DLX Simulator for Automated Checking of Assignments in a Computer Architecture Class

JIŘÍ KOCOUREK, JOSEF HLAVÁČ
Czech Technical University in Prague
Faculty of Information Technology
Thákurova 9, 160 00 Praha
CZECH REPUBLIC
{kocouji1,hlavacj2}@fit.cvut.cz

Abstract: In this paper, we introduce an automated simulator of the DLX educational processor. The simulator is intended for checking the correctness of homework assignments in a bachelor-level Computer Architecture course. We describe the nature of these assignments and describe the requirements, structure and capabilities of the new simulator.

Key–Words: DLX, WinDLX, processor, computer architecture, instruction-level parallelism, simulator, education

1 Introduction

One of the several important topics in the Computer Architecture course taught at the Faculty of Information Technology, Czech Technical University in Prague, is the concept of instruction-level parallelism and pipelining. The concept is demonstrated on the example of the DLX processor, introduced in the earlier editions of the classic textbooks [1, 2].

In order to get thoroughly acquainted with the concepts and issues of pipelining, hazards, stalls, etc., students are asked to write a moderately complex program in the DLX assembly language. The assignment is sufficiently complex to allow the use of various optimization techniques, primarily loop unrolling and instruction planning with respect to data dependencies, and to require thinking and planning ahead before actually writing the code. On the other hand, the assignment is sufficiently simple to allow coding, debugging and optimizing by hand directly in the assembly language.

An example of such program is matrix multiplication. Students are asked to write a program that multiplies two square matrices $\mathbf{A}, \mathbf{B}$, each of size $n \times n$, where $\mathbf{A}, \mathbf{B}, n$ are runtime parameters. The number of clock cycles needed to multiply two matrices of a certain size is then taken as the measure of performance and the basis for grading the assignment.

In addition to optimizing, students are also asked to write their code in a clean way and avoid potentially dangerous coding techniques. For instance, students usually understand that it is a bad programming practice to modify the contents of undeclared memory. However, students often fail to see that it is equally bad practice to even read the contents of unallocated memory, since this may lead, under certain conditions, to page faults and other "mysterious" errors. Other common errors include relying on the contents of uninitialized registers, failing to check for correct operation in border-case scenarios, etc.

The assignment also includes certain restrictions on register usage. For compatibility with high-level languages, it is commonly necessary for subroutines written in the assembly language to respect certain conventions, such as preserving the content of certain registers ("callee saves"), or not using certain registers at all ("reserved for the system").

2 Previous Approach

Currently, the assignments are checked and graded manually.

Students usually work in teams of two. The teams first submit their solutions by e-mail to their teaching assistant (TA), in order to create a record of timely (or untimely) completion. The TA then verifies the handed-in assignments, asks each team to explain their approach and the key features of their solution, and assigns a corresponding grade (or points).

It is obvious that the process of checking and grading the submissions is very time-consuming, and yet prone to errors. In many instances, the TA – no matter how experienced – fails to notice subtle errors in the code that may produce incorrect output in certain cases, or lead to other kinds of errors (such as the above-mentioned page faults).

When writing and optimizing the code, students

ISBN: 978-1-61804-140-1

use the WinDLX simulator developed at TU Vienna [4, 3]. Despite its age, it is still a very useful and very demonstrative tool for use in the Computer Architecture class. However, the tool does not allow automatic (scripted) simulation, nor does it check for the abovementioned bad programming practices.

Our Faculty already uses an automated system for checking C/C++ assignments. It is possible to add support for other programming languages, as long as a compatible testing environment is available.

We therefore undertook to write our own simulator that would be suitable for automatic execution. In addition, the new simulator should check for and report various bad programming practices as well as violations of other imposed conditions.

3 Requirements

Let us now summarize the requirements for the new simulator:

(i) The new simulator should support all parts of the DLX processor that are visible and accessible to user programs. Special instructions and registers intended for the operating system need not be supported. Therefore, all integer registers (*r0–r31*) and floating-point registers (*f0–f31*) should be supported. All instructions should be supported except for the `trap` instruction (which should terminate the program) and the `movi2s`, `movs2i` instructions that access special registers.

(ii) The simulator should simulate the entire memory range. However, it is expected that only a small portion of the memory space will be used at a time. This assumption should be used to optimize the memory efficiency of the simulator. In addition, the simulator should report access to undeclared memory. When the program terminates, the simulator should check the contents of pre-defined (configurable) memory locations against expected results.

(iii) The simulator should support the standard 5-stage DLX pipeline, with configurable numbers and latencies of floating-point execution units.

(iv) The simulator should keep execution statistics – the total number of clock cycles and the number of RAW, WAW, structural, control and trap stalls. The simulator should terminate the simulation when a predefined number of clock cycles is executed without reaching the `trap` instruction (to terminate endless loops and extremely inefficient submissions).

(v) The configuration should be easily changed, e.g. read from a XML file.

(vi) The output should be easily parseable. It is expected that the simulator will be eventually integrated into the Faculty's automated grading framework that is currently used for C/C++ code; it is therefore best to print any error messages and the final statistics to the standard output.

4 Implementation

4.1 Structure

The structure of the implemented simulator is presented in Fig. 1.

The simulator was implemented in the Java programming language. The primary reason for choosing Java over C/C++ was the authors' familiarity and experience with this language; added benefit includes easy portability to other operating systems and platforms. The only notable disadvantage is in potentially increased memory requirements and lower performance (increased simulation time). The performance is further discussed in section 5.

The simulator code attempts to mimic the operation of real hardware. An Instruction object is created for each instruction and filled with data as it passes through the individual pipeline stages (descendants of the PipelineStage class). Statistics are collected during the execution. Memory is represented with arrays of byte blocks, where each block (1 KiB) is allocated only when really needed. For each block, a bitmap with information about declared and undeclared bytes is maintained and checked upon every memory access.

The register file contains the zero register (*r0*), integer registers (*r1–r31*), single-precision floating point registers (*f0–f31*), and the floating point status register (*fps*). Pairs of floating point registers can be accessed for double-precision operations. Every access to the registers is checked against the configured register usage restrictions (either "unrestricted use", "callee saves", or "do not touch").

4.2 Input and output

There are two inputs to the simulator.

First, the parameters of the simulated DLX processor and the assignment to be solved are specified in a XML configuration file. The configuration specifies the number and latency of floating-point execution stages, enabled/disabled forwarding, initial values for defined memory blocks, and register usage restrictions. The configuration further includes the correct output (memory contents) and a grading scale table (essentially a clock cycle to percentage conversion table). The structure of the configuration XML file is described in a XSD file.

The second input is the submitted source code in the DLX assembly language.

ISBN: 978-1-61804-140-1

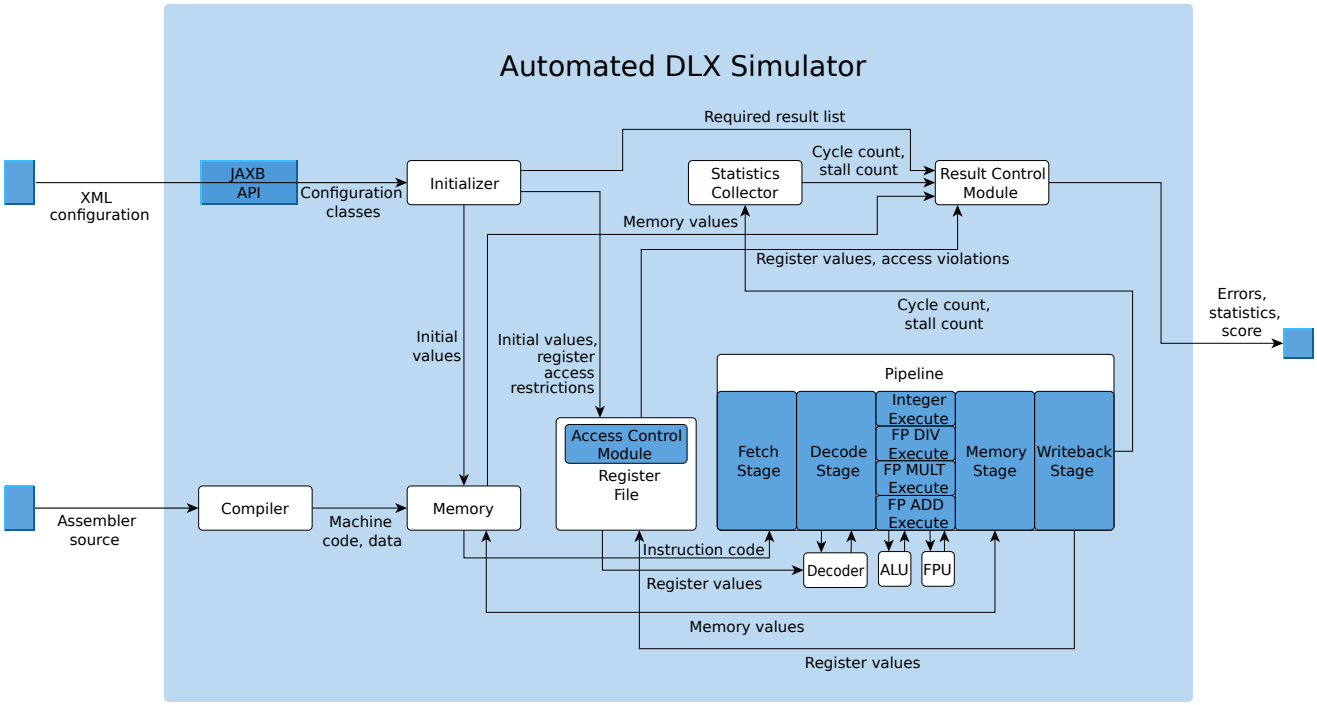

Figure 1: Structure of the simulator.

The simulator outputs compile-time errors and warnings (if any), simulation statistics (number of executed clock cycles, number of RAW, WAW, control, structural and trap stalls), and any violations of register usage restrictions or accesses to undeclared memory.

4.3 Operation

The simulator operates as follows (again, please refer to Fig. 1).

1. The configuration XML is read and processed.

2. The assembly language source file is read and compiled. The resulting machine code, which is binary compatible with the WinDLX simulator [3, 4], is stored to the simulated memory. In case of errors in the source code, the simulation terminates.

3. The DLX pipeline is created. The pipeline always contains five stages (Instruction Fetch, Instruction Decode, Execute, Memory Access, Write Back), with the superscalar floating-point execution stages as configured.

4. The program is executed. The simulator simulates the instructions passing through the pipeline just as the DLX processor would execute them. The simulation stops if the `trap` instruction is executed, if the maximum allowed number of clock cycles is reached (this means that the program has either entered an endless loop, or is extremely inefficient), or if a runtime error occurs (instruction decoding error, division by zero, access to an invalid memory address, or unaligned memory access). Access to undeclared memory is logged but does not stop the simulation.

5. Collected statistics are printed.

6. Results are evaluated. Non-critical runtime problems (access to undeclared memory, violations of register usage conventions) are printed and reflected in the student's overall score. Program outputs (register and memory contents) are checked for correctness and again reflected in the overall score, together with the achieved speed (number of clock cycles used). The simulator outputs a text line with the overall results.

5 Results

The simulator was tested on a set of DLX assembly language sources, mostly consisting of actual submitted solutions to the previous homework assignments. The number of simulated clock cycles and the results

ISBN: 978-1-61804-140-1

(memory contents) perfectly agree with the expectations and with the results of the WinDLX simulator.

There are some minor differences in the stalls statistics between our simulator and the WinDLX. They are due to a slightly different way of counting stalls. For instance, WinDLX counts a structural stall whenever a FP-EX unit is not available. Our simulator counts a structural stall whenever a unit of any type is unavailable (typically when a FP instruction and an integer instruction compete for the MEM pipeline stage). Other differences happen when a stall occurs for more than one reason.

There are also some minor differences in operation. Our simulator checks the source code more strictly for syntax errors; for instance, slightly incorrect `add r1, r2, #4` is permissible in WinDLX, while our simulator strictly requires the correct `addi` instruction. We do not distinguish the different `trap` instructions as WinDLX does; any `trap` instruction simply terminates the simulation. Naturally, the added memory and register access checks are also not present in WinDLX.

The biggest concern regarding the choice of Java as the implementation platform was the overall performance. Fortunately, it turns out that the performance is more than sufficient for our needs. On a common PC computer (quad-core i5 processor at 2.4 GHz, 4 GB RAM), the simulator can simulate approximately 140,000 clock cycles per second. The submissions are terminated after 20,000 clock cycles at the most. Thus, the total time for simulating one submission (including all overhead, such as starting the JVM and compiling the submitted source) is well under one second.

6 Conclusion

We elaborated on the background and reasons for developing a new DLX simulator, and described its structure and internals. The simulator fulfills all the requirements, and its performance is perfectly sufficient for the envisaged use.

At the time of writing this paper, the simulator is still undergoing final testing and debugging. First students will start using it in a few weeks, which will provide us with an opportunity for further extensive testing.

Acknowledgement: This work was supported by the Higher Education Development Fund of the Ministry of Education of the Czech Republic under project No. 1939/2012.

References:

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 2011.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, 2011.

[3] G. Raidl, *WinDLX DLX-Pipeline Simulator* (software), diploma thesis, TU Vienna, Inst. für Technische Informatik, 1991.

[4] H. Grunbacher and H. Khosravipour, WinDLX and MIPSim pipeline simulators for teaching computer architecture, *Proc. of IEEE Symposium and Workshop on Engineering fo Computer Based Systems*, 1996, pp. 412–417.