University of Edinburgh

Department of Computer Science

Online Teaching Aid: A Simjava DLX Simulation Environment

M.Sc. Project Report

Iain S. Patterson

Abstract

The aim of the project was to develop an on-line teaching aid for computer architecture. This was to address the difficulties associated with the teaching and understanding of modern high performance computer architectures. Developed in JAVA[1] and utilising the SIMJAVA[2] simulation package, the project successfully models a RISC load/store architecture, with pipelined instruction execution and multicycle arithmetic operations. The simulation can be customised by changing the instruction code to be simulated and also the latencies of various components of the system. This then provides the user with a powerful tool which can be used to follow the animated execution of assembly instructions through the model, as structural, data and control flow hazards are encountered and dealt with at run time.

[1]JAVA is a registered trademark of *Sun Microsystems, Inc.*

[2]SIMJAVA was developed at *The University of Edinburgh*

Acknowledgements

I would like to thank Professor Roland Ibbett for his invaluable help and guidance throughout the project.

I would also like to thank Dr Fred Howell for his incisive programming awareness and Paul Coe who provided valuable insight into the potential implementation pitfalls of the architecture simulation.

A special thanks to my parents and family who have put up with me longer than anyone else I know.

Last, but by no means least I would like to thank ALL my friends, new and old, without whom life just wouldn't be the same.

Any feedback is welcomed and I can be contacted at *isp@dcs.ed.ac.uk*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The aim of the project was to develop an online teaching aid for computer architecture. Central to this aim was the objective of providing an interactive environment for user defined simulations. This is in an attempt to address the difficulty of teaching modern high performance architectures, with their increased utilisation of concurrency techniques and instruction level parallelism.

The technical complexity of a modern microprocessor, when combined with the transparency of its operation and the millions of internal states associated with it at any given time are conceptually beyond the bounds of human comprehension. This in turn can make it difficult for students of computer architecture to gain a complete picture of the internal workings of a computer, beyond that of simple isolated examples e.g. pipelining. Indeed in the case of pipelining it can be difficult for the student to obtain proper insight into how an architecture might deal with pipeline stalls or the various methods used for branch control and prediction.

It is also the case that as computer manufacturers strive to build better, faster machines the inclusion of fundamental concurrency techniques are formed into hybrid solutions with unique characteristics specific to the exact implementation. Whilst these real issues are hardware specific such hybrids do not lend themselves to the fundamental teaching of concurrent computer architectures. Tools presently available for the teaching of computer architecture are limited to simple examples of lone components and for more detailed processor examples it is necessary to use packages requiring some level of familiarity themselves before being of use.

Coupled with the increasing performance of modern architectures, the advent of the World Wide Web (WWW) and more recently the introduction of the JAVA programming language has led increasingly to the availability of remote and complex information, from basic text or images to working JAVA applications. Indeed web based simulation tools have become a highly focused area of research, with their far reaching potential for global dissemination of information. One such tool, developed at the University of Edinburgh, is SIMJAVA [McN&H 97]. SIMJAVA is a process based discrete event simulation tool developed in JAVA and capable of graphically animating simulations included in web pages.

In order that a suitably detailed architectural model was simulated the project concentrated on the hypothetical model DLX architecture proposed by Hennessey & Patterson [H&P 96]. This architecture was developed primarily for the teaching of high performance computing techniques and as such has been reasonably well documented in terms of its behaviour. It has also been the focus of a number of attempts to simulate the behaviour in a web based environment such as the Casle project at Purdue [PURDUE 97]. However, of those simulations available today, many lack the necessary visual state changes that assist in comprehending the actions of the machine under running conditions. Some of the work focussed upon in Chapter 3 highlights this disparity between textual descriptive representations of behaviour and the more visual attempts.

This project also aimed to provide the user with more interaction than that which is presently possible with existing models. The Hierarchical computer Architecture Simulation Environment, also developed at the University of Edinburgh, provides for the rapid prototyping and simulation of computer architectural models. It provides the user with ample opportunity for interaction enabling abstraction of components at various levels and also selection of user settings. Indeed a basic working model of the DLX architecture was implemented in 1994, as part of an undergraduate project, that permitted a program to be run and through which the user could observe the various state transitions as an animated model. However, this model was limited by its level of accessibility and furthermore later releases of the HASE package have left the old model obsolete.

This project addresses these and other issues, ultimately providing a tool that can bridge the gap between the actual low level logic and the higher level concept of what computers actually do. The project successfully developed a working model of the DLX instruction set architecture with pipelined instruction execution, multiple parallel functional units and a scoreboarding mechanism of control. This model has the ability to simulate user defined instruction code and provides the option to modify the latencies of various entities within the system. As such it provides a useful teaching aid for computer architecture illustrating the effects of pipelining and the hazards incurred by a pipelined architecture.

Chapter 2 provides a brief overview of the necessary areas of the JAVA programming language, the DLX model architecture and also the SIMJAVA implementation issues that must be considered. A review of the current literature relating to the area of computer architecture simulation is presented in Chapter 3. This defined many of the goals of the project and how it could improve upon those tools currently available. The design issues brought to light from this research are presented in Chapter 4 and the implementation issues of the project feature in Chapter 5. A short discussion of the development and problems encountered is then highlighted in Chapter 6 before conclusions are drawn from the work undertaken.

Chapter 2

Java, Simjava and The Dlx Architecture: An Overview

This chapter highlights the essential features of JAVA and how the SIMJAVA package extends JAVA to handle discrete event simulation. The focus of the chapter is upon those issues pertinent to the development of the project, though recognises the significance of other JAVA language traits. Following this a brief overview of the main DLX architecture features are presented.

2.1 Java

JAVA was developed initially as a language for consumer electronic devices [SUN 96]. Based on *C*, *C++* and other existing languages it aimed to provide a simple, object oriented, statically typed, interpreted, architecture neutral, multi-threaded, garbage collected, robust, secure and extensible language [M^{c}N&S 97][N&P 96][FLAN 96]. These characteristics of JAVA make it one of the leading software development tools currently available.

2.1.1 Strengths of Java

For the purpose of web based simulation the key strengths of JAVA are:

Distributed & Dynamic Java is driven by its potential for handling the distributed environment of the Internet, with the ultimate ability to transmit dynamic objects, i.e. a dynamic self-executing program, across a network. Such dynamic objects become active agents on the client computer even though they are initiated by the server. This is of particular relevance to online simulation and modelling as it provides for remote use of applications without the need for download and storage facilities at the client site. Java also manages the cross-platform distributed environment of the Internet by providing TCP/IP protocol compatibility. Furthermore, Java permits the dynamic linking of code and in the *applet* environment small fragments of bytecode can be dynamically updated on a running system.

Platform Independent Java's ability to transmit dynamic objects is strengthened by the interpreted nature of the compiled intermediate bytecode and its platform independence. This intermediate bytecode can then be interpreted and executed on any processor with the presence of the Java runtime system. High performance cross platform program compatibility increases the potential to reach a wide base of users and an online teaching aid would benefit from this.

Multi-threaded Java supports a multi-threaded runtime environment and provides a sophisticated set of synchronisation primitives. This permits programs to be developed with greater emphasis on the actual program behaviour, as opposed to controlling the multitasking subsystem upon which the application is to be run. Furthermore, this capability provides for greater interactive responsiveness and improved real time behaviour. This is a desirable feature for the simulation of a complex system, itself representing concurrent activities.

2.1.2 The Java *Applet*

A Java *applet* is a small application that can be executed on a local machine in a 'safe' environment. This environment is provided by the language constructs eliminating many of the usual programming bugs, e.g. array bounds, and through the fact that the language is compiled into intermediate bytecode before it is transferred across the network. The bytecode interpretation also serves to provide verification of the *applet* prior to processing on the local machine. However, this verification also limits many of the capabilities of the application in respect to file or network access.

With their hyper-text-markup-language (HTML) compatibility, *applet*s have the potential to be incorporated as part of a web page. The *applet* class has a number of clearly defined constructs to provide for this. These are defined in terms of an *applet*'s application programming interface (API) and also the life-cycle of the *applet*. The *applet* API serves as the base API for the development of JAVA based applications and *applet* development. As such it includes all of the classes available in the core JAVA package. With regard to the lifetime of an *applet* there are four fundamental stages. These are:

- initialisation
- start
- stop
- destroy

2.2 Simulations in Java

The present euphoria surrounding the JAVA language has led to the growing interest in the field of web based simulation [WEBSIM 97][1]. Recent tools include JAVASIM, JSIM, SIMJAVA and SIMKIT. Such tools have provided simulations ranging from a canal lock system to virtual memory simulation. For the purposes of this project SIMJAVA was chosen as the desired tool for development. This was because of its capability to produce on-line web based simulation models and also the availability of 'in-house' support.

[1]It is noted that the WebSim'97 survey web page refers to all the sim-packages mentioned `http://ms.ie.org/websim/survey/survey.html`

2.2.1 The Simjava Package

SIMJAVA[2] is a discrete event simulation package developed in JAVA. It can produce stand alone simulations and also has the potential to include animated *applets*, visually depicting the states of the simulation entities over the simulation time frame. At its most basic level a SIMJAVA simulation is composed of a collection of entities with their behaviours described. These entities have the ability to communicate through ports with each other and are controlled by a *Sim_system* class, whose responsibility it is to maintain thread control, advance simulation time and maintain event queues. In brief the process involves the following four steps.

1. Extend the standard *Sim_entity* class, overriding the *body()* method with the entity behaviour using *Simjava* runtime methods.
2. Instances of the new entities must then be added to the *Sim_system* static object with the method defined by *Sim_system.add(entity)*
3. Ports between entities are then linked with the *Sim_system.link_ports()* method.
4. The simulation can then be run to generate a trace_file of the output results of the simulation.

2.2.2 The Simanim package

On top of the SIMJAVA package is the SIMANIM package which permits various run time methods of the system to be incorporated into an applet. Entities and ports can be represented by their own user defined icons and it is possible to display event objects being passed between ports. In addition it is also possible to animate the state changes of entities with various icons and display text string parameters next to entities, thus providing the user with further information. The updating of this is controlled from the trace-file output

[2]At the time of writing the most current version was *Simjava-V1.2*

by the simulation as it is run. This trace-file information records either the sending of an event or the updated information used to display an entity.

2.3 The Dlx Architecture

The DLX is a hypothetical RISC type load/store architecture. It was conceived to highlight the importance of pipelining capabilities, the benefits of an easily decoded instruction set and the compilation of high-level programs into efficient machine language code [H&P 96]. Fundamentally the DLX architecture has the following features:

- 32 general purpose registers
- 32 single-precision floating point registers
- Several miscellaneous registers e.g. for interrupt handling
- Word length of 32 bits
- Byte addressable memory (big endian) with 32-bit memory addressing

2.3.1 Register Usage

Of the 32 general purpose registers, R0 always contains the value 0 and R1-R31 are truly general purpose. Register R31 however, is used to remember the return address for jump instructions. It is possible to load a register with a byte, halfword or fullword quantity. These are numbered 0-31 with bit 0 being the most significant and bit 31 the least significant. This policy is also true for the Byte ordering of registers. See Figure 2.1 depicting the various numberings for the three types of register operands, namely general purpose, single precision floating point and also double precision floating point. Double precision is attained by use of two single precision registers, thus permitting only 16 double precision floating point registers at any one time.

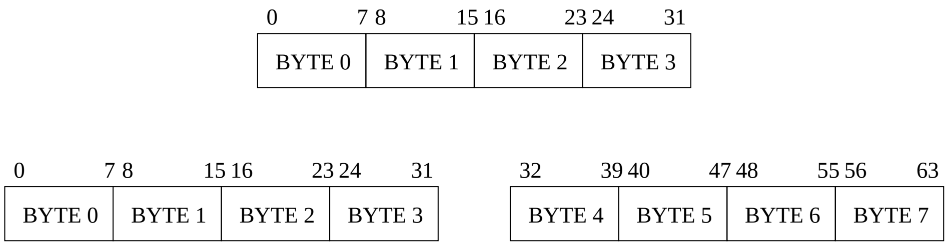

Figure 2.1: Register Numbering

It was indicated that the DLX architecture utilises several miscellaneous registers, these are the program counter (PC), the interrupt address register (IAR) and the floating point status register (FPSR). The PC contains the 32-bit address of the instruction currently being retrieved from memory for execution and is an implicit operand for all jump (non-register), branch and trap instructions. If the current instruction is not a taken branch, jump or trap instruction then the PC is incremented. Should a TRAP instruction be encountered then the 32-bit IAR maintains the return address for the interrupted program and is used by the return from exception (RFE) instruction to restore the PC. The single bit FPSR provides for conditional branching instructions to act depending upon the value of the FPSR.

2.3.2 Memory Transfers

The DLX architecture being purely load/store only permits memory access through explicit load/store instructions. These instructions are displacement addressed and follow the format below.

32-bit Unsigned Effective Address = 32-bit Register + Signed 16-bit Offset

2.3.3 Interrupts

The DLX has two interrupts within its instruction set, the TRAP instruction and the arithmetic overflow exception. Arithmetic overflow exception can occur as a result of a two's complement overflow, whilst the TRAP instruction causes the PC to be loaded with the specified absolute address and the IAR is loaded with the return address. The RFE as mentioned previously restores the PC from the IAR.

2.3.4 DLX Instruction Set Summary

Of the 92 instructions provided by the DLX there are found to be six distinct classes: (A complete summary of the instruction set can be found in Appendix A.1)

- Load and store instructions
- Move instructions
- Arithmetic and logical instructions
- Floating-point instructions
- Jump and branch instructions
- Special instructions

DLX instructions are aligned on a word boundary and there are three instruction formats used. This provides for the efficient decoding of instructions and are classed as follows:

I-type(immediate) The I-type instruction includes load and store instructions and also all immediate ALU operations and conditional branch instructions. (See Figure 2.2)

Opcode	rs1	rd	immediate
6	5	5	16

Figure 2.2: DLX I-Type Instruction Format

R-R ALU	rs1	rs2	rd	unused	func
6	5	5	5	5	6

R-R FPU	rs1	rs2	rd	unused	func
6	5	5	5	6	5

Figure 2.3: DLX R-Type Instruction Formats

R-type(register) R-type instructions are used for register-to-register ALU operations, reads and writes to and from the special registers and/or the floating point registers. (See Figure 2.3)

J-type(jump) J-type instructions include the jump, jump and link (JAL) and return from exception (RFE) instructions. (See Figure 2.4)

Opcode	name
6	26

Figure 2.4: DLX J-Type Instruction Format

Chapter 3

Computer Architecture Simulation & The Web

It is only in recent years that the area of web based simulation has come to the forefront of research. The advent of Java and its ability to incorporate executable programs at remote locations has furthered this drive. This chapter addresses the recent developments in the field of web based simulation, with specific reference to the tools available for the comprehension of computer architecture.

3.1 Existing Simulation Tools

In response to the question of how students are educated using simulation, it is recognised that of the packages available today most incorporate graphical front ends, thus permitting the student to learn through virtual world exploration. The benefits of the web, with its ability to hold information and permit access to it, without requiring local storage are highlighted. Furthermore it is recognised that web based simulation is still in its relative infancy and it is only through time that an improved way of performing simulations will emerge [FISH 96].

3.1.1 Pre-Java

Of the simulation tools for computer architecture available today, the DLXsim package developed at Purdue University is perhaps the most widely known and well documented. [H&M 96] This project has been in existence for a number of years and permits the textual simulation of the DLX instruction set architecture. The package, once downloaded, permits the user to run assembly code as though it were active on a real machine and the stages the code would go through are described in a step by step fashion. Recently the Casle project, of which DLXsim is a part, incorporated an extension, DLXview, to graphically display 'snapshots' of the architecture at the various cycles of running. Both packages require some familiarity and require the user to learn operating instructions. As a simulation is run the changes in the state of the system are shown by colouring of the relevant buses.

The Hierarchical computer Architecture Simulation Environment (HASE)[CW&I 97] project provides for extensive and relatively rapid development of hierarchical models of computer architecture. Indeed this is the purpose driving the project. It can incorporate graphical and textual commentary as a simulation progresses and benefits from its architectural focus, providing the user with associated component libraries with which new models can be built. In addition HASE provides the user with customisable textual windows and the ability to select the level of abstraction. However, HASE also requires that the user have access to storage space in order to compile and run simulations. Selecting various options from the user menu it is then necessary to recompile the simulation prior to running.

At yet another level of detail the DLX architecture has been taken down to a working VHDL model (Very high speed integrated circuit Hardware Description Language) [THER 97]. This details the behavioural description of the architecture in such a fashion so as to permit simulation of its logic at different levels of representation. This can ultimately be taken down through a data flow representation to gate level logic capable of synthesis and manufacture. However, as a potential aid for teaching this is more appropriate for students of electronic engineering and application specific integrated circuit (ASIC) design. As such it provides bit level representation of the behaviour of the logic but there is a loss of the abstract representation of the architecture. Verifying the behaviour requires that the user studies trace-file output and this must then be checked against the desired behaviour. It requires that the user be competent in VHDL and also that the operation of the system be clearly understood beforehand.

3.1.2 Java Based Tools

It is widely acknowledged that Java offers the potential to provide for the rapid development of intuitive user interfaces with its graphical capabilities [McN&S 97]. In turn this ability is identified as a powerful platform for developing object oriented applications and, more specifically, the Java language is recognised for its potential in the area of simulation applications development [B&S 96].

Some of the first online simulations in Java devised *applets* which addressed the aim of visually representing the changing state of a system as the simulation is run. This increased potential for *learn-ability* [DFAB 93], through visual interaction, attempts to improve the effectiveness of the tool as a teaching aid, leading to improved understanding of the model represented. However, as Java was in its infancy these tools offered little interactive option for the user, detracting from the flexibility of the system as a tool for the exchanging of information.

Current simulation tools are broadening their scope and beginning to recognise the benefits of online accessibility, user interaction and an abstract visual representation of the model in question. The SimPack [SIMPACK 97] simulation model of cpu/disc performance for example, incorporates basic entity state animation and also provides the user with an HTML forms based input mechanism. This allows the user to define a variety of options, for the model to simulate, and assists the user's ability to interpret the models behaviour [DFAB 93]. However, this is subject to movement between different web pages and necessary network latencies.

SIMJAVA [M^{c}N&H 97] and SIMKIT [B&S 96] are two recent non-commercial simulation tools developed in the JAVA programming language. They offer the user the ability to define and build simulations by extending the core classes of the respective packages. Both provide the potential for simulations to be included in a web page and run as embedded *applet*s. Further details regarding the implementation of these packages is outwith the scope of this project, though can be found on-line [M^{c}N&H 97] [B&S 96].

As SIMKIT was developed primarily for military purposes, it takes a more analytical approach to simulation, with emphasis given to specific military problems. SIMJAVA provides the ability to animate simulations diagrammatically with its accompanying SIMANIM package. With this extension, SIMJAVA has been used to develop working simulations of a computer cache memory and distributed computing protocol agreements. These benefit greatly over their simulation counterpart tools from their inherent ability to be executed at remote sites, without the need for storage and compilation.

3.2 Existing Architecture Simulations:A Critique

In order that these existing tools be critically assessed it was necessary to provide a framework for comparison. The framework chosen is presented by Dix, Finlay, Abowd & Beale [DFAB 93] for the evaluation of interactive computer systems. A number of paradigms for purposeful and principled design are highlighted. In brief the areas of primary concern are:

Learn-ability This reflects the ease with which new users can begin effective interaction and achieve maximal performance.

Flexibility The flexibility of a system is based upon the multiplicity of ways that the user and system exchange information.

Robustness Robustness is concerned with the support given to the user so that they determine the successful achievement and assessment of goals.

This framework is then extended to consider the central issues associated with providing an on line teaching aid for computer architecture. Factors such as on line accessibility and whether or not the system requires local storage facilities are implicated.

3.2.1 Applications Requiring Download

Prior to the introduction of JAVA, a criticism of existing tools centres around the need to download the application to a local site before the tool can be used. Dlxsim and HASE are in this category. It is necessary for the user to become involved with downloading, installation and other administrative tasks before any benefit from the package is realised. However, both these packages do benefit considerably from their extensive ongoing development, providing users with access to detailed implementation resources and backup support.

3.2.1.1 Textual Based Simulations

Dlxsim, once installed on a local site, provides an extensive environment for the simulation of the DLX instruction set architecture, though it is noted that certain architectural features are misrepresentative of the intended DLX architecture implementation as defined by [H&P 96]. For example, load stall cycles in DLXSim are not actually simulated due to the implementation of the behavioural descriptions of the entities [S&K 96].

Dlxsim provides the user with textual, step by step simulation. This representation requires that the user follow through each step of the model by reading the relevant text. Given the complexity of the model, and the number of concurrent activities within the system, this requires the absorption of a large volume of descriptive data to represent a specific point in time. The need for the user to learn additional commands to drive the package also detracts from the overall *learn-ability* of the model.

3.2.1.2 Graphical Simulations

The HASE system recognises the benefit of incorporating graphically animated simulations. In addition HASE also provides for the hierarchical viewing of certain simulation entities. This is a powerful feature and gives users the ability to control how they view simulations. Within the analytical framework, these features give HASE greater potential for usability over that of a purely textual simulator. The state of the system is displayed pictorially and animated event interactions can portray the system-wide changes, at any point in time. The additional feature of user defined menu selection and textual simulation scripts further add to the understanding of complex models.

3.2.2 Remotely Available Simulations

The recent extension to the Casle Project of DLXview [PURDUE 97], incorporates a more visual depiction of the behaviour of the DLX architecture. This is achieved through an HTML forms based interface permitting the user to cycle through steps of the execution of an assembly program, by viewing the updates to the system. These updates are presented by highlighting the active buses and displaying the textual updates to the tables involved. However, there is a loss of user control in that the simulation is dependent upon the transfer of data across the web for each successive update. Movement through the simulation can be slow and somewhat disjointed, though the flexibility of the system over the existing DLXsim is increased.

Of the JAVA simulation models currently in existence it is observed that no tool specifically provides for online architecture simulation modelling. A package which simulates the logic of architectural descriptions is available [IWAN 97], but as a teaching aid for computer architecture may be considered as of limited value. Indeed it is necessary to understand the behavioural model of the system in question before a simulation can be performed and the results verified. This model has associated problems for the teaching of computer architecture similar to those highlighted in reference to the VHDL model described in section 3.1.1.

3.2.2.1 Animation

The model presented by [CB 97] simulating the behaviour of a virtual memory system further addresses the aim of providing an animated *applet* with definite state changes. The model relies on colouring of data channels as they are used and values are updated in the various tables presented. However, this model does not possess the capability for user interaction other than starting, stopping and restarting the *applet*. Control of the system is primarily with the system itself and not the user. The flexibility of the system to provide multiple information exchange is therefore reduced. There is no provision of reasoning for many of the state changes and the user must ultimately rely upon inference and deduction from their existing knowledge base. This prevents a novice to the field of computer architecture from drawing sound conclusions about the behaviour of the system.

A number of remotely available models have been built using the SIMJAVA package. These are recognised as powerful in their communication of behaviour to the user. However, of the Simjava simulations available up to the present time, these models incorporate limited complexity. It can be inferred from the trends observed that such simulations provide the direction of online simulation and modelling.

3.2.2.2 Levels of Abstraction

A further criticism of the available teaching tools focuses on their ability to present abstract representations of the systems in question. Tools modelling architectural features often focus on the binary or hexadecimal level [THER 97][IWAN 97]. This representation does not provide for easily deduced behaviour, though if incorporated with higher level coding presents the user with meaningful interpretation of machine instructions.

Those tools incorporating animation merely highlight data buses as active or idle, and the user is not shown what it was that passed down the bus to change the system state. For example, [CB 97] and [PURDUE 97] use this colouring of buses to represent the passing of data along a bus. The SIMJAVA package tackles this issue and has the ability to pass information objects along ports. This provides a powerful mechanism for showing not only the state of the system and the fact that the bus is busy, but also provides for visual representation of 'cause and effect.'

It is seen that existing tools are beginning to move towards more flexible systems, providing the user with a number of ways to interact. It is also the case that simulation models are now better able to provide increased *learn-ability* and *robustness* through JAVA and its capabilities.

Chapter 4

DLX Simulation Design

This Chapter describes the design of the system and is split into four key sections. The first highlights the problem areas that must be addressed in order to provide an online teaching aid for computer architecture. An outline of the properties of the DLX architecture to be simulated and how an abstract representation of the architectural features was formed is then presented. Following on from this, a critique of the desirable features and functionality of the system is highlighted, before summarising the design decisions that were made.

4.1 Conceptual Problems

It is worthwhile summarising some of the key elements that an architectural model must show to ultimately be of value for the purposes of teaching. In brief these relate to dependencies in the assembly code and how the code can be processed by the system. A computer architecture is effectively a collection of resources available to process a given set of instructions. Consequently it is bound by factors pertaining to resource utilisation, instruction dependencies and control hazards.

These issues become crucial to the activity and behaviour of a high performance architecture. This is especially true for architectures implementing pipelining. It is therefore necessary for a simulation model of an architecture to show these problems and how they are handled by the system. The three main hazards associated with pipelining are highlighted in Table 4.1.

n.b. The table refers to instructions i and j, where instruction i occurs before instruction j.

Type of Hazard	**Description**
Structural	Functional unit required by j is busy with i
Write After Write	j tries to write destination before i writes it
Read After Write	j tries to read source before i write the value
Write After Read	j tries to write destination before i has read value
Control	caused by pipelining instructions which change the value of the PC

Table 4.1: Pipeline Hazards

In addition, the implementation of multicycle instruction execution and its control, data forwarding and the maintenance of precise interrupts are central issues associated with improving the effectiveness of instruction pipelining. These areas will be considered in the following sections outlining the design of the model.

4.2 Desired Model Features

In light of the weaknesses of existing computer architecture model representations, highlighted in the previous chapter, the project was guided by a number of objectives. A summary of these guiding factors is provided below.

- A **high level abstraction** is needed to bridge the gap between the gate level logic and that of the assembly code to be run.
- **User defined interaction** with existing online tools is limited and, where present, suffers from the need to transmit the changes through separate interface web pages.

The project needed to provide a stable user defined environment without the need to 'jump' between pages.

- Attempts to provide online representations of high performance architectural features are beginning to develop towards more visual and informative applications. However, it is clear that many of these fall short of the desired user defined animated simulation, now being seen in smaller examples. The goal was to present a **visually animated model** rather than textually descriptive model.

- **Pipelining and pipeline control**, for example, play key parts in textbook examples, yet are not available as online simulation models. Similarly, the memory hierarchy and its effect upon performance are well documented but specific examples of how a memory system is synchronised within an architecture are left to textual descriptions and thus prone to over simplification. Indeed, as the memory-processor disparity continues to affect performance, students are left to integrate different assumptions from separate models and group them together based solely upon presumption. This dictates that the project should provide a **comprehensive simulation tool** able to show the interactions of various components such as memory, the cpu and how they ultimately work together.

- The difficulties associated with comprehension of such a complex piece of machinery are heightened by the physical environment of the medium upon which the material is to be taught. The simulation demands that the user be presented with a model that works on a relatively small and compact two dimensional computer screen. This dictates that the model must have a **concise visual representation**.

4.3 DLX Model Features to be Simulated

4.3.1 Hardware Representation

At the logic level the design of the DLX architecture is a simplified pipeline representation with five key stages of operation. This is based on a model presented by [H&P 96]. (See Figure 4.1)

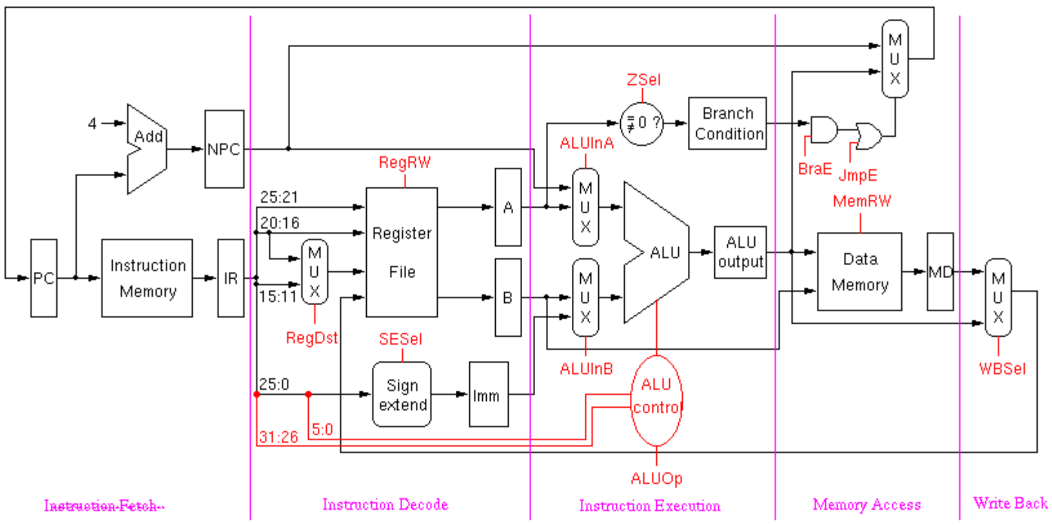

Figure 4.1: Fundamental DLX Logic Level

In brief the stages associated with it are:

Fetch The value of the program counter (PC) is sent to memory where the relevent instruction is retrieved. Assuming no changes in the flow of control, i.e. branch or other instructions affecting the PC, this is then forwarded to the decode stage.

Decode At this point the instruction retrieved from memory is decoded and the required operand source register(s) read.

Execution With the values of the necessary operands the instruction can be arithmetically computed.

Memory Access Load/store instructions are effectively 'executed' at this point and other instructions are passed onto the write back stage, where necessary.

Write Back Arithmetic results from execution are written back to the register file where they then become available for later dependent instructions.

This logic model provides a simplified representation of the architecture though it is recognised as a valuable starting point for the teaching of computer architecture. A number of other key issues should also be considered, which would further increase the necessary logic of the implementation, but also provide a more comprehensive and realistic model. Whilst pipelining itself is a powerful mechanism for increasing an architecture's performance, it is rarely a lone feature of an actual architecture. In this respect an isolated pipeline simulation was felt to be of little use other than as a means for teaching the rudimentary concepts associated with it.

To provide a more realistic model for simulation, it was also decided to incorporate multiple parallel functional units for the execution of instructions under the control of a scoreboarding mechanism. A key concept that must be understood by students is that of the variance in instruction execution depending on the nature of the instruction being executed. For all but the most basic introductory examples it is insufficient to assume that a floating point divide operation, for example, will be executed in the same time frame as a basic integer addition.

In addition, the DLX representation is governed by the assumption that all memory requests can be dealt with in a single clock cycle. This was also felt to be misleading and that with the disparity in timing of memory access and instruction execution, the provision for memory latency, and its effect upon the system should be shown. Furthermore, this would further permit the extension of the model to incorporate a memory cache hierarchy at a later date.

4.3.2 Instruction Set Representation

The model was designed to incorporate the three instruction class formats of the DLX described in section 2.3.4, namely immediate, register and jump. To provide a representative model, capable of further expansion, each of these classes was to be incorporated into the simulation.

It was also necessary to highlight the behaviour of the system on both integer and floating point operations. These instructions could then demonstrate the parallel operation of the four functional units requiring addition, multiplication and division.

The instruction subset also needed to highlight operations involving load and store memory interactions and also control flow hazards. It was decided to introduce the branch on integer not equal to zero (BNEZ) to the implemented sub-set of instructions. This would also provide for meaningful looping constructs to be incorporated in the code under simulation.

The initial implemented subset of instructions is shown in Table 4.3.2 below:

4.4 System Design Rationale

From the given description in [H&P 96], a clear abstract representation of the model is not one which can simply be drawn up. To provide a tool which is at a higher level than the underlying logic a number of issues had to be addressed. To show the pipeline execution of instructions there is the necessary abstraction and separation of the memory access and write back stages of the pipeline. In reality these operate concurrently, being fed from the output of the alu, and the purpose of the memory access is purely to deal with the loading and storing of data from memory. The abstraction does in effect draw a distinct boundary between these stages, but it is felt that this gives a more understandable picture of the sequence of events.

Implemented Instruction Subset

Instruction Format	Operation	Type
ADDF *rd, rs*$_1$*,rs*$_2$	$rd \longleftarrow_{32} (rs_1) + (rs_2)$	R
ADDI *rd, rs*$_1$*,immediate*	$rd \leftarrow_{32} (rs_1) + [(immediate_0)^{16} \| immediate]$	I
BNEZ *rs*$_1$*,name*	if$(rs_1) \neq 0 then\ PC \leftarrow_{32} \{[(PC)+4] + [(name_0)^{16} \| name]\}$	I
DIVF *rd, rs*$_1$*,rs*$_2$	$rd \longleftarrow_{32} (rs_1) \div (rs_2)$	R
LF *rd, offset(rs*$_1$*)*	$rd \leftarrow_{32} M\{[(offset_0)^{16} \| offset] + (rs_1)\}$	I
LW *rd, offset(rs*$_1$*)*	$rd \leftarrow_{32} M\{[(offset_0)^{16} \| offset] + (rs_1)\}$	I
MULTF *rd, rs*$_1$*,rs*$_2$	$rd \longleftarrow_{32} (rs_1) \times (rs_2)$	R
NOP	(none)	R
SF *offset(rs*$_1$*),rd*	$M\{[(offset_0)^{16} \| offset] + (rs_1)\} \leftarrow_{32} (rd)$	I
SUBF *rd, rs*$_1$*,rs*$_2$	$rd \longleftarrow_{32} (rs_1) - (rs_2)$	R
SUBI *rd, rs*$_1$*,immediate*	$rd \leftarrow_{32} (rs_1) + [(immediate_0)^{16} \| immediate]$	I
TRAP *name*	$IAR \leftarrow_{32} [(PC)+8]$ $PC \leftarrow_{32}$ ‘$0'^{6} \| name$	J J

Another key issue is that of how the architecture deals with load delays. In circumstances where the pipeline cannot cope with the instruction execution rate resulting from a load instruction, it may be necessary to introduce a pipeline interlock, which is effectively a stall or bubble in the system, affecting performance. In reality compilers have the ability to attempt to schedule useful instructions into such delay slots. However, it is still a necessary requirement that the architecture be able to handle such situations. The ability to demonstrate these compiler effects would be a useful application area for the model.

4.4.1 Model Representation

The design of the simulation model was governed by a number of important issues brought to light through the research undertaken pertaining to current web simulation tools for computer architecture. From an early stage it was decided that the model should be abstracted away from the detailed bit implementation and that the user should not have to learn system commands for the package itself. The user should ultimately be tasked with following and understanding each of the state changes of the simulation as it progresses. Thus it was felt that use of binary or hexadecimal values would be of little significance to the user. Instead it would be beneficial to the user if the actual instruction being manipulated could be represented throughout its lifetime in the system.

Without the ability for the system to define components hierarchically, a major concern related to providing the user with an acceptable level of detail without going into too much complexity. Conversely, it was clear that the simulation must also provide suitable abstraction to group together similar activities related to any given task.

Initially it was thought that the number of links to the memory component should show the different accesses from both the memory access stage and the instruction fetch unit. However this is inconsistent with reality and so was modified by adding a memory access controller to provide synchronisation. This permits the latency of memory to be shown more accurately and provides the ability to alter the access time for the memory component and witness the effect on the overall system.

4.4.2 Synchronisation

Having demonstrated the weakness of existing tools through their lack of real time visual information, a major emphasis of the project centred on the synchronisation of events within the system. The existing teaching aids provide largely textual descriptive representations of the states of the system at any given time. It became a necessary area to permit the clear visual synchronisation of events as would be the case in a 'clocked' architecture. Existing simulation practice relies heavily upon behavioural descriptions of entities and their unique responses to events scheduled to them. They largely rely upon the inherent timing of a given entity and provide few mechanisms for coordinated and well synchronised activity as would be typical of a synchronous high performance architecture. Again this is a key concept that users should understand. It is unrealistic to assume all activities are handshaked around the system as though it were asynchronous. As a result the decision to add a further, clock control entity was made.

Some ambiguity exists in present model implementations of certain architectural features. The reliance upon textual descriptions and static pictorial representations make the comprehension of architectural specific features more difficult to understand as part of the wider picture. Similarly, the comprehension of events as they occur must be portrayed to the user of the system. A representation of the three key components of the system was chosen, namely the memory, the registers and the pipeline of the processor itself. The introduction of the clocking synchronisation did however, add a fourth element to the system and its overall design. This is less significant for complex understanding and acts to reinforce the synchronous nature of the architecture.

4.4.3 Pipeline Design

Another key difference between this implementation and that of the logic level relates to the introduction of a sixth stage in the pipeline. This is in line with the inclusion of the scoreboarding mechanism of control. It was also designed into the system to provide a clearer representation of the different structural, WAW, RAW and WAR hazards. Furthermore this provides for the future inclusion of pipelined arithmetic functional units.

4.4.3.1 Multicycle Operations

As is representative of realistic architectures, the inclusion of arithmetic operations on floating point data is necessary in the DLX. [H&P 96] recognised that it is impractical merely to assume that the architecture can cope with floating point operations in the same one period clock cycle as a similar integer arithmetic operation. This would force the acceptance of either a slower clock speed, a vastly greater logic to cope with the necessary performance of floating point operations in the same time frame, or indeed both. More realistically it is necessary to include some form of parallel functional units with different latencies depending upon the nature of their operations. These then must be controlled as the individual integer unit would have been. Different instructions must then be scheduled to the appropriate functional units and both structural and data hazards detected in the light of this. This addition of multicycle execution to the model forms a necessary and difficult area for representation of the internal workings of an architecture.

Indeed it is universally accepted that as the complexity of the architecture increases, so too do the incurred hazards, their frequency and, in certain implementations, whether they are in fact present or not. Without multicycle operations instructions are forced to execute in order of issue. Introducing multicycle operations can permit certain instructions to complete out of order, depending on the nature of the instruction. An instruction passing through a low latency unit can be issued after one requiring a longer latency unit and effectively overtake the earlier instruction. This fact should be designed into the system and is an important area that should be highlighted to the user.

4.4.3.2 Interrupts

In turn, the addition of multicycle operations introduces the potential for the introduction of imprecise exceptions. Interrupt instructions can then be confronted by an unrestorable state if a source operand has subsequently been destroyed prior to the occurrence of the exception. A number of methods exist for handling interrupts. These rely on either accepting the inconsistent state, preventing an inconsistent state from arising or providing buffers for rolling back the state to rectify the problem. A desirable feature would be to present the user with the ability to roll back on encountering an exception. However, this is not a primary aim of the project and has not been included, though would be a useful extension to the design model. Provision is made to accept the TRAP instruction as a terminating instruction for the simulator.

4.5 Design Criteria Selected

This section sets out to highlight the design decisions made and is broken down into three key areas. However, these are not mutually exclusive and there were necessary overlaps between them in making specific decisions.

4.5.1 Component Selection & Specification

The component representation chosen to be shown on screen is summarised in Figure 4.2

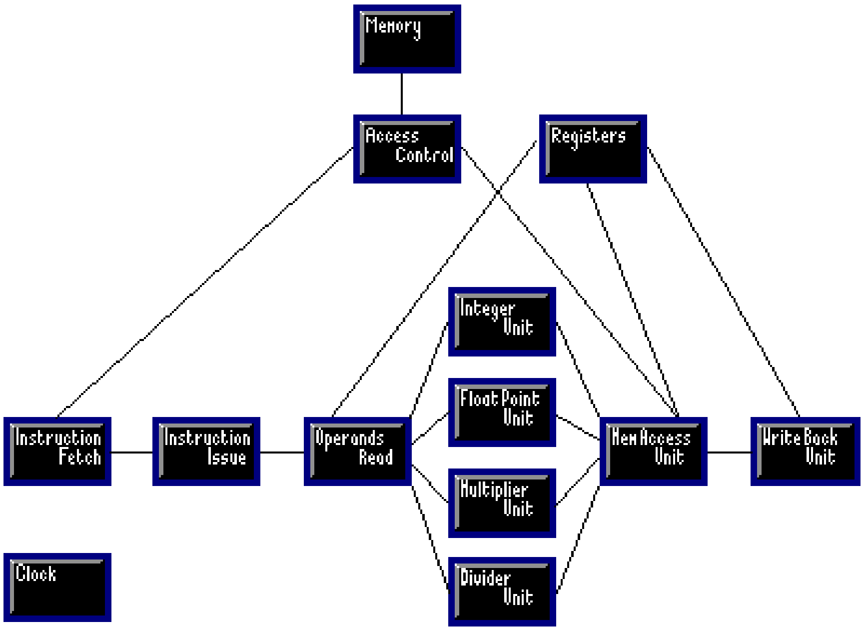

Figure 4.2: Entity Representation

In addition to the above representation, the design required that certain units incorporated queues to buffer instructions and also parameters to indicate the state of the entity throughout the lifetime of the simulation. It was also necessary to hide from the animation those events which might detract from the coherence of the model. All clock signals fell into this category and while the representation gives the outward appearance of only a limited number of entity interactions, there are in fact links between the clock control unit and each of the entities, with the exceptions of the memory and the register block.

Memory The memory entity is tasked with storage and retrieval of data. Distinction is made between the instruction memory and data memory. Accessing memory is bound by the single data bus, permitting either a read or a write operation at any one time, but not both.

Memory Access Control The memory access control is charged with the task of controlling the memory access and prioritising requests for memory access from both the memory access and instruction fetch pipeline stages. Memory access is designed to have priority for load/store operations over instruction fetch requests.

Instruction Fetch The role of the instruction fetch unit is to control the issuing of instruction memory requests and the detection of control flow hazards when memory requests are retrieved from memory.

Instruction Issue At this stage the instruction is decoded. The instruction is checked to see if the destination arithmetic unit is free and also whether or not there exists a WAW hazard.

Read Operands Operands are checked for availability and the existence of RAW hazards. Instructions are then forwarded to the relevant functional unit. Priority is given to the longer latency instructions.

Integer Execution Integer addition and subtraction is performed at this stage. In addition the address calculations for load and store memory transfer instructions, integer control flow conditional expression evaluation and logical operators are executed.

Floating Point This unit is responsible for all floating point addition and subtraction.

Multiplier Both floating point and integer multiplication are performed in this unit.

Divider Both floating point and integer division are performed in this unit.

Memory Access This unit is responsible for issuing the read and write instructions to memory for the purpose of loads and stores. In order that the model behave as closely to the actual logic as possible, this unit acts as the point where branch conditions are resolved.

Write Back The write back unit operates on all other arithmetic results and also the retrieved memory data from load instructions. It is the only write bus linked to the register file.

Register Block The register block is a simply a collection of fast access data registers organised into two blocks of 32 general purpose and 32 floating point registers. As is typical of a high performance system, there are two read buses and one to write back to the block. It is noted that the representation shows an additional read bus to clearly represent the necessary register read for a store operation.

Clock Controller In addition to the main functional units of the system the model aimed to incorporate a clocking mechanism. This controls the execution timing of the other entities and also serves as a reference for the clock cycle counter, thus removing the dependency upon interpretation of simulation time.

4.5.2 User Interface Design

In order that the simulation provide the user with an involved environment for learning, the addition of customisable features was a key aim. From the derived abstract model these options were to provide the user with a better understanding of factors affecting pipeline efficiency. As a result the objective was to provide for user defined code simulation and also the ability for the user to set different latencies in various components.

4.5.3 Instruction Code

The provision of **user defined code** to be simulated as part of the applet was a central aim. Indeed this could have been implemented in a number of ways from file access to HTML entry, similar to existing tools. However, these methods were felt to lack visual consistency and there are security implications involved with file access. It was felt that the provision of a small **text area** on the model itself could represent sufficient space for the user to input code and also this would serve as a point of reference when the model simulated the code. Additionally this would prevent the need to move between different web pages and could potentially be updated during run time.

4.5.4 Functional Unit Latencies

The provision of user defined latencies for certain entities was an aim for several reasons.

1. Increased user involvement
2. System-wide impact of individual unit latencies
3. More powerful model representation
4. Confronts new area in online simulation for architecture

The entities selected for customisation were the four arithmetic functional units, the memory and also the clock. These would then impact upon the efficiency of the pipeline and emphasise the fact that all components can affect the performance of the whole system.

4.5.5 Interaction Provided

Incorporation of these elements dictated that the ability to customise a simulation must enhance the basic model. It was not a desired option to have to move between different pages to instigate changes, nor was it desirable to change the representation of the model for each new option entered. It was an aim to provide a number of options, without detracting from the visual depiction of the model. As a result the user interface was designed to be incorporated by selecting the appropriate button and then being presented with the ability to change options in a supplementary window. This would leave the main model visible at all times and if the user chose not to implement customisation they would not have to 'bother' unnecessarily with options.

Again this overcomes the need to use file access or the need to reload or modify the present setup. Security is maintained and the user is given control over the timed behaviour of the system settings.

4.5.6 The Simjava Interface

Further to these design options, the model also demanded that the design be compatible with both JAVA and SIMJAVA. Some of the implications of the design are now presented in light of the necessary interactions.

The core behaviour of the simulation is designed upon the Simjava package functionality. Indeed the basic model could be implemented first as a stand alone simulation before adding any animated *applet* features. This direction was not taken. Rather it was decided that the development of the model should progress with the inclusion of observable animated behaviour.

In order that the model be animated, it is necessary to interface with the actions of the simulation and, using the trace-file output, present these as part of an applet. This requires the coordination of the core JAVA API, the SIMJAVA and SIMANIM packages, and also the behaviour of the designed model. These interactions are summarised in Figure 4.3.

The SIMANIM package extends the capabilities of the SIMJAVA simulation providing *applet* interaction. In essence the SIMANIM package extends the *applet* and provides new implementations for describing the life-cycle of an *applet*. (Ref: Chapter 2, section 2.1.2) The DLX simulation model extends the template provided by SIMANIM to build the simulation.

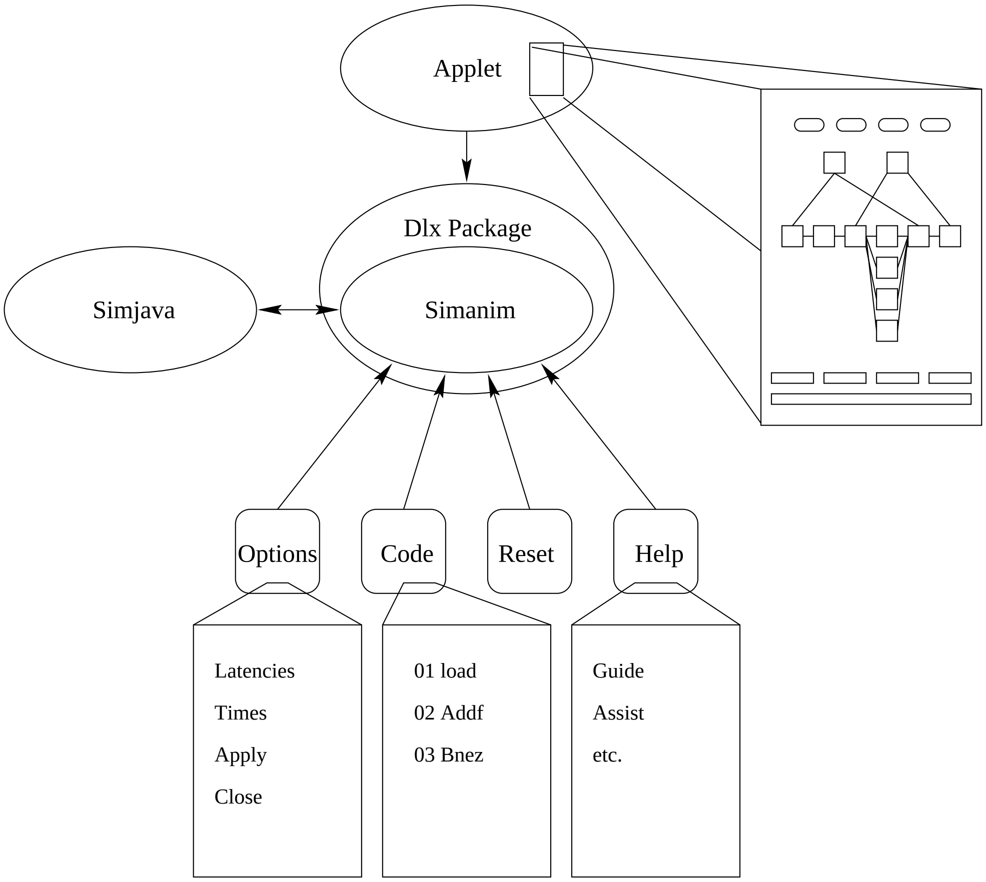

Figure 4.3: Simjava Class Interactions

These design decisions made then guided the implementation phase of the project. This implementation phase forms the focus of the following chapter.

Chapter 5

DLX Simulation Implementation

The focus of this chapter is upon essential implementation issues that arose throughout the development of the project. It concentrates on the implemented features and the functionality of the system. However, a brief overview of the coded behaviour of the system and further issues which impacted on the system development are discussed.

5.1 Entity Implementation

5.1.1 Data Storage

5.1.1.1 Memory

The memory was designed so as to be a separate entity outwith the control of the clock and with a user defined latency. The key issue was the representation of the instruction and data memory. The desire to provide the user with an easily understood mechanism for modelling of user code led to the implementation of a text area which the user has the ability to alter. This acted as instruction memory and a separate array representation was implemented to store data memory. This instruction memory would then remain visible to the user throughout the simulation and provide a central point of focus to assist the following of instructions as they progress through the system.

5.1.1.2 Registers

The register block is implemented outwith the control of the clocking mechanism. It performs two central functions, namely reading existing values and writing new values. Both the general purpose and floating point registers are represented by arrays, given that their size is known and remains consistent throughout. This is important in light of the fact that JAVA does not permit arrays to be allocated on the heap. They must be created with the *new* operator with an allocated size.

5.1.2 Pipeline

5.1.2.1 Instruction Fetch

The instruction fetch unit is responsible for the fetching and receiving of instructions from memory. This is achieved by sending the value of the program counter to memory. In addition it must detect control flow instructions and interrupts. Should a branch instruction be encountered, the instruction fetch unit must stall the issuing of new requests until the outcome of the branch is known. Detection of a TRAP instruction informs the instruction fetch to halt further requests.

5.1.2.2 Instruction Issue

The instruction issue stage of the pipeline is responsible for the detection of both structural and WAW hazards. It can then either issue or stall the appropriate instruction. If there is either a new instruction or a queued instruction awaiting issue, then a number of checks must be performed. The queue is checked in a first-in first-out fashion and if an instruction can issue it is sent to the **read operands** unit. To be eligible for issuing, an instruction must be able to access the necessary read operand buffer, no instruction before it may be a load or store operation and there must not be a WAW hazard present. Detection of a WAW is carried out by examining the the register status of active instructions and also those instructions before it in the issue queue. If it can be issued the instruction is removed from the issue queue and entered into the register status table and registered as issued. Upon detection of a WAW hazard, a warning message is displayed making the user immediately aware of the hazard.

5.1.2.3 Read Operands

The **read operands** stage is primarily responsible for the detection of RAW hazards and the accessing of available operands. When the operands are not available, the unit implementation provides textual output indicating that a hazard was encountered. This is clearly displayed at run time as the hazard is encountered. To incorporate a data forwarding mechanism is a potential extension of the model, but this is not yet fully supported. This could be implemented by adding an additional method to check the availability of arithmetic outputs for the destination register, before it is written back to the destination register. When available this can supply the necessary operand for the appropriate functional unit. This technique is common in pipelined architectures, though clearly representing this within the *applet* was felt to pose some problems which could detract from the understanding of the core model. One alternative would be to have an additional user option to select whether data forwarding was used or not. This would provide the user with a comparative model to investigate the benefits of data forwarding.

5.1.2.4 Execution

The execution stage is split into multiple parallel functional units for integer, floating point, multiplication and division arithmetic. These were implemented by inheriting a base execution class, and for each of the four units defining unique graphical entities to represent them. In addition each entity has its own user definable latency that can be altered, through the user options menu, both before and during execution. Once an arithmetic unit has completed its operation it must then check for structural hazards arising when two or more units wish to leave and enter the memory access stage of the pipeline. This is handled by prioritising the longer latency units to have precedence over the other units. Hence longer latency units are 'freed' as early as possible

5.1.2.5 Memory Access

When an instruction has been executed it is passed through to the memory access stage of the pipeline. Here a check is performed to detect load and store operations requiring memory access. If a load or store instruction is detected the relevant instruction is either sent directly to memory or the necessary registers are read and sent to memory. Other instructions simply pass on to the write back stage, with the exception of branch and jump statements. These are evaluated at this point to simulate the logic level implementation which is performed in parallel with the memory access. When evaluated, the branch operation is performed and the stall introduced by the hazard is removed.

5.1.2.6 Write Back

The write back unit is responsible for writing arithmetic results back to the register file for future instruction dependency. This is a straightforward operation and in this model, as with the logic level implementation, is carried out within one instruction cycle. However, as the pipeline introduces out of order execution with its multicycle operations the write back unit must ensure that no WAR hazards are introduced into the system. This is achieved by monitoring the active instructions and any dependencies shared with the instruction wishing to be written to the register block. Should a hazard be detected the user is alerted of the fact.

5.1.3 Control Entities

5.1.3.1 Synchronisation

The need to provide synchronisation for the architecture model was recognised at an early stage of the development. Its implementation however, raised a number of issues. The JAVA run time system is multi-threaded and the implementation of the SIMJAVA package means that each entity runs in its own thread. Rather than implement a handshake protocol a more visible synchronous control mechanism was required. This would assist in providing the system with predictable and observable data transfers.

The order in which events must be passed between entities is deterministic though the Java thread implementation is non-deterministic. The option of providing each entity with a very small inbuilt latency to account for the ordering of events was a potential implementation. However, with the complexity of the system being modelled, and the behavioural differences of each of the components represented by the system, this method was rejected as an inelegant solution. So too was the potential use of other techniques such as semaphores.

It was decided that a barrier synchronisation policy would be used. This would provide a single unit with the sole responsibility for the timing of the passing of data. Thus the synchronous architecture representation would be clearly represented, the underlying principle being that a single barrier is presented to all active processes. Processes must wait when this barrier is reached and only when all processes have reached the barrier can they progress. In the case of the clock controller, this barrier was devised by making use of a spin lock. This was responsible for tallying the entity responses until all had reached a fixed point. The entities themselves controlled the holding of data that would be processed or forwarded in the following period.

The clock controller is responsible for providing the units with a two phase barrier synchronisation mechanism. This simulates a rising edge 'go-do-it' phase (1), followed half a clock cycle later by a 'data-forward' signal (2). (See Figure 5.1) It was felt that this would add considerably to the comprehension of the model by providing clearly defined movements of data through the system by maintaining an array of entity identification numbers, it was then possible to schedule a start signal to and receive a done signal from each of the units, whilst keeping these communications hidden. The user then has the ability to adjust the timing of the clock period. A visual representation of the rising and falling clock edges is portrayed at run time, along with a clock cycle counter for the user to follow.

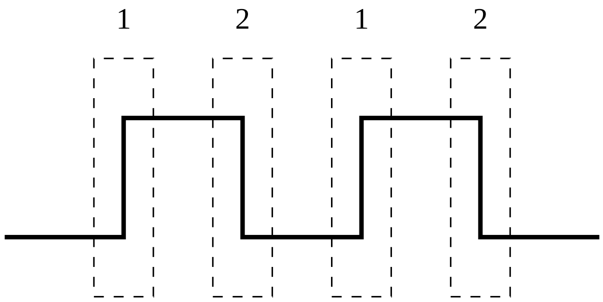

Figure 5.1: Clocking Mechanism

5.1.3.2 Scoreboard

The scoreboard mechanism for controlling the simulation relied on the maintenance and checking of a selection of tables. (These tables are shown in Appendix B). The implementation permits the system to maintain control over instruction execution by recognising when an instruction is active, and by tracking source and destination operands required for a given instruction. This enables out of order execution to be dealt with and in addition provides for destructive register use, i.e. where an instruction has the ability to read one register and overwrite this value with the result of the arithmetic computation. However, as the scoreboard must maintain complete control within a concurrent threaded environment, it was necessary to add a number of test-and-set synchronisation primitives. These ensure the ordering of certain events to prevent contention issues being introduced.

5.1.3.3 Memory Access Controller

The memory access controller provides the control mechanism for access to the memory by either the instruction fetch unit or the memory access unit. This is so as to provide a single bus to memory for either a read or a write operation, but not both at once. It was necessary to implement a queue of waiting requests that were scheduled in a first-in first-out fashion, but giving load/store requests priority over instruction fetch requests.

5.2 Applet Implementation

The aim of producing a real time simulation capable of clearly showing the states and behaviour of the DLX architecture was further bound by the constraints of the available screen size. This fact is of further importance when it is considered that the simulation is designed for inclusion within a web page.

5.2.1 Core Model Layout

The fundamental layout was described and presented in Chapter 4, Figure 4.2. However, it was also necessary to contain the size of the applet so that it presents to the user a clear model for understanding all the necessary information. For example, when a hazard is detected by the system this should be brought to the attention of the student in real time as it is dealt with. This enables the user to directly associate the problem with the given subset of active instructions as they are following the simulation through. The issue of structural hazards was not explicitly shown with text, but instead the graphical state information of each entity provides indication of why an instruction could not proceed.

5.2.1.1 Textual Representations

The implementation of the memory was based around the inclusion of instruction memory as visual text. This was implemented by the inclusion of a text area over the body icon of the memory. In order to accomplish this, a new layout manager was written for the applet initialisation and also the *Simanim* class was modified accordingly. The user has 'edit permission' to access this text area and can use both the mouse and general keyboard to access it. Once edited, the new text is only activated when the user presses the read new text button with the mouse pointer.

In addition it was also felt that the model would benefit greatly from the inclusion of a visual representation of the state of the registers. This was implemented to show the user not only whether the register block was being written to or read from, but also the register data contents. This provides the user with the ability to verify the code being simulated, checking that it performs the correct calculations on the data. This leads to a better understanding of instruction manipulation in a wider context, rather than a in a simple isolated example concerned with purely the movement of instructions. It is possible to perform simple high level mathematics to verify the computation of the machine assembly code. For example, the default program implemented contains a series of statements which act as a for loop, branching and repeating the loop five times.

5.2.1.2 Animated Representations

When the model encounters a conditional control flow instruction the pipeline must stall instruction fetching from memory until the outcome of the branch condition is known. This too is an important state that should be clearly portrayed to the user. This is achieved by the addition of a third entity state representation for the instruction fetch unit. When the pipeline detects a branch instruction the entity is updated to show the new stall state. As such the *applet* does not become any more cluttered.

In a similar fashion to the implementation of the instruction stall display mechanism, the other functional units all have state icons which are either idle or busy. This clearly indicates that data is being operated on, is stored in a buffer pending operation, or is idle. The exceptions are the memory and the register block. With the modified implementation to represent textual images for these two entities it became necessary to add additional name and value parameters, supported by SIMJAVA, which have the ability to display a parameter name and also its value. These values were updated depending upon the nature of the operation and it was then possible to display text information depicting either idle, reading or writing states. Again this implementation provides the user with extra information that is easily interpreted but which does not impinge on the overall load of information to be managed.

The culmination of these chosen implementations led to the main *applet* implementation as shown in Figure 5.2.

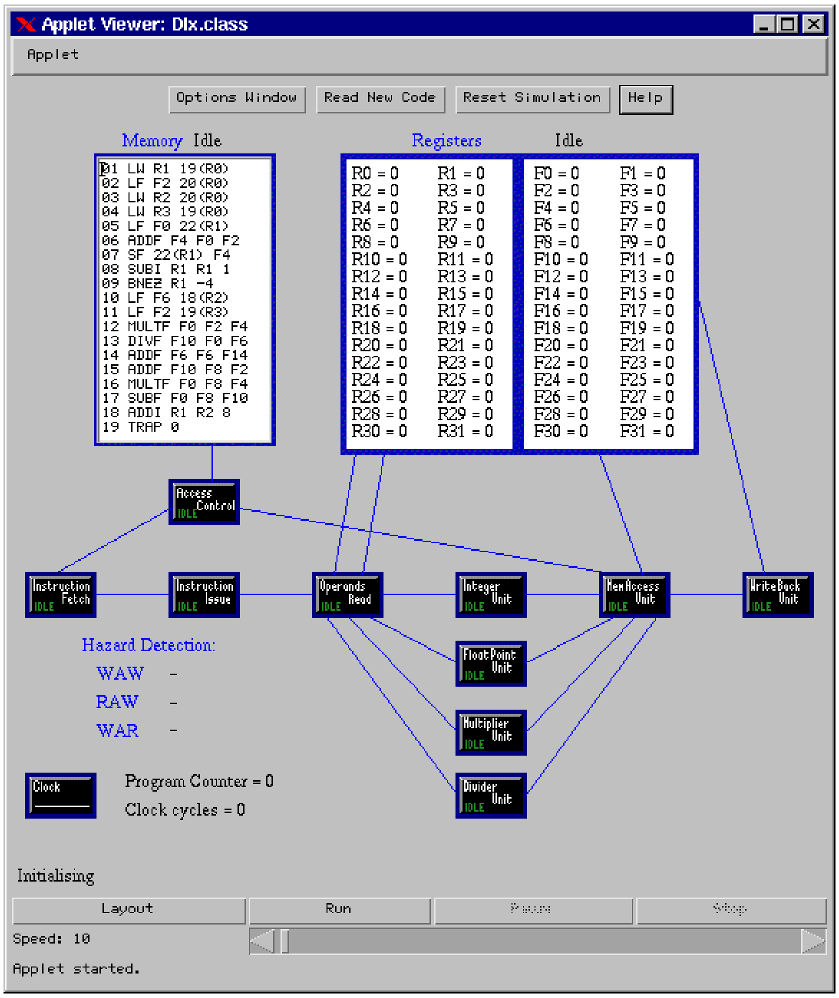

Figure 5.2: The Core Applet Design

5.2.2 User Interface

The implementation of the additional user defined options was bound by the goal of providing no interference with the existing display. As such the implementation of the options and a help display were based on user selectable frames. These frames, when selected, 'pop up' as additional windows that have the potential to remain active throughout the simulation lifetime without affecting it. A help window is provided which displays basic instructions on how to use the system and although not necessary, does provides the user with additional information should it be sought. This frame can be seen in Figure 5.3.

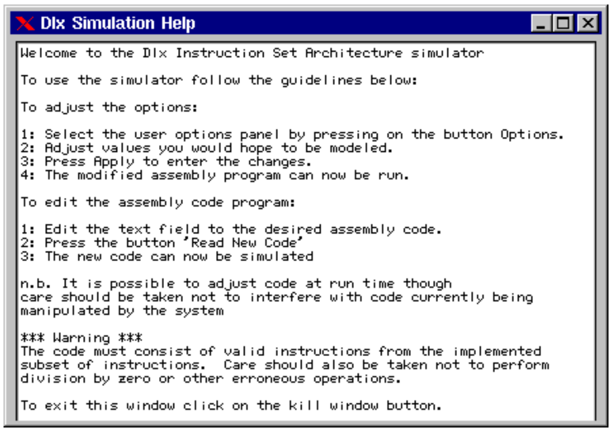

Figure 5.3: User Help Frame

The user option window was also implemented as an additional frame. This frame can be actively displayed at the same time as the main *applet* and its implementation further enables the user to perform run time modifications to the system. The options frame can be seen in Figure 5.4.

5.2.3 Code Implementation Issues

The aim of this section is to provide an indication of some of the JAVA coding issues relevant to the implementation. However, as there were in excess of 4000 lines of code it is recognised that only a brief overview can be presented.

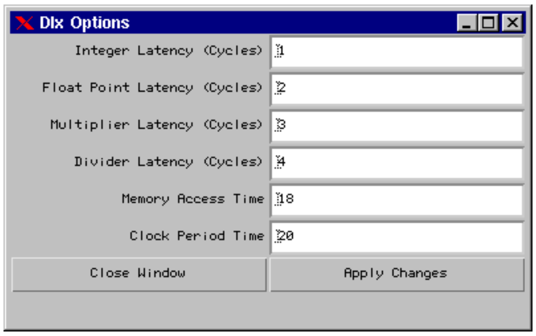

Figure 5.4: User Options Frame

5.2.3.1 Text Manipulation

In order that the model provide the capability for user defined code simulation, it was necessary to introduce a text parser capable of decoding the different instruction types. This also drove the need for instruction object manipulation throughout the life of a simulation run. Indeed the ability for a textual instruction to be 'wrapped' into an instruction object class was essential for the system development. It was then possible to act on instructions using the JAVA *instanceof* method which returns the type of the object in question. For example at the instruction fetch stage when checks must be performed for control flow instructions this was necessary. When an instruction is fetched from memory it can be 'wrapped' and manipulated by the model depending on its returned type. The parser also provides the system with error detection capabilities for instructions entered to the system which are not currently supported.

The required passing of JAVA defined objects between entities in the Simjava model permitted the code to be passed as a string object containing the full instruction. It removed the need to further code the text entry into numeric or other forms for manipulation by the system. However, the ability to translate the instruction object into a binary or hexadecimal form was recognised as a powerful additional feature that could be added.

The manipulation of text was also required to animate the data movements along the various buses. This permitted the user to directly associate an instruction within the system with its position in the code being modelled. It also provides for the clear interpretation of any code executing out of order and further emphasises the dependencies causing data hazards.

5.2.3.2 Queues

As Java does not permit memory allocation of arrays on the heap it was necessary to make use of the API *Vector* class for the various instruction buffers and queues required by the system. A prime example is the instruction issue queue. The system must cope with a variety of queue sizes, depending on the assembly program being simulated and the latencies of the system. A fixed size array lacks the dynamic capabilities to provide for multiple type object storage and also, given the customisable features of the system, was felt to provide for less effective resource utilisation. Use of the *Vector* class overcame this problem.

5.2.3.3 Synchronisation

The synchronisation of events became a major feature of the simulation. This added considerably to the behavioural descriptions of entities and provided some further complexity to the testing and development of the system. The two phase mechanism introduced the need for each entity to hold data between successive clock signals and manipulate the data as appropriate. Ensuring that each entity reached the 'barrier' and then halted execution until a further control signal was broadcast, proved complicated. However, this did provide some benefits for the debugging of entities. This is discussed more fully in Chapter 6.

A major benefit from the added complexity and extra coding required was the inclusion of load and stall cycles, associated with a 'real' architecture implementation. This addresses the existing weakness of the DLXsim representation which does not account for this [H&M 96]. The architecture is clearly simulated as a synchronous piece of machinery dependent upon the sending and receiving of data.

5.3 Final Web-Page Design

The entity and *applet* features highlighted in Section 5.1 and Section 5.2, respectively, then provide the necessary functionality for the inclusion of the *applet* into a web page. (See Figure 5.5) In summary the implementation provides:

- Focus on the model (options are simply minor feature buttons)
- No unnecessary clutter
- Buttons have obvious meaning
- Instruction memory represented at high level
- User option to modify instruction memory
- Register contents visible
- Hazard detection represented at run time
- Clock cycles displayed
- Value of the program counter visible
- Separate frame window options and help
- User is given high level of control over simulation
- Complements Simjava controls (layout,run,pause,stop & speed)

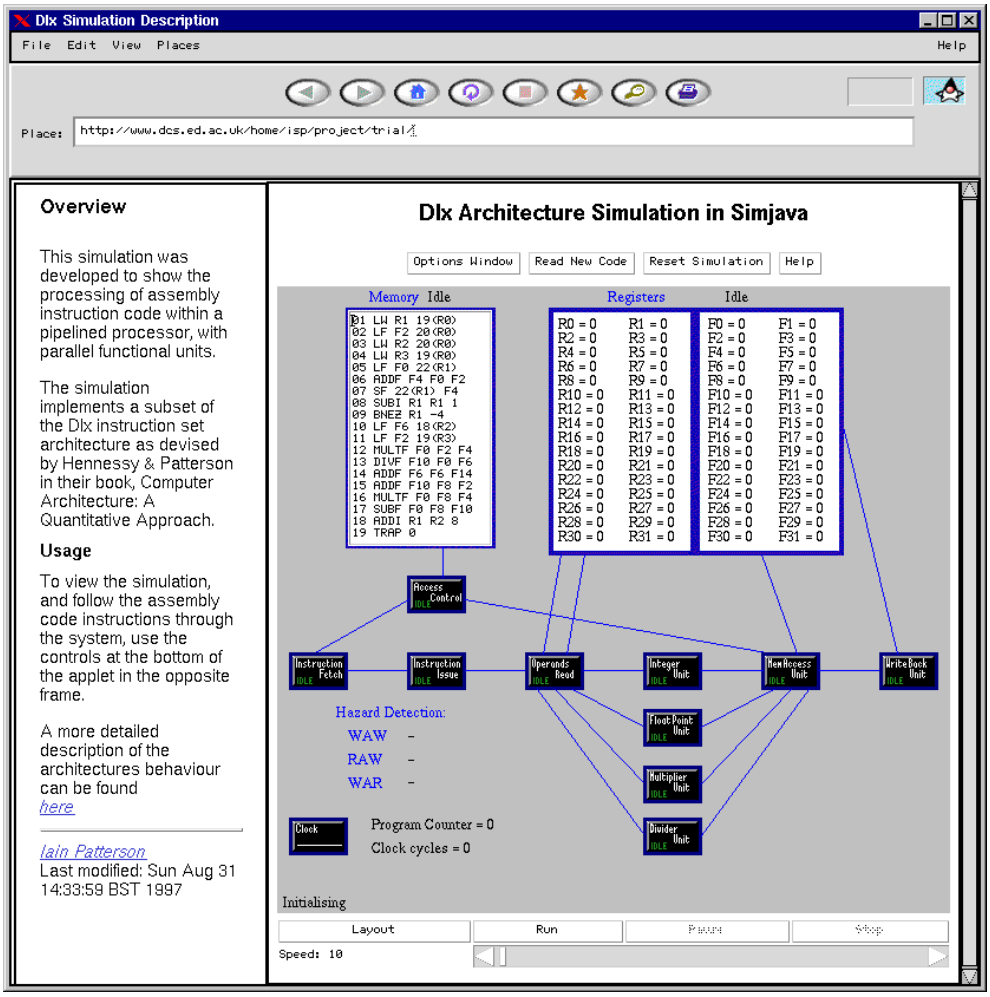

Figure 5.5: Web-Page Display

Chapter 6

Testing and Debugging

The successful implementation of the DLX model required that the simulation behave in a controlled and consistent fashion. The motivation for this chapter is to provide the reader with a brief insight into the variety of problems encountered as the project developed and how they were overcome.

6.1 Validation & Verification of Behaviour

The validation and verification of the model was perhaps the most important factor throughout the project. Consistency and predictability for the manipulation of instructions was vital. This becomes even more critical when it is considered that the model incorporates user defined program input.

Validation of the model was necessary to ensure that it conformed with the necessary behaviour of the DLX architecture. This was carried out in an ongoing manner, testing individual module behaviour by providing isolated examples before introducing them to the entire system. However, the pipeline behaviour was heavily dependent upon its ability to interact with other units in the system. True behaviour validation was only possible when all the elements were brought together.

Verification of the system behaviour was carried out initially by observing the instruction interactions with components. The lack of debugging facilities available required that a large number of screen output traces be added to the program. These then acted as indicators for the run time behaviour of various entities and pointed to areas of the coding which caused them. Understanding the necessary behaviour of the system was vital so that the path of an instruction could be followed and discrepancies observed.

The addition of the register file data contents aided considerably to the process of verification. Knowing the values of the registers, as instructions executed, provided a calculable measure which permitted confirmation of the actual system behaviour. It also provided important information for verifying the reading and writing of registers. Indeed this process further highlighted the benefit that a student could gain from the implemented representation of the system. The visual representation of information provides a powerful tool for associating a sequence of events with 'cause and effect' variables.

Known hazards were introduced to pieces of code. Initially these were added in a controlled fashion limiting their incidence of occurrence. As the model was developed the occurrence of multiple hazards was introduced. This was further extended to incorporate a variety of situations such as hazard identification while performing a pipeline stall.

The following example, (See Figure 6.1) highlights the system's capability to provide for multiple scenarios. At this point in the simulation the pipeline has been stalled by instruction number nine. The fetch unit has halted requests to memory and is awaiting the outcome of the branch instruction for the correct value of the program counter. Instruction nine has progressed to the stage where it needs to access the source register which it must test. Instruction seven is being sent to memory as it is a store instruction, having read the contents of the register F4 and computed the necessary store address should the branch be taken, i.e. 22(R1). Instruction eight is being computed in the integer unit and it is this result which the branch instruction is dependent upon for successful completion. The detection of the hazard is clearly represented to the student stating 'RAW -found'. This simple example further highlights the difficulty associated with providing a student with a concise representation of a complex synchronous model.

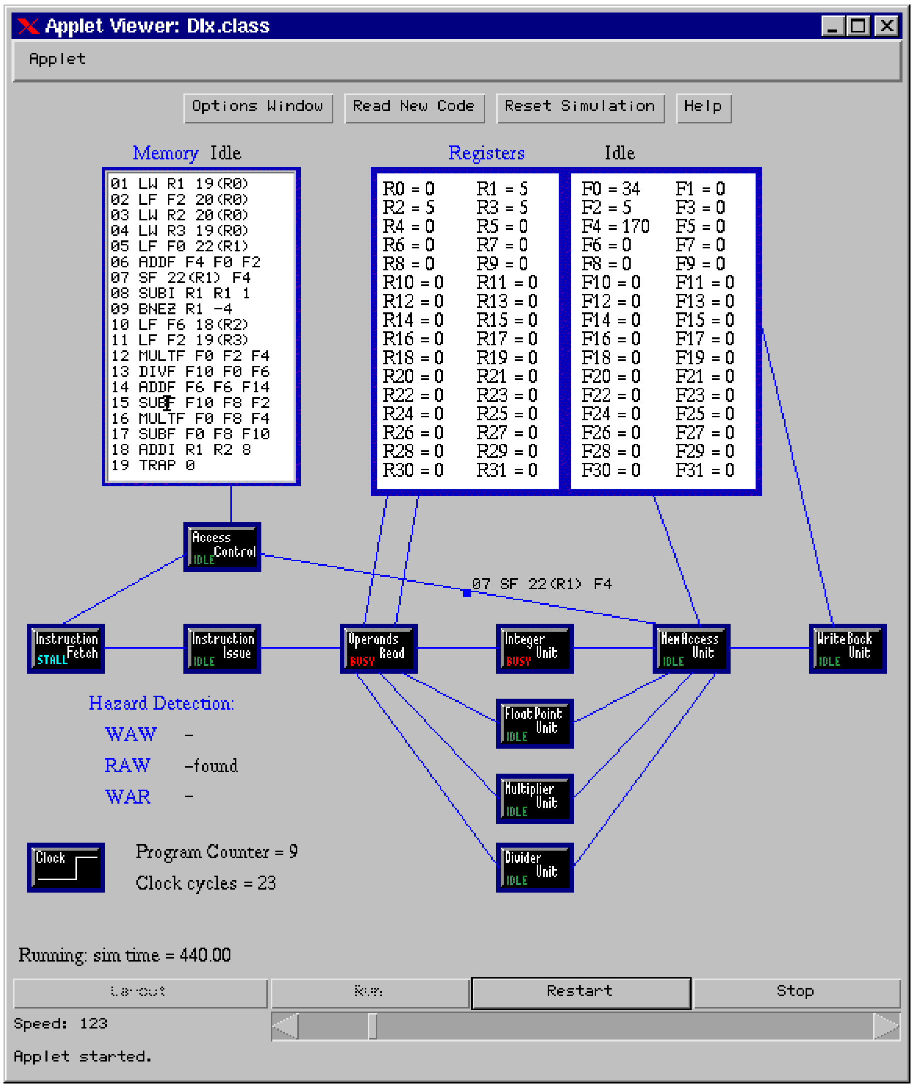

Figure 6.1: Hazard Detection and Pipeline Stall: Test Case

6.2 Debugging

As Java is still in its infancy the majority of tools for application development are limited in behaviour. This is especially true with regards the available tools for *applet* debugging. This problem is largely due to Java's multi-threaded concurrent nature and the problem was in many ways exacerbated by the highly concurrent nature of the simulation model being represented.

The increased complexity of the model, with the inclusion of the synchronous clocking mechanism, added considerably to the logic of the system and the necessary debugging, though it is felt that the benefits resulting from such a mechanism justify this. However, the introduction of the clock control, with its management of the simulation timing and event interactions, assisted the later development stages of the project as the system grew in complexity. It permitted critical errors in the system to be drawn out. As entities were activated, and their responses gathered, it was possible to detect the more obvious errors. Robustness was added to the system in that the clock was able to verify that an entity did not schedule out of order and that a single entity did not reply more than once in any given clock period.

As mentioned in Chapter 5, section 5.2.3.2 the issue of queues and their hidden nature provided considerable obstacles to the development. As the majority of the simulation behaviour must remain hidden, to provide a clear representation of the model, control queues and other features associated with the model had to be checked to verify their contents. The *applet* implementation required that queue verification be performed with system output. (i.e. print statements to the screen which were later removed)

While the Java environment has the capabilities for developing visual, distributed simulations, the development process within the Simjava framework was hampered by the weakness of the debugging facilities. The *Java Debugger* (JDB) is command line driven and offers the software developer only a limited number of options, especially with regards to the debugging of threads.

In addition, the development of *applets* is continually at the mercy of updates to the JDK API [JSOFT 97] and the consequential deprecation of various methods. This also presented problems associated with accessing up to date development information such as the available tools within the JAVA core API.

As with many simulation models attempting to simulate synchronous activities, the model relies upon zero simulation time synchronisation for the sending and receiving of multiple events. Several constructs in the SIMJAVA package provide for selection of events of either a known type, from a named entity or simply waiting for the next event. As the run time scheduling of threads, and their non-deterministic parallel implementation, has the potential to affect the ordering of such events, the selection methods utilised by the various entities were critical. Subtle run time variations in the thread scheduling provided the early model with errors that were difficult to reproduce and it was necessary to add a number of spin-lock variables to the system. These ensure the accessing of certain global variables used by the system, though do so without affecting the simulation time.

Chapter 7

Conclusions

The project successfully implemented a subset of the DLX instruction set architecture as an online animated simulation model. The model is implemented as part of an interactive *applet*. This addresses the identified weaknesses of existing models by providing a high level abstract representation of a pipelined, multicycle processor.

The model possesses the ability for user defined instructions to be entered into memory and run through the system. This can be accomplished prior to execution and also at run time. This addresses the necessary aim of providing the user with a tool that promotes learning through active user involvement. Hazards introduced by user defined code are detected and the user is informed when these occur.

The model also has the ability for users to define the cycle latencies for the four parallel functional units and both the clock and memory access times. The impact of a change in any of these can then be observed by the user as a simulation progresses. This too provides for the opportunity of improved learning through active user involvement with the system.

As a simulation progresses the user is presented with clearly defined movements of data through the system. Instructions are animated next to the appropriate bus as data is passed along it. For example, in the case of reading from the register file the data values read are displayed and when writing the outcome of an arithmetic operation back to the register-file the newly computed value is shown with the destination register in question. These features reinforce the state of the system and provide the user with a positive identification of where an instruction is as it progresses through the system.

The ability of the system to represent the actual data values of registers further permits meaningful assembly code to be entered. For example, the default code performs a for loop in which the test value is decremented and tested five times before exiting the loop. Using a wait-until-known branch scheme the user can see the effect that control flow instructions have on pipeline efficiency.

Definition of customised executions is further enhanced by the fact that the model *applet* always remains the central feature. The options are presented as additional frame windows which can remain hidden or be opened alongside the simulation window. This overcomes the problems associated with simulation tools that require separate HTML form submission to modify the simulation output.

As a tool for teaching computer architecture there is the added ability to show how compiler transformations affect the hardware usage. Various compiler optimisations can be represented by the instruction code and their effect on the utilisation of the system entities modelled.

The work undertaken also brought to light the weakness of the present tools for the debugging of complex threaded *applets*. This is especially true of the DLX model developed as it relies heavily upon the run time synchronisation primitives provided by JAVA to maintain realistic behaviour. The DLX model itself represents a synchronous system requiring a high level of concurrency for the interaction of events with each successive clock cycle.

The model was used by a number of fellow students to test its potential as a teaching aid. The response received was very favourable and they felt it assisted in their understanding of the behaviour of the architecture.

7.1 Further Directions

The recent extension of the SIMJAVA package to incorporate timing and graph information would be a potential inclusion for future updates to the model. This could highlight utilisation and other statistical information for the system.

As the model represents a pipelined architecture the introduction of additional branch control policies and prediction schemes would provide the user with a powerful tool for investigations into the effectiveness of such schemes. The different schemes could then be correlated directly with different pieces of code to see how they perform.

Data forwarding could be incorporated into the model in order that the user gains further insight into actual policies implemented which aim to improve pipeline efficiency. It was hoped that this might be provided though it was not achieved through pressure of time.

The model would also benefit from the extension of the working subset of instructions implemented. A full implementation of the entire DLX instruction set architecture could ultimately be introduced.

Teaching exercises could be devised around the model. For example, a piece of code which contains many hazards could be provided to a student. The object would then be for the student to rearrange the code to provide a more effective execution.

Bibliography

[H&P 96] '*Computer Architecture: A Quantitative Approach*' by Hennessy John L. & Patterson David A. Published by: *Morgan Kaufmann*, 1996

[S&K 96] '*The DLX Instruction Set Architecture Handbook*' by Sailer, Philip M. & Kaeli, David R. Published by: *Morgan Kaufmann*, 1996

[I&T 92] '*Architecture of High Performance Computers Volume I*' by Ibbett, R.N. & Topham, N.P. Published by: *Macmillan*, 1992

[CW&I 97] '*An Interactive Environment for the Teaching of Computer Architecture*' by Coe, P.S., Williams, L.M. & Ibbett, R.N. Published by: *Computer Systems Group, University of Edinburgh*, 1997

[M^c^N&S 97] '*Java: The Complete Reference*' by M^c^ Naughton, Patrick & Schildt, Herbert. Published by: *McGraw-Hill*, 1997

[CJA 97] '*Core JAVA*' by Cornell, Gary & Horstmann, Cay S. Published by: *SunSoft Press*, 1997, Second Edition

[N&P 96] '*Exploring Java*' by Niemeyer, Patrick & Peck, Joshua. Published by: *O'Reilly and Associates, Inc.*, May 1996

[FLAN 96] '*Java in a Nutshell*' by Flanagan, David. Published by: *O'Reilly and Associates, Inc.*, February 1996, First Edition

[SUN 96] '*The Java Language:An Overview*' Published by: *Sunsoft Press*, 1996. Web page location: `http://java.sun.com/docs/white/`

[M^cN&H 97] '*Simjava: A Discrete Event Simulation Tool (Release 1.2)*' By: M^cNab R.& Howell, F. Web page location:
`http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.2/`

[B&S 96] '*Discrete Event Simulation on the World Wide Web Using Java*' by Buss, Arnold H. & Stork, Kirk A. Published by: *Operations Research Department, Naval Postgraduate School, Monterey, California, USA*, 1996
Web page location: `http://131.120.142.115/~kastork/simkit_home.html`

[FISH 96] '*Web Based Simulation: Some Personal Observations*' by Fishwick, Paul A. Published by: *Dept. of Computer Science and Engineering, The University of Florida*, 1996

[CB 97] '*Virtual Memory Simulation in Java*' by Chakraborty, A., & Bhowmik, J.
Web page location:`http://earth.usc.edu/~avijitc/final/index.html`

[H&M 96] '*DLXsim*' by Hosteler,L.B. & Mirtich B. Web page location:
`http://hardy.ocs.mq.edu.au/mpce_courses/dlxsim/report.html`

[PURDUE 97] '*DLXview*' Web page locations:
`http://yara.ecn.purdue.edu/~teamaaa/dlxview`
`http://purcell.ecn.purdue.edu/~casle`

[JSOFT 97] '*The Java Development Kit JDK 1.1*' by JavaSoft (JDK1.1.3 Documentation Release) Web page location: `http://java.sum.com/`

[JDR 97] '*The Java Developer: Resources*'. Web page location:
`http://www.digitalfocus.com/digitalfocus/faq/resources.html`

[DFAB 93] '*Human-Computer Interaction*' by Dix, A., Finlay, J., Abowd, G. & Beale R. Published by: *Prentice Hall*, 1993

[Cxxsim 97] '$C^{++}sim$' Web page location:

`http://cxxsim.ncl.ac.uk/homepage.html`

[SDLX 97] '*SuperDlx-1.03*' Web page location:

`http://www-acaps.cs.mcgill.ca/superdlx/`

[THER 97] '*VHDL Dlx*' Web page location:

`http://www.rit.edu/~paf6282/thesis_description.html`

[DADD 97] '*Dlx additions*' Web page location:

`http://www.cs.nthu.edu.tw/~dr838312/architecture/hw2/dlx.html`

[IWAN 97] '*DigSim*' by Iwan van Rienwen Web page location:

`http://www.lookup.com/Homepages/96457/home.html`

[SIMPACK 97] Web page location:

`http://www.lookup.com/Homepages/96457/home.html`

[WEBSIM 97] Web page location:

`http://ms.ie.org/websim/survey/survey.html`

Appendix A

Dlx Instruction Set

Description	**Mnemonic**	**Operand(s)**
load	LB,LBU,LH,LHU,LW,LF,LD	$rd, offset(rs_1)$
store	SB,SH,SW,SF,SD	$offset(rs_1), rd$
move	MOVI2S,MOVS2I,MOVF	rd, rs_1
	MOVD,MOVFP2I,MOVI2FP	
integer arithmetic	ADD,ADDU,SUB,SUBU,MULT	rd, rs_1, rs_2
	MULTU,DIV,DIVU	
	ADDI,ADDUI,SUBI,SUBUI	rd, rs_1, rs_2
logical	AND,OR,XOR	$rd,$
	ANDI,ORI,XORI	$rd, rs_1, immediate$
	LHI	$rd, immediate$
shift	SLL,SRL,SRA	rd, rs_1, rs_2
	SLLI,SRLI,SRAI	$rd, rs_1, immediate$
floating-point arithmetic	ADDF,ADDD,SUBF,SUBD	rd, rs_1, rs_2
	MULTF,MULTD,DIVF,DIVD	
convert	CVTF2D,CVTF2I,CVTD2F	
	CVTD2I,CVTI2F,CVTI2I	rd, rs_1
set-on-comparison	LTF,LTD,GTF,GTD,LEF,LED	rs_1, rs_2
	GEF,GED,EQF,EQD,NEF,NED	
jump	J,JAL	$name$
	JR,JALR	rs_1
branch	BEQZ,BNEZ	$rs_1, name$
	BFPT,BFPF	$rs_1, name$
special	TRAP,RFE,NOP	$name$

Table A.1: Dlx Instruction Set

Appendix B

Scoreboard Tables

Instruction	Issue	Read Operands	Execution Complete	Write Result

Table B.1: Instruction Status Table: The Stage the Instruction is at

Unit Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk

Table B.2: Functional Unit Status

Busy refers to whether the unit is busy. Op refers to the necessary operation. Fi is the destination register. Fj&Fk are the source register numbers. Qj&Qk are the functional units producing the source registers Fj&Fk. Rj&Rk are flags which indicate the readiness of Fj&Fk.

Register	F0	F2	F4	F6	F8	F10	F12
Fj							

Table B.3: Register Result Status: Which Functional Unit Writes Each register