Development of an Interactive Visual Simulator for the Pipelining Concept

Joselyn C. Lotiba-Canuela
Michael Anthony Jay B. Regis
Visayas State University
Visca, Baybay City, Philippines

Abstract

Pipelining is an optimization technique in which each instruction is divided into sub units overlap with other instructions. Since it is tedious to visualize manually, this study provides a visualization of a five-stage pipeline based on the execution of Instruction Set Architecture (ISA) instruction. A lexical analyzer is used in extracting tokens from the source code and a parser in syntax checking. Instruction dependency and data dependency are also considered during execution with an optional hardware operand forwarding enabled or disabled before program execution. This study implemented in Java™ has a user friendly interface and was able to simulate the execution of sample code. Thus, expands the portfolio of pipeline simulators through using ISA instruction set.

Keywords: Lexical analyzer; Parser; Instruction set architecture; Optimization

Introduction

Computer organization and architecture is generally part of the computer science teaching curricula. This course brings theoretical knowledge and practical understanding to the concept of pipelining (Milutinovic, Radivojevic, & Djordjevic, 2009). The pipelining technique divides a task into subtask then executes each task concurrently by taking advantage of overlapping subtask from adjacent task. It operates in similar fashion to an assembly line in manufacturing. This technique is utilized by modern microprocessors, since it drastically increase its performance (Stallings, 2006). Take for example, the verification method of pipeline simulator generation by Toshiba® (United States of America Patent No. US20010034594A1, 2001), which aims to automate the activity on Registry Transfer Level (RTL).

However two general considerations are needed when formulating the reliability of a pipeline performance. First, when a pipeline doesn’t have prior information of the location for the next instruction to execute - also called as instruction dependency. Second, when a register (or any memory item) is shared between succeeding instructions - generally known as data dependency. So, a time-space chart (i.e. Gantt chart) is necessary to measure and visualize the actual performance of a pipeline (Abd-El-Barr & El-Rewini, 2005).

Computer simulations are key systems used by computer architects to not only evaluate innovation but also provide visualization of theoretical models. Given an actual workload, a simulated result will provide approximate result to actual operation (Martin, et al., 2005). For instance, a simulation of a Worst-Case Execution Time (WCET) analysis on a machine would provide a priority knowledge about the longest possible execution time needed for a program even without running the actual program on an actual machine.

*Correspondence: mbregis@vsu.edu.ph
ISSN 2545-9732
Having the necessary insights on this piece of program provides valuable knowledge for designing and testing real-time embedded systems (Engblom & Ermedahl, 1999).

The book *Computer Architecture: A Qualitative Approach*, Hennessy and Patterson (2007) provides a theoretical view of the pipeline using the DLX instruction set. This provides Zhang and Adams (1995) the model for their pipeline simulator named DLX view running on Unix environment that provides an interactive view of the pipeline during program execution.

On the other hand, a Windows® version aptly named WinDLX developed by Khosravipour and Grunbacher (1996) operates in similar fashion with their Unix counterparts. Furthermore, the authors expand their portfolio by developing a more advanced simulator for the MIPS machine known as MIPSim. This simulator provides a much detailed view to the functional units like register, multiplexers and pipeline registers. A web-based version of the same simulator called WebMIPS developed by Branovic, Giorgi, and Martinelli (2004) to provide ease of access through a web environment without the need for installation.

Since the available simulator operates using the DLX or MIPS instruction set, this directed the authors to implement a user-friendly simulator using the more simplified ISA instruction set, allowing the assortment of pipeline simulator to expand through a new instruction set and an intuitive user interface.

**Materials and Methods**

**Functional Requirements of the Pipeline Simulator**

The major capabilities of the simulator to be developed was thoroughly inspected. These include the implementation of the lexical analyzer and parser when interpreting of a given source code and hardware operand forwarding during occurrences of data dependency.

**User Interface Design**

In order for the user to navigate the simulator with ease, an interactive and a user- friendly user interface was developed (refer to Fig. 1). This allows the user to create a new file (i.e. ISA source code), open an existing file, and save a file. Before simulating the pipeline, the user is prompted to enable or disable hardware operand forwarding.

**Pipeline Simulator Workflow**

Given an ISA source code, the lexical analyzer would convert the characters into a sequence of tokens (i.e. String of characters) categorized according to the rules as symbol. These tokens will be used by the parser to perform syntactic analysis. It would determine the grammatical structure of the sequence of tokens with respect to the formal grammar. Thus, checking for the correct syntax in a given source code. After successfully parsing the code, a visual representation of the pipeline will be generated. Figure 2 shows the graphical view of the pipeline simulator.

**Lexical Analyzer**

Figure 3 shows the Deterministic Finite Automata (DFA) used by the lexical analyzer. DFA is the finite state machine which accepts or rejects the strings of symbol and only produces a unique computation of the automaton for each input string (Hopcroft, Motwani, & Ullman, 2001).
States are label q0 to q11, final states are q1, q2, q3, q5, q7, q8 and q11. States q1 and q2 accept token of memory or variable, state q3 accept token of registers, q5 accept token of integer, q7 for floating numbers, q8 accept comma (,) and q11 accept token of labels.

Parser: Context Free Grammar

Context-free grammar will be used to describe the notation of the syntax (Sebesta, 2012). The instructions included are LOAD, STORE, MOVE, DEC, INC, BEQZ, BENZ, BEQ, BNE, CLEAR, LABEL, ADD, SUB, MULT and DIV expression (refer to Listings 1).

Pipeline Stages Flowchart

Figure 4 shows the flowchart of pipeline stages that would operate concurrently. Every instruction is fetched from the memory, then decoded by the CPU, the necessary operands for the instruction is fetched, followed by the actual execution of the instruction. The results are stored either in the memory or on CPU registers.

During the initial fetch stage, the system would fetch the first instruction pointed by Program Counter (PC) and set Instruction flag (IF_flag) to one. Setting IF flag to one means that the fetch stage already has the instruction and is ready to pass it into the next stage.

If the time is greater than one and the PC is less than instruction size minus one, PC is incremented by one and fetches the next instruction indexed by PC, then set IF_flag to one. When no instruction is left (i.e. PC is equal to the instruction size minus one), the fetching stage is complete. Refer to Fig. 5 for more details.

Figure 6 shows the execution of the Instruction Decode (ID) stage. If IF_flag is equal to one the decode stage would be executed and set Instruction decode flag (ID_flag) to one. After finishing the decode stage the IF_flag is set to zero which means it is ready to fetch the next instruction.

If the decoded instruction is a branch instruction such as BENZ, BEQZ, BNE and BEQ, all stages would stall and wait until the recent instruction would be finished, inserting a stall or wasted clock cycle prevent fetching the wrong instruction. It will wait until Instruction Store (IS) stage will be finished,
<start> ::= <statements>
<statements> ::= <LOAD_expr> | <MOVE_expr> | <STORE_expr> | 
<CLEAR_expr> | <LOAD_expr> | <DEC_expr> | <INC_expr> | 
<BEQ_expr> | <BENZ_expr> | <BEQ_expr> | <BNE_expr> | 
<DIV_expr> | <MULT_expr> | <ADD_expr> | <SUB_expr> | 
<LABEL_expr> ::= LABEL
<LOAD_expr> ::= LOAD (REGISTER | MEMORY) COMMA (INTEGER | FLOAT | MEMORY)
<MOVE_expr> ::= MOVE REGISTER COMMA REGISTER
<STORE_expr> ::= STORE MEMORY COMMA REGISTER
<CLEAR_expr> ::= CLEAR (REGISTER | MEMORY)
<DEC_expr> ::= DEC (REGISTER | MEMORY)
<INC_expr> ::= INC (REGISTER | MEMORY)
<BEQ_expr> ::= BEQ (REGISTER | MEMORY) COMMA LABEL
<BENZ_expr> ::= BENZ (REGISTER | MEMORY) COMMA LABEL
<BEQ_expr> ::= BEQ (REGISTER | MEMORY) COMMA (REGISTER | MEMORY) COMMA 
LABEL
<BNE_expr> ::= BNE (REGISTER | MEMORY) COMMA (REGISTER | MEMORY) COMMA 
LABEL
<DIV_expr> ::= DIV REGISTER COMMA (REGISTER | MEMORY | INTEGER | FLOAT)
[MULT (REGISTER | MEMORY | INTEGER | FLOAT)]
<MULT_expr> ::= MULT REGISTER COMMA (REGISTER | MEMORY | INTEGER | FLOAT)
[MULT (REGISTER | MEMORY | INTEGER | FLOAT)]
<ADD_expr> ::= ADD REGISTER COMMA (REGISTER | MEMORY | INTEGER | FLOAT)
[ADDITION (REGISTER | MEMORY | INTEGER | FLOAT)]

Figure 4. Pipeline stages flowchart

Figure 5. Instruction fetch stage flowchart
only until that time IF stage would know what instruction to be fetched next. But if the decoded instruction is not a branch instruction then the next instruction will be fetched.

The execution of the Operand Fetch (OF) stage is illustrated on Fig. 7. Before fetching the value of the operand used in the instruction, it would check first if there was a data dependency that occurs on the current instruction with respect to the previous one. After checking the data dependencies and if ID_flag is equal to one, the OF stage would be executed. Then, set operand fetch flag (OF_flag) to one and ID_flag to zero. The decode stage will execute if IF_flag is equal to one, otherwise it will return to the position where OF function was called.

If ID_flag is equal to two, meaning OF stage would stall. This means that the current instruction would use the same register or memory from the previous instruction. If that instruction already passed storing stage, operand fetching of the current instruction may proceed otherwise OF stage is still a stall.

However, if the hardware operand forwarding is enabled, after the instruction execute (IE) stage of the previous instruction has finished, the current instruction could automatically proceed to the OF stage.

Figure 6. Instruction decode stage flowchart

Figure 7. Operand fetch stage flowchart

Figure 8. Instruction Store Stage Flowchart

Figure 9. Instruction Store Stage Flowchart
equal to one, the result from the current instruction being executed in IE stage will then be stored in the specified destination. Then set the IS_flag to one and the IE_flag to zero. Otherwise, if IE_flag is equal to two IS stage will stall and proceed to previous stage.

**System Testing and Evaluation**

The pipeline simulator was programmed using Java programming language which can run across different platform and NetBeans Integrated Development Environment (IDE) for easier and smarter way of making the graphical user interface of the simulator. The pipeline simulator was executed on Windows and Linux platform to test the speed of simulating the pipeline during execution of the program. Several sample source codes were also used to check the accuracy of the pipeline simulation.

**Results and Discussion**

**User Interface of the System**

The main user interface consists of text area wherein the source code is written, loaded and edited. The menus such as File, Compile, Run and Help are means of using the simulator (Fig. 10 a). The text area is just a plain editor and there is no automatic guide in making a source code. The supported instructions and its syntax are explained well under the help menu. Under the File Menu are the following menu items: New Open, Save, Save as and Close (Fig. 10 b). After opening or saving a file, the Compile file menu item will be enabled (Fig. 10 b). Upon compiling, the lexical analyzer and parser will extract the tokens and check the validity of the syntax respectively. If the compilation would be successful, a message in the status bar would indicate a successful parsing and the program is ready for simulation (Fig. 10 b). Otherwise, an error will prompt, for user to make the necessary corrections.

After the successful compilation, the Run menu for pipeline simulation will be enabled (Fig. 11a). After clicking this, a pop-up frame will then show up if the user will apply the Hardware Operand Forwarding (HOF) optimization or not (Fig. 11b).

**Figure 10. Screenshots of the user interface when creating and compiling an ISA code**

**Figure 11. Screenshot before pipeline simulation**

Figure 12a shows the Pipeline Generation Interface containing two panels- one for the time-space chart, and the other panel is for the Instruction table. The time-space chart contains control buttons Reset, Run, Step, Back and Exit while the instruction table is color coded for each stage with...
view register/memory values button and view performance measures button.

When the run button is clicked, the time-space chart would be generated to show how the instructions are scheduled on the pipeline (Fig. 12b). However, when the user would opt for manual stage generation it is necessary to clear the pipeline first as shown on Fig. 12c. By clicking the step button shown on Fig. 12d, a step by step simulation would be generated with respect to time. Somehow it will be disabled once no instruction is left to execute.

By clicking the View Register/Memory Values button, a frame would pop-up (refer to Fig. 12e) to show the content of the memory and the register. Finally, three performance measures of a pipeline as shown on Fig. 12f indicate the speed up (Equation 1), throughput (Equation 2) and efficiency (Equation 3).

![Figure 12. User Interface during pipeline simulation](image)

### System Testing and Evaluation

Table 1 shows the different sample programs tested for performance with Hardware Operand Forwarding (HOF) optimization and without optimization. As indicated, when HOF optimization is applied, it would result to a better performance of the pipeline.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Speedup</th>
<th>Throughput</th>
<th>Efficiency (%)</th>
<th>Speedup</th>
<th>Throughput</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>1.85</td>
<td>62.59</td>
<td>37.20</td>
<td>1.52</td>
<td>54.46</td>
<td>30.30</td>
</tr>
<tr>
<td>Accumulator</td>
<td>2.56</td>
<td>328.00</td>
<td>51.20</td>
<td>2.23</td>
<td>52.46</td>
<td>44.4</td>
</tr>
<tr>
<td>Math-Operations</td>
<td>3.33</td>
<td>333.30</td>
<td>68.60</td>
<td>3.12</td>
<td>312.00</td>
<td>62.00</td>
</tr>
<tr>
<td>Nested Branch</td>
<td>4.16</td>
<td>55.70</td>
<td>62.00</td>
<td>3.95</td>
<td>21.62</td>
<td>28.20</td>
</tr>
<tr>
<td>Simple Counter</td>
<td>1.64</td>
<td>185.70</td>
<td>32.70</td>
<td>1.46</td>
<td>97.65</td>
<td>28.20</td>
</tr>
</tbody>
</table>

* Instructions/sec is machine-dependent

### Summary, Conclusion and Recommendations

This study developed a teaching tool in simulating and scheduling a given instruction in each stage of the pipeline. It also demonstrates the aspect hardware operand forwarding as optimization. The system was implemented using Java making it cross platform.

The developed simulator has a user-friendly interface, and it generates the time-space chart representation of the pipeline enabling a better comprehension to the pipelining...
concept. Moreover, the design algorithm for each stage of the pipeline is effective in scheduling the instructions in the pipeline. Since it is cross platform and accurately depicts pipeline operation, it is of valuable help for those who are teaching pipelining in computer architecture.

It is recommended that the simulator should have more supported instructions and would also implement other techniques in optimizing the pipeline such as code reordering, software operand forwarding, and branch prediction. On the other hand, a printing capability may be added so a hard copy would be created out of the generated pipeline.

References Cited


