable Integer ins_no As
    Declare small_lex As New Lex
```

Integerextract_word(String str)

```
        Declare variable Integer k
        Declare variable Integer startindex
        Declare variable Integer endofstring = -1
        Declare Variable Integer slen
        Declare Variable Integer i
        Declare Variable Integer num_of_chars = 0
        For i = 0 To 10
            words(i) = ""
        Next i

        str = str + ";"
        slen = Length(str)

        startindex = 0
        k = 0

        While ((k < slen) And (Character(str(k) <> ";"))'look for ";" if found means
there is a comment

            k = k + 1
        End While
        endofstring = k
        num_of_chars = endofstring 'specify total number of character in the line

        i = 0
        k = 0
        Declare made As Boolean = False
        While (Length(str) > 0)
            If (Character(str(0)) = " ") Then 'looking for the spaces to extrct words
                If made Then
                    i = i + 1
                    made = False
                End If
                num_of_chars = num_of_chars - 1
                str = Substring(str(1, num_of_chars))
            ElseIf character(str(0)) = ":" Then 'look for ":", if found means here is
label
                If made Then
                    i = i + 1
                    made = False
                End If
                words(i) = ":"
                i = i + 1
                num_of_chars = num_of_chars - 1
                str = Substring(str(1, num_of_chars))
            Else character(str(0)) = "," Then 'look for "," if found means there is
two operands

            End If
        End While

        Return i
```

Figure 13: Pseudocode for code generation (codegen)

We now need to compare the extracted word with the tabled instruction. The function in Figure 14 searches for a matching instruction.

```
Boolean is_opcode(arg string w)
```

```
Declare integer i = 0
Declare boolean found = false

        Do
                If UpperCase(w) = opcode(i) then 'look for the coorect opcode
                        Found = true
                End if

        i = i + 1
        loop Until ((found = true) or ( i > = num_opcode)) 'number of loop is 32 is
equal to value of num_opcode
return found
```

Figure 14: Pseudo-code check for correct opcode

The instruction type is recognized next. The instruction type can be any one of the following: logical, arithmetic, control, jump, movement, or end. Figure 15 shows the pseudo-code.


```
Declare public class of Parser (for parsing)
```

```
Declare context local_cont ;extract word for parsing
```

```
Boolean is_Compute_instr()
```

```
If (is_arithmetic_inst() Or is_logical_inst() Or is_datamove_inst()) Then 'look for
the instruction type according to the mentioned function
        Return True
Else
        Return False
End If
```

```
Boolean is_arithmetic_inst()
```

```
                If (extracted word= "DEC") Or (extracted word= "INC") or (extracted
word= "MUL") Or (extracted word= "DIV") Or (extracted word= "ADD") Or (extracted word=
"SUB") Or (extracted word= "CMP")or (extracted word= "DEC") Or (extracted word=
"INC")or (extracted word= "MUL") Or (extracted word= "DIV") Or (extracted word= "ADD")
Or (word(0)) = "SUB") Or (extracted word= "CMP")Then
                        Return True
                Else
                        Return False
                End If
```

```
  Boolean is_logical_inst()
```

```
 If extracted word = "OR" Or extracted word = "SHR" Or extracted word = "XOR" Or
extracted word = "SHL" Then 'if the instruction is match
            Return True
        Else
            Return False
        End If
        "OR", "SHR", "XOR"
```

```
Booleanis_datamove_inst()
```

```
                 If (extracted word(1) = "MOV") Then 'look MOV instruction
                     If is_Register(word(2) Then  'MOV should follow with a register
name
                          If character(1)= "," then 'after register name should be a
","
                              Is_Register(word(3) or is_ID(word(3) or
is_Address(word(3)) than 'there should be a correct third word
                         Return True
                     Else
                         Return False
                           End If
                     End If
                 ElseIf (extracted word(1)) = "LEA") Then 'LEA is the same MOV
                     If is_Register(word(2) Then
                          If character(1)= "," then
                              Is_Register(word(3) or is_ID(word(3) or
is_Address(word(3)) than
                         Return True
                     Else
                         Return False
                          End If
                     End If

          ElseIf (word(1) = "PUSH") Then 'PUSH should follow with a correct second word
                        If (word(2) Is_Register(word(2) or is_ID(word(2) or
is_Address(word(2)) than
                 Return True
                Else
                 Return False
                    End If
                ElseIf (word(1)) = "POP") Then 'POP should follow with a correct second
word
            If (word(2) Is_Register(word(2) or is_ID(word(2) or is_Address(word(2))
than
                 Return True
                Else
                 Return False
                End If
        Else
            Return False
          End if
```

Booleanis_Control_instr()

```
      If (is_End_inst() Or is_jump_inst()) Then 'according  these function return
value
            Return True
        Else
            Return False
        End If
```

Boolean is_End_inst()

```
      If (extracted word = "END") Or (extracted word = "ENDP") Then 'shows the end of
program or end of procedure
            Return True
        Else
            Return False
        End If
```

Booleanis_jump_inst()

```
                 If (extracted word = "RET") or (extracted word = "JB") Or (extracted
word =  "JE") Or (extracted word =  "JG") Or (extracted word =  "JL") Or (extracted
word =  "JMP") Or (extracted word =  "JNC") Or (extracted word =  "JZ") Or (extracted
word =  "CALL") Then 'if the instruction is match
                     Return True
                     Else
                         Return False
                     End If
                 Else
                     Return False
                 End If
```

Figure 15: Pseudo-code to check instruction types

Like a real microprocessor, data entered in registers and memory is represented in hexadecimal base. The following functions in Figure 16 were developed for this purpose.

String hexdigit(Integer x)

```
  Declare variable String st = ""
If x = 0 Then
    st = "0"
ElseIf x = 1 Then
    st = "1"
ElseIf x = 2 Then
    st = "2"
ElseIf x = 3 Then
    st = "3"
ElseIf x = 4 Then
    st = "4"
ElseIf x = 5 Then
    st = "5"
ElseIf x = 6 Then
    st = "6"
ElseIf x = 7 Then
    st = "7"
ElseIf x = 8 Then
    st = "8"
ElseIf x = 9 Then
    st = "9"
ElseIf x = 10 Then
    st = "A"
ElseIf x = 11 Then
    st = "B"
ElseIf x = 12 Then
    st = "C"
ElseIf x = 13 Then
    st = "D"
ElseIf x = 14 Then
    st = "E"
ElseIf x = 15 Then
    st = "F"
Else
    st = "X"
End If
Return st
```


String tohexbyte(Byte x)

```
Declare st As String
Declare y As Integer
y = x \ 16
st = hexdigit(y)
y = x Mod 16
st = st + hexdigit(y)
Return st
```

String tohexint(Integer x)

```
Declare variable String st = ""
Declare variable String st1
Declare variable Integer y
Declare variable Integer safex
safex = x
Do
    y = x Mod 16
    st1 = hexdigit(y)
    x = (x \ 16)
    st = st1 + st
Loop Until (x < 16)
st = hexdigit(x) + st
If safex < 256 Then
    st = "00" + st
ElseIf safex < 4096 Then
```

```
            st = "0" + st
        End If
        Return st
```

Figure 16: Pseudo-code to change a decimal to a hexadecimal

We also need to find the correct operand for the introduced instruction. The functions in Figure 17 parse the operand.

```
Boolean is_Register(arg string w)
Declare integer i = 0
Declare boolean found = false

        Do
                If w = Register(i) then
                        Found = true
                End if

        i = i + 1
        loop Until ((found = true) or ( i > = num_reg)) 'look is w found as a correct
register name, till the loop is equal to the num_reg
return found
```

```
Boolean is_Address(arg1 string wrd, arg2 string w2, arg3 stirng w3)
If (is_ID (wrd)or (is_Register(wrd) and w2 = "[" and w3 = "]"then 'Address should
appear in [ ]
        Return true
Else return false
End if
```

```
Boolean is_ID(arg1 string wrd)
Declare string pattern 'the string should start with one small or capital letter a-z)
Declare Match Collection mc = match (wrd, pattern) 'in Visual Basic.NET we could
declare match collection
        If (length (mc) = 0 and len (mc) > 32) then
            Return false
        Else
            Return ture
        End if
```

Figure 17: Pseudo-code to check operand types

This pseudo-code program details the assembly cycle and is unlikely to change much in the implementation stage. This algorithm holds entered lines until the last line "ENDP" is reached. Note that a simple sequential search is used to find label, register, address, and comment, from separated words in each line.

4.6. Summary

This chapter has provided a breakdown of the proposed assembler simulator. Each component has been examined in detail. The design of the proposed implementation has been discussed. The main issues discussed in the chapter are: the instruction set supported by the simulator, assembler format, components of the 8086 microprocessor, the GUI for the simulator. Chapter 5 presents the implementation of the simulator.

Chapter 5

IMPLEMENTATION

5. Introduction

This chapter describes the implementation of the simulator, where each component as described in the specifications in Chapter 4 was implemented. The design was specified using a top-down process, where the original specification was refined until a detailed design was produced. It is at this stage that the feasibility of the ideas specified in Chapter 4 was really tested, since the actual code had to be processed. At this point the bottom-up implementation started. The design was implemented stage-by-stage until the final program was completed.

This section is not intended to explain each of the classes in the software in detail, but rather to give an idea of the operation of most of the classes and their interaction with each other. The simulator has two main components, namely the simulated GUI section, and the background computation component. The development of the assembler will be discussed, followed by the GUI functions and implementation parts such as the registers, which are general purpose, special purpose, and processor status word. Error handling and the reporting of errors to the user is also covered, along with the display of memory, which is the main function of the GUI.. Since our tool is logically separated into two parts, the GUI and assembler, separate classes and functions have been written.

5.1 Assembler Implementation

The structure of the assembler is summarized in Table 2. Since the source code of the assembler is about 1500 lines in Visual Basic.NET, it is impractical to describe all aspects of the code development in this chapter. However, the test and evaluation of the software is done in the next chapter to establish the correctness of the implementation of the Assembler. Usability and the developed classes and functions will next be described.

Since all assembly code is stored in a text area in the GUI, one class has been implemented to store the lines. The class *stack* has been written to generate spaces for holding lines of code. The stack can hold 1000 lines. The `storepgm()` method stores the entire user code in an array of strings.

The class `lex`, the lexical analyzer, the class `parse` and the class `symtable` interact and share data as follows: `lex` picks up a table and shares it with the parser, which uses `symtable` to determine its functions. This interaction proceeds on a line-for-line basis until the end is reached where the class is called *`codegen`*. The class `codegen` has been constructed to hold all values of the assembled code for simulation.

Many snippets of code were used to test the assembler's correctness, such as:

```
MOV AX, 10
MOV BX, 3
DIV BX
ENDP
```

This yields registers with the following values.

```
AL =  10
BL =  03
DL =  01
```

Some students used the developed tool at KU. Most comments given by the students concerned negative value movement to a register. Their suggestion was implemented, e.g.

```
MOV AX, -2
```

The result according to the real 8086 microprocessor which uses twos` complement is `FFFEH`.

5.1.1 Error Handling

As the code being assembled is typed in by the user, it is likely that there will be some errors. Information about the nature of these errors can serve as feedback to the user. When an error occurs the assembler throws and exception that is caught by the `lex` class.

Within the `lex` class many logical functions are used such as is_opcode, is_Register, is_Label, to check that the code's type is correct..

The `execute()` function works almost exactly as described in the design section, parsing the line, opcode, operand and comment. For example, if a user erroneously types MOV AX; 8, i.e. a semicolon is typed instead of a comma, the error message includes the line number where the error occurs:

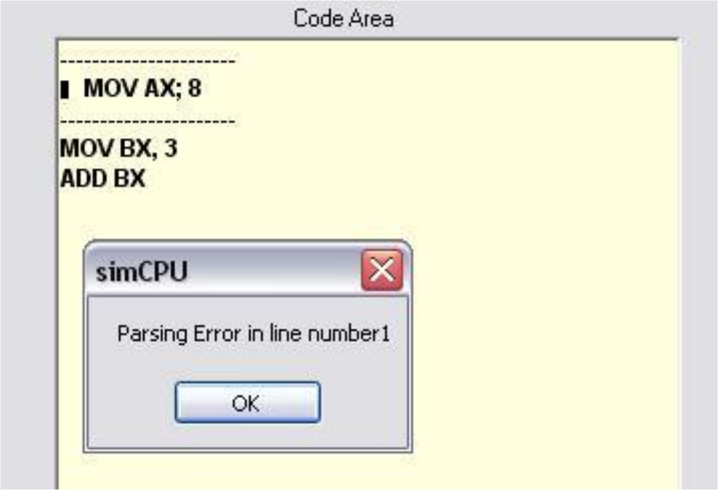

Figure 18: Example of error handling

5.1.2 The Error Display

There are many potential user errors. It is very important to display the errors clearly. A concise error message and the location of the error are displayed. If the user ignores the message, the user is prompted to correct the error, as in Figure 19.

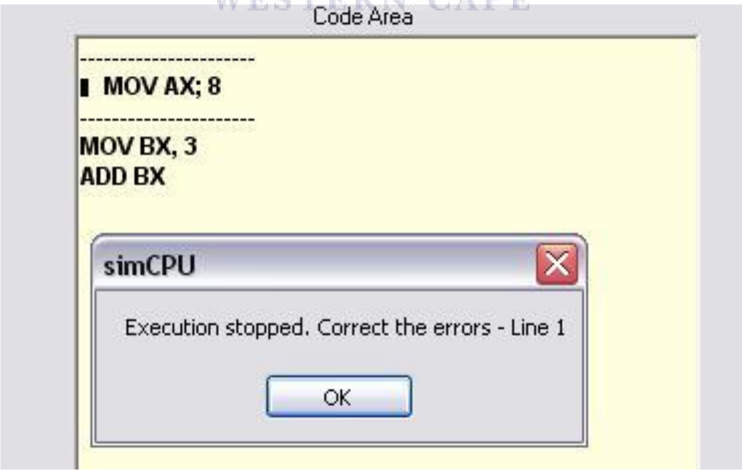

Figure 19: Error handling and correction

5.2 Graphical User Interface (GUI) Implementation

The GUI screen layout design proposed and drawn in Chapter 4 is a satisfactory screen layout and it covers all parts of 8086 architecture including the memory view. The proposed screen layout gives sufficient display space for each element, allowing all the suggestions and specifications of the design section to be attended to.

The set of registers, program area and memory area are the major blocks of the system. As such they have been coded as an extension of the Visual Basic.NET form. Visual Basic.NET is popular for implementing interfaces. Depcik and Assanis (2005) have used Visual Basic.NET to develop GUIs in an engineering educational environment.

GUI implementation is separated into the following parts as specified in Chapter 4.

5.2.1 Registers.

The general purpose registers, special purpose registers and the processor status word, i.e., the code condition registers are discussed next.

5.2.1.1 General Purpose Registers

In a real 8086 microprocessor there are 8 general purpose registers, each has its own name (Alpert & Avnon, 1993 and 2000). The following descriptions specify each part of the implemented registers.

`AX`: the accumulator register (divided into `AH` and `AL`):

1. One number must be in `AL` or `AX`
2. Multiplication and Division
3. Input and Output

`BX`—the base address register (divided into `BH` and `BL`):

1. Its value can be a decimal or hexadecimal

`CX` - the count register (divided into `CH` and `CL`):

1. Iterative code segments using the `LOOP` instruction
2. Repetitive operations on strings with the `REP` command
3. Count (in `CL`) of bits to shift and rotate

`DX`—the data register (divided into `DH` and `DL`):

1. `DX`: `AX` concatenated into a 32-bit register for certain `MUL` and `DIV` operations
2. Specifying ports in certain `IN` and `OUT` operations

These registers are created as follows:

According to the proposed screen layout for the GUI, two textboxes are used for each general purpose register. One for the low parts and another for the high parts, according to the division of these registers in a real 8086 microprocessor, i.e. AX is divided into AL and AH parts, with three labels for specifying the register and its parts.

Figure 20 shows the implemented general purpose registers.

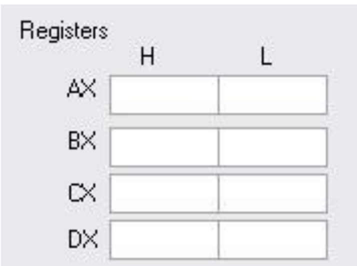

Figure 20: Implemented general purpose registers

5.2.1.2 Special Purpose Registers

The actual 8086 microprocessor uses special purpose registers such as the stack pointer (SP) and the instruction pointer (IP), which we have implemented.

These registers all use 16 bits.

Each register has a special function.The SP register points to the segment containing the machine instructions that are being executed at a given moment. Changing the value of the code segment (CS) register changes the code being executed. The counter register (CX) is used for loop instruction. The IP is sometimes referred to as the program counter (PC). This register cannot be accessed directly and is modified by the processor during execution. The PC points to the address of the next instruction and the instruction register holds the current instruction being executed. Since the PC is not directly used in the simulator, users cannot alter it, but it is always displayed. For example, after executing instruction number 2, the message shown in Figure 21 displays:

Figure 21: Message shown as a program counter

Special purpose registers are visualized similarly to general purpose registers.

Figure 22 shows the special purpose registers CS, IP and SP.

Figure 22: The special purpose registers CS, SP and IP

5.2.1.3 Processor Status Word: Code Condition Registers

The program status word (PSW) is a unique type of microprocessor register (described in Section 2.2) in the sense that its contents represent six different variable flags. Typical flags are OF, SF, ZF, AF, PF and CF. The processor has a variable for each flag. The PSW has the following parts:

- The OF (overflow flag) indicates an overflow status.
- The SF (sign flag) is set when the number resulting from a calculation is negative.
- The ZF (zero flag) is set, i.e. it becomes 1, when the number resulting from a calculation is zero.
- The AF (auxiliary carry flag) is a second carry flag.
- The PF (parity flag) indicates even or odd parity.
- The CF (carry flag) contains the carry bit of an arithmetic operation.

The PSW is illustrated in Figure 23.

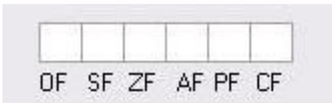

Figure 23: The processor status word (PSW)

5.2.2. Memory Display

This section first describes the memory discussed in Section 4.4.2.

Figure 24 shows the memory implementation.

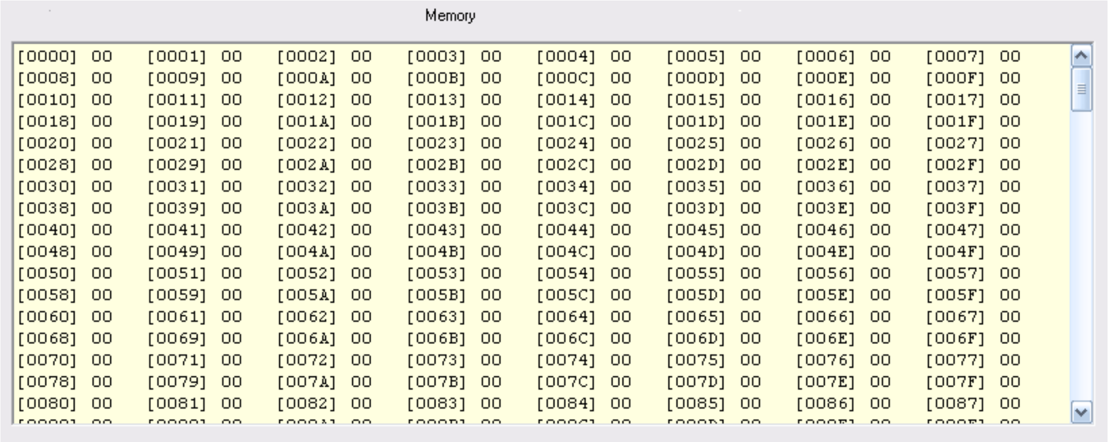

Figure 24: The memory display

Each memory cell occupies one byte, and has a unique address in brackets.

5.2.3. The Toolbar

So far it has been assumed that the simulation will start and stop and the user code will somehow be assembled. The toolbar, a small panel object containing several buttons, provides the following functions:

Step: only becomes enabled after a successful assembly, and when pressed animates the execution of one instruction.

Run: becomes enabled after a successful assembly, and when pressed begins animation of the entire program.

Stop: becomes enabled after run is pressed, and will stop the simulation after the current instruction.

5.2.4 The Task-bar

Like any other software the simulator has a taskbar with menus such as File, Edit View, Run, Math and Help. In the File menu, sub-instructions such as Open, Save, Save As, and Close are implemented. The Save option saves the file in a form that Open can load and saves the entire program for reuse later. The format of the file typed in the code area is text, so the extension of the file is .txt. Figure 25 shows the task- and toolbars.

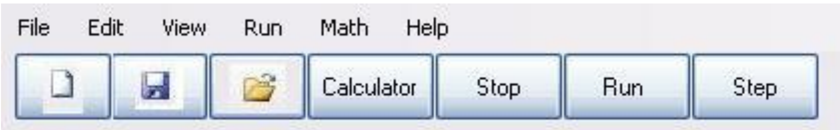

Figure 25: Menubar and toolbar view

Figure 26 shows the complete GUI:

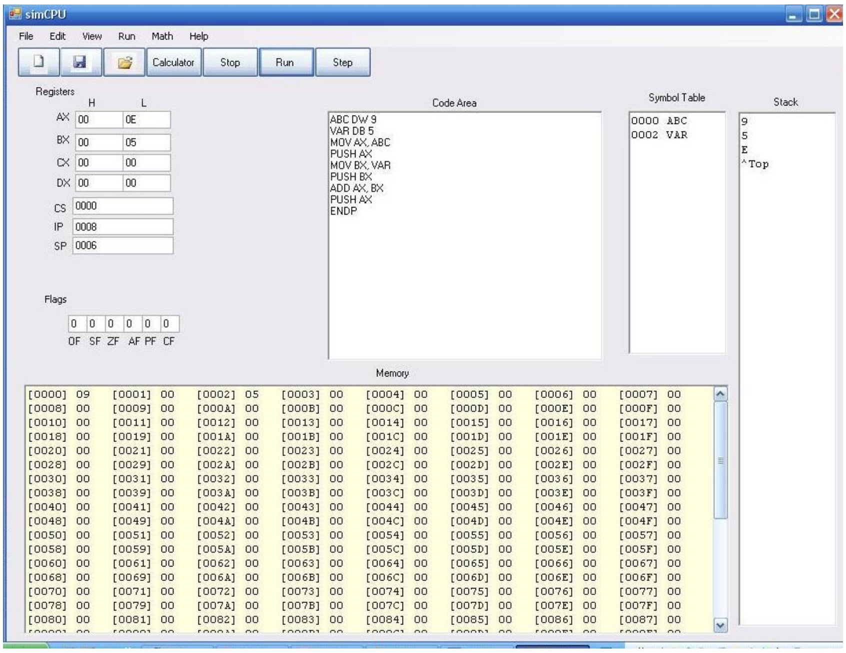

Figure 26: Complete developed GUI

The interface displays each component, i.e. the code area, register set, memory, symbol table and stack. In the code area the user can enter the program. The instructions can be executed step by step with the Step command button. The area for registers, memory and stack displays values and results after running the instructions. The symbol table shows the location of variables in memory.

5.3. Summary

We have described our simulated visualization tool for a subset of the instruction set of the 8086 microprocessor specified in the design chapter. This tool has two parts: the assembler and the GUI. In summary we have implemented the following:

An assembler with a set of 32 instructions

- A starting program counter-location for the instruction
- A text opcode for instructions (the same 8086 opcode)
- A hexadecimal version of the text operand if it is a symbol

A GUI

- The active parts of the 8086 microprocessor
- Highlighting the line of code that is being executed
- Visualization of 8KB of memory

In the next chapter we describe the testing of the correctness of the simulator and we assess its usability and user-friendliness.

Chapter 6

TESTING AND EVALUATION

6. Introduction

In this chapter an evaluation of the usability of the simulator and of its perceived benefits to the students at KU is presented. A survey was used to determine what impressions the students who used the simulator in the course had of the simulator. The simulator was used to teach students about easy microprocessor operations and elements such as the register set, stack usage, instructions for arithmetic, looping and memory manipulation with registers.

The completed questionnaires were analyzed to assess how using the simulator was received, how it compares with more traditional approaches, and the best and worst features of the tool. In summary the students found the tool practical and enjoyable, and a motivating, effective and stimulating learning tool.

6.1 Description of the Simulator

The *simCPU* is essentially a tool to familiarize students with the intricacies of assembly instructions and the 8086 microprocessor operations. The tool's aim is to help students understand the processor operations under different conditions and instructions. Understanding real microprocessor operations is a requirement for an advanced computer architecture course. Figure 27 shows a snapshot of the *simCPU*.

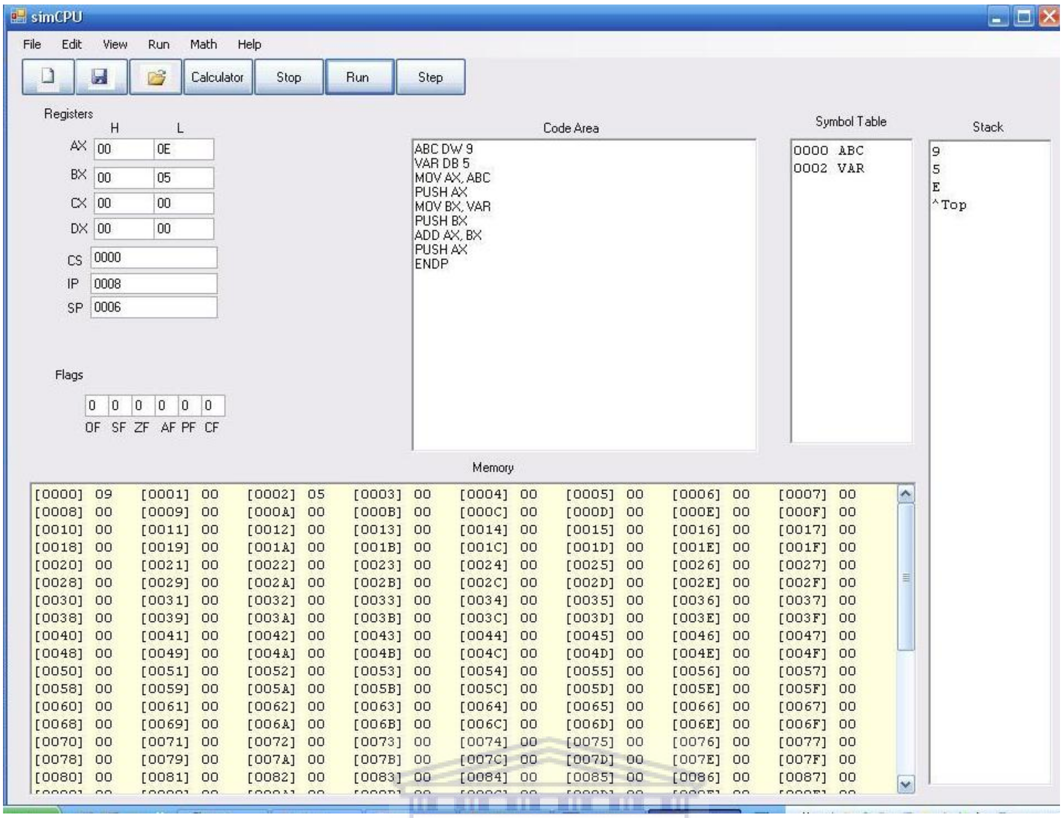

Figure 27: Complete snapshot of the tool

The interface components are the register set, code, the memory, a symbol table and the stack. In the code area users can enter the program. The instructions can be stepped using the Step command button. The registers, memory and stack are displayed in their text boxes after running the instructions. The symbol table shows where the variables are located in memory. As with a real microprocessor all entered data is displayed in hexadecimal.

6.1.1 Testing the Correctness of Instructions

Establishing the correctness of the instructions we have simulated turned out to be very simple. Each instruction was tested individually as follows:

Before executing any instruction the start state of the simulator is recorded. The specific instruction being tested is single-stepped once using the Step button and the resulting state of the simulator is recorded. The resulting state of the machine as displayed by the simulator is then compared with the specific expected outcome of the instruction. This process was completed for each instruction in turn. Some behaviours of the instructions, such as their effect on the carry flag or the sign flag, necessitated more than one test.

Once these instructions were tested for correctness, their collective behaviour when run as a sequence of instructions were necessarily correct because the instructions are mutually independent. Our confidence in the correctness of the simulator was justified when the simulator was used by 78 students in class and no new defects came to light.

6.1.2 Memory Visualization

8KB of memory was simulated, where the address starts at `0000H` and ends with `2000H` hexadecimal. A memory cell is displayed as two bytes preceded by its four-byte address in brackets. Displaying the address of each byte individually was found by students to simplify its usage.

6.1.3 Registers

The general purpose, special purpose, and flag registers are visualized in an obvious way as two-byte fields in hexadecimal. The flags in the PSW are visualized as single bits. Each general purpose register was fully tested to ensure that all actions were performed correctly. For example, in one of the general register boxes, such as in AL, a value would be set that is less than 256. This was tested to ensure that these values appeared on the screen. The next stage was tested to ensure that when a new value is entered, the old value is changed, or stayed the same. These tests were executed visually using an example such as the one in Figure 28, where we first moved 8 into the AL.

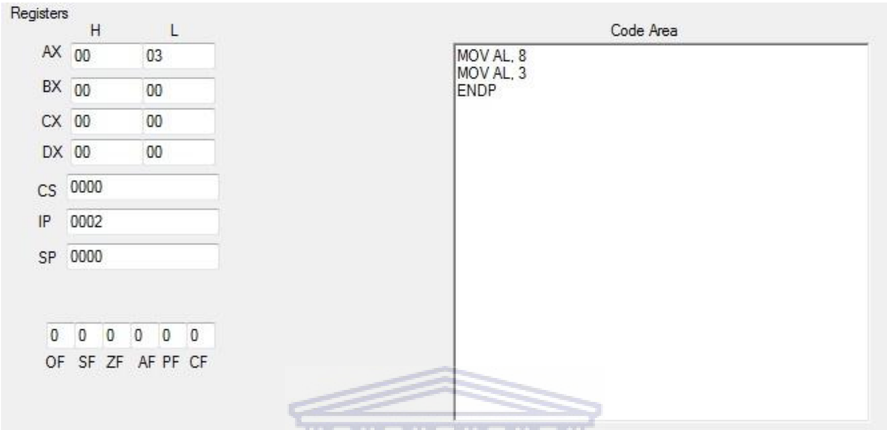

Figure 28: Refreshing of the AL register

Using `MOV AL, 8` and in the second line of code we move `3` to the same register, and observed the new state of the register. We see that the value of the IP is `2`, which shows that the second instruction has been executed and the machine is ready to execute the instruction in location 2. Note that the first instruction is at location 0, and the second is at location 1, so the IP must point at 2, i.e. at the third instruction. A further function that is added to the general register was the ability to display its value in hexadecimal. The user has a choice to enter either hexadecimal or decimal. The default value for input is decimal, but adding "H" at the end of the value in the code area specifies a hexadecimal input value. The results of this test are shown in Figure 29.

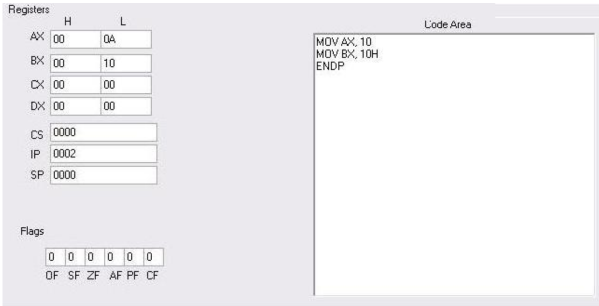

Figure 29: Specifying a hexadecimal number

6. 1. 4 Help System

As with any interactive teaching system it is desirable to have a help system to guide the user when necessary. The help for this program covers two areas:

1. The program operation help shows the user how to operate the tool, explaining each feature provided.
2. The instruction set help is provided for the implemented instruction set, giving examples of the usage of instructions, or describing their function and noting any peculiarities.

6. 1. 5 Syntax Highlighting

Almost all modern program development tools use a program editor that colours keywords according to their type. This would be an excellent addition to the program input area as it would give the user instant feedback on the accuracy of their program syntax. However, if a syntax error occurred using this tool the whole line of code will be highlighted. We therefore suggest specific word syntax highlighting. This is much better than requiring the user to assemble the code to find errors.

6.2 Usefulness and Benefits of the Tool

The simulator has been designed with a GUI which is easy to use. The stack and symbol table areas that are added to the GUI make the tool well suited to students. Integrating the simulator with our lectures enhanced the teaching and learning of various aspects of microprocessor operations. In the classroom, an in-class task that was given to the students to test registers, memory and stack values according to the instructions during or after execution time proved to benefit their understanding. After a month of using the simulator students reported their satisfaction of it as a learning device. (See Section 6.3.)

The main benefits of the tool are as follows:

- **Hands-on:** It facilitates interactive, hands-on learning of 8086 microprocessor concepts.
- **Simulation:** It simulates subset instructions of assembly language and hence enhances knowledge and understanding of a variety of 8086 microprocessor elements.
- **Easy to use:** The GUI makes the tool easy to use and user-friendly.

These claims are substantiated by our surveys described in Section 6.3 and our interpretation of the results of the survey in Section 6.4.

6.3 Evaluation of the Tool by Students

When any design has been completed it is important to evaluate its performance to ensure that it meets its goals (Barua, 2001). The specification for this research was set out in Chapter 4 and summed up as a series of points in Sections 4.1 and 4.5. We tested the simulator by letting 78 students in the computer architecture course at the Computer Science department of KU use it.

Students in the class were sampled for the survey, each having a different level of experience with the CPU, ranging from an overview to detailed design knowledge of microprocessor architecture. All were able to describe what was happening on the screen, while a few simple instructions were simulated. The more intelligent students stated that the system would be an excellent refresher on microprocessor operation for studying microprocessor architecture in one semester. In contrast the less intelligent students needed a little help in pointing out what was happening, but quickly recognized the simulated operations.

The simulator was used by the students for a month, after which they were asked to complete a questionnaire. The survey shows that the simulator helped them to facilitate their understanding of microprocessor operations and follow-up elements such as the register set, stack usage, assembly instructions, execution of arithmetic, looping and memory relations with the registers.

The scenarios were explained to the students for the following broad aspects of learning computer architecture topics:

- Tests of our subset of 8086 instructions that can be simulated
- Control of the flow of execution of the instructions from one location to another
- Memory visualization
- Checking of register values under different conditions
- Stack usage

To evaluate whether the simulation succeeded in making the learning process more effective in accordance with the course objectives, we posed a number of questions to the students regarding their understanding of the microprocessor operations: their ability to apply computer architecture concepts, the relevance of the course lecture material and the development of their skills in writing the code. The questionnaires were issued to the students at the end of the course, when they were asked to complete the survey before leaving the computer laboratory. A total of 78 questionnaires were issued to students. Of the total number of questionnaires, responses were received from all students, so that a 100% response rate was obtained.

6.4 Evaluation and Interpretation

The simulator has been evaluated extensively, both formally by students using student evaluation forms, and informally through discussion within the teaching team, in order to assess its educational value. As part of the formal evaluation process, students were asked to complete a questionnaire.

6.4.1 The Questionnaire

Students were asked the following twenty two questions, of which the first 10 will first be discussed:

1. **User-friendliness:** How convenient did you find the 'user interface' of the simulation tool to use?
2. **Simulation Tool information:** How useful did you find the information about the 8086 microprocessor to be?
3. **Easy to use:** How easy (overall) did you find the *simCPU* to use and follow?
4. **Navigation:** How easy did you find navigation through this simulation tool?

5. **Concept development:** How effective was the *simCPU* in helping you to improve your understanding of 8086 microprocessor concepts?
6. **Concept review:** How effective was the *simCPU* in helping you to improve your understanding of assembly instructions?
7. **Error reporting:** How effective was the *simCPU* for reporting the error in your program typed in the code area to you?
8. **Error recovery:** How effective was the *simCPU* for reporting an error for you to correct?
9. **Knowledge testing:** Did memory, registers and stack simulate reality in this tool?
10. **Hands-on:** Would you like to have more tools of this kind as part of your course?

A Likert scale with 5 points (1–5) was used in the questionnaire. For questions 1 to 8: 1 = Excellent, and 5 = Poor, and for questions 9 and 10: 1 = Yes, and 5 = No.

6.4.2 Response to the questionnaire

78 undergraduate students from the computer architecture course completed the questionnaire and their responses are plotted in Figures 30 to 39. The responses were interpreted as follows:

1. The GUI of the tool was found to be easy to use. About 87.2% of the students indicated that they were quite satisfied with the tool interface whereas the remaining 12.8% were neutral. See Figure 30.

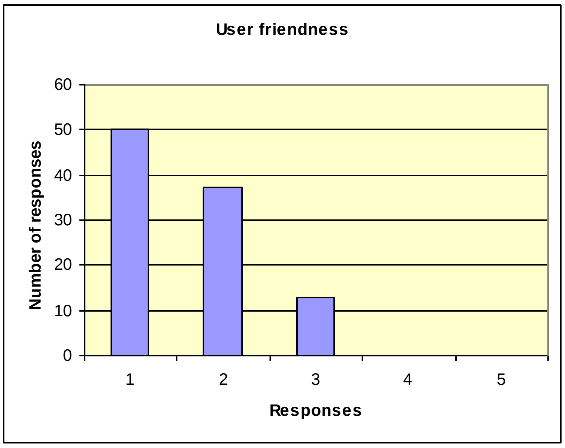

Figure 30: Students' responses to Question 1

2. About 84.62% of the students indicated that the simulation tool information presented in the platform is very useful. 1.28% of the students expressed some concern, and the rest (14.10%) were neutral. See Figure 31.

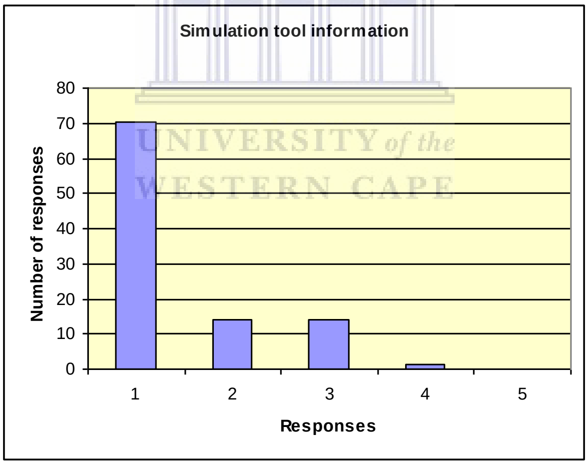

Figure 31: Students' responses to Question 2

3. The tool was found easy to use and to have a user-friendly interface. About 87.18% of the students were happy with the current version of the tool. However, 1.28% of the students indicated that they were not completely satisfied with the current version of the tool, and the remaining 11.54% were neutral. (See Figure 32.)

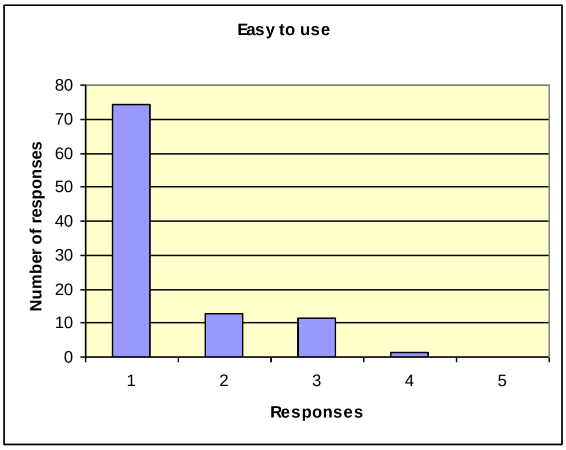

Figure 32: Students' responses to Question 3

4. About 93.59% of the students indicated that they found the tool to be easy to navigate, whereas the remaining 6.41% were neutral. (See Figure 33.)

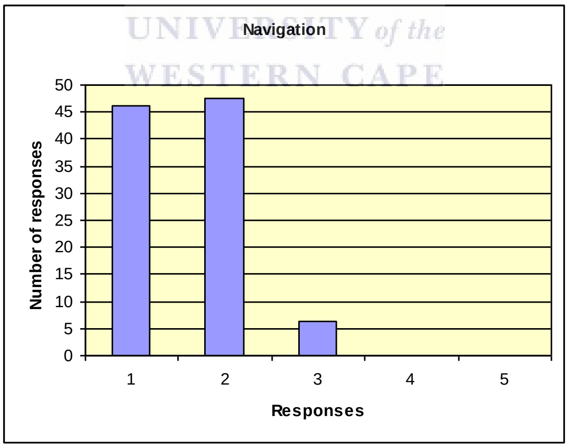

Figure 33: Students' responses to Question 4

5. All of the students indicated that the tool had assisted them in developing a better understanding of the concepts of the 8086 microprocessor. (See Figure 34.)

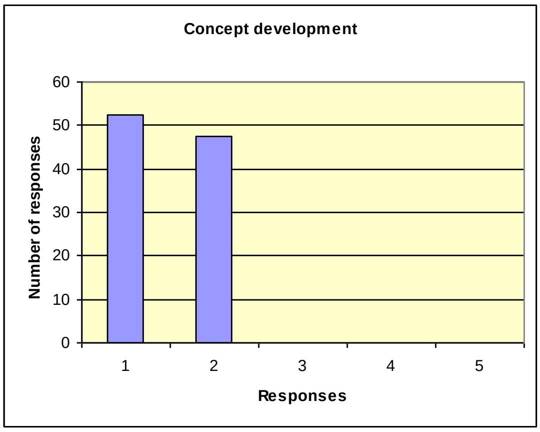

Figure 34: Students' responses to Question 5

6. About 85.9% of the students indicated that the *simCPU* was effective in helping them to improve their understanding of the assembly instruction set. Only 1.28% of the students felt that the simulator was not helpful, and the remaining 12.82% were neutral. (See Figure 35.)

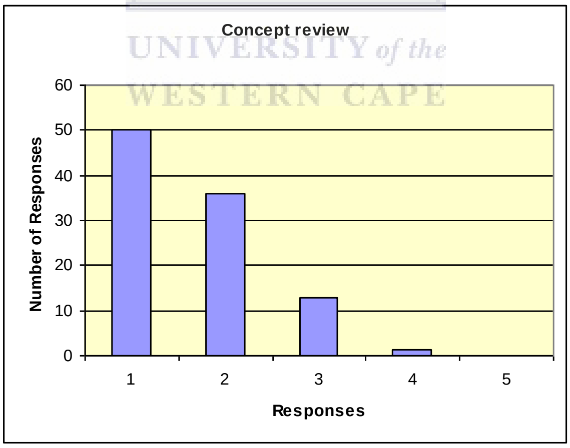

Figure 35: Students' responses to Question 6

7. About 91% of the students indicated that the errors reported were useful. About 9% of the students were neutral. (See Figure 36.)

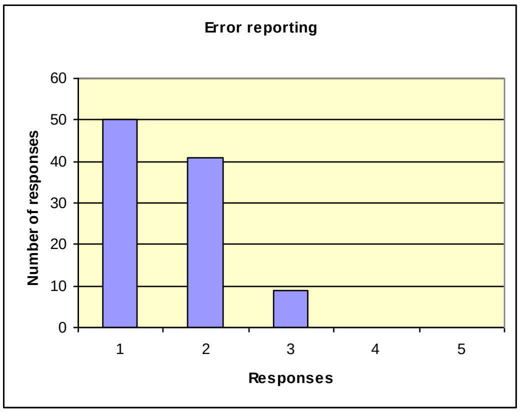

Figure 36: Students' responses to Question 7

8. About 80.8% of the students stated that errors reported by the simulator were easy to correct. About 1.28% of the students were not satisfied with the error reporting, and the remaining 17.92% were neutral. (See Figure 37.)

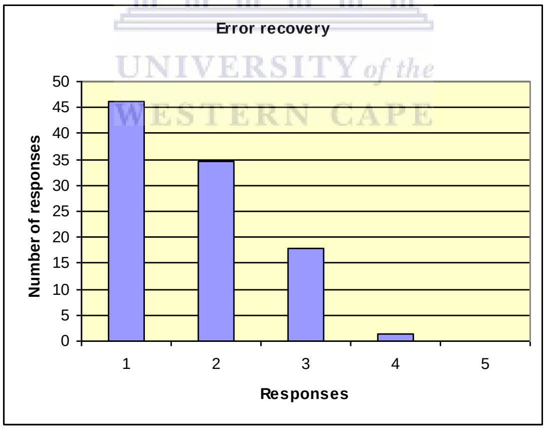

Figure 37: Students' responses to Question 8

9. About 91% of the students indicated that the simulator realistically simulates the memory, registers and stack, while the remaining 9% of students were neutral. (See Figure 38.)

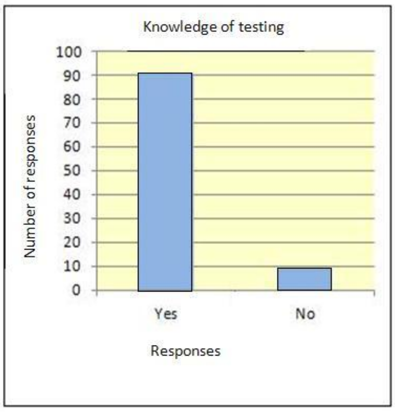

Figure 38: ***Students' Responses to Question 9***

10. All the students indicated that they would like to have more tools of this kind as part of their courses. (See Figure 39.)

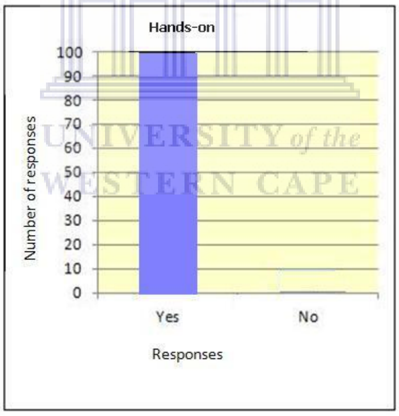

Figure 39: Students' responses to Question 10

The remaining twelve questions and their responses are given in Tables 3 to 8.

Table 3: Student feedback on the simulation tool

No.	Questions	Strongly agree (%)	Agree (%)	Neutral (%)	Disagree (%)	Strongly Disagree (%)
11	It helped me understand the intricacies of microprocessor operations.	11.25	78.75	10	0	0
12	It enhanced my ability to apply	46.25	40	13.75	0	0

	assembly instruction concepts and principles.					
13	It helped me understand the relevant lecture material.	56.25	43.75	0	0	0
14	It increased my programming skills.	10	27.5	25	25	12.5
15	It increased my knowledge about registers.	60	40	0	0	0

Table 3 tests the perception the students have of the usefulness of the simulator. Even though most answers referring to the understanding of specific concepts are positive, the responses to Question 14 are surprisingly more negative than expected.

Table 4: Students' views of their group when using the simulation tool

No.	Questions	Strongly agree (%)	Agree (%)	Neutral (%)	Disagree (%)	Strongly disagree (%)
16	All members contributed equally to the work.	47.5	41.25	11.25	0	0
17	There was a high level of cooperation in my group.	40	53.75	6.25	0	0

Table 4 indicates that the simulator can be used in group work.

Table 5: Students' views of the simulation tool compared to other types of learning

No.	Questions	Strongly agree (%)	Agree (%)	Neutral (%)	Disagree (%)	Strongly Disagree (%)
18	It motivated me to a greater extent.	36.25	38.75	25	0	0
19	It enabled me to learn more.	25	47.5	27.5	0	0

Table 5 clearly indicates that the simulator motivated students and that students felt that it enabled them to learn more.

Table 6: Students' preference for the simulation tool over traditional learning

20	Overall, given the choice between the simulation tool and more traditional learning, which one would you prefer?	Check your selection in this column, then explain why in the last column.

	Simulation tool	**100(%)**	
	Traditional learning	**0 (%)**	
	No Preference	**0 (%)**	

Table 6 indicates the students' overwhelming preference for using the simulator. In most comments students affirmed their preference of using the simulator.

Table 7: Students' reasons for preferring the computer-aided learning

21	Overall, given the choice between computer-aided learning and the traditional type of learning, which one would you prefer and why?	Check all that you believe are true	Comments
	More interesting and enjoyable	100 (%)	
	More practical	75 (%)	
	Facilitates learning process	100 (%)	
	More interactive	87.5 (%)	
	More motivating	87.5 (%)	
	Other aspects (%)		

In Table 7 students' specific reasons for preferring computer-aided learning are given. The students positively responded that the simulator is more interesting and enjoyable (100%), more practical (75%), more interactive (87.5%) and more motivating (87.5%) than traditional learning, and that it facilitates learning (100%).

Table 8: ***Students' Views on the Worst Aspects of Computer-Aided Learning***

22	Here, add any comments you have about the aspects of computer-aided learning that you do not like.
	No negative comments were made about computer-aided learning, but 6.25% expressed the opinion that more instructions could be added to the tool.

Question 22 requests the students to describe aspects that they do not like about computer-aided learning. Table 8 shows that there were only 6.25% that wanted the tool to have more instructions.

In the classroom we observed that students became increasingly motivated to learn more about 8086 microprocessor elements and operations, and that they enjoyed this course more than previous courses that consisted of lectures only. We regularly seek feedback from students for further improvements to the simulator.

6.5 Summary

A simulation software tool, the simCPU, has been developed that can be used either in the classroom or off campus to enhance the learning and teaching of various aspects of microprocessor operations. It was evaluated by students at KU, and their responses to the questionnaire about the SimCPU were overwhelmingly favourable. The students indicated that they had found the SimCPU easy to use, robust and that it helped them to gain an understanding of assembly subset instruction concepts. The SimCPU also had a positive impact on students' performance. Judged by the responses of the students, the simulator has a high valency and its usability has been proven.

Chapter 7

CONCLUSION AND FURTHER WORK

7. Conclusion

A new software tool has been designed and implemented to help students understand a microprocessor's operation. Instructions such as data movement between registers, and to and from memory have been visualized in a graphical representation of the CPU. The user code entered in the code area is assembled and executed correctly. In achieving the initial objectives the project has been a success, enabling students to use the microprocessor simulator to simulate the execution of their programs.

A carefully designed subset of the 8086's components allows the demonstration of most basic CPU features without being too simple. This tool was developed in Visual Basic.NET which turned out to be very suitable for developing this package.

The requirements of the tool were discussed in Chapter 4 and has come from my several years of teaching experience at KU. As mentioned in Chapter 4, the simulated instructions were selected from different instruction classes, i.e. declarations, data movement, control, arithmetic, and logical.

The tool was tested thoroughly and repeatedly during its development. Since the main objective was the development of a subset of 8086 instructions to help the students to understand the computer Architecture course, the effectiveness of the tool was also investigated using a questionnaire consisting of twenty two questions. The students' responses in the questionnaire revealed that the simulator appears to have lead to a better understanding of machine language by the students. Based on their feedback most of the students liked the tool and enjoyed using it. They expressed their wishes to have more such tools as part of their courses. Most of the students appreciated this method of teaching using the simulator and praised it for its usefulness and contribution to their learning.

The research goal of attaining a suitable instruction set was therefore achieved. The students were satisfied with the instruction set, and the instruction set is complete in the sense of containing all the necessary arithmetic, logical, control, and data movement instructions as well as some assembler declarations.

7.1. Suggestions for Further Work

From the outset this project had great potential. The final software is by no means "final", as there are many additions that can still be made to enhance the simulator. This is easy to do because the simulator has been developed using the object-oriented, easily maintainable Visual Basic.NET programming language. The thesis therefore concludes with an outline of possible extensions to the project in the sections that follow.

7.1.1. Implement Simulator on Internet

Since Visual Basic.NET can be used over a network, it would be useful to implement the simulator to run under a web interface also connected to a database of the students that use it. Then the instructor can help track the progress of students.

7.1.2 Expand Instruction Set

A full instruction set simulator has many uses beyond teaching. It could be used as a low-level language debugger.

7.1.3 Internationalize the Interface

The simulator currently only handles Farsi/Dari and English. A quite useful extension is to implement the entire interface so that it can be run in many more languages. It is quite easy to internationalize the simulator since it uses Unicode, and Visual Basic.NET provides for natural internationalization.

Bibliography

Abel, Peter. (2001). *IBM PC Assembly Language and Programming*, 5th edn. Upper Saddle River, NY: Prentice Hall. pp. 23–25.

Alpert, D. and Avnon, D. (1993). 'Architecture of the Pentium microprocessor'. *IEEE Micro* 13, 3 (May. 1993), pp. 11–21. =<http://dx.doi.org/10.1109/40.216745>

Alpert, D. and Avnon, D. (2000). 'Architecture of the Pentium microprocessor'. In *Readings in Computer Architecture*, Hill, M.D., Jouppi, N.P. and Sohi, G.S. eds. San Francisco, CA: Morgan Kaufmann Publishers. pp. 649–659.

Balena, F. (2002). *Programming Visual Basic.Net.* Washington, DC: Microsoft.

Barua, S. (2001). 'An interactive multimedia system on computer architecture, Organization, and Design'. *IEEE Transactions on Education* Vol. 44, pp. 41–46

Cerrato, T. (2002). *Swedish as a Second Language and Computer Aided Learning Language. Overview of the research area*. Department of Numerical Analysis and Computer Science TRITA-NA-P0206 • IPLab-203 • ISSN 0348–2952, Report number: TRITA-NA-P0206, IPLab-203 Publication.

Dickerson, M., Huang, T., and Russell, I. (2000). 'Using simulation across the curriculum'. In *Proceedings of the Second Annual CCSC on Computing in Small Colleges Northwestern Conference*) Consortium for Computing Sciences in Colleges. Beaverton, OR: Oregon Graduate Institute. pp. 56–64

Everingham, B. T., Thomas, B. T. , Troscianko, T. and Easty, D. (1998). 'Neural network virtual reality mobility aid for the severely visually impaired'. In *Proceedings of the 2nd European Conference on Disability, Virtual Reality and Associated Technologies*. Reading, UK: University of Reading. pp. 183–192

Fienup, M. and East, J. P. (2002). 'Improving computer architecture education through the use of questioning'. In *Proceedings of the 2002 Workshop on Computer Architecture Education, WCAE '02* (held in conjunction with the 29th International Symposium on Computer Architecture in Anchorage, AK). <http://doi.acm.org/10.1145/1275462.1275473>

Gonzalez, J.J. (1995). 'Computer assisted learning to prevent HIV-spread: Visions, delays and opportunities'. *Machine-Mediated Learning* 5(1), pp. 3–11

Herath, J., Ramnath, S., Herath, A., and Herath, S. (2002). 'An active learning environment for intermediate computer architecture courses'. In *Proceedings of the Workshop on Computer Architecture Education, WCAE '02* (held in conjunction with the 29th International Symposium on Computer Architecture in Anchorage, AK).<http://doi.acm.org/10.1145/1275462.1275474>

Imhanlahimi, E. O. and Imhanlahimi, R.E. (2008). ‘An evaluation of the effectiveness of computer assisted learning strategy and expository method of teaching biology: A case study of Lumen Christi International High School, Uromi, Nigeria’. *J. Soc. Sci.* 16(3), pp. 215–220

Martins, C. A. (2002). ‘A new learning method of microprocessor architecture’. In *Proceedings of 32nd ASEE/IEEE Frontiers in Education Conference*, November 6–9, 2002, Boston MA.. <http://fie.engrng.pitt.edu/fie 2002/papers/1455.pdf> [accessed 5 June 2008]

Navarro, J.I., Marchena, E. and Alcalde, C. (2004). ‘Stimulus control with computer assisted learning’. *Journal of Behavioral Education*Vol.13 , No. 2, June (2004), pp. 83–91

Phillips, J. (2007). ‘Simulation of a simple CPU design and its use as an instructional tool in a computer organization course’. *J. Comput. Small Coll.* 22, 6 (Jun. 2007), pp 140–146

Rebaudengo, M. and Reorda, M.S. (1998). ‘The training environment for the course on microprocessor systems at the Politecnico di Torino.’ In *Proceedings of the 1998 Workshop on Computer Architecture Education, WCAE '98*.<http://doi.acm.org/10.1145/1275182.1275190>

Sayed Razi, H. (2004). *Machine and Assembly Language in PC Computers*, 5th ed.Tehran: Naqoos Publishing Company. pp. 180–210 (in Farsi)

Stallings, W. (2002). *Computer Organization and Architecture*, 5th ed. New York: Macmillan. pp. 87–89.

Stanley, T.D. and Wang, M. (2005). ‘An emulated computer with assembler for teaching undergraduate computer architecture’. In *Proceedings of the 2005 Workshop on Computer Architecture Education: Held in Conjunction with the 32nd international Symposium on Computer Architecture, WCAE ‘05* (in Madison, WI). <http://doi.acm.org/10.1145/1275604.1275615>

Von Neumann, J., (1945), ‘First Draft of a Report on the EDVAC,’ Contract No. W-670-ORD-4926, U.S. Army Ordnance Department, Philadelphia: University of Pennsylvania, Moore School of Electrical Engineering, 30 June 1945.

Wainer, G.A., Daicz, S., De Simoni, L. F. and Wassermann, D. (2001). ‘Using the Alfa-1 simulated processor for educational purposes’.In *J. Educ. Resour. Comput.* 1, 4 (Dec. 2001). pp. 111–151. <http://doi.acm.org/10.1145/514144.514743>

Yehezkel, C. , Eliahu, M. and Ronen, M. (2003). ‘Learning computer organization and assembly language with the EasyCPU visual environment’. Paper presented at the *IEEE International Conference on Advanced Learning Technologies*, *ICALT ‘03* (in Athens, Greece).

Yehezkel, C. (2002). ‘A taxonomy of computer architecture visualizations’. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, *ITiCSE '02* (in Aarhus, Denmark, June 24–28). pp. 101–105. <http://doi.acm.org/10.1145/544414.544447>

Yu, Y. and Marut, C. (1992). *Assembly Language Programming and Organization of the IBM PC*. Singapore: McGraw-Hill.

Yurcik, W. and Gehringer, E.F. (2002). ‘A survey of web resources for teaching computer architecture’. In *Proceedings of the 2002 Workshop on Computer Architecture Education, WCAE ‘02 (in Anchorage, AK)* . <http://doi.acm.org/10.1145/1275462.1275492>

Yurcik, W. (2001). ‘A survey of simulators used in computer organization/ architecture course’. Paper presented at the *Summer Computer Simulation Conference (SCSC)* (in Orlando FL)..

Yushau B. (2004). *The predictors of success of computer aided learning of pre-calculus algebra.* University of South Africa, PhD Thesis.

Appendix A: Questionnaire

Kabul University Computer Science students' questionnaire
simCPU simulation tool evaluation at computer architecture course

User friendliness (1 = excellent; 5 = poor **)**

No.	Questions	1	2	3	4	5
1	How convenient did you find the 'user interface' of the Simulation tool to use?					

Simulation tool information (1 = excellent; 5 = poor**)**

No.	Questions	1	2	3	4	5
2	How useful did you find the information about 8086 microprocessor to be?					

Easy to use (1 = excellent; 5 = poor**)**

No.	Questions	1	2	3	4	5
3	How easy (overall) did you find the simCPU to use and follow?					

Navigation (1 = excellent; 5 = poor**)**

No.	Questions	1	2	3	4	5
4	How easy did you find navigating through this Simulation tool?					

Concept development (1 = excellent; 5 = poor**)**

No.	Questions	1	2	3	4	5
5	How effective was the simCPU in helping you to improve your understanding of 8086 microprocessor concepts?					

Concept review (1 = excellent; 5 = poor**)**

No.	Questions	1	2	3	4	5
6	How effective was the simCPU in helping you to improve your understanding of assembly instruction set?					

Error reporting (1 = excellent; 5 = poor**)**

No.	Questions	1	2	3	4	5
7	How effective was the simCPU for reporting the error from the written code in code area to you?					

Error recovery (1 = excellent; 5 = poor**)**

No.	**Questions**	1	2	3	4	5
8	How effective was the simCPU for reporting the error to you to correct it?					

Knowledge testing (1 = yes; 5 = no**)**

No.	**Questions**	1	2	3	4	5
9	How effective was the simCPU for reporting the error from the written code in code area to you?					

Hands-on (1= yes; 5 = no**)**

No.	**Questions**	1	2	3	4	5
10	Would you like to have more tools of this kind as part of your course?					

The rest twelve questions responses were described in the following tables:

Students' feedback of the Simulation Tool

No.	**Questions**	**Strongly agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
11	It helped me understand the intricacies of microprocessor operations					
12	It enhanced my ability to apply Assembly Instructions concepts and principles					
13	It helped me understand the relevant lecture material					
14	It increased my programming skills					
15	It increased my knowledge about Registers					

Students' views of their group when using the Simulation Tool

No.	**Questions**	**Strongly agree**	**Agree**	**Neutral**	**Disagree**	**Strongly disagree**
16	All members contributed equally to the work					
17	There was a high level of co-operation in my group					

Students' views comparing the Simulation Tool to other types of learning

No.	**Questions**	**Strongly agree**	**Agree**	**Neutral**	**Disagree**	**Strongly disagree(**
18	It motivated me to a greater Extent					
19	It enabled me to learn more					

Students' views on whether they prefer the Simulation Tool or Traditional learning

20	Overall, given the choice between the Simulation tool and more traditional learning which one would you	Check Your Selection in this column, then explain why in the last column

	prefer?		
	Simulation tool		
	Traditional learning		
	No Preference		

Students' views: why students prefer the computer-aided learning

21	Overall, given the choice between the computers aided learning and the traditional type of learning which one would you prefer and why?	Check all that you believe are true	Comments
	More interesting and enjoyable		
	More practical		
	Facilitates learning process		
	More interactive		
	More motivating		
	Other aspects (%)		

Students' view: worst aspects of the computer-aided learning

22	Here, add any comments you have about the aspects of computer aided learning that you do not like.

Appendix B: Code Generation Class

```
Public Class Codegen
    Dim opcodes As New Hashtable()
    Dim regs As New Hashtable()
    Dim ccon As New Context
    Dim words(10) As String
    Dim type(10) As Integer
    Dim ins_no As Integer
    Dim small_lex As New Lex
    Sub New(ByRef cont As Context, ByRef wrds As String(), ByRef tp
As Integer(), ByVal inst_no As Integer)
        ccon = cont
        Dim j As Integer
        For j = 0 To 9
            words(j) = wrds(j)
            type(j) = tp(j)
            'MsgBox(words(j))
        Next j
        addcodes()
        addregs()
    End Sub
    Private Sub addcodes()
        'declarative instructions
        opcodes.Add("DB", 5)
        opcodes.Add("DW", 6)
        opcodes.Add("PROC", 10)

        'data movement instructions 100 series
        opcodes.Add("LEA", 102)
        opcodes.Add("MOV", 100)
        opcodes.Add("PUSH", 110)
        opcodes.Add("POP", 111)

        'Control instructions 200 Series
        opcodes.Add("INT", 200)
        opcodes.Add("JB", 210)
        opcodes.Add("JE", 211)
        opcodes.Add("JG", 212)
        opcodes.Add("JL", 213)
        opcodes.Add("JMP", 214)
        opcodes.Add("JNC", 215)
        opcodes.Add("JZ", 216)
        opcodes.Add("LODSB", 250)
        opcodes.Add("LOOP", 255)
        opcodes.Add("REPE", 257)
        opcodes.Add("CALL", 300)
        opcodes.Add("RET", 301)
        opcodes.Add("END", 400)
        'Compute instructions 500 series

        'Arithmetic instructions 500 Series
        opcodes.Add("INC", 500)
        opcodes.Add("DEC", 501)
        opcodes.Add("ADD", 502)
        opcodes.Add("CMP", 510)
        opcodes.Add("SUB", 515)
        opcodes.Add("MUL", 520)
```

```
        opcodes.Add("DIV", 525)

        'Logical instructions 600 series
        opcodes.Add("OR", 600)
        opcodes.Add("SHR", 602)
        opcodes.Add("SHL", 603)
        opcodes.Add("XOR", 605)
        'MsgBox(opcodes.Item("MOV"))
    End Sub
    Private Sub addregs()
        'data movement instructions 100 series
        regs.Add("AL", 0)
        regs.Add("AH", 1)
        regs.Add("AX", 2)
        regs.Add("BL", 3)
        regs.Add("BH", 4)
        regs.Add("BX", 5)
        regs.Add("CL", 6)
        regs.Add("CH", 7)
        regs.Add("CX", 8)
        regs.Add("DL", 9)
        regs.Add("DH", 10)
        regs.Add("DX", 11)
        regs.Add("SP", 12)
        regs.Add("BP", 13)
        regs.Add("DI", 14)
        regs.Add("SI", 15)
    End Sub
    Private Function get_case(ByVal st As String) As Integer
        Return opcodes.Item(st.ToUpper)
    End Function
    Private Sub normalize_instr()
        If (type(0) = 5) Then 'first word is a label
            Dim j As Integer
            For j = 2 To 9 'since words(0) is label and words(1) is :
                words(j - 2) = words(j)
                type(j - 2) = type(j)
            Next
        Else
            'the instruction is in normal format
        End If
    End Sub
    Private Sub increment_IP()
        If ccon.IP = 65535 Then
            MsgBox("Segment exceeded. Too many instructions")
            Exit Sub
        Else
            ccon.IP = ccon.IP + 1
        End If
    End Sub
    Private Function read_reg(ByVal r As String) As Integer
        Dim val As Integer
        Dim tempreg As Integer
        'MsgBox(r)
        tempreg = regs.Item(r.ToUpper)
        'MsgBox(tempreg)
        'MsgBox(regs.Item("BX"))
        Select Case tempreg
            Case 0
                val = ccon.AX(0)
            Case 1
```

```
                val = ccon.AX(1)
            Case 2
                val = ccon.AX(1) * 256 + ccon.AX(0)
            Case 3
                val = ccon.BX(0)
            Case 4
                val = ccon.BX(1)
            Case 5
                val = ccon.BX(1) * 256 + ccon.BX(0)
            Case 6
                val = ccon.CX(0)
            Case 7
                val = ccon.CX(1)
            Case 8
                val = ccon.CX(1) * 256 + ccon.CX(0)
            Case 9
                val = ccon.DX(0)
            Case 10
                val = ccon.DX(1)
            Case 11
                val = ccon.DX(1) * 256 + ccon.DX(0)
            Case 12
                val = ccon.SP
            Case 13
                val = ccon.BP
            Case 14
                val = ccon.DI
            Case 15
                val = ccon.SI
        End Select
        Return val
    End Function
    Public Sub execute()
        Dim tag As Integer
        Dim adr As Integer
        Dim temptoken As symtable.item
        Dim tempval As Integer
        increment_IP()
        normalize_instr()
        'MsgBox("here" + words(0))
        If type(0) = 3 Then
            tag = get_case(words(1))
        ElseIf type(0) = 1 Then
            tag = get_case(words(0))
        Else
            MsgBox("Runtime Error. Parsing suspect.")
            Exit Sub
        End If
        'MsgBox(tag)
        Select Case tag
            Case 5 'DB
                If type(2) = 4 Then
                    ccon.mem.write_memory(1,
ccon.symtb.symbol(ccon.symtb.searchtoken(words(0))).address,
words(2))
                    'MsgBox(ccon.mem.read_memory(1,
ccon.symtb.symbol(ccon.symtb.searchtoken(words(0))).address))
                ElseIf type(2) = 7 Then
                    Dim strlen = small_lex.literal_val(words(2))
```

```
                    ccon.mem.write_memory_str(1,
ccon.symtb.symbol(ccon.symtb.searchtoken(words(0))).address,
words(2))
                Else
                    'do nothing
                End If
            Case 6 'DW
                ccon.mem.write_memory(2,
ccon.symtb.symbol(ccon.symtb.searchtoken(words(0))).address,
words(2))
                'MsgBox(ccon.mem.read_memory(2,
ccon.symtb.symbol(ccon.symtb.searchtoken(words(0))).address))
            Case 10 'PROC
                ccon.symtb.addtoken(words(1), 5, ins_no)
            Case 100 'MOV
                If type(1) = 2 And type(3) = 4 Then
                    Dim xx As Integer
                    Dim neg As Boolean = False
                    If words(3).Chars(0) = "-" Then
                        words(3) = words(3).Substring(1,
words(3).Length - 1)
                        neg = True
                    End If
                    xx = small_lex.literal_val(words(3))
                    'this is for the time being
                    If neg Then
                        words(1) = words(1).ToUpper
                        If xx < 256 Then
                            If (words(1) = "AL") Or (words(1) = "BL")
Or (words(1) = "CL") Or (words(1) = "DL") Then
                                xx = compl_2_8(-1 * xx)
                            ElseIf (words(1) = "AX") Or (words(1) =
"BX") Or (words(1) = "CX") Or (words(1) = "DX") Then
                                xx = compl_2_16(-1 * xx)
                            Else
                                MsgBox("Overflow occured while moving
data. Check the value you are moving.")
                            End If
                        ElseIf xx < 65536 Then
                            If (words(1) = "AL") Or (words(1) = "BL")
Or (words(1) = "CL") Or (words(1) = "DL") Then
                                MsgBox("Overflow occured while moving
data. Check the value you are moving.")
                            ElseIf (words(1) = "AX") Or (words(1) =
"BX") Or (words(1) = "CX") Or (words(1) = "DX") Then
                                xx = compl_2_16(-1 * xx)
                            Else
                                MsgBox("Overflow occured while moving
data. Check the value you are moving.")
                            End If
                        Else
                            MsgBox("Type Mismatch")
                        End If
                    End If
                    'upto here is for the time being
                    MOV_R_L(regs.Item(words(1).ToUpper), xx)
                ElseIf type(1) = 2 And type(3) = 3 Then
                    temptoken = ccon.symtb.get_token(words(3))
                    adr = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    MOV_R_L(regs.Item(words(1).ToUpper), adr)
```

```
                ElseIf type(1) = 2 And type(3) = 2 Then
                    MOV_R_L(regs.Item(words(1).ToUpper),
read_reg(words(3)))
                Else
                    MsgBox("Runtime Error: Unable to access
data/resolve source address")
                End If
            Case 102 'LEA
                Dim x As Integer
                Dim ad As Integer
                Dim y As Integer

                ad = ccon.symtb.searchtoken(words(3))
                ad = ccon.symtb.symbol(ad).address
                y = ccon.symtb.symbol(ad).value
                x = ccon.mem.read_memory(ad, y)
                MOV_R_L(words(1), x)
            Case 110 'PUSH
                'MsgBox(words(1))
                If ccon.SP < 65535 Then
                    ccon.SP = ccon.SP + 2
                    If type(1) = 2 Then

ccon.stack_seg.push(read_reg(words(1).ToUpper))
                    ElseIf type(1) = 3 Then
                        temptoken = ccon.symtb.get_token(words(1))
                        adr = ccon.mem.read_memory(temptoken.value,
temptoken.address)

                        ccon.stack_seg.push(adr)
                    Else
                        MsgBox("Error: Cannot move data to stack,
check instruction")
                    End If
                Else
                    MsgBox("Error: Stack Overflow")
                End If
            Case 111 'POP
                Dim x As Integer
                If (ccon.SP > 0) Then
                    If type(1) = 2 Then
                        ccon.SP = ccon.SP - 2
                        x = ccon.stack_seg.pop()
                        MOV_R_L(regs.Item(words(1).ToUpper), x)
                    ElseIf type(1) = 3 Then
                        Dim t As Integer
                        temptoken = ccon.symtb.get_token(words(1))
                        t = temptoken.value
                        ccon.mem.write_memory(t, temptoken.address,
ccon.stack_seg.pop())
                    Else
                        MsgBox("Error: Cannot move data from stack,
check instruction")
                    End If

                Else
                    MsgBox("Error: Trying to pop empty stack.")
                    Exit Sub
                End If
                'Control instructions 200 Series
            Case 200 'INT
            Case 210 'JB
```

```
        If ccon.PSW.Get(0) Then
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 211 'JE
        If ccon.PSW.Get(3) Then
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 212 'JG
        If Not ccon.PSW.Get(4) And Not ccon.PSW.Get(3) Then
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 213 'JL
        If ccon.PSW.Get(4) Then
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 214 'JMP
        Dim jmploc = label_locate(words(1))
        ccon.sys_stack.push(ccon.IP)
        ccon.IP = jmploc
    Case 215 'JNC
        If Not ccon.PSW.Get(0) Then
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 216 'JZ
        If ccon.PSW.Get(3) Then
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 250 'LODSB
    Case 255 'LOOP
        Dim count As Integer
        count = ccon.CX(1) * 256 + ccon.CX(0)
        If count > 0 And count < 32768 Then
            count = count - 1
            MOV_R_L(regs.Item("CX"), count)
            Dim jmploc = label_locate(words(1))
            ccon.sys_stack.push(ccon.IP)
            ccon.IP = jmploc
        End If
    Case 257 'REPE
    Case 300 'CALL
        Dim jmploc = proc_locate(words(1))
        ccon.sys_stack.push(ccon.IP)
        ccon.IP = jmploc
    Case 301 'RET
        ccon.IP = ccon.sys_stack.pop()
    Case 400 'END
        MsgBox("Execution complete")
        ccon.finish = True
        Exit Sub
```

```
                    'Compute instructions 500 series
                    'Arithmetic instructions 500 Series
                Case 500 'INC
                    Dim x As Integer
                    x = read_reg(words(1).ToUpper)
                    x = x + 1
                    If x > 65535 Then
                        ccon.PSW.Set(5, True)
                        x = x Mod 65536
                    End If
                    MOV_R_L(regs.Item(words(1).ToUpper), x)
                Case 501 'DEC
                    Dim x As Integer
                    x = read_reg(words(1).ToUpper)
                    x = x - 1
                    If x < 0 Then
                        ccon.PSW.Set(4, True)
                        x = compl_2_16(x)
                    End If
                    MOV_R_L(regs.Item(words(1).ToUpper), x)
                Case 502 'ADD
                    If type(1) = 2 And type(3) = 4 Then
                        ADD_R_L(regs.Item(words(1).ToUpper),
small_lex.literal_val(words(3)))
                    ElseIf type(1) = 2 And type(3) = 3 Then
                        temptoken = ccon.symtb.get_token(words(3))
                        adr = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                        ADD_R_L(regs.Item(words(1).ToUpper), adr)
                    ElseIf type(1) = 2 And type(3) = 2 Then
                        'MsgBox(words(3))
                        tempval = read_reg(words(3).ToUpper)
                        ADD_R_L(regs.Item(words(1).ToUpper), tempval)
                    Else
                        MsgBox("Runtime Error: Unable to access
data/resolve source address")
                    End If
                Case 510 'CMP
                    Dim xx As Integer
                    Dim yy As Integer
                    Dim result As Integer
                    If type(1) = 2 And type(3) = 2 And words(2) = ","
Then
                        xx = read_reg(words(1).ToUpper)
                        yy = read_reg(words(3).ToUpper)
                    ElseIf type(1) = 2 And type(3) = 3 And words(2) = ","
Then
                        xx = read_reg(words(1).ToUpper)
                        temptoken = ccon.symtb.get_token(words(3))
                        yy = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    Else
                        MsgBox("Error: Cannot compare, check
instruction")
                    End If
                    result = xx - yy
                    If result < 0 Then
                        ccon.PSW.Set(4, True)
                    ElseIf result = 0 Then
                        ccon.PSW.Set(3, True)
                    Else
```

```
                    'do nothing
                End If
            Case 515 'SUB
                If type(1) = 2 And type(3) = 4 Then
                    Dim xx As Integer
                    xx = small_lex.literal_val(words(3))
                    If xx < 255 Then
                        SUBTR(regs.Item(words(1).ToUpper), xx)
                    Else
                        MsgBox("Runtime Error: Cannot subtract word
from byte")
                    End If
                ElseIf type(1) = 2 And type(3) = 3 Then
                    temptoken = ccon.symtb.get_token(words(3))
                    adr = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    If adr < 255 Then
                        SUBTR(regs.Item(words(1).ToUpper), adr)
                    Else
                        MsgBox("Runtime Error: Cannot subtract word
from byte")
                    End If
                ElseIf type(1) = 2 And type(3) = 2 Then
                    tempval = read_reg(words(3).ToUpper)
                    If tempval < 255 Then
                        SUBTR(regs.Item(words(1).ToUpper), tempval)
                    Else
                        MsgBox("Runtime Error: Cannot subtract word
from byte")
                    End If
                Else
                    MsgBox("Runtime error: unable to access data /
resolve source address")
                End If
            Case 520 'MUL
                Dim x As Integer
                If type(1) = 2 Then
                    x = read_reg(words(1).ToUpper)
                ElseIf type(1) = 3 Then
                    temptoken = ccon.symtb.get_token(words(1))
                    x = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                Else
                    MsgBox("Error: Cannot multiply, check the code")
                    Exit Sub
                End If
                x = x * read_reg("AX")
                If x > 4294967295 Then
                    ccon.PSW.Set(5, True)
                    x = x \ 4294967296
                End If
                If x < 65535 Then
                    MOV_R_L(regs.Item("AX"), x)
                Else
                    MOV_R_L(regs.Item("DX"), x \ 65536)
                    MOV_R_L(regs.Item("AX"), x Mod 65536)
                End If
            Case 525 'DIV
                Dim divident As Integer
                Dim divisor As Integer
                Dim quotient As Integer
```

```
                Dim remainder As Integer
                If type(1) = 2 Then
                    If (words(1).ToUpper = "BL") Or (words(1).ToUpper
= "CL") Or (words(1).ToUpper = "DL") Then
                        divident = read_reg("AX")
                        divisor = read_reg(words(1).ToUpper)
                        quotient = divident \ divisor
                        remainder = divident Mod divisor
                        MOV_R_L(regs.Item("AL"), quotient)
                        MOV_R_L(regs.Item("AH"), remainder)
                    ElseIf (words(1).ToUpper = "BX") Or
(words(1).ToUpper = "CX") Then
                        divident = read_reg("DX") * 65536 +
read_reg("AX")

                        divisor = read_reg(words(1).ToUpper)
                        quotient = divident \ divisor
                        remainder = divident Mod divisor
                        MOV_R_L(regs.Item("AX"), quotient)
                        MOV_R_L(regs.Item("DX"), remainder)
                    Else
                        MsgBox("Error: Cannot divide, check the
code")

                        Exit Sub
                    End If
                ElseIf type(1) = 3 Then
                    temptoken = ccon.symtb.get_token(words(1))
                    divisor = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    If temptoken.value = 1 Then
                        divident = read_reg("AX")
                        quotient = divident \ divisor
                        remainder = divident Mod divisor
                        MOV_R_L(regs.Item("AL"), quotient)
                        MOV_R_L(regs.Item("AH"), remainder)
                    ElseIf temptoken.value = 2 Then
                        divident = read_reg("DX") * 65536 +
read_reg("AX")

                        quotient = divident \ divisor
                        remainder = divident Mod divisor
                        MOV_R_L(regs.Item("AX"), quotient)
                        MOV_R_L(regs.Item("DX"), remainder)
                    Else
                        MsgBox("Error: Cannot divide, check the
code")

                        Exit Sub
                    End If
                Else
                    MsgBox("Error: Cannot divide, check the code")
                    Exit Sub
                End If
                'Logical instructions 600 series
            Case 600 'OR
                Dim x As Integer
                Dim y As Integer
                Dim z As Integer
                Dim zz As Integer
                Dim resu As Integer
                Dim st1, st2, st3 As String
                st3 = ""
                MsgBox("in OR")
                If type(1) = 2 And type(3) = 4 Then
```

```
            MsgBox("here in reg-lit")
            x = read_reg(regs.Item(words(1).ToUpper))
            y = small_lex.literal_val(words(3))
            z = x Xor y
            If x <> y Then
                zz = 0
            Else
                zz = x
            End If
            resu = zz Xor z
            MOV_R_L(regs.Item(words(1).ToUpper), resu)
        ElseIf type(1) = 2 And type(3) = 3 Then
            MsgBox("here in reg-mem")
            temptoken = ccon.symtb.get_token(words(3))
            y = ccon.mem.read_memory(temptoken.value,
temptoken.address)
            x = read_reg(regs.Item(words(1).ToUpper) Xor adr)
            resu = x Or y
            'z = x Xor y
            'If x <> y Then
            'zz = 0
            'Else
            'zz = x
            'End If
            'resu = zz Xor z
            ccon.mem.write_memory(temptoken.value,
temptoken.address, resu)
        ElseIf type(1) = 2 And type(3) = 2 Then
            MsgBox("here in or reg-reg")
            y = read_reg(words(3).ToUpper)
            x = read_reg(regs.Item(words(1).ToUpper))
            st1 = num_to_binary_str(x)
            st2 = num_to_binary_str(y)
            While st1.Length > 1 And st2.Length > 1
                If (st1.Chars(0) = "1") Or (st2.Chars(0) =
"1") Then
                    st3 = st3 + "1"
                Else
                    st3 = st3 + "0"
                End If
                st1 = st1.Substring(1, st1.Length - 1)
                st2 = st2.Substring(1, st2.Length - 1)
            End While
            While st1.Length > 1
                st3 = st3 + st1.Chars(0)
                st1.Substring(1, st1.Length - 1)
            End While
            While st2.Length > 1
                st3 = st3 + st2.Chars(0)
                st2.Substring(1, st2.Length - 1)
            End While
            resu = binary_str_to_num(st3)
            'resu = x Or y
            'z = x Xor y
            'If x <> y Then
            ' zz = 0
            ' Else
            ' zz = x
            'End If
            'resu = zz Xor z
            MOV_R_L(regs.Item(words(1).ToUpper), resu)
```

```
                Else
                    MsgBox("Runtime Error: OR operator")
                End If


            Case 602 'SHR
                Dim x As Integer
                If type(1) = 2 Then
                    x = read_reg(words(1).ToUpper)
                    x = x / 2
                    MOV_R_L(regs.Item(words(1).ToUpper), x)
                ElseIf type(1) = 3 Then
                    temptoken = ccon.symtb.get_token(words(1))
                    x = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    x = x / 2
                    ccon.mem.write_memory(temptoken.value,
temptoken.address, x)
                Else
                    MsgBox("Error: Cannot shift")
                End If
            Case 603 'SHL
                Dim x As Integer
                If type(1) = 2 Then
                    x = read_reg(words(1).ToUpper)
                    x = x * 2
                    MOV_R_L(regs.Item(words(1).ToUpper), x)
                ElseIf type(1) = 3 Then
                    temptoken = ccon.symtb.get_token(words(1))
                    x = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    x = x * 2
                    ccon.mem.write_memory(temptoken.value,
temptoken.address, x)
                Else
                    MsgBox("Error: Cannot shift")
                End If
            Case 605 'XOR
                Dim x As Integer
                If type(1) = 2 And type(3) = 4 Then
                    x = (read_reg(regs.Item(words(1).ToUpper))) Xor
(small_lex.literal_val(words(3)))
                    MOV_R_L(regs.Item(words(1).ToUpper), x)
                ElseIf type(1) = 2 And type(3) = 3 Then
                    temptoken = ccon.symtb.get_token(words(3))
                    adr = ccon.mem.read_memory(temptoken.value,
temptoken.address)
                    x = read_reg(regs.Item(words(1).ToUpper) Xor adr)
                    ccon.mem.write_memory(temptoken.value,
temptoken.address, x)
                ElseIf type(1) = 2 And type(3) = 2 Then
                    'MsgBox(words(3))
                    tempval = read_reg(words(3).ToUpper)
                    x = read_reg(regs.Item(words(1).ToUpper)) Xor
tempval
                    MOV_R_L(regs.Item(words(1).ToUpper), x)
                Else
                    MsgBox("Runtime Error: Unable to access
data/resolve source address")
                End If
        End Select
```

```
End Sub

Public Sub MOV_R_L(ByVal reg As Integer, ByVal int As Integer)
    'MsgBox(int)
    Select Case reg
        Case 0 'AL
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.AX(0) = int
            End If
        Case 1 'AH
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.AX(1) = int
            End If
        Case 2 'AX
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.AX(1) = int \ 256
                ccon.AX(0) = int Mod 256
            End If
        Case 3 'BL
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.BX(0) = int
            End If
        Case 4 'BH
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.BX(1) = int
            End If
        Case 5 'BX
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.BX(1) = int \ 256
                ccon.BX(0) = int Mod 256
            End If
        Case 6 'CL
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.CX(0) = int
            End If
        Case 7 'CH
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.CX(1) = int
            End If
        Case 8 'CX
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.CX(1) = int \ 256
```

```
                ccon.CX(0) = int Mod 256
            End If
        Case 9 'DL
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.DX(0) = int
            End If
        Case 10 'DH
            If int > 255 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.DX(1) = int
            End If
        Case 11 'DX
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.DX(1) = int \ 256
                ccon.DX(0) = int Mod 256
            End If
        Case 12 'SP
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.SP = int
            End If
        Case 13 'BP
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.BP = int
            End If
        Case 14 'DI
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.DI = int
            End If
        Case 15 'SI
            If int > 65535 Then
                MsgBox("Runtime Error, destination is smaller")
            Else
                ccon.SI = int
            End If
    End Select
End Sub
Public Sub LEA()

End Sub
Public Sub ADD_R_L(ByVal reg As Integer, ByVal int As Integer)
    Dim sum As Integer
    Dim temp As Integer
    Select Case reg
        Case 0 'AL
            sum = ccon.AX(0) + int
            If sum > 65535 Then
                ccon.PSW.Set(5, True)
            End If
            temp = sum \ 256
            If temp < 256 Then
```

```
            ccon.AX(1) = temp
        Else
            ccon.AX(1) = temp Mod 256
        End If
        ccon.AX(0) = sum Mod 256
    Case 1 'AH
        sum = ccon.AX(1) + int
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.AX(1) = sum Mod 256
        Else
            ccon.AX(1) = sum
        End If
    Case 2 'AX
        sum = ccon.AX(1) * 256 + ccon.AX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.AX(1) = temp
        Else
            ccon.AX(1) = temp Mod 256
        End If
        ccon.AX(0) = sum Mod 256
    Case 3 'BL
        sum = ccon.BX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.BX(1) = temp
        Else
            ccon.BX(1) = temp Mod 256
        End If
        ccon.BX(0) = sum Mod 256
    Case 4 'BH
        sum = ccon.BX(1) + int
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.BX(1) = sum Mod 256
        Else
            ccon.BX(1) = sum
        End If
    Case 5 'BX
        sum = ccon.BX(1) * 256 + ccon.BX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.BX(1) = temp
        Else
            ccon.BX(1) = temp Mod 256
        End If
        ccon.BX(0) = sum Mod 256
    Case 6 'CL
        sum = ccon.CX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
```

```
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.CX(1) = temp
        Else
            ccon.CX(1) = temp Mod 256
        End If
        ccon.CX(0) = sum Mod 256
    Case 7 'CH
        sum = ccon.CX(1) + int
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.CX(1) = sum Mod 256
        Else
            ccon.CX(1) = sum
        End If
    Case 8 'CX
        sum = ccon.CX(1) * 256 + ccon.CX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.CX(1) = temp
        Else
            ccon.CX(1) = temp Mod 256
        End If
        ccon.CX(0) = sum Mod 256
    Case 9 'DL
        sum = ccon.DX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.DX(1) = temp
        Else
            ccon.DX(1) = temp Mod 256
        End If
        ccon.DX(0) = sum Mod 256
    Case 10 'DH
        sum = ccon.DX(1) + int
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.DX(1) = sum Mod 256
        Else
            ccon.DX(1) = sum
        End If
    Case 11 'DX
        sum = ccon.DX(1) * 256 + ccon.DX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.DX(1) = temp
        Else
            ccon.DX(1) = temp Mod 256
        End If
        ccon.DX(0) = sum Mod 256
        'Case 12 'SP
```

```
                'ccon.SP = int
                'Case 13 'BP
                'ccon.BP = int
                'Case 14 'DI
                'ccon.DI = int
                'Case 15 'SI
                'ccon.SI = int
        End Select
    End Sub
    Private Function compl_2_8(ByVal int As Integer) As Integer
        Return 256 + int
    End Function
    Private Function compl_2_16(ByVal int As Integer) As Integer
        Return 65536 + int
    End Function
    Public Sub SUBTR(ByVal REG As Integer, ByVal INT As Integer)
        Dim sum As Integer
        Dim temp As Integer
        Select Case REG
            Case 0 'AL
                sum = ccon.AX(0) - int
                If sum < 0 Then
                    ccon.PSW.Set(4, True)
                    sum = compl_2_16(sum)
                End If
                temp = sum \ 256
                If temp < 256 Then
                    ccon.AX(1) = temp
                Else
                    ccon.AX(1) = temp Mod 256
                End If
                ccon.AX(0) = sum Mod 256
            Case 1 'AH
                sum = ccon.AX(1) - INT
                If sum < 0 Then
                    ccon.PSW.Set(4, True)
                    sum = compl_2_8(sum)
                End If
                ccon.AX(1) = sum
            Case 2 'AX
                sum = ccon.AX(0) - INT
                If sum < 0 Then
                    ccon.PSW.Set(4, True)
                    sum = compl_2_16(sum)

                End If
                temp = sum \ 256
                If temp < 256 Then
                    ccon.AX(1) = temp
                Else
                    ccon.AX(1) = temp Mod 256
                End If
                ccon.AX(0) = sum Mod 256
            Case 3 'BL
                sum = ccon.BX(0) + int
                If sum > 65535 Then
                    ccon.PSW.Set(5, True)
                End If
                temp = sum \ 256
                If temp < 256 Then
                    ccon.BX(1) = temp
```

```
        Else
            ccon.BX(1) = temp Mod 256
        End If
        ccon.BX(0) = sum Mod 256
    Case 4 'BH
        sum = ccon.BX(1) + int
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.BX(1) = sum Mod 256
        Else
            ccon.BX(1) = sum
        End If
    Case 5 'BX
        sum = ccon.BX(1) * 256 + ccon.BX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.BX(1) = temp
        Else
            ccon.BX(1) = temp Mod 256
        End If
        ccon.BX(0) = sum Mod 256
    Case 6 'CL
        sum = ccon.CX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.CX(1) = temp
        Else
            ccon.CX(1) = temp Mod 256
        End If
        ccon.CX(0) = sum Mod 256
    Case 7 'CH
        sum = ccon.CX(1) + int
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.CX(1) = sum Mod 256
        Else
            ccon.CX(1) = sum
        End If
    Case 8 'CX
        sum = ccon.CX(1) * 256 + ccon.CX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.CX(1) = temp
        Else
            ccon.CX(1) = temp Mod 256
        End If
        ccon.CX(0) = sum Mod 256
    Case 9 'DL
        sum = ccon.DX(0) + int
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
```

```
                temp = sum \ 256
                If temp < 256 Then
                    ccon.DX(1) = temp
                Else
                    ccon.DX(1) = temp Mod 256
                End If
                ccon.DX(0) = sum Mod 256
            Case 10 'DH
                sum = ccon.DX(1) + int
                If sum > 255 Then
                    ccon.PSW.Set(5, True)
                    ccon.DX(1) = sum Mod 256
                Else
                    ccon.DX(1) = sum
                End If
            Case 11 'DX
                sum = ccon.DX(1) * 256 + ccon.DX(0) + int
                If sum > 65535 Then
                    ccon.PSW.Set(5, True)
                End If
                temp = sum \ 256
                If temp < 256 Then
                    ccon.DX(1) = temp
                Else
                    ccon.DX(1) = temp Mod 256
                End If
                ccon.DX(0) = sum Mod 256
                'Case 12 'SP
                'ccon.SP = int
                'Case 13 'BP
                'ccon.BP = int
                'Case 14 'DI
                'ccon.DI = int
                'Case 15 'SI
                'ccon.SI = int
        End Select

    End Sub
    Public Sub mul(ByVal REG As Integer, ByVal INT As Integer)
        Dim sum As Integer
        Dim temp As Integer
        Select Case reg
            Case 0 'AL
                sum = ccon.AX(0) * INT
                If sum > 65535 Then
                    ccon.PSW.Set(5, True)
                End If
                temp = sum \ 256
                If temp < 256 Then
                    ccon.AX(1) = temp
                Else
                    ccon.AX(1) = temp Mod 256
                End If
                ccon.AX(0) = sum Mod 256
            Case 1 'AH
                sum = ccon.AX(1) * INT
                If sum > 255 Then
                    ccon.PSW.Set(5, True)
                    ccon.AX(1) = sum Mod 256
                Else
                    ccon.AX(1) = sum
```

```
        End If
    Case 2 'AX
        sum = ccon.AX(1) * 256 + ccon.AX(0) * INT
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.AX(1) = temp
        Else
            ccon.AX(1) = temp Mod 256
        End If
        ccon.AX(0) = sum Mod 256
    Case 3 'BL
        sum = ccon.BX(0) * INT
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.BX(1) = temp
        Else
            ccon.BX(1) = temp Mod 256
        End If
        ccon.BX(0) = sum Mod 256
    Case 4 'BH
        sum = ccon.BX(1) + INT
        If sum > 255 Then
            ccon.PSW.Set(5, True)
            ccon.BX(1) = sum Mod 256
        Else
            ccon.BX(1) = sum
        End If
    Case 5 'BX
        sum = ccon.BX(1) * 256 + ccon.BX(0) + INT
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.BX(1) = temp
        Else
            ccon.BX(1) = temp Mod 256
        End If
        ccon.BX(0) = sum Mod 256
    Case 6 'CL
        sum = ccon.CX(0) + INT
        If sum > 65535 Then
            ccon.PSW.Set(5, True)
        End If
        temp = sum \ 256
        If temp < 256 Then
            ccon.CX(1) = temp
        Else
            ccon.CX(1) = temp Mod 256
        End If
        ccon.CX(0) = sum Mod 256
    Case 7 'CH
        sum = ccon.CX(1) + INT
        If sum > 255 Then
            ccon.PSW.Set(5, True)
```

```
                ccon.CX(1) = sum Mod 256
            Else
                ccon.CX(1) = sum
            End If
        Case 8 'CX
            sum = ccon.CX(1) * 256 + ccon.CX(0) + INT
            If sum > 65535 Then
                ccon.PSW.Set(5, True)
            End If
            temp = sum \ 256
            If temp < 256 Then
                ccon.CX(1) = temp
            Else
                ccon.CX(1) = temp Mod 256
            End If
            ccon.CX(0) = sum Mod 256
        Case 9 'DL
            sum = ccon.DX(0) + INT
            If sum > 65535 Then
                ccon.PSW.Set(5, True)
            End If
            temp = sum \ 256
            If temp < 256 Then
                ccon.DX(1) = temp
            Else
                ccon.DX(1) = temp Mod 256
            End If
            ccon.DX(0) = sum Mod 256
        Case 10 'DH
            sum = ccon.DX(1) + INT
            If sum > 255 Then
                ccon.PSW.Set(5, True)
                ccon.DX(1) = sum Mod 256
            Else
                ccon.DX(1) = sum
            End If
        Case 11 'DX
            sum = ccon.DX(1) * 256 + ccon.DX(0) + INT
            If sum > 65535 Then
                ccon.PSW.Set(5, True)
            End If
            temp = sum \ 256
            If temp < 256 Then
                ccon.DX(1) = temp
            Else
                ccon.DX(1) = temp Mod 256
            End If
            ccon.DX(0) = sum Mod 256
            'Case 12 'SP
            'ccon.SP = int
            'Case 13 'BP
            'ccon.BP = int
            'Case 14 'DI
            'ccon.DI = int
            'Case 15 'SI
            'ccon.SI = int
    End Select

End Sub
Public Sub div()
```

```
End Sub
Private Function label_locate(ByVal labl As String) As Integer
    Dim place As Integer = -1
    Dim result As Boolean
    place = ccon.symtb.searchtoken(labl)
    If (ccon.symtb.symbol(place).type = 5) Then
        Return ccon.symtb.symbol(place).address
    Else
        ccon.symtb.symbol(place).type = 5
        Dim found = False
        Dim i As Integer
        i = ins_no + 1
        While (Not found) And (i < ccon.pgmlen)
            Dim cp As New CodePass(ccon)
            result = cp.locate_token(ccon.instru(i), labl)
            If result Then
                'MsgBox(ccon.symtb.symbol(place).address)
                ccon.symtb.symbol(place).address = i
                'MsgBox(ccon.symtb.symbol(place).address)
                'MsgBox(i)
                found = True
                place = i
            End If
            i = i + 1
        End While
    End If
    Return place
End Function
Private Function proc_locate(ByVal labl As String) As Integer
    Dim place As Integer = -1
    Dim result As Boolean
    place = ccon.symtb.searchtoken(labl)
    If ccon.symtb.symbol(place).type = 5 Then
        Return ccon.symtb.symbol(place).address
    Else
        ccon.symtb.symbol(place).type = 5
        Dim found = False
        Dim i As Integer
        i = ins_no + 1
        While (Not found) And (i < ccon.pgmlen)
            Dim cp As New CodePass(ccon)
            result = cp.locate_proc(ccon.instru(i), labl)
            If result Then
                ccon.symtb.symbol(place).address = i
                found = True
                place = i
            End If
            i = i + 1
        End While
    End If
    Return place
End Function
Private Function num_to_binary_str(ByVal x As Integer) As String
    Dim y As Integer
    Dim digit As Integer
    Dim s As String
    s = ""
    y = x
    While y > 1
        digit = y Mod 2
        If digit = 0 Then
```

```
                s = s + "0"
            Else
                s = s + "1"
            End If
            y = y \ 2
        End While
        Return s
    End Function
    Private Function binary_str_to_num(ByVal str As String) As
Integer
        'Dim i As Integer
        Dim x As Integer
        x = 0
        While str.Length > 1
            If str.Chars(0) = "1" Then
                x = x * 2 + 1
            Else
                x = x * 2
            End If
            str = str.Substring(1, str.Length - 1)
        End While
    End Function
End Class
```

Appendix C: Lexical Analyze Class

```
Imports system.text.RegularExpressions
Public Class Lex
    Dim num_opcode As Integer = 33
    Dim opcode() As String = {"DB", "DW", "INT", "JB", "JE", "JG",
"JL", "JMP", "JNC", "JZ", "LODSB", "LOOP", "REPE", "RET", "STD",
"CALL", "DEC", "INC", "MUL", "POP", "PUSH", "DIV", "ADD", "CMP",
"LEA", "MOV", "OR", "SHR", "SHL", "END", "SUB", "PROC", "XOR"}
    Dim num_reg As Integer = 16
    Dim Register() = {"AX", "BX", "CX", "DX", "AH", "AL", "BH", "BL",
"CH", "CL", "DH", "DL", "SP", "BP", "SI", "DI"}

    Public Function is_opcode(ByVal w As String) As Boolean
        Dim i As Integer = 0
        Dim found = False
        Do
            If w.ToUpper = opcode(i) Then
                found = True
            End If
            i = i + 1
        Loop Until ((found = True) Or (i >= num_opcode))
        Return found
    End Function
    Public Function is_Register(ByVal w As String) As Boolean
        Dim i As Integer = 0
        Dim found = False
        Do
            If w.ToUpper = Register(i) Then
                found = True
            End If
            i = i + 1
        Loop Until ((found = True) Or (i >= num_reg))
        Return found
    End Function
    Public Function is_Label(ByVal wrd As String, ByVal w1 As String)
As Boolean
        'Dim pattern As String = "\b:{1}\b"
        If is_ID(wrd) And w1 = ":" Then
            Return True
        Else
            Return False
        End If
    End Function
    Public Function is_ID(ByVal wrd As String) As Boolean
        Dim pattern As String = "\b[a-zA-Z]{1}\w*\b"
        Dim mc As MatchCollection = Regex.Matches(wrd, pattern)
        If (mc.Count = 0 Or mc.Count > 32) Then
            Return False
        Else
            Return True
        End If
    End Function
    Public Function is_Literal(ByVal wrd As String) As Integer
        Dim pattern1 As String = "\b-?[0-9]{1,}\b"
        Dim pattern2 As String = "\b[0-9a-fA-F]{1}[0-9a-fA-
F]*[hH]{1}\b"
        Dim pattern3 As String = "\b[0-9a-zA-Z]*\b" '"\w*"
        Dim mc1 As MatchCollection = Regex.Matches(wrd, pattern1)
```

```
    Dim mc2 As MatchCollection = Regex.Matches(wrd, pattern2)
    Dim mc3 As MatchCollection = Regex.Matches(wrd, pattern3)
    Dim result As Integer = -1
    'MsgBox("here in is_Literal")
    'MsgBox(mc3.Count)
    If (mc1.Count <> 0) Then
        result = 1
    End If
    If (mc2.Count <> 0) And (result = -1) Then
        result = 2
    End If
    If (mc3.Count <> 0) And (result = -1) Then
        result = 3
    End If
    If wrd.Chars(0) = "'" Then
        result = 3
    End If
    'MsgBox(result)
    Return result
End Function
Private Function hexdigit_todigit(ByVal ch As Char) As Integer
    Select Case ch
        Case "0"
            Return 0
        Case "1"
            Return 1
        Case "2"
            Return 2
        Case "3"
            Return 3
        Case "4"
            Return 4
        Case "5"
            Return 5
        Case "6"
            Return 6
        Case "7"
            Return 7
        Case "8"
            Return 8
        Case "9"
            Return 9
        Case "A"
            Return 10
        Case "B"
            Return 11
        Case "C"
            Return 12
        Case "D"
            Return 13
        Case "E"
            Return 14
        Case "F"
            Return 15
    End Select
End Function
Private Function hextoint(ByVal str As String) As Integer
    Dim x As Char
    Dim i As Integer
    Dim num As Integer
    Dim y As Integer
```

```
        i = 0
        num = 0
        str = str.ToUpper
        While (i < str.Length - 1)
            x = str.Chars(i)
            y = hexdigit_todigit(x)
            num = num * 16 + y
            'MsgBox(num)
            i = i + 1
        End While
        Return num
    End Function
    Public Function literal_val(ByVal wrd As String) As Integer
        Dim pattern1 As String = "\b-?[0-9]{1,}\b"
        Dim pattern2 As String = "\b[0-9a-fA-F]{1}[0-9a-fA-
F]*[hH]{1}\b"
        Dim pattern3 As String = "\w*"
        Dim mc1 As MatchCollection = Regex.Matches(wrd, pattern1)
        Dim mc2 As MatchCollection = Regex.Matches(wrd, pattern2)
        Dim mc3 As MatchCollection = Regex.Matches(wrd, pattern3)
        Dim result As Integer = -1
        If (mc1.Count <> 0) Then
            result = CInt(wrd)
        End If
        If (mc2.Count <> 0) And (result = -1) Then
            result = hextoint(wrd)
        End If
        If (mc3.Count <> 0) And (result = -1) Then
            result = -1
        End If
        If wrd.Chars(0) = "'" Then
            result = wrd.Length - 2
        End If
        Return result
    End Function
End Class
```

Appendix D: Code Pass Class

```
Public Class CodePass
    Dim local_s As New Context
    Dim words(10) As String
    Dim typ(10) As Integer
    Dim val(10) As Integer
    Sub New(ByRef s As Context)
        local_s = s
        'Dim j As Integer
        'For j = 0 To 9
        'words(j) = local_s.words(j)
        'typ(j) = local_s.typ(j)
        'Next j
    End Sub
    Public Function extract_word(ByVal str As String) As Integer
        'MsgBox("in extract word")
        Dim k As Integer
        Dim startindex As Integer
        Dim endofstring As Integer = -1
        Dim slen As Integer
        Dim i As Integer
        Dim num_of_chars As Integer = 0
        For i = 0 To 10
            words(i) = ""
        Next i
        str = LTrim(str)
        str = str + ";"
        slen = Len(str)
        'MsgBox(slen)
        startindex = 0
        k = 0
        'MsgBox(str)
        While ((k < slen) And (str.Chars(k) <> ";"))
            'MsgBox(str.Chars(k))
            k = k + 1
        End While
        endofstring = k
        num_of_chars = endofstring
        'MsgBox(endofstring)
        i = 0
        k = 0
        Dim made As Boolean = False
        While (str.Length > 0)
            If (str.Chars(0) = " ") Then
                If made Then
                    i = i + 1
                    made = False
                End If
                num_of_chars = num_of_chars - 1
                str = str.Substring(1, num_of_chars)
            ElseIf str.Chars(0) = ":" Then
                If made Then
                    i = i + 1
                    made = False
                End If
                words(i) = ":"
                i = i + 1
                num_of_chars = num_of_chars - 1
```

```
                str = str.Substring(1, num_of_chars)
            ElseIf str.Chars(0) = "," Then
                If made Then
                    i = i + 1
                    made = False
                End If
                words(i) = ","
                i = i + 1
                num_of_chars = num_of_chars - 1
                str = str.Substring(1, num_of_chars)
            Else
                words(i) = words(i) + str.Chars(0)
                num_of_chars = num_of_chars - 1
                str = str.Substring(1, num_of_chars)
                made = True
            End If
        End While
        'Dim j As Integer
        'For j = 0 To i
        'MsgBox(words(j))
        'Next j
        Return i
    End Function

    Private Function find_type(ByVal word As String, ByVal w1 As
String) As Integer
        Dim lexthis As New Lex
        Dim typ As Integer
        If (word = "," Or word = ":") Then
            typ = 0
        ElseIf lexthis.is_opcode(word) Then
            typ = 1
        ElseIf lexthis.is_Register(word) Then
            typ = 2
        ElseIf lexthis.is_ID(word) Then
            'MsgBox(word + " " + w1)
            If w1 = ":" Then
                typ = 5
                'MsgBox("here")
            ElseIf word.Chars(0) = "'" Then
                If word.Chars(word.Length - 1) = "'" Then
                    typ = 7
                Else
                    MsgBox("String not terminated")
                End If
            Else
                typ = 3
            End If
        ElseIf lexthis.is_Literal(word) Then
            typ = 4
        Else
            typ = 6
        End If
        'MsgBox(typ)
        Return typ
    End Function
    Private Sub install_token(ByVal wrd As String, ByVal typ As
Integer, ByVal val As Integer, ByVal lineno As Integer)
        'MsgBox("in install_token")
        Dim x As Integer
        If (typ = 3) Then 'it is a identifier
```

```
            'MsgBox("adding identifier")
            x = local_s.symtb.searchtoken(wrd)
            If (x = -1) Then
                'val = 1 'assuming byte operand
                'MsgBox(val)
                local_s.symtb.addtoken(wrd, typ, val)
                'MsgBox("token added" + wrd)

'MsgBox(local_s.symtb.symbol(local_s.symtb.count).token)

'MsgBox(local_s.symtb.symbol(local_s.symtb.count).type)

'MsgBox(local_s.symtb.symbol(local_s.symtb.count).value)

'MsgBox(local_s.symtb.symbol(local_s.symtb.count).address)
            Else
                'If (local_s.symtb.symbol(x).type <> typ) Then = this
is not needed of course
                'MsgBox(local_s.symtb.symbol(x).type)
                'MsgBox(typ)
                'wrd = lineno + 1
                'MsgBox("Inconsistent symbol. Line " + wrd)
                'Exit Sub
                'End If
                'otherwise do not add the token
            End If
        ElseIf typ = 5 Then 'it is a label
            'MsgBox("adding label")
            x = local_s.symtb.searchtoken(wrd)
            If (x = -1) Then
                local_s.symtb.addtoken(wrd, typ, lineno)
            Else
                'do not add the label
            End If
        ElseIf typ = 7 Then
            x = local_s.symtb.searchtoken(wrd)
            If (x = -1) Then

                local_s.symtb.addtoken(wrd, typ, 10)
            End If
        Else
            'MsgBox("nothing to add")
            'do nothing
        End If
    End Sub
    Public Function lexicalise(ByVal instruction As String, ByVal
ins_no As Integer) As Boolean
        'MsgBox("in lexicalize")
        'Dim i As Integer
        Dim j As Integer
        Dim num As Integer
        'If local_s.pgmlen > 0 Then
        num = extract_word(local_s.instru(ins_no))
        'MsgBox(num)
        For j = 0 To num - 1
            typ(j) = find_type(words(j), words(j + 1))
            'MsgBox(typ(j))
            If (typ(j) = 6) Then
                MsgBox("Unrecognizable token found in input. Line " +
ToString(ins_no))
                Return False
```

```
                Exit For
            End If
            If words(j + 1).ToUpper = "DB" Then
                'MsgBox("it is db")
                val(j) = 1
            ElseIf words(j + 1).ToUpper = "DW" Then
                'MsgBox("it is dw")
                val(j) = 2
            ElseIf typ(j) = 1 And typ(j + 1) = 3 Then
                typ(j + 1) = 5
            Else
                'not used so far, to add later
            End If
            install_token(words(j), typ(j), val(j), ins_no)
        Next j
        typ(num) = find_type(words(num), " ")
        install_token(words(num), typ(num), val(num), ins_no)
        Return True
        'End If
    End Function

    Public Function parse() As Boolean
        Dim p As New parser(local_s, words, typ)
        If p.wfinst() Then
            Return True
        Else
            Return False
        End If
    End Function
    Public Function execute(ByVal inst As String, ByVal ins_no As
Integer) As Boolean
        'MsgBox("in Execute")
        Dim line_str As String = ins_no + 1
        If local_s.pgmlen >= 0 Then
            If lexicalise(inst, ins_no) Then
                If parse() Then
                    Dim cdgen As New Codegen(local_s, words, typ,
ins_no)
                    cdgen.execute()
                    'MsgBox("completed instruction" + line_str)
                    'cdgen.show_regs()
                    'Dim ev As New executeVisualize(local_s)
                    Return True
                    'ev.show_before()
                    'ev.execute_inst()
                    'ev.show_after()
                Else
                    MsgBox("Parsing Error in line number" + line_str)
                    Return False
                End If
            Else
                MsgBox("Lexical error in line no" + line_str)
                Return False
            End If
        Else
            MsgBox("No Code")
        End If
    End Function
    Public Function locate_token(ByVal instr As String, ByVal lbl As
String) As Boolean
        Dim num As Integer
```

```
        num = extract_word(instr)
        If words(0) = lbl And words(1) = ":" Then
            Return True
        Else
            Return False
        End If
    End Function
    Public Function locate_proc(ByVal instr As String, ByVal lbl As
String) As Boolean
        Dim num As Integer
        num = extract_word(instr)
        If words(1) = lbl And words(0).ToUpper = "PROC" Then
            Return True
        ElseIf words(1).ToUpper = "PROC" And words(0) = lbl Then
            Return True
        Else
            Return False
        End If
    End Function
End Class
```

Appendix E: Execute and Visualize Class

```
Public Class executeVisualize
    Dim run_context As Context
    Dim temp_ip As Integer
    Dim form As Form1
    Sub New(ByRef frm As Form1, ByRef local_s As Context)
        local_s.IP = 0
        run_context = local_s
        form = frm
        'show_before()
        execute_inst()
        show_after(0)
    End Sub

    Private Function hexdigit(ByVal x As Integer) As String
        Dim st As String = ""
        If x = 0 Then
            st = "0"
        ElseIf x = 1 Then
            st = "1"
        ElseIf x = 2 Then
            st = "2"
        ElseIf x = 3 Then
            st = "3"
        ElseIf x = 4 Then
            st = "4"
        ElseIf x = 5 Then
            st = "5"
        ElseIf x = 6 Then
            st = "6"
        ElseIf x = 7 Then
            st = "7"
        ElseIf x = 8 Then
            st = "8"
        ElseIf x = 9 Then
            st = "9"
        ElseIf x = 10 Then
            st = "A"
        ElseIf x = 11 Then
            st = "B"
        ElseIf x = 12 Then
            st = "C"
        ElseIf x = 13 Then
            st = "D"
        ElseIf x = 14 Then
            st = "E"
        ElseIf x = 15 Then
            st = "F"
        Else
            st = "X"
        End If
        Return st
    End Function
    Private Function tohexbyte(ByVal x As Byte) As String
        Dim st As String
        Dim y As Integer
        y = x \ 16
        st = hexdigit(y)
```

```
        y = x Mod 16
        st = st + hexdigit(y)
        Return st
    End Function
    Private Function tohexint(ByVal x As Integer) As String
        Dim st As String = ""
        Dim st1 As String
        Dim y As Integer
        Dim safex As Integer
        safex = x
        Do
            y = x Mod 16
            st1 = hexdigit(y)
            x = (x \ 16)
            st = st1 + st
        Loop Until (x < 16)
        st = hexdigit(x) + st
        If safex < 256 Then
            st = "00" + st
        ElseIf safex < 4096 Then
            st = "0" + st
        End If
        Return st
    End Function
    Private Sub displaymemory()
        'MsgBox("now")
        Dim sLine As String = ""
        Dim i As Integer
        'Dim rich2 As New TextBox
        form.RichTextBox2.Text = ""
        'RichTextBox2.BorderStyle = BorderStyle.Fixed3D
        form.RichTextBox2.ForeColor = Color.Black
        form.RichTextBox2.BackColor = Color.LightYellow
        For i = 0 To 255
            'RichTextBox2.BackColor = Color.Yellow
            sLine = sLine + "[" + tohexint(i) + "] " +
tohexbyte(run_context.mem.location(i)) + "    " 'vbCrLf
            'sLine = sLine + tohex(s.mem.location(i)) + "   |   "
'vbCrLf
        Next i
        'RichTextBox2.BackColor = Color.Black
        form.RichTextBox2.Text = form.RichTextBox2.Text + sLine
    End Sub

    Public Sub showsymboltable()
        'MsgBox("now")
        Dim sLine As String = ""
        Dim texttoken As String
        Dim i As Integer
        Dim texttype, textaddress, textvalue As String
        'Dim rich2 As New TextBox
        form.RichTextBox3.Clear()
        form.RichTextBox3.Text = ""
        For i = 0 To run_context.symtb.count
            'MsgBox(run_context.symtb.symbol(i).type)
            If run_context.symtb.symbol(i).type <> 5 Then
                texttoken = run_context.symtb.symbol(i).token
                While texttoken.Length < 9
                    texttoken = texttoken + " "
                End While
                texttype = run_context.symtb.symbol(i).type
```

```
                textaddress =
tohexint(run_context.symtb.symbol(i).address)
                'MsgBox(run_context.symtb.symbol(i).address)
                textvalue = run_context.symtb.symbol(i).value
                'sLine = sLine + "  " +
run_context.symtb.symbol(i).token + "  " + texttype + "  " +
textaddress + "  " + textvalue
                sLine = textaddress + " " + texttoken + " "
'run_context.symtb.symbol(i).token + "  " + textaddress + "  "
                form.RichTextBox3.Text = form.RichTextBox3.Text +
sLine
                texttype = ""
                textaddress = ""
                textvalue = ""
            End If
        Next i
        'form.RichTextBox3.Text = form.RichTextBox3.Text + sLine
    End Sub
    Public Sub show_regs()
        Dim sline As String = ""
        sline = tohexbyte(run_context.AX(1))
        form.TextBox1.Text = sline
        sline = tohexbyte(run_context.AX(0))
        form.TextBox2.Text = sline
        form.TextBox3.Text = tohexbyte(run_context.BX(1))
        form.TextBox4.Text = tohexbyte(run_context.BX(0))
        form.TextBox5.Text = tohexbyte(run_context.CX(1))
        form.TextBox6.Text = tohexbyte(run_context.CX(0))
        form.TextBox7.Text = tohexbyte(run_context.DX(1))
        form.TextBox8.Text = tohexbyte(run_context.DX(0))
        form.TextBox9.Text = tohexint(run_context.CS)
        form.TextBox10.Text = tohexint(run_context.DS)
        form.TextBox11.Text = tohexint(run_context.ES)
        form.TextBox12.Text = tohexint(run_context.SS)
        form.TextBox13.Text = tohexint(run_context.DI)
        form.TextBox14.Text = tohexint(run_context.SI)
        form.TextBox15.Text = tohexint(run_context.IP)
        form.TextBox16.Text = tohexint(run_context.BP)
        form.TextBox169.Text = tohexint(run_context.SP)
        'this code show flags, check for the sequence of bits in the
PSW of 8086
        If run_context.PSW.Get(0) Then 'CF
            form.TextBox199.Text = 1
        Else
            form.TextBox199.Text = 0
        End If
        If run_context.PSW.Get(1) Then 'PF
            form.TextBox200.Text = 1
        Else
            form.TextBox200.Text = 0
        End If
        If run_context.PSW.Get(2) Then 'AF
            form.TextBox195.Text = 1
        Else
            form.TextBox195.Text = 0
        End If
        If run_context.PSW.Get(3) Then 'ZF
            form.TextBox196.Text = 1
        Else
            form.TextBox196.Text = 0
        End If
```

```
            If run_context.PSW.Get(4) Then 'SF
                form.TextBox194.Text = 1
            Else
                form.TextBox194.Text = 0
            End If
            If run_context.PSW.Get(5) Then 'OVF - overflow flag
                form.TextBox193.Text = 1
            Else
                form.TextBox193.Text = 0
            End If

        End Sub
        Private Sub show_stack_seg()
            Dim sline As String = "^Top"
            Dim tempstack As New stack
            Dim x As Integer
            Dim y As String
            form.RichTextBox5.Clear()
            While Not (run_context.stack_seg.stackempty())
                x = run_context.stack_seg.pop()
                y = Hex(x)
                sline = y & vbCrLf + sline
                tempstack.push(x)
            End While
            'sline = sline + "TOP"
            form.RichTextBox5.Text = sline
            While Not tempstack.stackempty()
                run_context.stack_seg.push(tempstack.pop())
            End While
        End Sub

        Private Sub execute_inst()

            'set type in the parser itself equivalent to code generation
        End Sub
        Public Sub show_before()
            show_pgm_pointer()
            show_context(run_context)
        End Sub
        Public Sub show_after(ByVal place As Integer)
            code_cue(place)
            show_pgm_pointer()
            show_context(run_context)
        End Sub
        Public Sub show_context(ByRef con As Context)
            show_regs()
            showsymboltable()
            displaymemory()
            show_stack_seg()
        End Sub
        Public Sub show_pgm_pointer()

        End Sub
        Public Sub code_cue(ByVal current As Integer)
            Dim i As Integer
            If run_context.pgmlen > 0 Then
                form.RichTextBox4.Clear()
                For i = 0 To current - 1
    form.RichTextBox4.AppendText(run_context.instru(i).ToUpper & vbCrLf)
                Next i
```

```
            form.RichTextBox4.AppendText("----------------------" & 
vbCrLf)
            form.RichTextBox4.AppendText(Chr(7) + " " + 
run_context.instru(current).ToUpper & vbCrLf)
            form.RichTextBox4.AppendText("----------------------" & 
vbCrLf)
            For i = current + 1 To run_context.pgmlen

form.RichTextBox4.AppendText(run_context.instru(i).ToUpper & vbCrLf)
            Next
        End If
    End Sub
End Class
```

Appendix F: Context Class

```
Public Class symtable
    Structure item
        Dim token As String
        Dim type As Integer
        Dim address As Integer
        Dim value As Integer 'can be float also, to be extended later
    End Structure
    Dim maxsize As Integer = 100
    Public symbol(maxsize) As item
    Public count As Integer
    Public novalue = -9999
    Sub New()
        Dim i As Integer
        count = -1
        For i = 0 To 100
            symbol(i).token = ""
            symbol(i).type = -1
            symbol(i).address = -1
            symbol(i).value = novalue
        Next i
    End Sub
    Public Sub addtoken(ByVal s1 As String, ByVal typ As Integer,
ByVal val As Integer)
        If (count < maxsize - 1) Then
            count = count + 1
            symbol(count).token = s1
            symbol(count).type = typ
            symbol(count).value = val 'here val is the type of data
            If (typ = 3) Then
                symbol(count).address =
memory.set_memory_location(val)
            ElseIf typ = 5 Then
                symbol(count).address = val
                'at the moment label is not stored in memory, keep in
symbol table itself
            ElseIf typ = 7 Then
                symbol(count).address =
memory.set_memory_location(10)
            End If
        Else
            MsgBox("Symbol Table Overflow: Too many tokens in the
Program. Quitting")
        End If
    End Sub
    Public Function searchtoken(ByVal key As String) As Integer
        Dim i As Integer
        For i = 0 To count
            If symbol(i).token = key Then
                Return i
            End If
        Next i
        Return -1
    End Function
    Public Function get_address_of_token(ByVal s1 As String) As
Integer
        Dim place As Integer
        place = searchtoken(s1)
```

```
        If place <> -1 Then
            Return symbol(place).address
        Else
            Return -1
        End If
    End Function
    Public Function find_value_of_token(ByVal s1 As String) As
Integer
        Dim place As Integer
        place = searchtoken(s1)
        If place <> -1 Then
            Return symbol(place).value
        Else
            Return novalue
        End If
    End Function
    Public Sub updatetoken(ByVal name As String, ByVal newval As
Integer)
        Dim x As Integer
        x = searchtoken(name)
        symbol(x).value = newval
    End Sub
    Public Function get_token(ByVal str As String) As item
        Dim place As Integer
        place = searchtoken(str)
        Return symbol(place)
    End Function
End Class
Public Class memory
    Shared maxmemory As Integer = 8192
    Public location(maxmemory) As Byte
    Public Shared usedlist(maxmemory) As Char
    Public Shared current As Integer = -1
    Sub New()
        Dim i As Integer
        For i = 0 To maxmemory
            usedlist(i) = "n"
        Next i
    End Sub
    Public Shared Function set_memory_location(ByVal typ As Integer)
As Integer 'returns the first byte address of allocated memory
        Dim temp As Integer
        Dim i As Integer
        'MsgBox(typ)
        If current = maxmemory Then ' write composite condition using
typ to take care of less memory available than to be allocated
            MsgBox("Memory full. Quitting...")
        Else
            If typ = 1 Then 'character type or byte type
                current = current + 1
                usedlist(current) = "y"
                Return current
            ElseIf typ = 2 Then 'integer type or word type
                current = current + 1
                usedlist(current) = "y"
                temp = current
                current = current + 1
                usedlist(current) = "y"
                Return temp
            ElseIf typ = 3 Then 'Float type
                current = current + 1
```

```
                usedlist(current) = "y"
                temp = current
                For i = 1 To 3
                    current = current + i
                    usedlist(current) = "y"
                Next i
                Return temp
            ElseIf typ = 10 Then 'number of bytes to be allocated is
specified in typ
                current = current + 1
                usedlist(current) = "y"
                temp = current
                For i = 1 To typ
                    current = current + i
                    usedlist(current) = "y"
                Next i
                Return temp
            End If
        End If
    End Function
    Public Function read_memory(ByVal typ As Integer, ByVal address
As Integer) As Integer
        If typ = 1 Then 'character
            Return location(address)
        ElseIf typ = 2 Then 'integer
            Return location(address + 1) * 256 + location(address)
        ElseIf typ = 3 Then 'float
            Return -9999 'here construct the float number using
characteristic and mantiss --- do it later
        End If
    End Function
    Public Sub write_memory(ByVal typ As Integer, ByVal address As
Integer, ByVal value As Integer)
        If typ = 1 Then 'character
            If value < 256 Then
                location(address) = value
            Else
                MsgBox("Cannot Assign Data to Byte Variable")
            End If
        ElseIf typ = 2 Then 'integer
            location(address + 1) = value \ 256
            location(address) = value Mod 256
        ElseIf typ = 3 Then 'float
            MsgBox("storing float yet not coded")
            'here construct the float number using characteristic and
mantissa --- do it later
        End If
    End Sub
    Public Sub write_memory_str(ByVal typ As Integer, ByVal address
As Integer, ByVal value As String)
        If typ = 1 Then 'String copied bytewise
            Dim k As Integer
            k = 1
            While k < value.Length - 1
                location(address + k - 1) =
Microsoft.VisualBasic.Asc(value.Chars(k))
                k = k + 1
            End While
            MsgBox(k)
        Else
            MsgBox("Error writing String to memory")
```

```
            End If
        End Sub
        Public Sub show_memory()
            'here connect to the visualization module that displays the
memory chart on screen
        End Sub
End Class
Public Class Context
        Public mem As New memory
        'Dim line(100) As String
        Public instru(100) As String 'copy the instructions line by line
here
        Public words(10) As String 'the words of the instruction are
placed here
        Public pgmlen = 0
        Public AX(2) As Byte
        Public BX(2) As Byte
        Public CX(2) As Byte
        Public DX(2) As Byte
        Public CS As Integer
        Public DS As Integer
        Public ES As Integer
        Public SS As Integer
        Public SI As Integer
        Public BP As Integer
        Public DI As Integer
        Public PSW As New BitArray(16)
        Public IP As Integer
        Public SP As Integer
        Public E As Byte 'only one bit should be used
        Public CF = PSW.Get(0)
        Public PF = PSW.Get(1)
        Public AF = PSW.Get(2)
        Public ZF = PSW.Get(3)
        Public SF = PSW.Get(4)
        Public OVF = PSW.Get(5)
        Public symtb As New symtable
        Public sys_stack As New stack
        Public stack_seg As New stack
        Public finish As Boolean
        'Public codegen As Integer
        Sub New()
            Dim i As Boolean = False
            PSW.SetAll(i)
            'MsgBox(PSW.Get(0))
            finish = False
        End Sub
End Class
```

Appendix G: Parser Class

```
Public Class parser
    Dim local_cont As New Context
    Dim word(10) As String
    Dim typ(10) As Integer
    Sub New(ByRef cont As Context, ByRef wrds As String(), ByRef type
As Integer())
        local_cont = cont
        Dim j As Integer
        For j = 0 To 10
            word(j) = wrds(j)
            typ(j) = type(j)
        Next
        'MsgBox(wrds(0))
    End Sub
    Private Function is_SRC(ByVal tp As Integer) As Boolean
        If ((tp = 2) Or (tp = 3) Or (tp = 4)) Then 'register,
identifier and literal
            Return True
        Else
            Return False
        End If
    End Function
    Private Function is_DST(ByVal tp As Integer) As Boolean
        If ((tp = 2) Or (tp = 3)) Then 'register or identifier
            Return True
        Else
            Return False
        End If
    End Function
    Private Function is_Compute_instr() As Boolean
        If (is_arithmetic_inst() Or is_logical_inst() Or
is_datamove_inst()) Then
            Return True
        Else
            Return False
        End If
    End Function
    Private Function is_arithmetic_inst() As Boolean
        If (typ(0) = 5) Then
            If word(1) = ":" Then
                If (word(2).ToUpper = "DEC") Or (word(2).ToUpper =
"INC") Then
                    If typ(3) = 2 Or typ(3) = 3 Then
                        Return True
                    Else
                        Return False
                    End If
                ElseIf (word(2).ToUpper = "MUL") Or (word(2).ToUpper
= "DIV") Or (word(2).ToUpper = "ADD") Or (word(2).ToUpper = "SUB") Or
(word(2).ToUpper = "CMP") Then
                    Return True '"POP", "PUSH"
                Else
                    Return False
                End If
            Else
                Return False
            End If
```

```
        Else
            If (word(0).ToUpper = "DEC") Or (word(0).ToUpper = "INC")
Then
                If typ(1) = 2 Or typ(1) = 3 Then
                    Return True
                Else
                    Return False
                End If
            ElseIf (word(0).ToUpper = "MUL") Or (word(0).ToUpper =
"DIV") Or (word(0).ToUpper = "ADD") Or (word(0).ToUpper = "SUB") Or
(word(0).ToUpper = "CMP") Then
                Return True
            Else
                Return False
            End If
        End If
    End Function
    Private Function is_logical_inst() As Boolean
        word(0) = word(0).ToUpper
        If word(0) = "OR" Or word(0) = "SHR" Or word(0) = "XOR" Or
word(0) = "SHL" Then
            Return True
        Else
            Return False
        End If
        ' "OR", "SHR", "XOR"
    End Function
    Private Function is_datamove_inst() As Boolean
        If (typ(0) = 5) Then
            If word(1) = ":" Then
                If (word(2).ToUpper = "MOV") Then
                    If is_DST(typ(3)) And (word(4) = ",") And
is_SRC(typ(5)) Then
                        Return True
                    Else
                        Return False
                    End If
                ElseIf (word(2).ToUpper = "LEA") Then
                    If is_DST(typ(1)) And (word(2) = ",") And
is_SRC(typ(3)) And (typ(1) = 2) And (typ(3) = 3) Then
                        Return True
                    Else
                        Return False
                    End If
                Else
                    Return False
                End If
            Else
                Return False
            End If
        ElseIf (word(0).ToUpper = "MOV") Then
            If is_DST(typ(1)) And (word(2) = ",") And is_SRC(typ(3))
Then
                Return True
            Else
                Return False
            End If
        ElseIf (word(0).ToUpper = "LEA") Then
            If is_DST(typ(1)) And (word(2) = ",") And is_SRC(typ(3))
And (typ(1) = 2) And (typ(3) = 3) Then
                Return True
```

```
            Else
                Return False
            End If
        ElseIf (word(0).ToUpper = "PUSH") Then
            'MsgBox("IN PUSH")
            If typ(1) = 2 Or typ(1) = 3 Then
                Return True
            Else
                Return False
            End If
        ElseIf (word(0).ToUpper = "POP") Then
            If typ(1) = 2 Or typ(1) = 3 Then
                Return True
            Else
                Return False
            End If
        Else
            Return False
        End If
    End Function
    Private Function is_Control_instr() As Boolean
        If (is_End_inst() Or is_Start_inst() Or is_jump_inst()) Then
            Return True
        Else
            Return False
        End If
    End Function
    Private Function is_End_inst() As Boolean
        If (word(0).ToUpper = "END") Or (word(0).ToUpper = "ENDP")
Then
            'MsgBox("yes")
            Return True
        Else
            Return False
        End If
    End Function
    Private Function is_Start_inst() As Boolean
        'this can be used for other cases like .code etc.
    End Function
    Private Function is_jump_inst() As Boolean
        If typ(0) = 5 Then
            If word(1) = ":" Then
                word(2) = word(2).ToUpper
                If word(2) = "RET" Then
                    Return True
                End If
                If (word(2) = "JB") Or (word(2) = "JE") Or (word(2) =
"JG") Or (word(2) = "JL") Or (word(2) = "JMP") Or (word(2) = "JNC")
Or (word(2) = "JZ") Or (word(2) = "CALL") Then
                    If typ(3) = 5 Then
                        Return True
                    Else
                        Return False
                    End If
                Else
                    Return False
                End If
            Else
                Return False
            End If
        ElseIf typ(0) = 1 Then
```

```
            'MsgBox("here 1")
            word(0) = word(0).ToUpper
            If word(0) = "RET" Then
                Return True
            End If
            If (word(0) = "JB") Or (word(0) = "JE") Or (word(0) =
"JG") Or (word(0) = "JL") Or (word(0) = "JMP") Or (word(0) = "JNC")
Or (word(0) = "JZ") Or (word(0) = "CALL") Or (word(0) = "LOOP") Then
                'MsgBox(typ(1))
                Dim x As Integer
                x = local_cont.symtb.searchtoken(word(1))
                If (x = -1) Then
                    Return False
                Else
                    Return True
                End If
            Else
                Return False
            End If
        Else
            'MsgBox("Unexpected token, resolving instruction")
        End If
    End Function
    Private Function is_decl_instr() As Boolean
        If (is_data_dec_inst() Or is_proc_dec_inst()) Or
is_proc2_dec_inst() Then
            Return True
        Else
            Return False
        End If
    End Function
    Private Function is_data_dec_inst() As Boolean
        If (typ(0) = 3) Then
            If (word(1).ToUpper = "DB") Or (word(1).ToUpper = "DW")
Then
                If (typ(2) = 4) Or (word(2) = "?") Or (typ(2) = 7)
Then
                    Return True
                Else
                    Return False
                End If
            Else
                Return False
            End If
        Else
            Return False
        End If
    End Function
    Private Function is_proc_dec_inst() As Boolean
        If (typ(0) = 1) Then
            If (word(0).ToUpper = "PROC") Then
                If ((word(1).ToUpper = "FAR") Or (word(1).ToUpper =
"NEAR")) Then
                    If (word(2) <> "") Then
                        Return True
                    Else
                        Return False
                    End If
                ElseIf word(1) <> "" Then
                    Return True
                Else
```

```
                    Return False
                End If
            Else
                Return False
            End If
        Else
            Return False
        End If
    End Function
    Private Function is_proc2_dec_inst() As Boolean
        If (typ(0) = 3) Then
            Dim temp As String
            If (word(1).ToUpper = "PROC") Then
                If ((word(2).ToUpper = "FAR") Or (word(2).ToUpper =
"NEAR")) Then
                    If (word(3) <> "") Then
                        temp = word(0)
                        word(0) = word(1)
                        word(1) = temp
                        Return True
                    Else
                        Return False
                    End If
                ElseIf word(1) <> "" Then
                    temp = word(0)
                    word(0) = word(1)
                    word(1) = temp
                    Return True
                Else
                    Return False
                End If
            Else
                Return False
            End If
        Else
            Return False
        End If
    End Function
    Public Function wfinst() As Boolean
        If is_Compute_instr() Or is_Control_instr() Or
is_decl_instr() Then
            Return True
        Else
            Return False
        End If
    End Function
End Class
```

Appendix H: Stack Class

```
Public Class stack
    Dim elements(100) As Integer
    Dim top As Integer
    Sub New()
        top = -1
    End Sub
    Public Function stackempty() As Boolean
        If top = -1 Then
            Return True
        End If
        Return False

    End Function
    Private Function stackfull() As Boolean
        If top = 100 Then
            Return True
        End If
        Return False

    End Function
    Public Function push(ByVal x As Integer)
        If Not (stackfull()) Then
            top = top + 1
            elements(top) = x
        Else
            MsgBox("Stack Full error")

        End If
        Return 0
    End Function
    Public Function pop() As Integer
        Dim x As Integer
        If Not (stackempty()) Then
            x = elements(top)
            ' MsgBox(x)
            top = top - 1
            Return x
        Else
            MsgBox("Stack Empty Error")
        End If
    End Function
End Class
```