

Virtual Machines as an Aid in Teaching Computer Concepts

Ola Ågren
Department of Computing Science
Umeå University
SE-901 87 Umeå, SWEDEN
E-mail: Ola.Agren@cs.umu.se

Abstract

A debugger containing a set of virtual assemblers and a virtual machine are used as teaching tools in order to teach students differences and similarities between architectural styles of computer processors. Programs written by the students in the virtual assemblers are compiled and executed in this environment so that the students can follow the execution of the programs step by step or at full speed.

1 Introduction

The course in Computer Architecture is one of the most popular courses in Computing Science at Umeå University. It is a graduate level course, mandatory for students of the MSC/E program and optional for students of the MSC and BSC/E programs. The main text of the course is the very popular textbook by Patterson and Hennessy [4].

1.1 Problem

The course has grown over the years and had two years ago an enrollment of over 100 students, which meant that the assignments had to be changed because of resource limitations. The first assignment has been to build an instruction tracer that executed other programs while collecting vital statistics about it (i.e., size of data used in the operations, register usage, length of basic blocks and calls, bits used in instructions, etc.). This assignment unfortunately required that all students had to have access to machines with MIPS processors. This assignment had to be scrapped since there are only a total of 28 SGI workstations and two SGI servers available and test runs of the assignment led to an unacceptably high load level on the machines running it.

Since the author wanted to highlight the issue of different architectural styles a new assignment was planned. A virtual machine [5] running on all Unix platforms available (SGI Irix, Sun Solaris and IBM AIX) was created together with virtual assemblers that converted assembler code of the different architectural styles to the virtual target machine

assembler.

Unfortunately, some of the students complained that the terminal based interface was outdated and wanted a graphical user interface instead. Since we had already planned to use the same type of assignment last fall (with an enrollment of over 120), we felt a need to supply one. The choices available at the time were to either set up a “wrapper” around the older programs¹ or create a new program from scratch.

1.2 Goals

There were three goals for the new assignment:

1. It should still provide a deeper understanding of how different computer architectures behave,
2. it had to be easy to understand and use, and
3. it had to be portable to all platforms available at the department (including Windows-based PCs).

All of the goals were deemed imperative for the actual implementation to be successful.

2 Solution

New hardware is expensive both to acquire and maintain, especially if it makes the machine park heterogeneous. Moreover, very few machines with unorthodox system architectures are currently in production; especially accumulator and stack machines have become very rare lately. This indicated that another route had to be followed: The Virtual Machine.

The basis of virtual machinery is the actual hardware that is supposed to build up a machine. This is constructed in such a way that it “executes” the given program in the same way as an actual machine would do, but in software. Virtual machines have been used for a multitude of reasons. One use of virtual machines has been to test hardware systems

¹In the same way that `ddd` is a wrapper for `gdb`.

prior to their actual construction (often in VHDL or Verilog). Another use is to give system developers the availability of their own machine without blocking it for all others (providing the user with the same interface as the underlying hardware) [5].

3 The Different Types of Virtual Machines

There are four basic types of architectures ([4, 1]): Accumulator, Stack, Memory-Memory and Load-Store. In addition to these four we added a fifth, an Index Machine. Instead of developing a virtual machine (with user interface) for each style we opted for a debugger containing all five virtual assemblers as autonomous systems.

The virtual assemblers were created with small instruction sets (between 20 and 35 for each machine) to emphasize the main principles as clearly as possible. The different assembler languages had some parts that were similar (e.g., name of the normal operations) and some that were not (i.e., branching/jumping, how to create variables, and architecture-specific instructions). Schematics for the virtual machines are found in Appendix A.

3.1 Accumulator

The accumulator machine is one of the most basic processor architectures created. This type of machine has one register (called the accumulator) that is implicitly used as one of the operands in all instructions and is the destination of all calculations. It is the target of all loads and the source for all stores.

3.2 Index

This is basically an accumulator machine with the addition of two index registers that can be used as temporary storage and/or offset register in load and store instructions. These extra operations make the machine more suitable for implementation of high-level languages and data constructs, e.g. vector operations.

We have modeled these operations after the 6502 processor, since that processor used to be quite popular and has a user friendly instruction set.

3.3 Stack

A strict stack machine does not have any general purpose registers at all. All data is handled in a last-in, first-out stack. Operations take their operands from the stack and the result is pushed back onto the stack, except for operations that move data from memory to stack or vice versa.

3.4 Memory-Memory

A strict memory-memory machine does not have any general purpose registers. The main difference with respect to stack machines is that operations in a memory-memory machine use the values in memory cells as operands and the result is stored in a memory cell.

3.5 Load-Store

The load-store machine has a fixed number of registers (in our case 32) that are used as operands and the destination of the result for operations, except for operations that transfer data between registers and memory or vice versa.

4 Implementation

In order to get the required portability even in the graphical user interface (thereby ruling out C and X11) Java² was chosen as the implementation language. Another benefit of this choice was that some of the code written in C could be used with few changes. The basic structure of both the virtual machine and most of the virtual assemblers could be retained, thereby increasing the probability that the new system would work in much the same way as the old.

The program uses the suffix of the input files to decide which virtual assembler to use when parsing the file (i.e. *ac*, *ix*, *st*, *mm* and *ls*, respectively).

4.1 Implementation-Specific Bug

The virtual assembler for the memory-memory machine used to be implemented partly using *yacc*, and *java_cup* (by Scott Hudson, Frank Flannery and C. Scott Ananian) was used as a substitute for *yacc*. The two program packages are unfortunately not plug-and-play replacements for each other, so there is a difference in behavior in the end product. The main difference is that reductions are performed eagerly (as soon as a projection is fulfilled) [2, 3] in *yacc*, but require yet another token in *java_cup*. This meant that if a code contained an “end *fct*” (indicating that *fct* is the label to start execution at) at the end, execution would in the *java_cup* version start at the first valid operation in the file even if *fct* was further down in the source file.

5 Assignments

We have now used this assignment for two years running, with some smaller changes. During the first course we let the students decide for themselves which mathematical function to implement in the four virtual assemblers, disallowing factorial functions in the mark-up assignments. During the second course the students had to choose two

²Java is a trademark of Sun Microsystems, Inc.

functions to implement in all five available virtual assemblers, of which one had to be a fifth degree polynomial (to show the strengths of the stack machine).

6 Sample Screenshot

The screenshot (Figure 6) in Appendix B shows the virtual machine/debugger in operation. It is currently processing a stack machine program that calculates the factorial of 10, and is just about to multiply the top two numbers on the stack.

7 Availability

The virtual machine is together with some example assembler files available for download through <http://www.cs.umu.se/~ola/Dark/>. Current documentation is unfortunately in Swedish, but a full technical report in English will be available this autumn.

8 Conclusions

The students have found it much easier to understand how different architectural styles affect code (generality, parameter passing, layout, execution, porting, etc.) after this assignment was added to the curriculum. Since this was one of the main goals for the assignment it can be regarded as a success so far.

The system appears to be easy to use, thereby fulfilling the second goal. So far, only a few bugs turned out to exist and those will be corrected before next course. The graphical user interface was seen as a boost by most of the students but some used the older scripts instead. Some students made requests for minor changes, mainly in error presentation and default behavior of certain character combinations.

The third major goal of this assignment was that any workstation or server could be used as a platform for programming and debugging of the students' assignments, thereby reducing the burden of the SGI machines. This goal was clearly achieved since the program could be installed on any machine that supports Java 1.1 or higher.

9 Future Updates and Additions

The known bugs will be eliminated. The major thing to change will be to rewrite the memory-memory assembler in a recursive-descent parser in Java, thus correcting the difference in behavior between yacc and java_cup described in section 4.1.

The virtual assembler part of the previous version of this assignment will be used as a front-end in a later assignment. The students will implement their own virtual machine that

will be required to match the virtual assembler, thereby executing the code that they wrote in the assignment with the virtual machine.

References

- [1] ÅGREN, O. Teaching Computer Concepts Using Virtual Machines. *SIGCSE Bulletin* 31, 2 (June 1999).
- [2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: principles, techniques and tools*. Addison-Wesley, Reading, Massachusetts, 1986.
- [3] LEVINE, J. R., MASON, T., AND BROWN, D. *lex & yacc*, second ed. O'Reilly & Associates, Inc., Sebastopol, California, 1992.
- [4] PATTERSON, D. A., AND HENNESSY, J. L. *Computer organization & design: the hardware/software interface*, first ed. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.
- [5] SILBERSCHATZ, A., AND GALVIN, P. B. *Operating System Concepts*, fifth ed. Addison-Wesley, Reading, Massachusetts, 1997.

Appendix A: Schematics of the Virtual Machines

Figures 1 to 4 shows the programmers' view of the virtual machines schematics, while Figure 5 is the layout of the basic virtual machine actually executing the code.

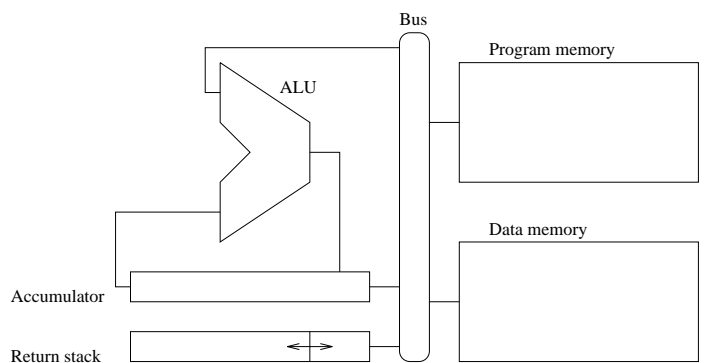


Figure 1: Schematics of the accumulator machine.

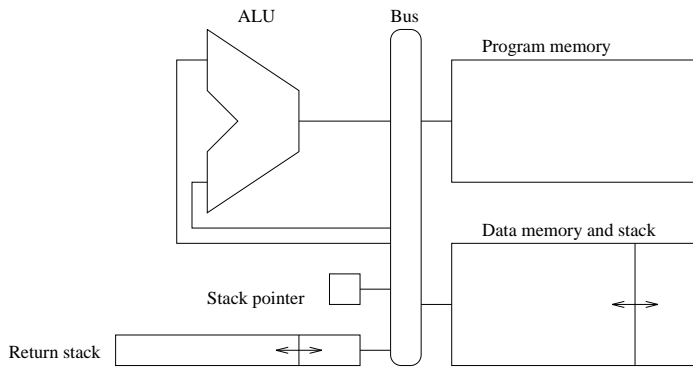


Figure 2: Schematics of the stack machine.

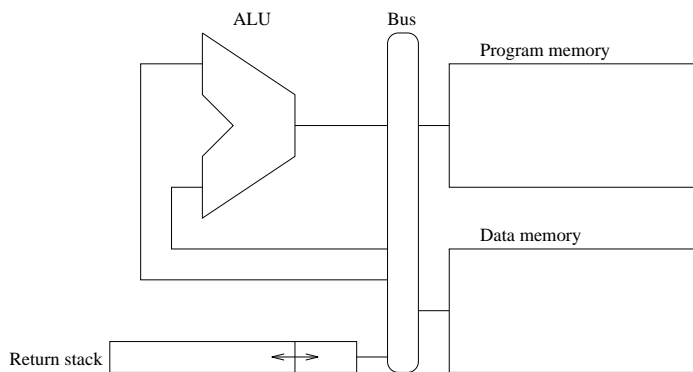


Figure 3: Schematics of the memory-memory machine.

Appendix B: The User Interface

The different parts of the user interface are (as can be seen in Figure 6):

- **Menu bar:** This gives access to the “File” pop-up menu and a help window. The commands available are “Load file”, “Reload file”, “Print to file” and “Quit”;
- **Top:** Current information and debugging commands, i.e. name of currently loaded file, number of assembler lines and internal steps executed, a “step” button (executes one assembler command), and a “go” button (execute code at full speed until a “stop”-instruction is executed or an invalid memory location is reached);
- **Left column:** Source view of the currently loaded file, with the line that is just about to be executed highlighted;

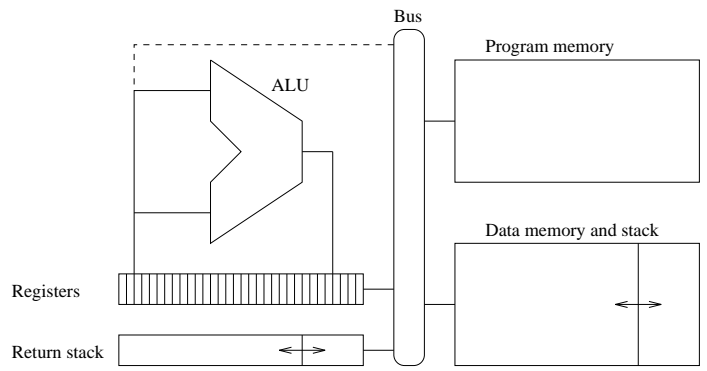


Figure 4: Schematics of the load-store machine.

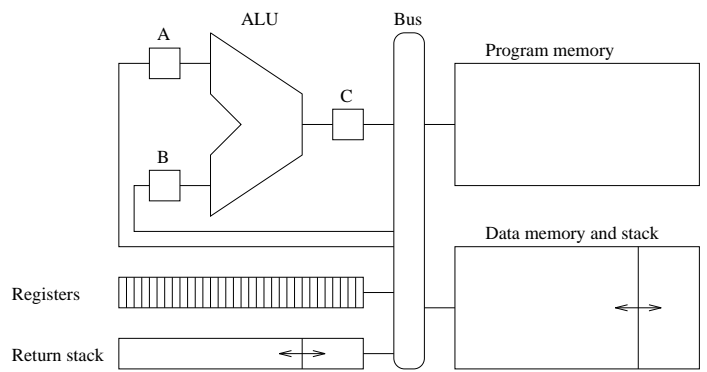


Figure 5: Schematics of the virtual machine.

- **Upper right:** The variable window. It shows the current value of all variables defined in the program. Double clicking on a variable brings up a dialogue to set the value of that variable;
- **Lower right:** The miscellaneous window. It will always show non-zero memory locations in the lower part of memory, but also information that depends on the type of source code that is currently loaded:
 - **Accumulator:** For accumulator and index machine code;
 - **Index registers:** For index machine code;
 - **Stack:** For stack and load-store machine code;
 - **Non-zero registers:** For load-store machine code.

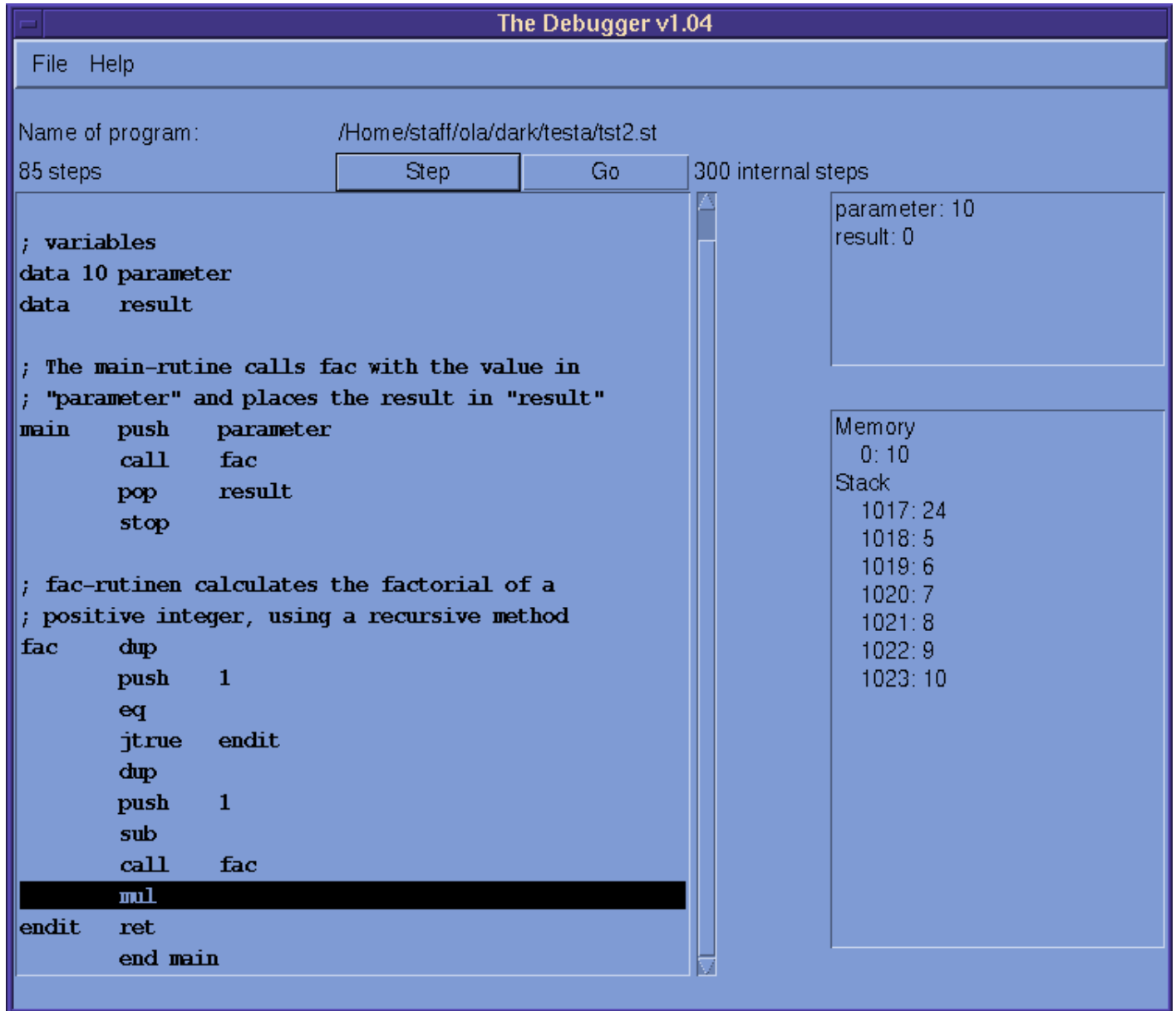


Figure 6: Screenshot of the user interface.