

ASSEMBLY PROGRAMMING ON A VIRTUAL COMPUTER

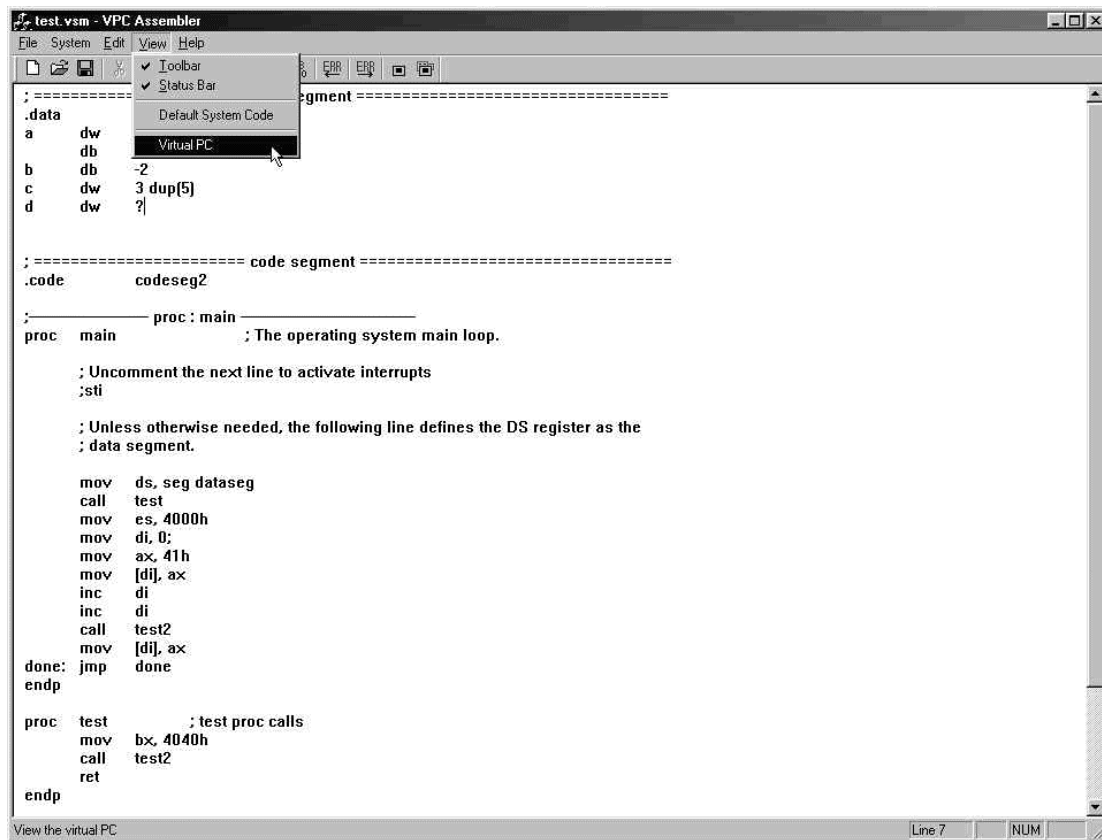
Pierre A. von Kaenel
Mathematics and Computer Science Department
Skidmore College
Saratoga Springs, NY 12866
(518) 580-5292
pvonk@skidmore.edu

ABSTRACT

This paper introduces the VPC Assembler, a Windows 95/98 assembly programming environment that targets the Virtual PC, a simulator of a small computer system based on the Intel 8086 architecture [1]. The assembler provides an editor, a debugger, and views of the assembly program's variables, the CPU's registers, and the Virtual PC's output. The VPC Assembler was designed as a learning tool for courses that introduce assembly programming or for courses, such as computer architecture or organization, that briefly cover the fundamentals of assembly.

1 INTRODUCTION

A number of years ago, Skidmore College included a course in assembly programming in its core CS curriculum. With time, however, programming at the assembly level on Intel-based computers running Microsoft Windows proved problematic, especially on lab PCs that were protected from user hacking with utilities such as Fool Proof. In addition, the inclusion of assembly programming as a core course seemed to become less important in computer science education. Assembly, however, is still an important topic when studying computer organization or any area that examines the connection between hardware and software. The problem with introducing hands-on assembly programming using a real system in such courses is the learning curve and time associated with understanding the compiler, language, BIOS, and hardware system. Another simpler approach is to use DEBUG in DOS mode to get a feel for the way assembly programs behave – if running DOS programs is an option. The VPC assembler was designed as a simple environment, running under Windows, that offers students the opportunity to write programs that do not require a major investment in time to master the language and compiler. The assembly language supported by the compiler is a subset of Intel 8086 assembly, but it strips away many of the special instructions needed to couple with the operating system. In addition, the VPC assembler is based on a simple machine language that students can also study, should that be a required topic in the course.



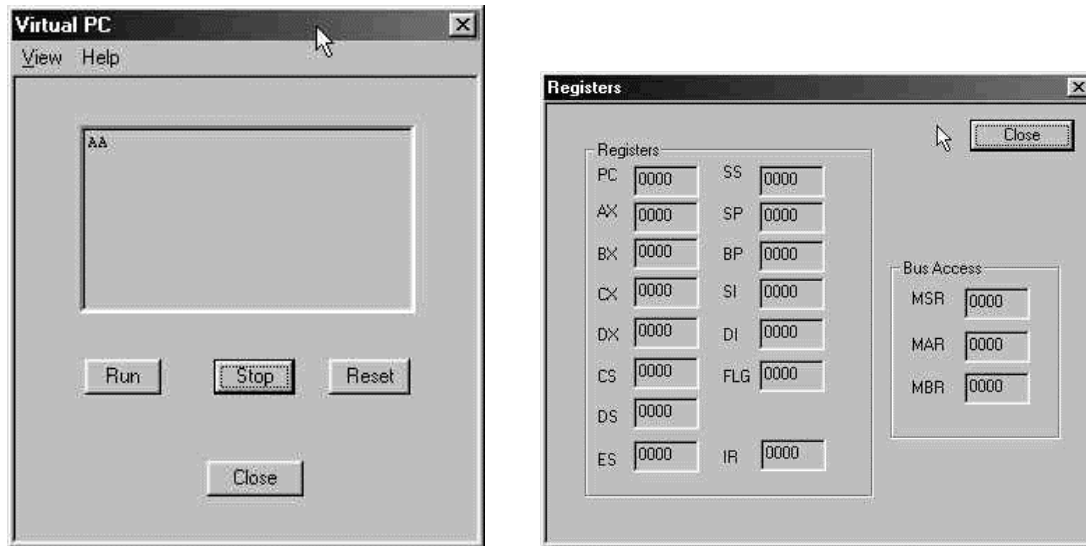
The editor

2 THE VIRTUAL PC

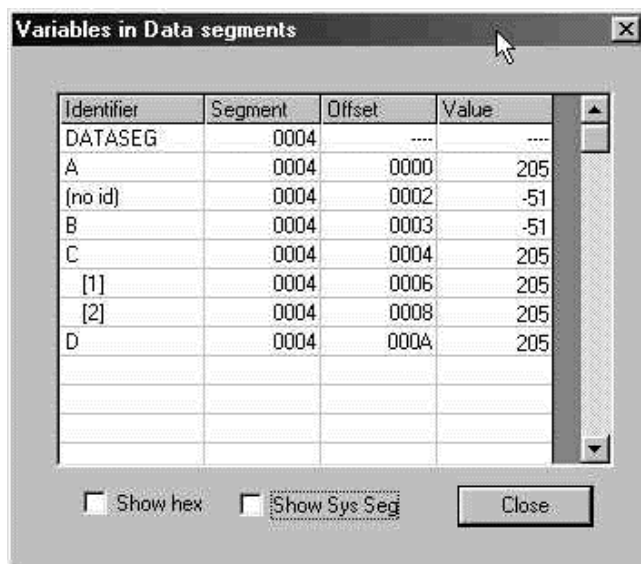
To provide our students in the computer organization course with a programming system at the microcode level, the author designed the Virtual PC and a set of tools for writing and running microprograms [3]. These tools comprise a set of independent applications and include the hardware simulator, a microprogram assembler, a sample set of microprograms, including a microprogram interpreter of a simple machine language, and an assembler that produces machine code that runs on top of the microcode. While this system provides a comprehensive set of tools for studying a PC at various levels, these tools do not provide an integrated system for writing, running, and debugging assembly programs. Running the code produced by the stand-alone assembler requires an understanding of the underlying microcode-supported hardware and does not provide a visual debugger or a dynamic view of program variables (it does provide a view of raw memory). The VPC Assembler introduced in this paper is based on this earlier work but does provide these features in one integrated application without requiring any knowledge of the underlying microcode. In fact, there is no microcode level implemented in the current software.

The VPC simulates a system that consists of a microprocessor based on the Intel 8086 (a segmented architecture), 256K of RAM, video output, hardware interrupt controller, keyboard input, and a system clock. To run a program on the VPC, a student must first write the source code using the VPC Assembler's built-in editor and then compile it. Assuming there are no errors, the next step is to pop up the VPC and run the code. To view the execution, a student may open internal views of the registers and memory variables. To step through a program (step into or over proc and

interrupt calls), the visual debugger is used where the source code is displayed and breakpoints can be defined. The VPC Assembler also includes comprehensive help windows to aid the student.



The VPC and Registers views.



The Variables views.

3. THE VPC ASSEMBLY LANGUAGE

The language is a subset of Intel's 16-bit assembly language and includes often-used commands such as moves, compares, arithmetic and logical operators, jumps, procedure calls, and interrupt calls. The same registers are used (AX, AL and AH, etc), including the flag register and segment/offset registers. Indirect addressing is supported by move instructions, and the use of a system stack with push and pop operations is also included. In all, over 130 opcodes are defined in the associated machine language. In addition several directives are used to declare procs (functions), data variables, and segments – again, like the Intel language. As discussed later, the absence of a basic operating system normally found in ROM requires an additional directive that isn't found on Intel systems. This directive defines a special system

segment that begins at low memory where the interrupt vector table and other system variables are maintained. To shield beginning student from these details, a “system file” can be used to store this part of the code, separate from the student’s source file. This system code can also include other code and data segments, whatever might be needed to provide basic input/output services along with other operations.

3.1 Interacting with Memory

Like the original PC, a portion of main memory is reserved for the video memory map. The rest of memory is used to store the system segment and stack, machine code, and data. One primary difference between the Virtual PC and a real PC is that all memory accesses and other operations are executed in one clock cycle. Students access memory by moving data to and from memory addresses. For most of these operations, the DS (data segment) register is used to designate which segment of memory to access. A second register, ES, can be used when data is moved from one data segment to another. Again, these operations are identical to those used on the Intel platform, so most assembly programming texts that focus on the Intel architecture can be used in class as references.

3.2 Output

Like the Intel architecture, writing a character to the video memory (the video uses text mode only) involves two bytes – the ASCII code and the attribute. The video does not support color, blinking, etc. so the attribute byte is not used, but it is included for completeness.

3.3 Input and Hardware Interrupts

When the user enters data at the keyboard, the ASCII code is trapped by the keyboard controller. The controller signals the interrupt controller, which in turn signals the CPU for attention. At this moment, assuming the CPU is not processing a clock interrupt (which has higher precedence) and interrupts are not masked, the microprocessor switches tasks by loading the CS and PC registers with the segment and offset addresses found in the interrupt vector table and begins executing the interrupt routine. The system code includes the table addresses as well as the interrupt routines. As mentioned earlier, this level of detail can be hidden from users, so they need not be overwhelmed with this material.

3.4 Defining Software Interrupts – An Advanced Topic

The interrupt vector table, which begins at absolute address 0, contains the segment and offset addresses of interrupts. Interrupts 4 and 5 are used for the two hardware interrupts (clock and keyboard), but other software interrupts can also be defined. To do this a system file must be written, which includes a system segment as well as a code segment for the interrupt routines.

4. THE SYSTEM SEGMENT

Since the VPC does not include an operating system, a mechanism is needed to set up low memory where the interrupt table and other system addresses are stored. In the VPC assembly language a special system segment can be defined. This segment is

nothing more than a data segment that maps to low memory. An example of two entries in this segment follows:

```
dw seg      _int4      ; segment address of int 4
dw offset   _int4      ; offset address of int 4
```

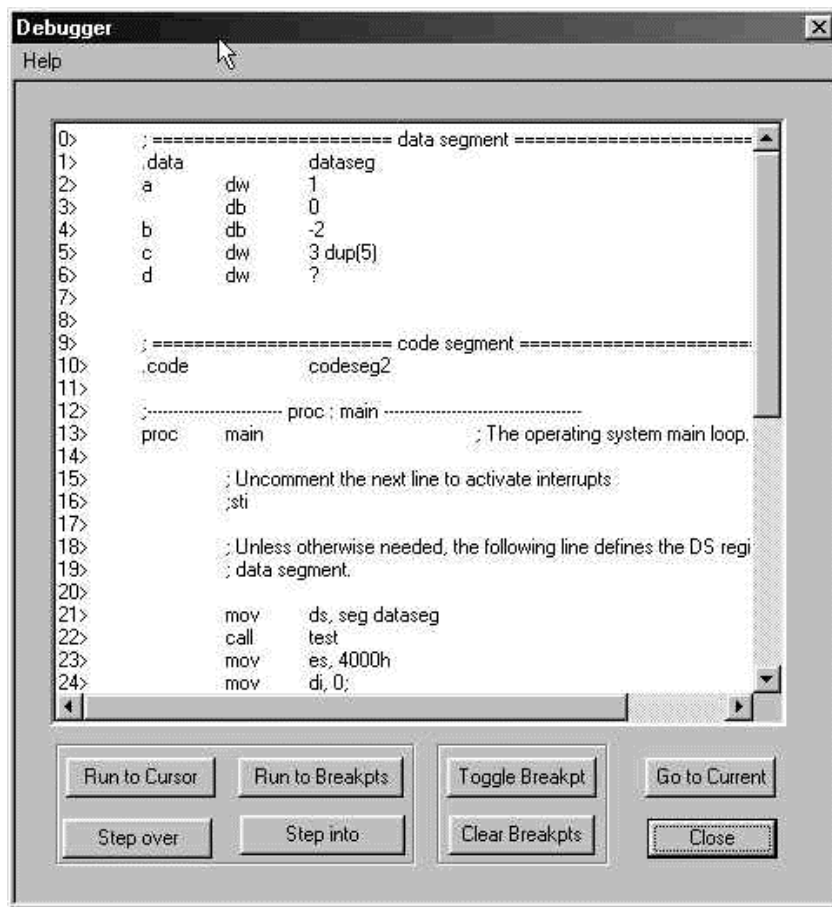
These statements define two 16-bit words; one whose value is the segment address of a proc named “_int4” and the other is its offset. Assuming these statements appear at offsets 10h-13h within the segment (the table entry for an interrupt 4 – the clock interrupt), then a hardware interrupt 4 would trigger the _int4 proc. Another important entry in this system segment at offsets 20h-23h is the address of the first proc to be executed when the VPC begins running. While these details sound technical and may confuse beginning students, they are not vital to using the assembler.

5. THE SYSTEM CODE – AN ELEMENTARY OPERATING SYSTEM

The VPC Assembler compiles two types of files; a “normal” source file, and a system file. All assembly files use the .vsm extension. A normal source file is one that does not include the system segment, nor any special code segment for interrupt routines. In most cases, students would write normal source files. A system file, on the other hand, does include these special segments and may consist of a complete, self-contained program. The reason for supporting these two types of files is to hide the system code from the novice programmer. When a normal source file is compiled, the VPC Assembler looks in the file’s folder for a special system file, system.vsm. If it is not found, then default system code is used. This code is built into the assembler and provides a basic interrupt table with a do-nothing hardware interrupt routine. In either case, the system code is merged with the source code to produce the resulting machine code. Thus, whether the instructor provides system.vsm or not, students can program their own source code without being concerned with this system code. On the other hand, for advanced projects, students can write their own system files.

6. USING THE DEBUGGER

Since the VPC Assembler was designed to learn assembly language programming at an introductory level, providing easy-to-use debugging tools is important. Aside from running an assembly program, the user can open a debugger window and step through the source code (stepping into or over proc and interrupt calls) or set breakpoints and run the code. If the user’s source code is not a system file, then the system code that is merged in is not visible to the user, and stepping into this code is equivalent to stepping over. Again, the intention is to hide the system code from the student. The assembler does provide a few safety valves. If stepping over hidden system code results in a suspiciously large number of clock ticks, a pop-up window provides the user an opportunity to stop execution (instead of staying in a potential infinite loop). The window keeps popping up after long clock intervals if the user decides not to abort. In addition to the debugger window, other views can be opened. The Variables view displays all the variables in each data segment, their offsets and values. A check box allows the user to decide whether to display the system variables, including the interrupt vector table. Finally, the Registers view displays the current settings of the CPU’s registers. With both the Variables and Registers views, the values change dynamically when the program is run or stepped.



7. VIEWING THE MACHINE CODE

The VPC Assembler is not a source code interpreter. It actually compiles the source into machine language and loads the code into memory (256K of RAM) before the assembly program is run. Students can produce listing and map files and study both the segment/offset addresses of segments, procs, and data, and view the machine code for each instruction. This is useful in a computer organization course where both the machine and assembly levels are studied.

8. STUDENT PROJECTS

Because of its flexible design, the VPC Assembler can be used at both introductory and intermediate levels. Simple projects include writing a character or a string to the monitor. For strings, the program can use a loop to write one character at a time. Another project might involve defining an array of integers (or characters) in a data segment and copying that data to another array. Other exercises include performing arithmetic operations on data and writing the solutions back to memory or to the monitor (which is more complex since integer values must be converted to ASCII). Advanced projects may involve writing a system file that includes clock and input routines. For the keyboard interrupt, the incoming key can be stored in a queue, which is maintained by that routine and another software interrupt that is called to retrieve keys from the queue. It is even possible to write a multiprocessing application in which two procs are alternately given slices of time to perform their algorithms. In this project, the clock interrupt routine must determine when it's time to stop one routine and start the other. The project is complex since important registers must be

saved with each routine when swapped and the housekeeping involved can be tricky. A group project might involve writing a series of INT 21h-like functions that handle input/output tasks, similar to the DOS functions found in the Intel BIOS. Each student is assigned one or more routines, which they test separately. When done the combined code is distributed to the class members who then comment on the code and use it in a main program.

9. CONCLUSIONS

While the VPC Assembler is not rich enough for use in an assembly course, it is an ideal tool for courses in which assembly programming is introduced. For example, a computer organization course that follows the spirit of Tanenbaum's text [2] may spend a few weeks studying the machine and assembly levels. In such a course, the assembler can be used to provide a simple platform on which students get some hands-on experience. The assembler's simplicity also provides a good platform for doing independent study projects, especially if assembly is not offered as a regular course. See the references for a link to download the application.

REFERENCES

1. Morse, Stephin P., The 8086/8088 Primer, 2nd Edition, Hayden Books, 1982.
2. Tanenbaum, Andrew S., Structured Computer Organization, 4th Edition, Prentice-Hall, 1999.
3. von Kaenel, Pierre, *The Virtual PC: A Tool for Studying Hardware and Software*, Proceedings of the Fourteenth Annual Eastern Small College Computing Conference, October 1998.
4. VPC Assembler – web site for downloads:
<http://www.skidmore.edu/~pvonk/home/vpcasm>