SIMPLIFIED ASSEMBLY LANGUAGE PROGRAMMING

James T. Streib
Department of Computer Science
Illinois College
Jacksonville, IL 62650
jtstreib@hilltop.ic.edu

ABSTRACT

Since it is sometimes difficult for students to make the transition from programming in high-level languages to low-level languages, it is important that steps should be taken to simplify the assembly language programming process. First, this paper briefly looks at ways to simplify register usage. Next, some possibilities to simplify input/output are considered and ways to structure assembly language are examined. Lastly, this paper reports some informal student responses to the structured programming techniques. Although this paper examines techniques that apply specifically to Intel assembly language, some of the suggestions are applicable to other microprocessors and various mainframes.

INTRODUCTION

Although the market for assembly programmers is much smaller than it was in the past, there still appears to be a need to learn assembly language programming. Probably the most important reason to learn assembly language programming is that it is a good educational tool that allows students to learn about the inner workings of a computer. Given the one to one relationship between assembly language and machine language, students can gain an understanding of such topics as registers, addressing schemes, instruction formats, and machine language in a computer systems course.

The computer systems course, what previously might have been called the assembly language course and where aspects of assembly language are taught, is often scheduled during the sophomore year of an undergraduate curriculum. Usually, the prerequisite is at least one semester of high-level programming and often two semesters of high-level programming experience in the Computer Science I and II sequence taken during the student's first year. Some college curriculums even require a course in digital circuits prior to a computer systems course. At the author's institution, students take Computer Science I and II as prerequisites using the Delphi and/or C++ programming languages, and many students take Data and File Structures concurrently using the C++ programming language.

Although the intention of a computer systems course should not be to discourage students from continuing in the field of computer science, it is sometimes difficult for students to learn assembly language programming. As a result, the course might give the impression of being a

weeder course. If we accept the above, that assembly language is probably a good way for computer science students to learn about computer systems and that it is also difficult for some students to learn assembly language, then methods to make assembly language simpler for students should be explored.

SIMPLIFIED REGISTER USAGE

One of the first things in assembly language programming that can cause difficulty with students is register usage. As has happened at least once, a student has filled all the registers with various values and asked the instructor, "I have used up all the registers, now what should I do?" Although one of the reasons for programming in assembly language is to use registers effectively to speed up execution time, this can be confusing to new students. As a result, a possible solution is to have students convert the high-level instructions to low-level instructions, one at a time, without being overly concerned about redundant register to memory and memory to register transfers. Although this causes the creation of some unnecessary code and slows execution time, it helps eliminate the above mentioned problem. At a later time, students can then be shown how to optimize their assembly code and avoid costly references to memory. The following are examples of C++ code, non-optimized assembly code, and optimized assembly code.

C++ Code	Assembly Code	Optimized Assembly Code
total = total + amount;	mov ax, total	mov ax,total
balance = total - debits;	add ax, amount	add ax,amount
	mov total,ax	mov total,ax
	mov ax,total	sub ax,debits
	sub ax,debits	mov balance,ax
	mov balance,ax	

SIMPLIFIED INPUT/OUTPUT

Another technique to simplify assembly language programming is to make the I/O simpler for the student, since it can be very difficult at the assembler level. Given today's modern, visual programming environments, assembly language I/O can appear quite archaic and can be very frustrating to the novice assembly programmer.

One possibility is to create subroutines or macros that perform the I/O and have the students just call or invoke the various routines, respectively. The mechanics of how procedures or macros work and the details of the I/O can be deferred until later in the course. More than one enterprising computer science instructor has probably tried to put together a library of either procedures or macros to make life a little easier for their students.

Some might argue, however, that an understanding of how I/O works in assembly language is also an important concept of how computer systems work. As an alternative, the basics of I/O can be explained and simple macros using parameters can be introduced early in the course. An advantage with using parameters is that the values can automatically be stored

in specified variables and the code can be better integrated with the simplified register technique mentioned previously. The result is that the students will have at least some knowledge of how I/O works, how simple macros work, and yet the code is cleaner. Unfortunately, many texts introduce macros and parameters in a later chapter, so there may need to be some jumping around in the text. An example of simplifying I/O using macros and parameters is given immediately below.

```
Straight Assembly Code          Macro Definition                Simplified Assembly Code
    mov ah,1                    input   macro parm                  input value
    int 21h                             mov ah,1
    mov value,al                        int 21h
                                        mov parm,al
                                        endm
```

New software products have made I/O even easier, such as Inprise's (Borland's) Turbo C++ compiler which has an in-line assembler called BASM (Built-in Assembler).[1] For example, it is easier to input an integer via a C++ cin statement, invoke an in-line assembler code segment, and then later display the output using a C++ cout statement as demonstrated below. Now, the focus of the student is on register usage and not some cryptic assembly code just to input a numeric character, convert it to a binary integer, and so on. The disadvantage is that the student is unaware of the techniques necessary to perform I/O in low-level programming, but again this could be explained later in the course after the student has had more experience with low-level programming techniques.

```
C++ Code                C++ with BASM
cin>>dog;               cin>>dog;
cat=dog;                asm
cout<<cat<<endl;        {
                        mov ax,dog
                        mov cat,ax
                        }
                        cout<<cat<<endl;
```

STRUCTURED PROGRAMMING TECHNIQUES

In addition to using simplified I/O, another way to improve student understanding of assembly language programming is not to undo what the student has already learned in the previous one or two courses that are a prerequisite to the computer systems course. In particular, the prerequisite courses have usually stressed structured programming techniques (and possibly OOP), where not only are the programs structured in terms of using sub-programs, but also structured in the sense of limited or no use of the GOTO statement. When students have had previous classes without the GOTO, and then they are introduced to the unconditional jump or branch instruction, they tend to have many difficulties writing code with a limited number of logic errors.

A possible solution is to use labels that help describe the high-level structure and assist in the structured use of the GOTO statement. This is accomplished by using unconditional jumps