A report on the Mini-Project

MIPS Microprocessor
and
Cache Simulation

Submitted in partial fulfilment of

the requirements for the award of the Degree of

Bachelor of Technology

in

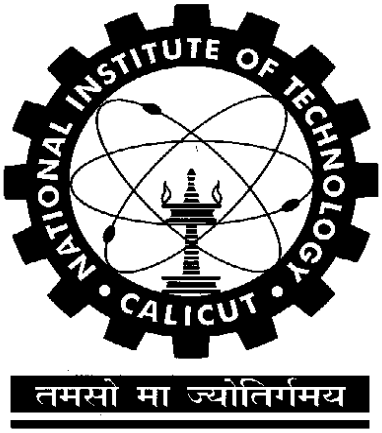
Computer Science and Engineering

at

The National Institute of Technology, Calicut

by

Varghese Mathew (Y2.049)



DEPARTMENT OF COMPUTER ENGINEERING

National Institute of Technology

(A Deemed University)

CALICUT, KERALA - 673 601

2005

National Institute of Technology, Calicut

A Deemed University

Calicut REC (P.O.), Kerala 673 601, India.

Telephone : 0495 2286100
0495 2286101
Fax : 0495 2287250
e-mail : nitc@nitc.ac.in
website : www.nitc.ac.in

CERTIFICATE

This is to certify that the report entitled **"MIPS Microprocessor and Cache Simulation"** is a bona fide record of the project work done by **Mr. VARGHESE MATHEW**, (Roll No Y2.049) under our supervision and guidance. The project is submitted to the National Institute of Technology, Calicut, in partial fulfilment of the requirements for the award of the Degree of **Bachelor of Technology** in **Computer Science and Engineering**.

Dr. V.K. Govindan
Professor and Head,
Department of Computer Engineering,
NIT, Calicut.

Dr. Priya Chandran
Project Guide,
Assistant Professor,
Department of Computer Engineering,
NIT, Calicut.

Abstract

This project is an attempt to simulate the pipeline of an MIPS microprocessor, and thereby obtain an insight into the nuances of design, efficiency considerations etc. The five stage pipeline of the MIPS microprocessor has been simulated for integer instructions. Also provided is a facility for simulation of a cache system, allowing means for performance analysis of various cache configurations. The project is specifically oriented at the academic community, to form a study tool for aspirants of Computer Architecture, Operating systems, Compilers etc.

To my mother, my father, and my sister …

Acknowledgements

Working with others, leaning onto them when in need of help, support and encouragement, is the most joyful experience in working on any project; however small, or however big may it be. And I would like to express my heartfelt gratitude to everyone who stood by me, helped me, and encouraged me all through this project, right from its conception, to its successful consummation.

At this point, I first express my sincere gratitude to **Dr. V. K. Govindan**, Professor and Head of the Department of Computer Engineering (CSED), National Institute of Technology, Calicut (NITC), for his support and encouragement throughout the project.

I also express my heartfelt gratitude to **Mr. Vinod Pathari**, Lecturer, CSED, NITC, for his invaluable suggestions, support, and guidance during the conception of this project.

I express my profound and heartfelt gratitude to my guide, **Dr. Priya Chandran**, Assistant Professor, CSED, NITC, for her expert guidance, continued help and support, and limitless encouragement, throughout this project. It is my wish to state that this project would never have been so successfully completed, had she not made me believe in myself, guided me and propelled me in the right direction.

I extend my gratefulness to the entire faculty & staff of the Department of Computer Engineering, as well as of the whole university, who aided me, guided me, and stood by me, in big and small ways, all through the proceeding of this project.

I remember, with deep pleasure, all the help & encouragement given to me, by my parents, my family, and each and every one of my friends; the sacrifices they made for my sake, and their continued concern about my project; and express, within limitations words impose, my immeasurable gratitude to each and every one…

Varghese Mathew

Contents

List of Figures

Enclosure

CD ROM containing the project work and soft copy of the report.

1. Introduction

Man's laziness, his perennial quest for an easier way out, and his curiosity to know the unknown, has always been the driving forces behind all innovations ever made. We have progressed from the wheel to machines, to electricity, to vehicles, gigantic edifices, global communication systems and the like. But one invention stands out and may be rightfully accredited as the most significant in history; the Computer. With it came a tool that could do nearly anything so long as appropriately instructed.

But like brine quenching thirst, man always wanted more. The computers of bygone days were never fast enough; and we went on to discover microprocessors, caches, pipelining, multiprocessors, super-scalars, distributed systems, and a lot more.

In this project, I try to mainly simulate two such mechanisms to speed up the computer; Pipelining and Caches.

A set of integer instructions from a MIPS microprocessor [Appendix A], the MIPS R2000, was chosen to form the instruction language for the simulated microprocessor. A simple 5 stage pipeline studied in ref. [1] was chosen for simulation. Also, the environment was designed in such a way as to allow various types of cache modules to be written and interoperated.

1.1. The Pipeline

Pipelining is a technique used in implementing processors, wherein execution of multiple instructions is overlapped to enhance the performance by reducing the aggregate execution time.

We may draw an analogy with a car assembly line to describe pipelining and discover its advantage.

Assembling a car may be divided into 6 steps.

1. Set up the base structure
2. Install the engine
3. Fix the chassis
4. Fix the tyres
5. Paint the car
6. Fix the Windows

So to assemble 'n' cars, a simple approach would be to first perform the 6 actions to make the first car; then perform these 6 actions again to make the second car; then again for the third, the fourth and so on…

But under an Assembly Line approach, or a pipelined approach, you have six people; the first person sets up the base structure, passes that to the second person, and starts working on the base structure for the next car. The second person accepts a base structure from the first, installs the engine, passes that on to the third person, gets the next base structure from the first person and starts working on it. The third person gets the car from the second, fixes the chassis, passes it on to the fourth and takes up the next one from the second. The fourth gets the car from the third, fixes the tyres, passes it onto the fifth and then gets the next car from the third. The fifth person gets the car from the fourth, paints it, passes it onto the sixth and gets the next car from the fourth. The sixth person gets the car from the fifth, fixes the window, rolls out the finished car, and takes up the next one from the fifth.

Thus, in our simple approach, a new car is rolled out in every six units of time, but in the pipelined approach, after the first five units of time, a new car is rolled out every unit of time. In other words, for a large number of cars, our pipelined approach is six times faster than the simple approach.

Pipelining instruction execution in a processor can now be approached in a similar manner. First we identify the different stages in the execution of an instruction in the processor; these may be instruction fetch, instruction decode, register fetch, arithmetic computation, memory access, register write, etc. In order to maximise pipeline performance, the various stages should have similar execution times.

Next we implement the processor in such a manner that when an instruction is in one stage of execution, the succeeding instruction will be in the just previous stage of execution, an so on…

1.2. The Cache

As programs became larger, and as technology advanced, larger memories became available for use in computer. At the same time, the processors too were growing faster, and these memories which were removed from the processor, turned out to form bottlenecks due to their large access delays.

The principles of locality of reference came to the rescue at this juncture. These principles state that locations recently referenced as well as those adjacent to them have a greater probability for future reference than any other arbitrary location. Thus was introduced, the **"cache"**, which formed a cache or storage place on the processor, for locations in memory which were either recently referenced or are adjacent to those which were recently referenced.

Because the cache is closer to the processor than the memory, is smaller than the memory, and is implemented in faster hardware than the memory; accesses fulfilled from the cache take significantly lesser time for fulfilment than accesses requiring reading from / writing to the memory.

At this point, the reader may refer to ref. [1] for a rigorous study of caches.

2. Software Requirements Specifications

The following gives the list of requirements which form the objectives to be fulfilled through the development of this application.

2.1. Scope of the Product

This simulation suite is oriented for use by the academic community. The suite is expected to give an insight into the nuances in implementation of pipelines in processors. The suite also will form a platform for simulating, and analysing some of the performance parameters of Caches. Additionally, implementing Branch Target Buffers, Compilers, Operating systems etc. for the suite will form an interesting assignment to the enterprising academician.

2.2. The Instruction Set

As mentioned earlier, the subset of integer instructions from the instruction set of the 32bit MIPS R2000 processor was chosen for simulation. (The entire instruction set for the same can be found in ref. [2].) The semantics of some of these instructions were altered slightly for this project. In addition, a few instructions for providing input and output capabilities have been added.

The complete list of instructions selected for simulation has been enumerated in [Appendix A]. The Software developed is expected to adhere strictly to this instruction set. However, the processor may be implemented as big endian / little endian depending on whichever turns out to be convenient.

2.3. The Pipeline

The simple 5 stage pipeline described and studied in ref. [1] has been chosen for implementation. The different stages of the pipeline and their functions relating to different classes of instructions have been summarised below.

2.3.1. The IF (Instruction Fetch) stage

This stage fetches the instruction from the instruction cache on the processor and increments the PC registers.

2.3.2. The ID (Instruction Decode) stage

This stage determines what instruction was fetched, fetches the different source register values required by the instruction, pre-processes the immediate values, and in the case of some branch instructions, executes the branch instruction.

2.3.3. The EX (Execution) stage

This stage handles the actual execution of most other instructions, except the branches mentioned above. For arithmetic and logic instructions, this stage performs the ALU operation. For memory access instructions, this stage performs the target address computation. And for the remaining branch instructions, which were not executed in ID stage, this stage performs their execution.

2.3.4. The MEM (Memory access) stage

This stage handles all memory and I/O device accesses. For simplicity, a memory system which always completes the requested operation within the current clock cycle has been assumed. However it is required that the memory system is modularised so that this can be modified to include access latencies for simulating caches more accurately.

2.3.5. The WR(Register Write) stage

This stage handles storing values into destination registers of instructions. Storing of values into the register file in this stage should precede the reading of values from the register file in the ID stage.

2.3.6. Data forwarding

In the pipeline, it may happen that an instruction needs to read a register which one of the preceding instructions is supposed to write, but the preceding instruction has not reached the WR stage, wherefore the value has not yet been stored in the register. In such cases, the value needs to be retrieved from the source instruction and not from the register file. This process of retrieving the value directly from the source instruction is called Data Forwarding. The simulation is expected to simulate Data Forwarding as well.

2.3.7. Multithreading

The simulation is expected to simulate the pipeline in a multithreaded fashion, with each stage being handled by a thread. The synchronisation of these threads should also be handled.

2.4. The Cache

As far as Cache simulation is concerned, the software is expected to provide a cache interface which subsequent developers may instantiate to simulate different types of caches. The software must also allow simulation of multilevel caches by stacking different types of caches one on top of the other.

2.5. Additional Devices

The software should simulate ports which connect the processor to peripheral devices. These may be implemented using network sockets. Also a demonstrative module simulating a simple I/O terminal should be coded and connected to the processor using this feature. (As mentioned earlier, instructions to write to / read from these ports were included in the chosen instruction set.)

2.6. Assembler

As an emergent requirement, a simple assembler needs to be implemented for the simulated processor. This assembler should support all the instructions in our selected instruction set. The assembler should take its input from a named file and write its output to another named file.

2.7. Platform requirements

The application should be developed targeting POSIX compliant Operating Systems like UNIX, LINUX etc., running on an Intel x86 architecture machine.

2.8. Miscellaneous requirements

2.8.1. The application should display informative messages to the user, as to what goes on in each of these stages in each of the clock cycles.

2.8.2. The application should give facilities by which the user may trace the simulated execution of a user program on the simulated processor.

3. Design

The entire software was divided into mainly 3 applications: Assembler, Processor Simulator, and I/O device.

3.1. The Assembler

The assembler was designed to be implemented using **bison** (parser generation tool) and **flex** (lexical analyser generator tool). The grammar for the assembly language for MIPS was identified [Appendix B] and the corresponding actions were formulated.

3.2. The Processor Simulator

This application is the core of this project and thus necessitates a detailed study of its design.

3.2.1. The Processor Class

The Processor class represents the processor of the machine, i.e., the simulated processor. The design of the processor class is detailed in the class diagram below

Figure 3.1 *Class Diagram for the Processor Class.*

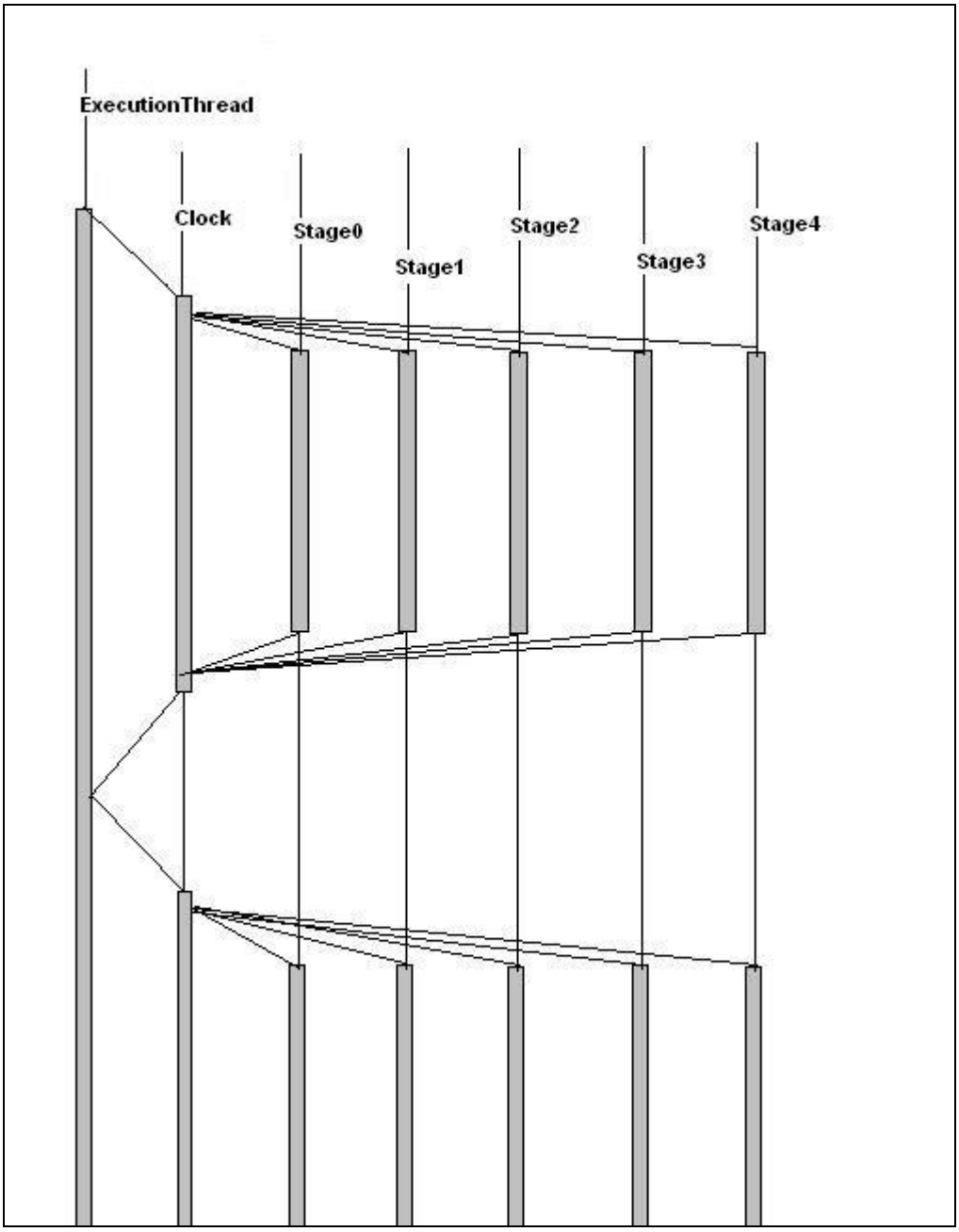
Processor
word_32 reg[32] // Register file word_32 Hi, Lo // Hi and Lo registers u_word_32 PCreg u_word_32 NPCreg PCWritingStage NPCfrom // Which stage writes the NPC bool flushStage[5] // Whether an instruction needs to be flushed MainMemory * mem Cache * dataCache Cache * instrCache PortManager * pman // Module that maps ports to sockets Latch inLatch[5] // Latches between stages Latch outLatch[5]

bool blockUpdate // Disable all writes
Processor (MainMemory * m, Cache * dc, Cache * ic, PortManager * pm) void Execute ( ) void ExecutionThread ( ) void Clock (int clk) bool RegisterFetch (RegisterFetchTarget target, int regNumber, bool noFail = false) bool RegisterFetch_Stage2 (RegisterFetchTarget target, int regNumber) bool RegisterWrite (int regNumber, RegisterWriteSource source) bool ReadMem (word_32 address, word_32 & result, int noOfBytes) bool WriteMem (word_32 address, word_32 value, int noOfBytes) void UpdatePC_Stage1 (word_32 value, PCUpdateType updateType) void UpdatePC_Stage2 (word_32 value, PCUpdateType updateType) void Stage0 ( ) void Stage1 ( ) void Stage2 ( ) void Stage3 ( ) void Stage4 ( )

This class works with 6 threads, 5 for the various stages and one for managing the clock, copying contents of the latches from the output side to the next stage's input side etc.

The sequence diagram for the threads is shown on the next page.

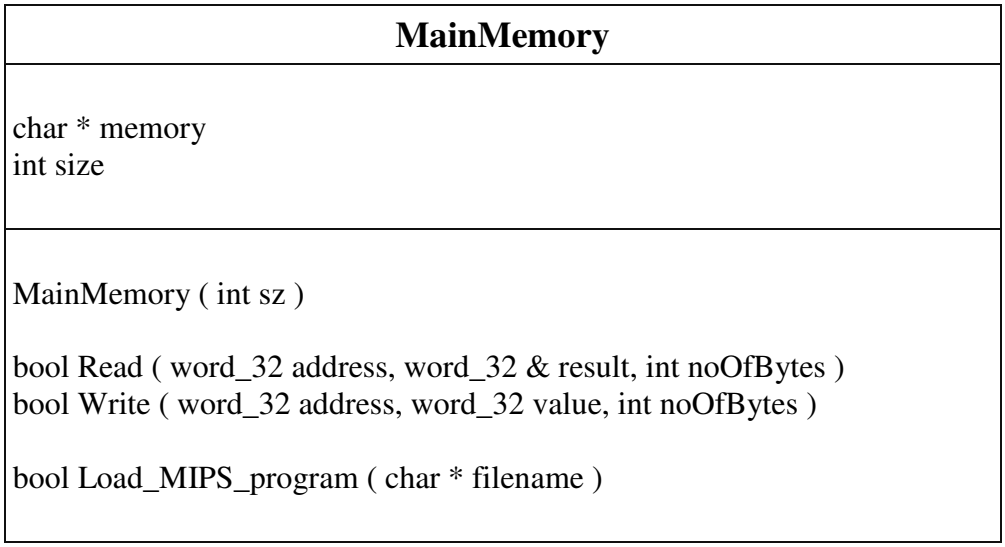
Figure 3.2 *The Sequence diagram for the key methods in the Processor Class.*



3.2.2. The MainMemory class

The MainMemory class represents the Memory or RAM of the simulated machine. The class diagram is detailed below.

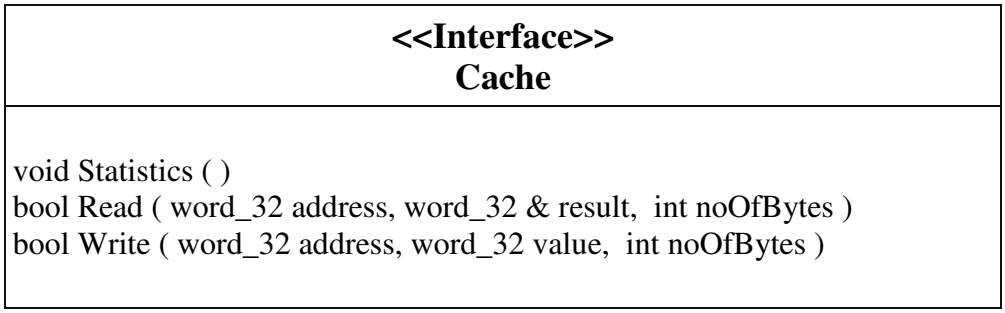
Figure 3.3 *The class Diagram for the MainMemory class*



3.2.3. The Cache Interface

This is the interface for all caches that may be implemented on this simulation suite. The diagrammatic representation follows.

Figure 3.4 *The Cache Interface*



3.2.4. The PortManager Class

The PortManager class manages the mapping between the ports on the simulated machine and the networks sockets using which they are implemented.

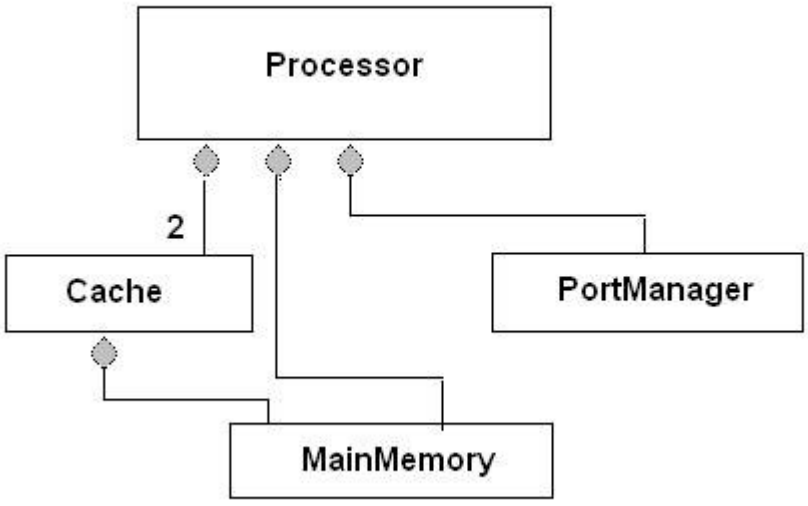
Figure 3.5 *The Class Diagram for the PortManager Class*

PortManager
char hostname[32] int portMap [MAX_PORTS]
PortManager ( ) int AddPort (int portNo, int mapsTo) int RemovePort (int portNo) int Write (int portNo, word_32 oneWord) int Read (int portNo, word_32 & oneWord)

3.2.5. The Class Diagram

Given below is the class diagram depicting the major classes in the application and the associations between them.

Figure 3.6 *The Class Diagram for the entire application*



3.3. The I/O device

The I/O device is mainly divided into two threads, the input thread and the output thread.

3.3.1. The Input thread

This thread opens a port, and listens for connections. Once the Processor Simulator makes a connection with this thread, this thread reads characters from the keyboard and writes them out onto the port.

3.3.2. The Output thread

This thread too opens a port, and listens for connections. Once the Processor Simulator makes a connection with this thread, this thread reads characters from the port and writes them out onto the screen / console.

4. Implementation Aspects

4.1. The system was developed using **C++ (ANSI / ISO)**, **flex**, and **bison**; for **Intel x86 machines**, running **LINUX or other POSIX compliant** operating systems.

4.2. As mentioned, the simulation suite is implemented in three applications, the assembler, the processor simulator, and the I/O device. These have been respectively named "**asm"**, "**mips**", "**dumbterminal**".

4.3. The directory structure in the accompanying source code is as follows

./mips	:	The processor simulation sources
./asm	:	The assembler sources
./io	:	The sources for the I/O devices
./include	:	Header files common to the suite
./test	:	Location where the suite is compiled and where the user programs reside

4.4. The invocation formats for the different applications are

asm : asm <outputfile> <inputfile>

mips : This does not take any command line arguments. However the first program executed will be the program named "**a.out**".

dumbterminal : This too doesn't take any command line arguments. It takes the input from a socket and writes it onto the screen, and takes user input from the keyboard and writes it onto another socket.

4.5. The different commands available on the "**mips**" interface are as follows

```
'n' execute one clock cycle of the processor.
'c <number>' execute <number> number of clock cycles.
'p' print values of registers. (small 'p')
'P' print values of registers for registers whose value
      is not zero. (capital 'P')
'q' quit.
'm <address>' display the value stored at memory addres <address>.
'd <address>' display the value stored at 1-level
            data cache address <address>.
'i <address>' display the value stored at 1-level
      instruction cache address <address>.
's' display stastics for the caches.
'b <breakpoint no.> <break address>' set one of the 0-15
      breakpoints. to unset a breakpoint, set its address as -1.
'B' view all breakpoints.
```

5. Conclusion

The simulation of the 5 stage MIPS pipeline was carried out successfully. The project gave an insight into the implementation of various machine instructions in hardware. The division of execution between the different stages for each of these instructions could be studied closely. Also there turned out to be some additional complexities because of the asynchrony in the execution of threads as against the completely synchronised execution of stages in an actual processor, but these complexities were also tackled successfully.

A simple single level write-back cache taking number of blocks, block size in words, and associativity as parameters was implemented on this simulation, and tested out. A multilevel cache consisting of 'n' levels of this single level cache stacked one on top of the other was also simulated successfully.

Appendix A: List of implemented machine instructions

Arithmetic and logic instructions

1. add rs, rt, rd

0	rs	rt	rd	0	0x20

Puts the sum of registers ‘rs’ and ‘rt’ into register ‘rd’

2. addi rt, rs, imm

8	rs	rt	imm

Puts the sum of register ‘rs’ and the sign-extended immediate into register ‘rt’

3. and rd, rs, rt

0	rs	rt	rd	0	0x24

Put the logical AND of registers ‘rs’ and ‘rt’ into register ‘rd’

4. andi rt, rs, imm

0xc	rs	rt	imm

Puts the logical AND of register ‘rs’ and sign-extended immediate into register ‘rt’

5. div rs, rt

0	rs	rt	0	0	0x1a

Divide register ‘rs’ by register ‘rt’, leave quotient in register ‘lo’ and remainder in register ‘hi’.

6. mult rs, rt

0	rs	rt	0	0	0x1b

Multiply registers ‘rs’ and ‘rt’, leave the lower order word of the product in register ‘lo’ and the higher order word in the register ‘hi’

7. nor rd, rs, rt

0	rs	rt	rd	0	0x27

Put the logical NOR of registers ‘rs’ and ‘rt’ into register ‘rd’

8. or rd, rs, rt

0	rs	rt	rd	0	0x25

Put the logical OR of registers ‘rs’ and ‘rt’ into register ‘rd’

9. ori rt, rs, imm

0xd	rs	rt	imm

Put the logical OR of register ‘rs’ and sign extended immediate into register ‘rt’

10. sll rd, rt, shamt

0	0	rt	rd	shamt	0

Shift register ‘rt’ left by shamt bits and store the result in register ‘rd’

11. sllv rd, rt, rs

0	rs	rt	rd	0	4

Shift register ‘rt’ left by amt. indicated by register ‘rs’ and store the result in register ‘rd’

12. sra rd, rt, shamt

0	0	rt	rd	shamt	3

Shift register ‘rt’ right by shamt bits and store the result in register ‘rd’. The sign bit is used as padding on the left

13. srav rd, rt, rs

0	rs	rt	rd	0	7

Shift register ‘rt’ right by amt. indicated by register ‘rs’ and store the result in register ‘rd’. The sign bit is used as padding on the left

14. srl rd, rt, shamt

0	0	rt	rd	shamt	2

Shift register ‘rt’ right by shamt bits and store the result in register ‘rd’. Zero bit is used as padding on the left

15. srlv rd, rt, rs

0	rs	rt	rd	0	6

Shift register ‘rt’ right by amt. indicated by register ‘rs’ and store the result in register ‘rd’. Zero bit is used as padding on the left

16. sub rd, rs, rt

0	rs	rt	rd	0	0x22

Put the difference of registers ‘rs’ and ‘rt’ into register ‘rd’

17. xor rd, rs, rt

0	rs	rt	rd	0	0x26

Put the logical XOR of registers ‘rs’ and ‘rt’ into register ‘rd’

18. xori rt, rs, imm

0xe	rs	rt	imm

Put the logical XOR of register ‘rs’ and the sign-extended immediate into register ‘rt’

Constant manipulation instructions

19. lui rt, integer

0xf	0	rt	imm

Load the upper half-word of the integer into the upper half-word of the register ‘rt’

Comparison instructions

20. slt rd, rs, rt

0	rs	rt	rd	0x2a	

Set register ‘rd’ to 1 if register ‘rs’ is less than ‘rt’, and to 0 otherwise

21. slti rt, rs, imm

0xa	rs	rt	imm

Set register ‘rt’ to 1 if register ‘rs’ is less than the sign-extended immediate, and to 0 otherwise

Branch instructions

22. beq rs, rt, label

4	rs	rt	offset

Conditionally branch the number of instructions specified by the offset if register ‘rs’ equals register ‘rt’

23. bgez rs, label

1	rs	1	offset

Conditionally branch the number of instructions specified by the offset if register ‘rs’ is greater than or equal to zero

24. bgtz rs, label

7	rs	0	offset

Conditionally branch the number of instructions specified by the offset if register ‘rs’ is greater than zero

25. blez rs, label

6	rs	0	offset

Conditionally branch the number of instructions specified by the offset if register ‘rs’ is less than or equal to zero

26. bltz rs. label

1	rs	0	offset

Conditionally branch the number of instructions specified by the offset if register ‘rs’ is less than zero

27. bne rs, rt, label

5	rs	rt	offset

Conditionally branch the number of instructions specified by the offset if register ‘rs’ is not equal to ‘rt’

Jump instructions

28. j target

2	target

Unconditionally jump to the instruction at target

29. jal target

3	target

Unconditionally jump to the instruction at target. Save the address of the next instruction in register 'ra' (31)

30. jalr rs, rd

0	rs	0	rd	0	9

Unconditionally jump to the instruction whose address is in register 'rs'. Save the address of the next instruction in register 'rd'

31. jr rs

0	rs	0	0	0	0x8

Unconditionally jump to the instruction whose address is in register 'rs'

Memory reference instructions

32. lw rt, offset (rs)

0x23	rs	rt	offset

Load the word at address "contents of register 'rs' + offset" into register 'rt'

33. sw rt, offset (rs)

0x2b	rs	rt	offset

Store the contents of register 'rt' into the address "contents of register 'rs' + offset"

Input / Output instructions

34. din rt, devno

0x3f	0	rt	devno

Read a word from device 'devno' and store it into register 'rt'

35. dout rs, devno

0x3e	rs	0	devno

Write the word in register 'rs' onto device 'devno'

36. rdin rd, rt

0	0	rt	rd	0	0x3f

Read a word from the device whose number is in register 'rt' and store it into register 'rd'

37. rdout rs, rt

0	rs	rt	0	0	0x3e

Write the word in register 'rs' onto the device whose number is in register 'rt'

Data movement instructions

38. mfhi rd

0	0	0	rd	0	0x10

Move the contents of register ‘hi’ to register ‘rd’

39. mflo rd

0	0	0	rd	0	0x12

Move the contents of register ‘lo’ to register ‘rd’

40. mthi rs

0	rs	0	0	0	0x11

Move the contents of register ‘rs’ to register ‘hi’

41. mtlo rs

0	rs	0	0	0	0x13

Move the contents of register ‘rs’ to register ‘lo’

Other instructions

42. syscall

0	0	0	0	0	0xc

Causes a system call / software interrupt. Register ‘v0’ contains the number of the system call.

43. nop

0	0	0	0	0	0

Does nothing

Appendix B: The grammar for the assembler

```
program: newlines begin_stmt newlines start_stmt instructions
      newlines END newlines

begin_stmt: BEG INTCONSTANT '\n'

start_stmt: START INTCONSTANT '\n'

instructions: newlines label_one_instruction

label_one_instruction: IDENTIFIER
      | one_instruction '\n'

one_instruction: DW INTCONSTANT
      | DW '[' INTCONSTANT ']'
      | DW STRING
      | ADD REGISTER ',' REGISTER ',' REGISTER
      | ADDI REGISTER ',' REGISTER ',' INTCONSTANT
      | ADDI REGISTER ',' REGISTER ',' IDENTIFIER
      | AND REGISTER ',' REGISTER ',' REGISTER
      | ANDI REGISTER ',' REGISTER ',' INTCONSTANT
      | DIV REGISTER ',' REGISTER
      | MULT REGISTER ',' REGISTER
      | NOR REGISTER ',' REGISTER ',' REGISTER
      | OR REGISTER ',' REGISTER ',' REGISTER
      | ORI REGISTER ',' REGISTER ',' INTCONSTANT
      | SLL REGISTER ',' REGISTER ',' INTCONSTANT
      | SLLV REGISTER ',' REGISTER ',' REGISTER
      | SRA REGISTER ',' REGISTER ',' INTCONSTANT
      | SRAV REGISTER ',' REGISTER ',' REGISTER
      | SRL REGISTER ',' REGISTER ',' INTCONSTANT
      | SRLV REGISTER ',' REGISTER ',' REGISTER
      | SUB REGISTER ',' REGISTER ',' REGISTER
      | XOR REGISTER ',' REGISTER ',' REGISTER
      | XORI REGISTER ',' REGISTER ',' INTCONSTANT
      | LUI REGISTER ',' INTCONSTANT
      | LUI REGISTER ',' IDENTIFIER
      | SLT REGISTER ',' REGISTER ',' REGISTER
```

```
            | SLTI REGISTER ',' REGISTER ',' INTCONSTANT
            | BEQ REGISTER ',' REGISTER ',' IDENTIFIER
            | BGEZ REGISTER ',' IDENTIFIER
            | BGTZ REGISTER ',' IDENTIFIER
            | BLEZ REGISTER ',' IDENTIFIER
            | BLTZ REGISTER ',' IDENTIFIER
            | BNE REGISTER ',' REGISTER ',' IDENTIFIER
            | J INTCONSTANT
            | J IDENTIFIER
            | JAL INTCONSTANT
            | JAL IDENTIFIER
            | JALR      REGISTER ',' REGISTER
            | JR REGISTER
            | LW REGISTER ',' INTCONSTANT '(' REGISTER ')'
            | SW REGISTER ',' INTCONSTANT '(' REGISTER ')'
            | MFHI REGISTER
            | MFLO REGISTER
            | MTHI REGISTER
            | MTLO REGISTER
            | SYSCALL
            | NOP
            | DIN REGISTER ',' INTCONSTANT
            | DOUT REGISTER ',' INTCONSTANT
            | RDIN REGISTER ',' REGISTER
            | RDOUT REGISTER ',' REGISTER
```

Appendix C: List of viable future enhancements

C-1. Graphical User Interface

The application was designed with future development of a graphical user interface also in perspective. Each of the status messages displayed on the standard output device has been formatted in a manner facilitating parsing. Thus a potential graphical interface design would be to set up a FIFO pipe between the graphical front end and the application. The Graphical Front End can parse the text it reads from the pipe to trace the status of the simulated processor. Likewise, when the Front End needs to issue commands to the simulator, it can do so by issuing them onto the pipe. Thus a graphical front end development may be carried out independently of the simulator core.

C-2. Exception Handling & Multiple Modes of Operation

Exceptions have not been simulated presently in the simulated processor. Also, the processor has been implemented to work in a single mode of operation only. However, both exceptions and multiple modes of operation are central to the idea of a practical processor. It would be an enterprising assignment for a future developer to implement these within the simulation.

C-3. Paging Support

The processor simulated is presently incapable of supporting Virtual Memory and other memory management paradigms because of the absence of a paging facility. But once the Multiple Modes of Operation mentioned above are implemented, a paging scheme can be set up using a TLB and a set of kernel mode instructions to manage its contents.

C-4. Compiler

As mentioned, a simple assembler has been developed for the simulated processor. However, assembly coding places limitations on the complexities of the user program. Also, since it is compiler generated code which finds more contemporary practical application, studies like cache performance analysis should preferentially be done on compiler generated code. Thus, implementation of a compiler for the suite will greatly benefit the suite.

C-5. Additional Devices

As has already been mentioned, a simple I/O device has been coded for the simulation suite. However, the design of the simulation suite allows implementation of additional devices independently. Viable assignments would be to implement a disk storage device, a more realistic console device, etc.

C-6. Operating System

Once the above mentioned enhancements have been made, an enterprising developer may further go in to implement a simple operating system over the simulation suite. Such an Operating System can make good use of the multiple modes of operation to differentiate the kernel and user modes of program execution, as well as the paging facility to implement multitasking.

Another viable enhancement to the processor would be implementation of branch prediction schemes like Branch Target Buffers etc.

Appendix D: Sample Interaction with the simulator

The "**fact.mips**" program:

```
        begin  1024
        start  1024

        j      MAIN

        dw     [200]
STACK
STR1    dw     "\nThe factorial of "
        dw     0
STR2    dw     " is "
        dw     0
STR3    dw     "\nEnter the number whose factorial is to be found : "
        dw     0

MAIN
        addi   $sp, $zero, STACK
        lui    $sp, STACK

        addi   $a0, $zero, STR3
        lui    $a0, STR3
        jal    PRINTSTR

        jal    GETINT

        add    $t0, $zero, $v0

        addi   $a0, $zero, STR1
        lui    $a0, STR1
        jal    PRINTSTR

        add    $a0, $zero, $t0
        jal    PRINTINT

        addi   $a0, $zero, STR2
        lui    $a0, STR2
        jal    PRINTSTR

        add    $a0, $zero, $t0
        jal    FACT

        add    $a0, $zero, $v0
        jal    PRINTINT

        j      EXIT


#-------------------------------------------------------------
# Procedure to find the factorial of a number
# a0 - number whose factorial is to be found
# v0 - returns the factorial

FACT    addi   $sp, $sp, -8
        sw     $ra, 4($sp)
        sw     $a0, 0($sp)

        slti   $t0, $a0, 1
        beq    $t0, $zero, FACT_L1

        addi   $v0, $zero,1
        addi   $sp, $sp, 8
        jr     $ra
```

```
FACT_L1       addi   $a0, $a0, -1
       jal    FACT

       lw     $a0, 0($sp)
       lw     $ra, 4($sp)
       addi   $sp, $sp, 8

       mult   $a0, $v0
       mflo   $v0

       jr     $ra


#------------------------------------------------------------
# Procedure to print an integer
# a0 - integer to be printed

PRINTINT
       addi   $sp, $sp, -16
       sw     $a0, 0($sp)
       sw     $t0, 4($sp)
       sw     $t1, 8($sp)
       sw     $t2, 12($sp)

       addi   $t0, $zero, 10
       add    $t2, $zero, $zero

       bgez   $a0, PRINTINT_LOOP1
       addi   $t1, $zero, '-'
       dout   $t1, 2              # print the - sign

PRINTINT_LOOP1
       div    $a0, $t0
       mflo   $a0                 # quotient
       mfhi   $t1                 # the remainder
       addi   $t1, $t1, '0'       # make it a character
       addi   $t2, $t2, 1         # keep count of number of digits placed
on stack
       addi   $sp, $sp, -4
       sw     $t1, 0($sp)         # place the digit on the stack
       bne    $a0, $zero, PRINTINT_LOOP1

PRINTINT_LOOP2
       lw     $t1, 0($sp)         # take the digit off the stack
       addi   $sp, $sp, 4
       addi   $t2, $t2, -1        # decrement count
       dout   $t1, 2              # display the digit
       bne    $t2, $zero, PRINTINT_LOOP2 # if there are more digits, loop

       lw     $a0, 0($sp)
       lw     $t0, 4($sp)
       lw     $t1, 8($sp)
       lw     $t2, 12($sp)
       addi   $sp, $sp, 16
       jr     $ra

#------------------------------------------------------------


#------------------------------------------------------------
# Procedure to print a string.
# a0 - start address of null terminated string

PRINTSTR
       addi   $sp, $sp, -8
       sw     $a0, 0($sp)
       sw     $t0, 4($sp)
```

```
PRINTSTR_LOOP
       lw     $t0, 0($a0)
       addi   $a0, $a0, 4
       beq    $t0, $zero, PRINTSTR_LOOPEND
       dout   $t0, 2
       j      PRINTSTR_LOOP

PRINTSTR_LOOPEND

       lw     $a0, 0($sp)
       lw     $t0, 4($sp)
       addi   $sp, $sp, 8

       jr     $ra
#-------------------------------------------------------------


#-------------------------------------------------------------
# procedure to get a number from the user
# v0 - the number input

GETINT
       addi   $sp, $sp, -16
       sw     $t0, 0($sp)
       sw     $t1, 4($sp)
       sw     $t2, 8($sp)
       sw     $t3, 12($sp)

       addi   $t1, $zero, '0'
       addi   $t2, $zero, '\n'
       addi   $t3, $zero, 10
       add    $v0, $zero, $zero

GETINT_LOOP
       din    $t0, 1              # read the digit
       beq    $t0, $t2, GETINT_END       # if digit is '\r' stop looping

       sub    $t0, $t0, $t1       # compute the arithmetic value of digit
       mult   $v0, $t3                   # $v0 = $v0 * 10
       mflo   $v0

       add    $v0, $v0, $t0       # $v0 = $v0 + new digit.
       j      GETINT_LOOP

GETINT_END

       lw     $t0, 0($sp)
       lw     $t1, 4($sp)
       lw     $t2, 8($sp)
       lw     $t3, 12($sp)

       jr     $ra

#-------------------------------------------------------------

EXIT
       end
```

Simulator output for the “fact.mips” program

```
[miniproject@localhost test]$ ./mips

Choose the type of cache you want for Data
1-level DATA Cache : The choices available are
 1. No Cache
 2. Simple single level FIFO writeback cache
 3. Multilevel Cache
Please enter your choice : 3

Enter the number of levels in the Cache : 2

Choose the 2-level cache :
2-level DATA Cache : The choices available are
 1. No Cache
 2. Simple single level FIFO writeback cache
Please enter your choice : 2

Enter number of Blocks in cache : 128

Enter number of words per block : 32

Enter associativity : 4

Select extent of cache info displayed :
 1.verbose
 2.silent
Enter your choice : 2

Choose the 1-level cache :
1-level DATA Cache : The choices available are
 1. No Cache
 2. Simple single level FIFO writeback cache
Please enter your choice : 2

Enter number of Blocks in cache : 16

Enter number of words per block : 4

Enter associativity : 2

Select extent of cache info displayed :
 1.verbose
 2.silent
Enter your choice : 2

Choose the type of cache you want for Instructions
1-level INSTRUCTION Cache : The choices available are
 1. No Cache
 2. Simple single level FIFO writeback cache
 3. Multilevel Cache
Please enter your choice : 2

Enter number of Blocks in cache : 62

Enter number of words per block : 16

Enter associativity : 4

Select extent of cache info displayed :
 1.verbose
 2.silent
Enter your choice : 2

[ PortManager :: AddPort ] Connected Device 1 to socket 5678 successfully
```

```
[ PortManager :: AddPort ] Connected Device 2 to socket 5680 successfully
[** Clock: 0 **] Executed...

mips > c 990
[__ PC_update_control __] NPC updated by stage 0 to 1028
[ Stage0 ] PC to fetch = 1024, Instruction = 34114
[ Stage1 ] NOP
[ Stage2 ] NOP
[ Stage3 ] NOP
[ Stage4 ] NOP
[** Clock: 1 **] Executed...

[__ PC_update_control __] NPC updated by stage 0 to 1032
[ Stage0 ] PC to fetch = 1028, Instruction = 0
[ Stage2 ] NOP
[ Stage3 ] NOP
[ Stage4 ] NOP
[__ PC_update_control __] NPC updated by stage 1 to 2132
[ Stage1:UpdatePC ] updated NPC with absolute address
[ Stage1 ] J (jump) to absolute address 2132
[** Clock: 2 **] Executed...
[** Clock: 2 **] Flushing stage 0

[__ PC_update_control __] NPC updated by stage 0 to 2136
[ Stage0 ] PC to fetch = 2132, Instruction = 119859208
[ Stage1 ] NOP
[ Stage2 ] J idle
[ Stage3 ] NOP
[ Stage4 ] NOP
[** Clock: 3 **] Executed...

[__ PC_update_control __] NPC updated by stage 0 to 2140
[ Stage0 ] PC to fetch = 2136, Instruction = 59407
[ Stage1:RegisterFetch ] Register $zero is always 0
[ Stage1:RegisterFetch ] A = 0
[ Stage1 ] ADDI r0 + imm 1828 -> r29
[ Stage2 ] NOP
[ Stage3 ] J idle
[ Stage4 ] NOP
[** Clock: 4 **] Executed...
```

<< clock cycle outputs 5 to 986 have been omitted >>

```
[__ PC_update_control __] NPC updated by stage 0 to 2212
[ Stage0 ] PC to fetch = 2208, Instruction = 40386
[ Stage2 ] NOP
[ Stage3 ] JR idle
[ Stage4:RegisterWrite ] writing value 1812 to register r29
[ Stage4 ] ADDI stored 1812 into r29
[ Stage1 ] NOP
[** Clock: 987 **] Executed...

[__ PC_update_control __] NPC updated by stage 0 to 2216
[ Stage0 ] PC to fetch = 2212, Instruction = -463032
[ Stage2 ] NOP
[ Stage3 ] NOP
[ Stage4 ] JR idle
[__ PC_update_control __] NPC updated by stage 1 to 2524
[ Stage1:UpdatePC ] updated NPC with absolute address
[ Stage1 ] J (jump) to absolute address 2524
[** Clock: 988 **] Executed...
[** Clock: 988 **] Flushing stage 0

[__ PC_update_control __] NPC updated by stage 0 to 2528
[ Stage0 ] PC to fetch = 2524, Instruction = 0
[ Stage2 ] J idle
```

```
[ Stage3 ] NOP
[ Stage4 ] NOP
[ Stage1 ] NOP
[** Clock: 989 **] Executed...

[__ PC_update_control __] NPC updated by stage 0 to 2532
[ Stage0 ] PC to fetch = 2528, Instruction = 0
[ Stage2 ] NOP
[ Stage3 ] J idle
[ Stage4 ] NOP
[ Stage1 ] NOP
[** Clock: 990 **] Executed...

mips > s

dataCache Statistics :
[ SimpleCache::Statistics ] Statistics for the 1-level
       DATA cache are displayed
Number of Blocks : 16
Words per block : 4
Associativity : 2
Number of sets : 8
Total Cache Size in Bytes : 256

Total Number of reads : 129
Total Number of read Hits : 107
Total Number of read Misses : 22
Read Hit Ratio : 0.829457

Total Number of writes : 55
Total Number of write Hits : 46
Total Number of write Misses : 9
Write Hit Ratio : 0.836364

Total Number of accesses : 184
Total Number of Hits : 153
Total Number of Misses : 31
Overall Hit Ratio : 0.831522
[ SimpleCache::Statistics ] Statistics for the 2-level
       DATA cache are displayed
Number of Blocks : 128
Words per block : 32
Associativity : 4
Number of sets : 32
Total Cache Size in Bytes : 16384

Total Number of reads : 124
Total Number of read Hits : 120
Total Number of read Misses : 4
Read Hit Ratio : 0.967742

Total Number of writes : 8
Total Number of write Hits : 8
Total Number of write Misses : 0
Write Hit Ratio : 1

Total Number of accesses : 132
Total Number of Hits : 128
Total Number of Misses : 4
Overall Hit Ratio : 0.969697
instrCache Statistics :
[ SimpleCache::Statistics ] Statistics for the 1-level
       INSTRUCTION cache are displayed
Number of Blocks : 64
Words per block : 16
Associativity : 4
Number of sets : 16
Total Cache Size in Bytes : 4096
```

```
Total Number of reads : 990
Total Number of read Hits : 982
Total Number of read Misses : 8
Read Hit Ratio : 0.991919

Total Number of writes : 0
Total Number of write Hits : 0
Total Number of write Misses : 0
Write Hit Ratio : 0

Total Number of accesses : 990
Total Number of Hits : 982
Total Number of Misses : 8
Overall Hit Ratio : 0.991919
mips > q

[__ PC_update_control __] NPC updated by stage 0 to 2536
[ Stage0 ] PC to fetch = 2532, Instruction = 0
[ Stage2 ] NOP
[ Stage3 ] NOP
[ Stage4 ] J idle
[ Stage1 ] NOP
```

The output from dumbterminal

```
[miniproject@localhost test]$ ./dumbterminal

Input Thread made contact !!

Output Thread made contact !!

Enter the number whose factorial is to be found : 12

The factorial of 12 is 479001600
```

[Bibliography]

[1] John L. Hennessy, and David A. Patterson, "Computer Architecture: A Quantitative Approach," 3rd ed., Morgan Kaufmann Publishers Inc. 2004.

[2] David A. Patterson, and John L. Hennessy, "Computer Organisation & Design: The Hardware / Software Interface," 2nd ed., Harcourt (India) Private Limited / Morgan Kaufmann Publishers Inc. 2002.

[3] Terrence Chan, "UNIX System Programming Using C++," Prentice-Hall of India Private Limited, 1999.

[4] Danny Kalev, "ANSI/ISO C++ Professional Programmer's Handbook," Prentice-Hall of India Private Limited, 2000.

[5] Rebecca Thomas, Lawrence R. Rogers, and Jean L. Yates, "Advanced Programmer's Guide to UNIX System V," McGraw Hill International Edition, Computer Science Series.

[6] Barry B. Brey, "The Intel Microprocessors: Architecture, Programming, and Interfacing," 6th ed., Pearson Education, 2003.