DARC2: 2nd Generation DLX Architecture Simulator

Roger Luis Uy
De La Salle University
2401 Taft Avenue
Malate, Manila
+(632) 524-0402
uyr@ccs.dlsu.edu.ph

Marizel Bernardo
De La Salle University
2401 Taft Avenue
Malate, Manila
+(632) 524-0402
marizel@yahoo.com

Josiel Erica
De La Salle University
2401 Taft Avenue
Malate Manila
+ (632) 524-0402
Osie_22@yahoo.com

ABSTRACT

Renewed interest in computer architecture education in our university started three years ago. Since then, research framework in computer architecture has been established with emphasis on simulation of different computer architecture concepts. One of the concepts, which have generated a lot of excitement, is the topic on pipelining. Our research group had already developed a pipeline simulator based on the DLX architecture called DARC [1]. The simulator was used as a supplementary tool for both undergraduate and graduate students. It was received favorably and at the same time, they gave feedbacks and suggestions on improving the simulator. With those suggestions, DARC2, the 2nd generation pipeline simulator based on DLX architecture was developed. This paper describes the DARC2 system.

Keywords

Computer Architecture, undergraduate teaching, graduate teaching, pipelining, DLX Architecture

1. INTRODUCTION

In the past, computer architecture education was given less emphasis in our university. But this has change in recent years due to the following reasons:

a.) Creation of Academic Area Chair. The Academic Area Chair is in charge of the development of the curriculum in their respective area. In the past, faculty members who were assigned to teach a particular subject in an academic area, developed their own syllabi and course contents. Thus, there is no continuity in the development of this area. With the academic area chair, he is task to develop a research framework, which will serve as a roadmap for the development of its area. He is also task to make sure that all appropriate textbooks and reference materials are up-to-date. Computer Architecture is classified as one academic area.

b.) Adoption of the computer architecture book "Computer Architecture, A Quantitative Approach" by Patterson & Hennessy [2]. The university has adopted many references but not textbook for computer architecture. Finally, in our opinion, a good book in computer architecture.

The current research framework of computer architecture in our university is focused on the development of simulation tools for the different concepts of computer architecture. In an environment where financial resources are limited, finding less expensive alternatives is always welcomed. With simulators, a quality-learning environment that is equivalent to the actual system itself is presented to students without incurring additional expenditures. Initial project is centered on the pipelining concept based from the DLX architecture. Many students are fascinated with the concept though they have a hard time visualizing them. Initially, simulation is through Microsoft® Excel file, then some students volunteered to write a module, then another. Eventually, a research group was formed to develop a "full-blown" pipeline simulator based on DLX architecture called DARC. Though there are simulation tools on this concept, but each institution is unique in their learning needs and the learning process of developing a pipeline simulation more than justify the development of our own pipeline simulation project.

2. The DARC2 ORIGIN: DARC

DARC2 is the 2nd generation of DLX Architecture Pipeline Simulator (DARC). DARC is a windows-based software system with a built-in text editor. It simulates a DLX code segment using different pipeline algorithms. We defined Pipeline 0 as the un-pipelined version of the algorithm; Pipeline 1 as the pipelined version and Pipeline 2 as the modified pipelined to minimized branch hazard. All of the algorithms are based from [2]. As an added bonus, dynamic scheduling algorithms - Scoreboarding and Tomasulo, are also provided. Users may enter up to 1,024 instructions, with the provision for saving the program. It incorporates a compiler for identifying syntax errors, and a help file that aids the user in correcting such errors. Pipelining results obtained are displayed through a trace of the pipeline stages, while dynamic scheduling algorithms are processed in the standard table form.

The system uses two simulation modes: one-pass and stepwise. One-pass mode allows continuous execution. Stepwise mode, on the other hand, allows instructions to be simulated one at a time. The simulator can be configured to support either shared or separate memory as option to illustrate structural hazard. Forwarding and non-forwarding are used as option to visualize data hazard. While control hazard can be resolved using pipeline freeze, predict-not-taken and pipeline 2. During simulation, hazards encountered are displayed and explained to the user.

As seen, DARC demonstrates great aid in the study of the DLX architecture. The simulator was initially offered to around 160 students in four separate classes. It was a sighed of relief for them since they could now visualize and experiment different options and situations in pipelining. They also suggested several changes, correction and enhancements. One such change is that the memory contents should be displayed as 32-bit word instead of displaying it as byte. Another suggestion is that there should be a provision for breaking out of infinite loops. The system also does not support floating-point values in IEEE standards. Students also reported some inconsistencies with the results obtained. They also suggest that besides that standard "pipeline" view, they could also visualize the flow of data to the individual components of the DLX architecture (i.e., pipeline registers, program counter) as well as the generation of the control signals. These suggestions warrant a major design of the simulation system. Thus, a new version of pipeline simulator is created – DARC2.

3. DARC2: DLX ARCHITECTURE SIMULATOR 2

DLX Architecture Pipeline Simulator 2 (DARC2) is an improved version of DARC. It is a windows-based system that utilizes graphical user interface to simulate both DLX pipelining algorithms and dynamic scheduling algorithms. Unpipelined instruction execution, Pipeline 1 and Pipeline 2 algorithms are now illustrated both through the standard tracing of the pipeline stages and through an animated diagram of the DLX data path. The animation shows the data flow through the major components of DLX architecture. DLX instructions are showed as they are processed, together with the control signals associated with them. Through the animation, the user is informed on how each internal component works and on the actual process of passing along data from each unit. To ensure consistency, all the diagrams are presented in the same manner as in [2], matching both the look and the function of the architecture. This improves the learning process and the usability of the simulator. On the other hand, results obtained through the dynamic scheduling algorithms – Scoreboarding and Tomasulo – are shown in the usual tabular form.

As with the original version, there are two modes of simulation: one-pass and step mode. Apart from this, the user also chooses among other simulation options specific of each pipelining algorithm. These options are forwarding, pipeline freeze, pipeline flush, predict-not-taken, and the use of unified or separate memory. The chosen options are greatly important since they set the conditions to be followed during the simulation process.

Moreover, users are given the freedom to choose which memory address to be assigned as base address for the instructions in the program code. This is especially useful when unified memory is utilized. The system also enables the user to change the value of the registers and the memory at any point during the simulation. With this, users can test different register values without having to construct another program code. Changes made would apply to DLX instructions that are not yet decoded in the Instruction Decode (ID) stage.

Another feature integrated to the system is the option for backtracking. DARC 2 allows values stored in the pipeline registers, as well as the other registers to be viewed at any clock cycle. From the final trace of the pipeline diagram, the user may choose the required clock cycle and the corresponding data path diagram showing data stored in the different components is displayed. During the simulation process, the system also gathers statistical data that aids in determining the effectiveness of the DLX program code used as input. The number of hazards encountered is considered and updated per clock cycle of the simulation, after which it is displayed.

A text editor, syntax checker and help system are also incorporated into the system. The text editor allows the user to key in the DLX instruction code to be simulated. The syntax checker checks the instruction code for syntax errors and notifies the user if such are encountered, providing opportunity to correct them. The help function provides topics and references related to the DLX Architecture.

4. SYSTEM STRUCTURE

The DARC 2 system is an integration of several major components that work together to achieve the requirements and specifications of the simulator. The interaction of these components - the text editor, the assembler and the function manager – is illustrated in the system's block diagram. A more detailed description of each of these components follows.

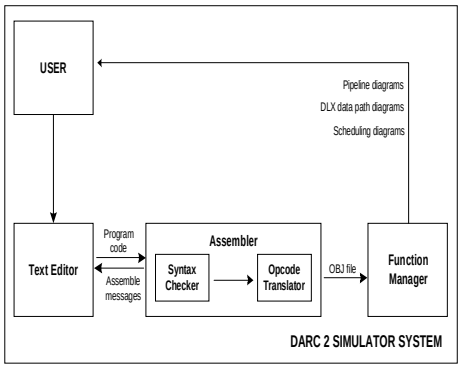

Figure 1. Block Diagram of DARC2

4.1 Text Editor

The built-in text editor of the system, as illustrated in Figure 2, accepts a DLX assembly code consisting of at most 1,024 instructions. The program code comprises of instructions written in DLX mnemonics, constructed either from scratch or chosen among the initial sample programs installed in the system. The text editor also allows the user to save written assembly codes for later use.

The program code entered contains not only DLX instructions, but also data variable declarations as well. The user defines all variable declarations in the. DATA segment. All DLX instructions are included in the separate .CODE segment. Immediate addresses included within the instruction code itself may be in decimal, binary or hexadecimal form. For decimal, the format is *#immediate*. For binary, the format is *$immediate*. Unless written in any of these formats, the immediate is considered as hexadecimal.

After the user has keyed in the DLX instruction code through the text editor and clicking the Assemble button, illustrated in Figure 2, the assemble options window is displayed. The user inputs the base address of the instructions in the program code to be simulated. Upon which, the program code together with the value set as base address is passed on to the assembler for processing.

4.2 Assembler

The assembler module handles the processing and translation of the DLX mnemonics. The module is comprised of two sub-modules: the syntax checker and the opcode translator. After the user has created the instruction code through the text editor, the syntax checker checks the code for syntax errors. If no errors are identified, the program code is then passed on to the opcode translator. However, if errors arise, assembling is considered unsuccessful and error messages are displayed to the user. The opcode translator then gets the instruction mnemonics and translates them into opcodes, that are in turn used as input to the simulator.

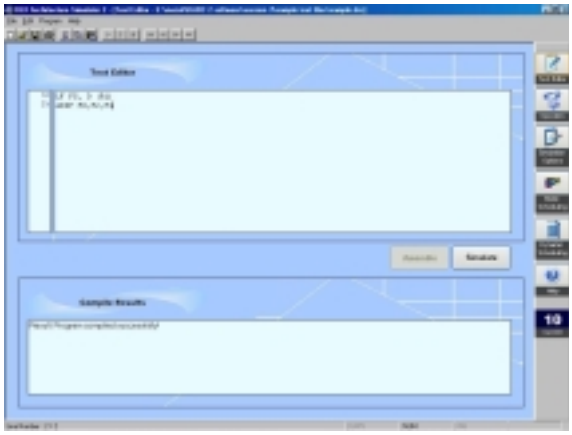

Figure 2. DARC 2 Text Editor

4.2.1 Syntax Checker

The Syntax Checker handles the instruction code written through the text editor. Each instruction line is checked for syntax errors. If there are no errors, the program code is passed on to the Opcode Translator. In the case that an error is encountered, the Syntax Checker takes note of the line number and proceeds with checking the remaining instructions, until the last instruction is reached. A message box is then displayed to inform the user of unsuccessful assembling and of the line numbers where the errors were found.

Upon receiving the program code, the Syntax Checker identifies variable declarations by checking the presence of a .DATA segment. If such exists, variable declarations are checked for correctness. Format should be *label = target address.* Label names or variables can be alphanumeric. Special characters, except for the underscore (_), are not allowed. The labels and their corresponding target addresses are stored in a temporary array for reference during simulation. In case there is no defined .DATA segment, all text input are considered DLX instructions.

Following the .DATA segment is the .CODE segment, consisting of the DLX instructions to be simulated. With each instruction, the DLX mnemonic is compared with each entry in a library of DLX instructions, called *main.lib*. The format of R-type instructions contained in the library is *mnemonic : instruction type : opcode;*. I-type and J-type instructions, on the other hand, have the format *mnemonic : instruction type : opcode : EX : MEM : WB;*. The opcode is an assigned two-digit binary code for each of the instructions. The *EX*, *MEM* and *WB* portions of the library entries

pertain to the attributes of the particular instruction during simulation, specifically during the Execute (EX), Memory (MEM) and Write Back (WB) stages. In *EX*, the type of ALU operation (e.g., memory reference, register to register, register to immediate, or branch), the type of operation (e.g., basic arithmetic, logical comparison, shifting, conversion, move, or comparison) and the type of registers (e.g., immediate, floating-point or double precision) are all stated. Instructions may include additional details such as the sign of the result (e.g., signed, unsigned or floating-point) and the type of the extension (e.g., sign or zero). This is primarily due to the differences in the way instructions are executed. Thus, their parameters also vary. In the *MEM* portion, it is noted if the instruction is active during the MEM stage. For active instructions such as Load/Store instructions, the length of data to be loaded or stored (e.g., byte, halfword, word, single-precision floating point or double-precision floating point), together with the type and sign of the destination register, is indicated. Evidently, entries for other instructions inactive during this stage do not include such parameters. In *WB*, the type of operation done (e.g., register-to-register integer, register-to-register floating-operation, register to immediate, among others) is specified. All information indicated in *EX*, *MEM* and *WB* portions of the library entries are accessed later on during simulation and are used as bases in the methods that are to be performed.

Once the corresponding mnemonic is found, a library, called *type.lib*, which contains the different formats for each R-type, I-type and J-type instruction, is accessed. The library is searched for the corresponding mnemonic in order to find the correct format of that particular instruction, and compare the instruction against it. For *r-type instructions*, the format is *mnemonic <operandtype datatype>,<operandtype datatype>,<operandtype datatype>;*. *Operandtype* may be *rd*(destination register), *rs1* (source register 1) or *rs2* (source register 2). *Datatype* may be *i* (integer), *f* (single precision floating point) or *d* (double precision floating point). Operands are enclosed in brackets and are separated by commas. For *i-type instructions*, instruction formats vary for every instruction available. For instructions with memory access, the immediate is checked if it has been declared. The temporary array previously initialized is searched for a match. If the label is not found, it is then checked if it decimal, binary, or hexadecimal, by checking the first character. # denotes a decimal immediate, while *$* denotes a binary immediate. In any case, the immediate is treated as hexadecimal.

Aside from ensuring that the instructions in the program code are of correct format, the validity of the operands used is also verified. Registers are ensured to reach until R31 only, and that R0 is not the destination field. For branch instructions, it is first determined whether the target address exists or not. For double precision floating point instructions, only even-numbered floating-point registers should be used.

In general, the Syntax Checker is not case-sensitive. The presence of commas and spaces between instruction fields are considered. Operands should always be separated by commas. Space in between the mnemonic and the first operand is important, while those in between commas and operands is negligible.

4.2.2 Opcode Translator

After the program code is checked for syntax errors, it is then passed on to the Opcode Translator. This sub-module decodes the program into codes conforming to the DLX instruction formats. For every instruction in the program code, the DLX instruction library *main.lib* is accessed. Each entry in the library contains the instruction type and opcode of different instruction mnemonics. The format for each entry is *mnemonic : instruction type : opcode*. The mnemonic of the instruction being translated is compared against each entry in the library. Once the matching mnemonic is found, its corresponding opcode is obtained. If the opcode obtained is *00*, a separated library, *special.lib*, is accessed. This library contains the mnemonics and the corresponding opcode of DLX instructions that involve general purpose registers. If the opcode obtained is *01*, the *fparith.lib* library is accessed. *Fparith.lib* contains entries of DLX instructions using floating-point registers. The entries contained in *special.lib* and *fparith.lib* follows the same format as those in the *main.lib*, each consisting of the mnemonic and a corresponding code. The difference is that, this code is for the *function* of that instruction, which constitutes the last 11 bits of the code. After obtaining the opcode, the operands are translated into their corresponding binary codes.

To illustrate further the decoding process, suppose the instruction ADD R1, R0, R2 is to be decoded into its corresponding code. This instruction is an R-type instruction and follows the format, illustrated in Figure 3.

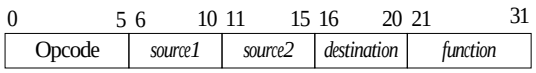

Figure 3. R-type instruction format

From *main.lib*, the entry of the ADD instruction would be *ADD : R-type : 00*. The value of *opcode* field is then 00. Then, the value of the *function* field is divided into two – the first 5 bits, which is unused and has the value of 0, and the last 6 bits, which contains the opcode from the *special.lib*. In this case, the opcode of the ADD instruction is 00. Therefore, the final code of the instruction in binary code is shown in Figure 4.

0 5	6 10	11 15	16 20	21 31
000000	00000	00010	00001	00000 000000

Figure 4. Sample ADD code

If errors are encountered before decoding is finished, the translator terminates without completing and error messages are displayed to the user. If there is no error, an object file is created. This file contains the corresponding binary code of each of the instructions in the program and is forwarded to the function manager to execute the necessary algorithms.

4.3 Function Manager

The function manager serves as the core of the DARC 2 system. It handles all the algorithms that the system uses, and implements the specifications defined by the user for each algorithm. It receives as input the binary codes of all the instructions in the program, as generated by the assembler.

It receives as input the object file generated by the assembler. The file contains the binary codes corresponding to each instruction of the program. The function manager analyzes the first 6 bits of each code and determines the type of instruction that will be executed and the type of operands that it will have. There are only three types of instruction: ALU operations, Load/Store operations and Branch operations. Look-up tables, containing the instructions under each type and their corresponding 6-bit code, are used in executing the different algorithms since there are different executions for different types of instructions.

The function manager implements pipelining algorithms – unpipelined, pipeline 1 and pipeline 2. After creating the codes, the count of the clock cycle starts. The memory, as well as registers and pipeline registers, are updated every clock cycle and whose values are stored in a text file.

The configurations defined by the user among the simulation options have different effects and implementations on the algorithms. These differences are reflected more in the data path diagrams than the pipeline diagrams, since it is the data flow that differs mainly with each pipeline algorithm. The data path diagram and pipeline diagrams are illustrated in Figures 5 and 6.

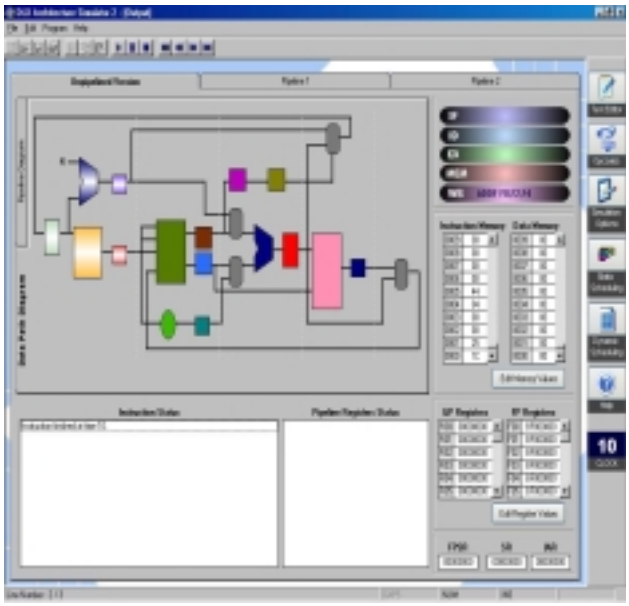

Figure 5. Output Window for Pipelining (Data Path Diagram)

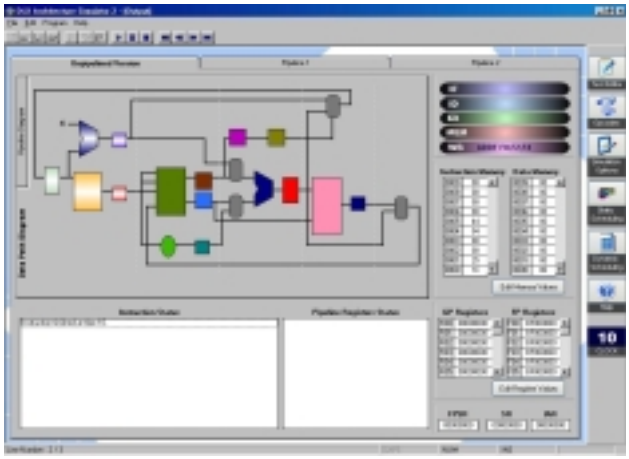

Figure 6. Pipeline Diagram of the Output Window

Unpipelined execution, Pipeline 1 and Pipeline 2 are each treated as modules. The modules contain different procedures each representing the pipeline stages. Each procedure involves only the components present within the stage. It accepts input, such as register values, from one stage, and the necessary methods are performed. The required output produced is then passed on to the next procedure. If it is pipelined execution, this does not necessarily mean the next pipeline stage. Thus, each stage is indifferent of what instruction is currently processed, and is concerned only with the methods it needs to accomplish.

Before the simulation begins, the value of the Program Counter (PC) of the last instruction is noted. This is for purposes of monitoring the end of the simulation. Then, during the IF stage, the first Program Counter (PC) is used as input. The Instruction Register (IR) takes the value of the memory location pertained to by PC. This is the first instruction, whose first six bits are then analyzed in the ID stage. Since instructions differ mainly in the EX, MEM and WB stages, each instruction is treated differently. The conditions to be followed during these stages are indicated in the corresponding entry of each instruction in the library files *main.lib*, *special.lib* and *fparith.lib*. Each instruction entry found in any of these libraries include the EX After noting these conditions, appropriate methods are performed. After the WB stage, the Next Program Counter (NPC) is checked and is compared against the value of PC of the last instruction noted earlier. If the NPC is greater, this means that the end of the program code has been reached and that the simulation is finished. Else, the next instruction is fetched and the simulation continues.

4.4 Infinite Loop

An instruction is said to be an infinite loop when it has exceeded the intended frequency of execution, usually brought about by logical errors made. To ensure accurate results, the system is equipped with the ability to detect infinite loops. This facility is also presented in the simulation options windows, wherein the user defines the number of times a certain instruction is executed before it can be considered an infinite loop. This becomes the threshold of the frequency of every instruction execution, and is applicable to both unpipelined and pipelined algorithms. To monitor infinite loops, each instruction is then assigned a counter, which counts the number of times that particular instruction is executed during simulation. If the counter value exceeds the threshold defined, that instruction is said to be an infinite loop. Whenever an infinite loop is encountered in either the unpipelined or the pipelined algorithms, simulation is terminated for that particular algorithm. Simulation of other algorithms continues unless an infinite loop is also met.

For instance, a user defines ten (10) as the frequency threshold in the simulation options window. This means that if a certain instruction is executed more than 10 times as indicated by the instruction's counter, it will be considered an infinite loop. There may be cases wherein an infinite loop is encountered only in one pipeline algorithm and not in the others. An instruction may be loop infinitely in pipeline 1, but not in the unpipelined execution and in pipeline 2. In this case, only the simulation of pipeline 1 terminates; simulation of unpipelined execution and pipeline 2 continues.

5. CONCLUSIONS AND FUTURE WORK

DARC 2 provides an effective environment for the simulation and exploration of the DLX Architecture. The animation facility in the system proves useful for allowing the students to visualize the interaction of different components employed in the DLX architecture. It allows students to understand better the flow of data through the architecture and how each instruction is executed, while removing the difficulty that is often experienced with manual tracing and redrawing of pipeline and data path diagrams. With all the improvements incorporated in DARC2, it is hopeful that new batch of students will have a clearer understanding of the pipeline concept. It is hopeful that this will boost the appreciation of studying computer architecture even higher and thus, creating great possibilities of constructing new architectures.

Subsequent improvement involves adding advanced concepts such as superpipelining, superscalar execution, cache memory, branch target buffers and others to the simulators. Eventually, the simulator will evolve to a DLX Virtual machine. This is similar to the "Java machine" concept. In the virtual machine, actual DLX code will be executed in x86 machine. The DLX Virtual Machine projects will involve modules relating to runtime operating system, a compiler module to convert high-level language to DLX and others. With the research framework in place, our university is excited with the revival of Computer Architecture field.

6. ACKNOWLEDGEMENT

The author would like to acknowledge the 1st generation DARC development team composed of Mr. Jonathan Lee, Jonathan Ray Roque, John Jerrick Sy, and Aldrich Nino Lorenzo. This project, which is the development team's undergraduate thesis project also won grand prize award sponsored by a local software institution.

7. REFERENCES

[1] Uy, Roger Luis, Lee, J., Roque, J.R. Sy, J.J., and A.N. Lorenzo. *DARC.* Undergraduate Thesis, De La Salle University, Manila, Philippines, 2003.

[2] Hennessy, John L., and Patterson, David A. *Computer Architecture: A Quantitative Approach 3rd Edition*. Morgan Kaufmann Publishers, 2003.