A Revised Assembly Language Programming Course for a Computer Engineering Technology Program

Dean Lance Smith, Robert Douglas
The University of Memphis

Abstract

A new text was selected which teaches programming and uses the 80x86 family assembly language as the vehicle. Laboratory exercises have been written or revised to support the text. Students assemble and run the programs on new networked Microsoft Windows NT personal computers. The programs are assembled with Microsoft MASM 6.11. Microsoft Visual C++ Professional version 4.0 is used to assemble the software when assembly is mixed with C.

I. Introduction

TECH 3251, Assembly Language Programming, is a required first semester senior class in Computer Engineering Technology. Other engineering technology students take the course as an elective. The class has three 55-minute lecture periods and three hours of unscheduled laboratory each week during a 14 week semester. The students receive 4 hours credit for the course. Prerequisites include programming in C and Pascal, and an introductory microprocessors course on the 80x86 family of processors. Assembly Language Programming is offered every fall during the day and every two and one-half years in the evening.

A new text[1] was selected for the course to replace the previous text[2], and the laboratory exercises were revised to accommodate the new text. Laboratory exercises are the focus of the course. Grades assigned to the laboratory exercises count one-third of the course grade. The exercises emphasize programming and the reuse of existing code. Lectures are used to go over reading assignments and discuss some short problem and question assignments. Often, the students write short assembly language programs in class that help them do the laboratory exercises. The solutions to the in-class programming exercises are critiqued in class.

II. Grading

Table 1 shows the grading criteria used for the laboratory exercises. Up to 100 points can be earned for each assignment. Assignments turned in late receive no credit. The more a student accomplishes and the higher the quality of the accomplishments, the better the score. For example, up to 10 points can be earned for high quality documentation. All procedures will have a header block that describes a) what the procedure does, b) the procedure's inputs, c) the procedure's outputs, and d) what functions or procedures are called. Significant action blocks or data definitions will have adequate comments if they are not self-documenting. (e.g. NOMATCH DB "Strings do not match." is an example of a self-documenting data definition.)

All errors are identified on graded laboratory exercises, and only the first two exercises are not collected and graded. Errors made by two or more students are discussed in class when graded exercises are returned.

The University mandates a two-hour final exam for the course. Two 55-minute exams are given during the semester to help prepare the students for the final exam.

Each 55-minute exam has a programming problem based on the laboratory exercises that counts approximately 25% of the points available on the exam. Any student who understands the laboratory exercise solutions should be able to earn most of the points for the problem.

At least one problem on a 55-minute exam will have code that has one or more errors in it. Students are expected to correct the code. Again, any student who understands the laboratory solutions should be able to earn most of the points for this type of problem.

Twenty-five percent of each exam consists of sentence correction problems[3]. Each sentence states or misstates a major concept of assembly language programming. The sentence correction problems and the graded written comments on the laboratory exercises are the writing samples that are collected and graded to comply with both ABET's (Accreditation Board for Engineering and Technology) and the University's writing expectations.

The remaining exam problems are short answer questions based primarily on concepts that should have been learned doing the laboratory exercises. A final exam is equivalent to two hour exams and is counted that way in course grading.

III. Exercises

Table 2 list the exercises used during the Fall 1998 semester. Several of the exercises are new, but some are revisions of existing exercises.

Exercise 1 expects students to enter a working program with Microsoft Programmers WorkBench (PWB), assemble the program with Microsoft MASM 6.11, and run the program on Microsoft Windows NT. Most of the students have already done this in a previous class. However, some junior college student may not have had this experience, or at least have not assembled 80x86 mnemonics with MASM 6.11 and run the machine code in a Windows NT environment.

The students are given procedures that will perform character and string input and output through the keyboard and console of an IBM PC compatible computer. The listing for the procedures includes a main program that permits strings to be entered and printed on the console, and assembler directives for the data, code, and stack segments that the students can use in their own programs. The students will find the procedures useful for getting data from the keyboard and printing results on the console in later exercises. This exercise is not collected and graded since a student must complete the exercise to successfully complete the remaining exercises.

The students trace the .EXE file from Exercise 1 with Microsoft CodeView in Exercise 2. The .LST file from Exercise 1 is used to determine where to insert break points, single-step through the program, trace loops, examine registers, and examine memory locations. Next, the students reassemble the program with the /Zd option and examine the machine code with the partial symbolic information produced by this option. Finally, the students reassemble the program with the /Zi option to get line numbers and complete symbolic information, and examine the code with the complete information.

The students use immediate addressing and most of the data segment addressing modes to print a test message on the screen in Exercise 3. They are told, with words, how to write the statements that get each character from a string and print the string on the console. The purpose of the exercise is to give the students practice in coding and using the complicated data segment addressing modes in the 80x86 family of processors.

Exercise 4 is the first real programming exercises. Students are expected to modify the program supplied with Exercise 1 so that the program will a) ask the user what string they wish to enter, b) skip a line, c) get the string from the user, d) skip a line, and e) print the string supplied by the user.

The students write and test two procedures in Exercise 5. One will print a string from the code segment. The other will print a string from the extra segment. The purpose of the exercise is to give the students practice using segment override prefixes and setting up the segment directives for the extra segment. A side effect is to teach the students that code and data in the code segment cannot be mixed without caution. Data in the code segment must be out of the flow of execution.

A colored rectangle is drawn on the video display in Exercise 6. The students are expected to write and link several procedures. One will set the display to the proper mode (video or text). Another procedure will put a pixel on the graphics display. A third will draw a horizontal line on the graphics display. A fourth procedure will draw a vertical line. The linking of procedures is fairly complicated for students at this stage of their career, particular the weaker students. However, the visual output seems to motivate many of the weaker students to at least produce a working solution, if not a well designed working solution.

A program is written for Exercise 7 that asks the user for two strings, then compares the strings to determine if they are identical. The students are expected to use two 80x86 family string instructions. SCASB is used to scan each string, locate the end, and determine the length of each string entered. (Strings entered with the functions supplied with Exercise 1 end with a null.) CMPSB is used to compare the strings character-by-character. The students get experience overlaying the data and extra segments and using the REPx prefix for string instructions.

In Exercise 8 the students convert an ASCII (American Standard Code for Information Interchange) character for a digit to BCD (Binary Coded Decimal), then convert the BCD digit back to ASCII and print the result. The students must use CodeView to verify their program is working, otherwise it is easy to print the input data and believe the program is working. The program is a simple test of the use of arithmetic and/or logical instructions in addition to being a good application for CodeView.

The students use a look-up table in Exercise 9 to convert an ASCII character for a hexadigit entered through the keyboard to its binary value, then back to its ASCII equivalent which is displayed on the console. Again, the exercise is a good application for CodeView. The students use the 80x86 XLAT instruction to do the conversion from binary to ASCII and an exhaustive search using SCASB to do the conversion from ASCII to binary. Failure to find the ASCII value in the table is a good error check technique for data entry. Only one table of ASCII values for 0 to F is needed.

Exercise 10 prepares the students for mixing assembly with higher level languages. Exercise 8 is repeated. However, arguments are passed on the stack to procedures that do the conversions, and the return values are also passed on the stack. Students have the option of having either the calling routine or the procedure clean up arguments from the stack. Register indirect addressing using the BP (base pointer) registers is used by the procedures to address the arguments and return values on the stack.

The students combine C and assembly language in Exercise 11. In-line assembly with a compiler directive, such as asm, was used in the past with the Borland C/C++ compilers. Unfortunately, problems were encountered using the in-line assembly directive, _asm, with the Microsoft Visual C++ Professional Version 4.0 compilers installed on the new equipment. Not all assembly language mnemonics compile correctly when preceded by _asm. None of the BIOS or MS-DOS interrupt instructions executed properly. There were no compiler errors, but Windows NT intercepted the interrupt calls and terminated the program. The same problem occurred with Windows 95. Many other instructions seem to execute correctly. As this is written, an acceptable patch is not available from Microsoft.

IV. The Future of the Course

If recent plans are approved, TECH 3251 will cease to be a required course, probably after the Fall 1999 semester. Approximately the first one-third of the course will be incorporated into the course on microprocessors. Most of the last one-third of the course will be incorporated into a course on computer interfacing. Assembly Language Programming will become an elective, although it probably will not be offered after the Fall 1999 semester.

V. Conclusions

The techniques and the exercises have proven effective. Exercise 2, on the use of CodeView, and Exercise 11, on embedding assembly programming in Visual C++ Professional, still need revision. A permanent solution for Exercise 11 may be to compile a main program in C, assemble interface routines in assembly, and link the modules before execution.

References

1. Peter Abel, *IBM PC Assembly Language and Programming*, 4th ed. Upper Saddle River, NJ: Prentice-Hall, 1998. (ISBN 0-13-756610-7)

2. Barry B. Brey, *The Intel 32-Bit Microprocessors, 80386, 80486, and Pentium*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

3. G. J. Lipovski, *Single- and Multiple-Chip Microcomputer Interfacing*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

DEAN LANCE SMITH
Dean Lance Smith received B.S.E.E., M.S.E.(E.E.), and Ph.D. (electrical engineering) degrees from The University of Michigan. Dr. Smith has taught in both engineering technology and engineering programs at several universities, and is currently an Assistant Professor of Engineering Technology at The University of Memphis. Dr. Smith has held summer faculty fellowships with the U.S. Air Force and NASA, and worked for several industrial firms, including his own engineering consulting firm. Dr. Smith is a senior member of the IEEE, and a member of the ACM and ASEE. He has held several section and chapter offices in the IEEE and other engineering organizations. He received the Teetor Award from the SAE for outstanding teaching and three service awards from the Memphis Section of the IEEE. He is a registered engineer in Texas, Louisiana, and Illinois and holds an FCC General Radiotelephone license with a ship radar endorsement.

ROBERT DOUGLAS
Robert Douglas received a B.S.E.E. from The University of Mississippi in 1962 and an M.S.E.E. from The University of Houston in 1967. He has taught engineering technology at Mississippi State University, and is currently an Associate Professor of Engineering Technology at The University of Memphis. Mr. Douglas has been a Manager of Systems Engineering for Ingalls Shipbuilding Division of Litton Industries, a Senior Engineer with General Electric and NASA/JSC, and an engineer with HD Electronics and Texaco Research Laboratory. He served in the USAF. Mr. Douglas has won several awards for his work in industry. He is a member of the ASEE and Tau Alpha Pi, and a registered engineer in Mississippi and Tennessee.

Table 1
Program Grading Criteria

Feature		Points
1. Listing of program submitted on time (No partial credit)		20
2. Test results submitted on time (No partial credit)		20
3. Program works as submitted		20
4. Program style		20
a. Efficient use of language	10	
b. Adequate remarks or comments	10	
5. Program tests		20
a. Adequate test plan	10	
b. Execution of that plan	10	

Total Points		100

Table 2
Laboratory Exercises for the Fall 1998 Semester

1 - Using the Editor and MASM

2 – Codeview

3 – Data Segment Addressing Modes

4 – Getting Strings from the Keyboard and Printing Them on the Console

5 – Segment Override Prefixes

6 – Video Displays

7 – String Instructions

8 – ASCII to Binary Coded Decimal and Binary Coded Decimal to ASCII Conversion

9 – Converting an ASCII Character for a Hexadigit to Binary and an Unpacked Hexadigit to ASCII Using Look-Up Tables

10 – Stack Addressing and Procedures

11 – Embedding Assembly Language in C