# The Use of Java to Develop a Microprocessor Emulator

Clint Cooper
*Department of Computer Science*
*University of Otago*
*Dunedin, New Zealand*
email: clint@atlas.otago.ac.nz

Paul Werstein
*Department of Computer Science*
*University of Otago*
*Dunedin, New Zealand*
email: werstein@cs.otago.ac.nz

## Abstract

*This paper describes the use of Java to develop a software based emulator for a microprocessor. This emulator is used in a second year computer architecture course to teach the basics of assembly language programming. The implementation of the emulator in Java is described in detail.*

## 1. Introduction

Part of the second year computer architecture course at the University of Otago includes assembly language programming. The goal is to enable students to understand program execution at the machine level. Assembly language programming occupies 25% of the course.

A paper based method was used in the past; i.e., the students were given a short problem and developed assembly language code to solve the problem. There was no adequate method for them to check their code other than to simulate it by hand.

In this paper, we describe the use of Java to develop an emulator for a microcomputer. The emulator allows students to develop assembly language programs and test them on an "actual machine." In this case, the "actual machine" is a Java program which has the appearance of a real processor.

The paper is organised as follows. In the next section, the goals of teaching assembly language are covered. Section 3 details the previous method used in the course along with some alternatives. The implementation details of the microprocessor emulator are covered in section 4. Section 5 discusses the results of using the emulator in conjunction with an assignment. Finally, some conclusions are given in section 6.

## 2. Goals of teaching assembly language

The second year course on computer architecture covers digital design, including boolean algebra, gates and basic integrated circuits, sequential and combinatorial circuits. Building on that knowledge, the basic parts of a computer such as memory, I/O, registers, and the CPU are studied in detail.

A few lectures cover assembly language so that students gain a better understanding of program execution. The goal is for students to understand how high level language programs, such as those written in Pascal or C, actually get executed. In order to accomplish this goal, the Intel 8085 microprocessor is covered in some depth.

The Intel 8085 is a very basic but complete microprocessor. Although it is an old processor, its descendants continue today through the 80x86 and Pentium based machines. The advantage of teaching with the 8085 is its simple register structure and a straight-forward, easy to understand assembly language.

More modern processors were considered. However the modern processors have many advanced features to enhance performance. These advanced features are difficult for first time students to understand and tend to hinder learning the basics of a computer system.

## 3. Previous method

In their study of assembly language, students are given two types of problems: problem definitions that can be coded directly into assembly language, and high level language constructs to code into assembly language. In the former case, the students receive an English statement of a simple problem. The goal is to code a solution in assembly language considering the architecture of the Intel 8085 and its instruction set.

In the latter case, students receive a high level language program construct. Example constructs are if-then-else, for loop, while loop, and repeat loop. The goal is to code the given high level language construct in assembly language. Understanding how high level language constructs are executed at the machine level also provides some foundation for a later course in compilers.

Having written an assembly language program, the students need to test it. Unfortunately there was no simple way to do so. What is needed is a way for students to enter their code and easily observe the registers, memory locations, condition flags, and program counter while the program slowly executes.

The PC and Macintosh were considered as candidates but rejected. The basic tools for assembly language on a PC do not allow easy debugging and observation of the machine state. The more advanced coding tools such as Code Warrior are too complex to learn and use in the short time span allowed. The high cost of these advanced tools is also a factor.

Modern processors such as the DEC Alpha are too complex at this teaching level. In addition, the design tools for these processors require too much training to be useful for students who have only a few weeks for this part of a computer architecture course.

There is also a need for students to experiment with instructions to gain a better understanding of them. Many of the more inquisitive students want to try their hand at more advanced problems in their own time, but are frustrated since there is no way to check their solutions.

For these reasons, a Java software program was developed which emulates the Intel 8085 microprocessor.

## 4. The Java based emulator

The Intel 8085 emulator is written in the Java programming language [1, 2]. The decision to use Java to implement the 8085 emulator is based on a number of factors. These reasons include platform independence, a good user interface library, and the object-oriented nature of the language.

Platform independence is needed due to the large variety of computers found in a computer science department. History shows that the machines used currently are different from the machines used twenty, ten and even five years ago. Platform independence allows a change in the computers used for teaching without the time consuming process of porting software.

The user interface library of Java allows rapid development of the display for the emulator. The user interface of the emulator is shown in Figure 1. Java is an object-oriented language, and this feature helps greatly in the implementation of the 8085 emulator as is discussed in Section 4.2.

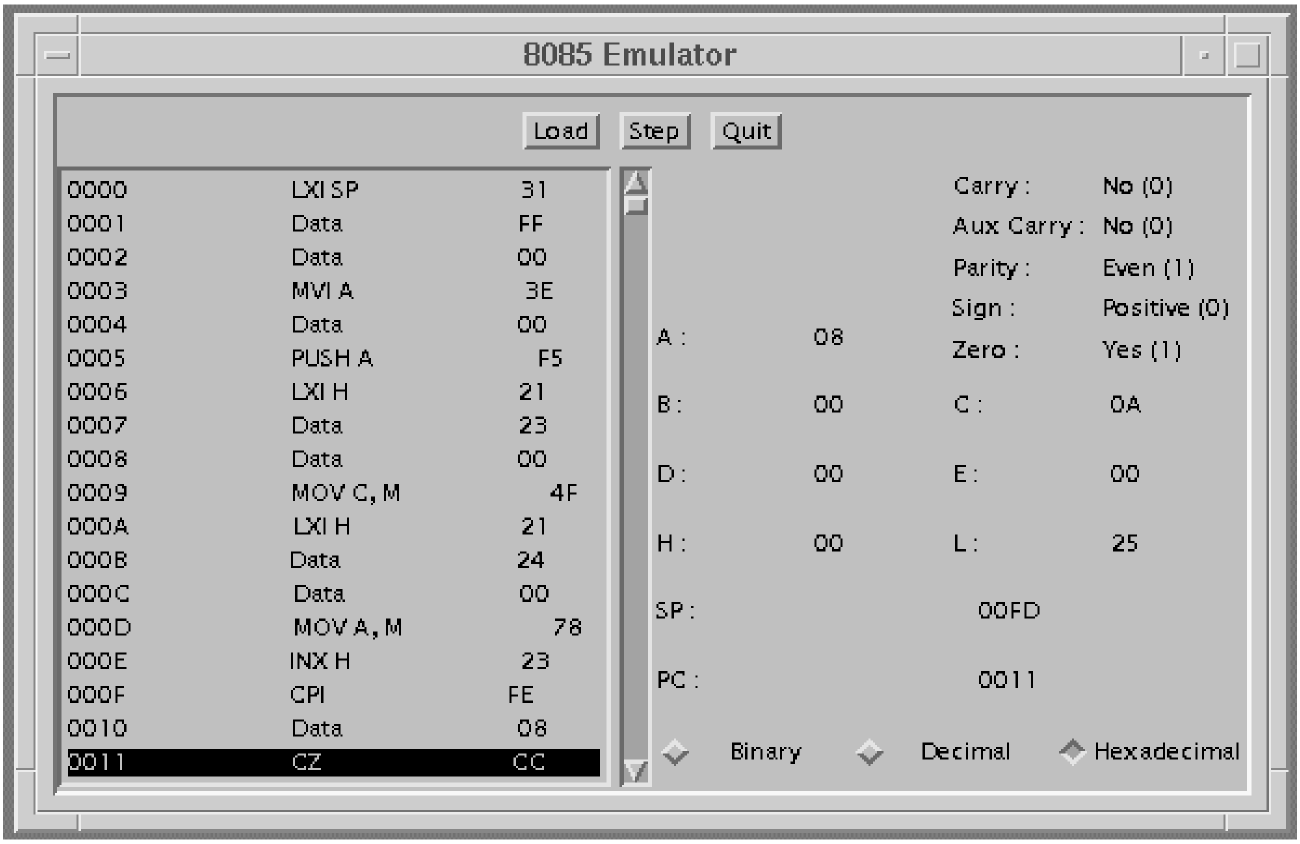


**Figure 1. Emulator interface screen**

## 4.1. Overview of the program

The 8085 emulator program is logically divided into two components: the memory and the processor. These are kept separate to allow a layer of abstraction between each of the components. From the user's viewpoint, the memory is visible as the left half of figure 1. The processor is visible as the right half of the figure.

The memory of the emulator is used for storage of instructions and data. It is responsible for displaying its current contents and keeping the display updated for any changes. The displayed information includes the memory address, the instruction mnemonic, and the appropriate hexadecimal code for the instruction.

In the case of data, a special mnemonic, *data*, is shown instead of an instruction. The value of the data and the address are also shown. The *data* mnemonic allows a user to determine if the information in a particular memory location is an instruction code or a data item.

Instructions are not stored internally in the emulator program as values, but as Java objects. This allows the emulator to catch certain user errors.

The other component of the emulator is the processor. It retrieves instructions from the memory for execution. It also fetches data from memory and stores data within memory. Like the memory component, the processor is responsible for displaying the current status of its internal state. The processor is divided into general purpose registers, special registers, and the processor status word.

The architecture of the Intel 8085 microprocessor is shown in Figure 2 [3, 4]. It has seven general purpose registers, a program counter, a stack pointer, and a processor status word which stores the condition flags.

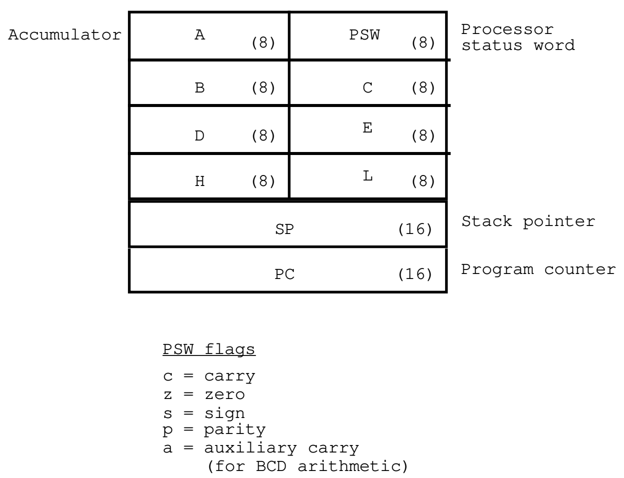


**Figure 2. Intel 8085 architecture**

The contents of the registers in the emulator can be displayed in binary, decimal or hexadecimal. The user selects the desired numeric base.

The seven general purpose registers (A, B, C, D, E, H, L) of the 8085 processor are supported. They are accessed as eight bit words. Alternatively some instructions access register pairs (B-C, D-E, H-L) allowing sixteen bit words to be used.

The special registers are the program counter (PC) and the stack pointer (SP). These two registers are sixteen bits wide and have special functions. The program counter holds the address of the next instruction to be executed. The stack pointer gives the current location of the stack in memory. The stack is used by the processor to store the return address for subroutine calls. It can also be used for temporary storage of data by the programmer.

Finally, the processor status word contains individual bits for the internal condition flags of the processor. This status word stores conditions such as a zero or a carry resulting from instruction execution. These condition flags are used by conditional instructions to determine the flow of program execution.

## 4.2. Details of implementation

The 8085 microprocessor emulator is implemented in the Java programming language. Since Java is an object-oriented language, an object based design is used for the emulator. The basic object types are register, processor status word, instruction, memory and chip. A class diagram of the design is shown in figure 3. The class relationship notation is from [5].

The registers are implemented as a register class. This register class makes no distinction between eight bit registers and sixteen bit register pairs. Instead a general number class is created with child classes for eight and sixteen bit words. This enables the register class to display numbers of any bit length, as long as it has an appropriate child class derived from the number class.

The processor status word is implemented as the storage object for the values of the status flags. The internal flags are able to be retrieved and set using the methods of this object. This class is responsible for displaying the current status of these flags to the user. We elect to display the individual bits and their meaning instead of a single eight bit word. This display method enhances the information gained by students.

Holding the above two objects is the chip class. This class holds instances of the registers and the processor status word objects. The chip object is responsible for displaying these instances to the user. The chip object has methods to allow external access to the registers and processor status word.

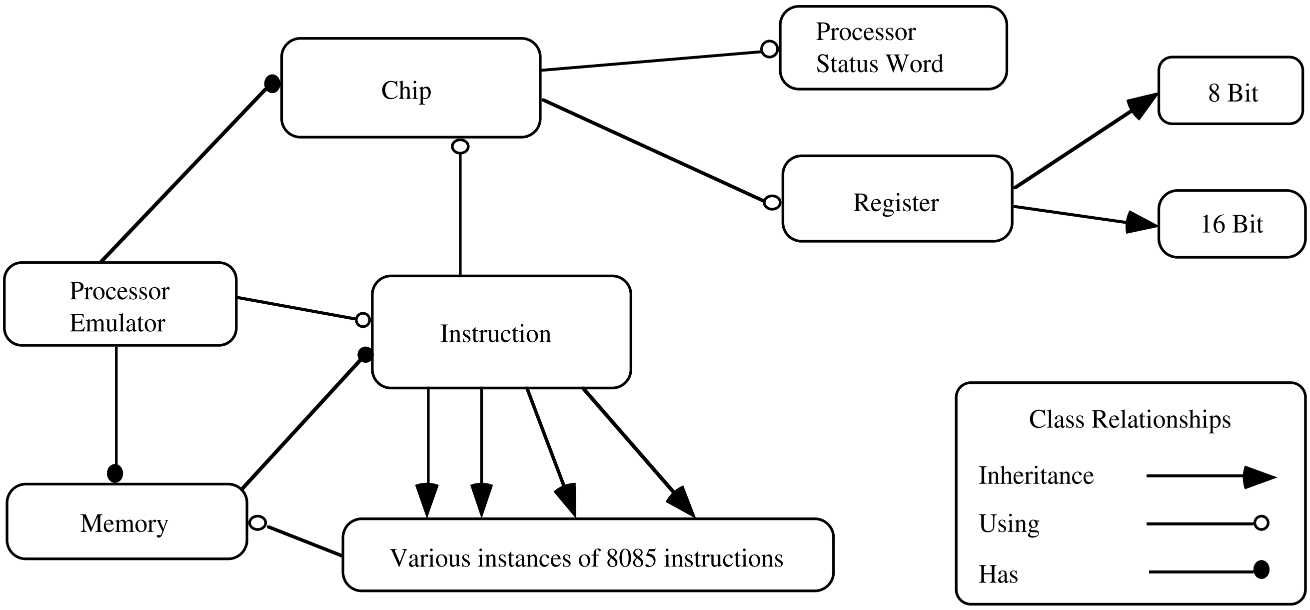


**Figure 3. Details of implementation**

The memory object is the storage class for the instruction object instances loaded into memory. It is responsible for displaying the current instructions or data at each location in memory. The emulator is able to store and retrieve instructions from this object.

The instruction object is the parent class of all instructions implemented within the emulator. This object provides the basic functionality for each instruction. An exception is the data class which is used only for storage of data values.

An instruction is implemented within the emulator through inheritance of the instruction class. The methods of the instruction class are overridden in the child object. The most important method overridden is the execute() method. This method is used to implement the functionality of each of the instructions. Since this method provides the core implementation of an instruction, the instances of the chip and memory classes are passed as parameters. This allows the instruction to perform its operation on the processor.

A small number of instructions of the 8085 microprocessor were unimplemented. These unimplemented instructions are the functions which deal with hardware signals such as *reset* and the interrupts.

Finally, the processor emulator object is responsible for the display of the instances of the memory and chip classes. This object is the one with which the user interacts. The user interaction consists of loading assembly files, stepping through the assembly program and exiting the program.

The loading of assembly files involves the retrieval of the text file from the operating system and the compilation into suitable intermediate form for loading into memory. This intermediate form is converted into the appropriate instances of instruction and data objects. Exiting the program involves shutting down the program and quitting.

The step command executes the 8085 program in memory one instruction at a time. The instruction at the current location of the program counter register is retrieved, and the execute() method of the retrieved instruction object is called.

As described above, the 8085 microprocessor emulator is implemented in modular manner. This allows for future expansion. A possible future expansion would be extending this emulator to a later model processor such as the Intel 80486. The registers can easily be extended to accommodate the wider bit size of the 80486 microprocessor registers [6]. The chip class can store the additional registers. Additional instructions can be implemented readily by inheriting the instruction object for each of the new instructions of the 80486 microprocessor.

### 4.3. Use of the program

Use of the emulator involves three stages: writing the program, loading and execution. The writing stage occurs when the user creates the assembly source code to be used by the emulator. This source file is created with any text editor and consists of 8085 assembly instructions and assembler directives. The instructions are described in detail in [4].

The second stage is the loading and parsing of the source file by the emulator. Parsing the source code converts the text file into the internal representation used by the emulator. This internal representation consists of instances of instruction and data objects.

In the final stage, the user executes the program. Execution of the program is done by stepping through the program. The user presses the step button to execute each instruction. This stepping action allows the user to see the effect of each instruction on the state of the processor and memory.

A screen shot of the emulator is shown above as Figure 1. The left hand side of the emulator displays the contents of the emulator's memory. On the right hand side is the internal state of the processor. It shows the contents of the registers and status word.

A feature of the emulator is the error handling. Error handling within the emulator informs the user of incorrect code. There are four types of errors: invalid instruction syntax, out of range memory access, unimplemented instructions, and an attempt to execute data as program code.

Instruction syntax is checked while the emulator is loading the source file. If there is a syntax error, an error screen appears informing the user of the error. The screen displays the line number, the code, what it was expecting and what it found.

An out of range memory access error occurs when the user's program tries to access a memory address outside of the available memory. In many real world systems, an access to a location beyond the memory size may access a valid memory location. This error occurs because the higher order bits of the address bus often are unused. The result is equivalent to performing modulo division on the memory address. The program flags this condition as an error.

Unimplemented instructions are the third type of error. The user is informed of this error since the emulator cannot execute unimplemented instructions.

Finally, the last error is execution of an invalid instruction. An invalid instruction is an instance of a data object. A data object is an object where the user has stored values. This error, if not flagged, would be an attempt to execute data as if it were an instruction.

Although an actual processor makes no distinction between data and instructions in memory, the authors have chosen to make the distinction in the emulator to assist in learning. This distinction is made since we assume a user's program is incorrect if it tries to execute data.

A valid reason for an assembly language program to execute data values occurs when the program is self modifying. The authors consider this to be beyond the scope of a second year course.

## 5. Results

The development of the emulator was done as a project in another course. We did not set out to prove scientifically that the learning environment would be improved. Therefore we can only infer the effect of the emulator on learning.

We used three methods to gauge the use of the program and its effectiveness: verbal student feedback, a written survey, and analysis of system usage records. To determine actual usage of the program, the UNIX 'lastcomm' command was used. The data returned from the command was written to a file and analysed. It showed that 85% of the students used the emulator at least once during the course.

Verbal feedback was generally positive. Students appreciated being able to view the results of program execution. Some commented that the use of the emulator helped them gain a better understanding of individual instructions.

The students were also given a written survey to complete. Of 31 students enrolled in the course, nineteen students completed the survey. Eighteen (95%) of them had used the emulator and found it to be helpful. The same eighteen students used the emulator to complete an assignment on assembly language programming.

One question on the survey asked the reasons for using the emulator and had the following three choices:

- to gain a better understanding of individual instructions (50%)
- to assist in completing the assignment (94%)
- to gain a better understanding of assembly language programming in general (39%)

The numbers in brackets () indicate the percentage of students marking that reason. These answers confirm our belief that the learning environment was enhanced.

The written comments on the survey suggest two features which should be added to the program. The first request is for a 'run' button that would let the program run to completion or until an input was needed. The second request is for a 'reset' button to stop the program in the event of an infinite loop.

## 6. Conclusion

This paper describes a microprocessor emulator developed using Java. The use of the Java programming language allows the human interface to be developed quickly. Since Java is used, the software can be run on a variety of platforms.

In the future, the ‘run’ and ‘reset’ buttons will be added. Additionally, a method to add the hardware oriented instructions is being investigated. The plan is to add an additional screen with buttons to simulate the hardware signals. This will allow experimentation in interrupt handling and programming.

We believe the project is very successful. Most of the students used the emulator at least once, and all feedback is positive.

## 7. References


[1] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Reading: Addison Wesley, 1996.

[2] T. Lindholm, F. Yellin, B. Joy, and K. Walrath, *The Java Virtual Machine*. Reading: Addison-Wesley, 1996.

[3] R. D. Dowsing and F. W. D. Woodhams, *Computers - From Logic to Architecture*. London: Van Nostrand Reinhold (International), 1990.

[4] R. S. Gaonkar, *Microprocessor Architecture, Programming, and Applications with the 8085/8080A*. Columbus: Charles E. Merrill Pub. Co., 1984.

[5] G. Booch, *Object-Oriented Analysis and Design with Applications*, Second ed. Menlo Park: Addison-Wesley, 1994.

[6] T. Shanley, *80486 System Architecture*, Third ed. Reading: Addison-Wesley, 1995.