ABSTRACT

COMPUTER SCIENCE

FOWLER, CHANTICE B.S., SPELMAN COLLEGE, 1992
M.S., CLARK ATLANTA UNIVERSITY, 1994

THE COMPUTER SCIENCE GRADUATE RECORD EXAM TUTORIAL COURSEWARE

Advisor: Dr. Kenneth Perry

Thesis dated May, 1994

The design and development of Computer Science Graduate Record Examination Tutorial Software will be discussed. The courseware reviews Computer Design, File Structures, Data Structures, and Discrete Math to thoroughly prepare students for the exam. A demonstration of the software is included on diskette.

THE COMPUTER SCIENCE GRADUATE RECORD EXAM

TUTORIAL COURSEWARE

A THESIS

SUBMITTED TO THE FACULTY OF CLARK ATLANTA UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

BY

CHANTICE M. FOWLER

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE

ATLANTA, GA

MAY 1994

R= viii T= 103

ACKNOWLEDGEMENTS

This entire tutorial courseware project and thesis would not be possible had it not been for the encouragement, support, dedication, patience, insight, and expertise of my advisor, Dr. Kenneth Perry. To him I say, "Thank You!!" The Atlanta University Center's Computer Science & Engineering Programs would definitely reign as the best in the country if all the professors were as nearly dedicated and hardworking as he. Clark Atlanta University did not make any mistakes when they gave him tenure.

Let me acknowledge Clayton Collie who contributed the report on File Structures. In addition, Byron Roberson, Janice Barlow, Katina McKinney, and Courtney Smith programmed the cache demonstration used within the CS-GRE Tutorial Courseware.

Special thanks are given to the National Science Foundation for funding my entire graduate studies education. This project would not have been possible had it not been for their financial support.

Finally, let me thank Clark Atlanta University for being one exceptional university.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF ILLUSTRATIONS vi

LIST OF TABLES viii

CHAPTER 1

INTRODUCTION 1

CHAPTER 2

STUDENT COMPUTER SCIENCE

GRADUATE RECORD EXAM TEST RESULTS 4

CHAPTER 3

PROGRAM DESIGN 17

Main Menu 19

A. Computer Design 25

A.1. Introduction 25

A.2. IAS Computer 26

A.3. Modern Computers 26

A.4. CPU 29

A.5. Memory Systems 33

A.6. Microinstruction Sequencing 55

A.7. Boolean Algebra (Switching Algebra) . . 56

B. File Structures 58

B.1. Sequential File Organization 58

B.2. Random File Organization 58

B.3. Indexed Sequential File Organization . 59

B.4. Multikey File Organization 59

B.5. Blocking 60

B.6. Buffering 61

B.7. File Storage Devices 61

B.8. Timing of Access Methods 62

B.9. Record Keys 64

B.10. Hashing 64

B.11. Trees 68

C. Data Structures 71

C.1. Array 71

C.2. Linked List 71

C.3. Stacks 72

C.4. Queues 73

C.5. Trees 74

C.6. Measuring a Program's Performance . . . 75

C.7. Algorithm Characteristics 76

C.8. Allocation of Storage Space 77

C.9. Program Performance Improvements . . . 77

C.10. An Explanation of the Questions & Answers 78

D. Discrete Mathematics (Set Theory) 80

D.1. Logic, Induction, & Recursion 80

D.2. Propositional Logic 86

D.3. Predicate Logic 88

E. Summary 88

CHAPTER 4

CONCLUSIONS & FUTURE WORK 90

APPENDIX A

THE CS-GRE TUTORIAL COURSEWARE 102

SELECTED BIBLIOGRAPHY 103

LIST OF ILLUSTRATIONS

Figure Page

2-1. Computer Science GRE Evaluation Results for All Students. 9

2-2. Computer Science GRE Evaluation Results for CAU Undergraduate Students. 10

2-3. Computer Science GRE Evaluation Results for CAU Graduate Students 11

2-4. Computer Science GRE Evaluation Results for CAU Graduate Students from CAU. 12

2-5. Computer Science GRE Evaluation Results for CAU Graduate Students from Other Schools 13

2-6. Structure Diagram 15

3-1. CS-GRE Tutorial Courseware Main Menu Screen . . 19

3-2. CS-GRE Tutorial Courseware Computer Design Screen 20

3-3. CS-GRE Tutorial Courseware File Structures Screen 21

3-4. CS-GRE Tutorial Courseware Data Structures Screen 22

3-5. CS-GRE Tutorial Courseware Discrete Math Screen 23

3-6. CS-GRE Tutorial Courseware Details Screen . . . 24

3-7. The Original von Neumann Machine. 27

3-8. Basic Computer Organization 28

3-9. Memory Hierarchy. 34

3-10. Memory Represented By Array M of Size N 38

3-11. The Moving of Pages Between Auxiliary & Primary Memory. 40

3-12. The CS-GRE Tutorial Courseware Diagram 1. . . . 43

3-13. The CS-GRE Tutorial Courseware Diagram 2. . . . 44

3-14. The CS-GRE Tutorial Courseware Diagram 3. . . . 47

3-15. The CS-GRE Tutorial Courseware Diagram 4. . . . 49

3-16. The CS-GRE Tutorial Courseware Diagram 5. . . . 51

4-1. CS-GRE Tutorial Courseware Menu Selections. . . 92

4-2. CS-GRE Tutorial Courseware Computer Design Selections. 93

4-3. CS-GRE Tutorial Courseware Introduction Selections. 94

4-4. CS-GRE Tutorial Courseware Introduction Selections. 95

4-5. CS-GRE Tutorial Courseware Introduction Selections. 98

LIST OF TABLES

Table Page

2-1. Computer Science GRE Evaluation Results 6

2-1. Computer Science GRE Evaluation Results, cont'd. 7

CHAPTER 1

INTRODUCTION

This thesis presents the design and implementation of interactive courseware for the Computer Science Graduate Record Examination (CS-GRE). The motivation for the courseware arose after Dr. Kenneth Perry administered the Computer Science GRE to Clark Atlanta University (CAU) students. They did not perform as well as expected. The test results are in Chapter 2. The software provides tutorial information on four subject areas: Computer Design, File Structures, Data Structures, and Discrete Mathematics. Initial feedback indicates the CS-GRE tutorial using interactive multi-media modalities to present many basic computer science concepts does help in preparing students for the GRE Computer Science Exam.

The tutorial courseware began as a group project. We tested 17 CAU students on a previously given Computer Science GRE for questions numbered 1 through 39; and, the students did not perform very well. A group of students who needed to prepare for the actual GRE Computer Science Exam realized there was no commercially-available tutorial software for this particular exam. Amazing, since there are other software

packages on the market for other GRE subject exams. After all, computer science is the discipline upon which software originates.

Consequently, this thesis was initiated after one student realized the need for the tutorial courseware based on the CAU students test scores and decided to make one. After analyzing previous CS-GRE exams and several meetings with my advisor, Dr. Kenneth Perry, we decided on the computer science subjects to be included in the package: Data Structures & Algorithm Analysis, Computer Organization, File Structures, Operating Systems & Computer Architecture, Discrete Mathematics, and Pascal. A large percentage of the Computer Science GRE questions cover material taught within these particular subjects. I prepared in-depth summaries of many important concepts reviewed within each course. Clayton Collie, another Clark Atlanta University (CAU) graduate student, contributed the report on File Structures.

I designed the courseware structure and programmed the computer interface using an event-driven software package, Microsoft Visual Basic, which is an application programmers interface language for Microsoft Windows environments. (An executable copy of the software is included in Appendix A.) The software is currently available to students in the CIS Computer Lab. One objective of the tutorial courseware is to target the largest install base which is the personal computer market and provide the software at an affordable price for

students.

Chapter 2 presents and analyzes the Computer Science GRE score results when 17 CAU students took the sample exam. Chapter 3 introduces the design philosophy used to develop this tutorial courseware, gives a detailed description of the software design, and discusses its features. Chapter 4 provides a list of potential future enhancements for the next version and concludes the tutorial courseware thesis.

CHAPTER 2

STUDENT COMPUTER SCIENCE GRADUATE RECORD EXAM TEST RESULTS

This chapter provides an extensive discussion of the CAU student CS-GRE test results. Based on the results, the CS-GRE tutorial courseware was created to increase our students' test scores. The courseware reviews information most often asked within the Computer Science Graduate Record Examination.

An actual Computer Science Graduate Record Examination (GRE) was administered to 17 Clark Atlanta University (CAU) students in the spring of 1993. Dr. Kenneth Perry, a CAU Computer Science professor, officiated the exam. There is a total number of 80 questions within the Computer Science GRE, and the exam is given over a 4 hour time period. However, the CAU students were only required to answer questions 1-39 due to the time factor of 2 hours. The CAU students comprised both graduates and undergraduates.

The graduate students represented a diverse group in terms of their different ages, interests, occupations, work experience, undergraduate majors, and undergraduate colleges. Most of the graduate students came from other schools. The undergraduate colleges they attended were among Tennessee

State University, Spelman College, Rensselaer, and Clark Atlanta University. Their undergraduate majors varied from Computer Science and Management Information Systems to Physics. Their current graduate grade point averages had to be above 3.0 since they remained in excellent standing with CAU's Computer & Information Systems Department. The range of ages is from 22 years to 37 years.

There were two graduate students who attended CAU as undergraduates. Their undergraduate majors were Physics and Computer Science. Their undergraduate college grade point averages were 3.0 and 3.9, respectively. Both students earned high school grade point averages above 3.5.

The undergraduates also represented a diverse group. Several students were Computer Information Systems majors, while others were Computer Science majors. Their classifications were either junior or senior. Their current college grade point averages ranged from 2.77 to 3.75. In reference to their high school statistics, their grade point averages ranged from 2.7 to 3.8. Their SAT scores ranged from 790 to 1080.

Given the diverse group of students taking the exam, the results were very revealing. The evaluation results may be viewed in Table 2-1. The rows list the number of the specific exam questions from 1 to 39. The columns give the assigned student identification numbers from 1 to 17. The average of all the students' scores was approximately 38% correct, 34%

Table 2-1.
Computer Science GRE Evaluation Results

Assigned Student ID Number

Question Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	R	R	R	R	R	W	R	R	R	R	R	R	R	R	R	R	R
2	W	W	W	R	N	N	N	R	W	R	R	R	R	R	R	R	R
3	W	N	W	N	W	W	R	W	W	R	W	W	R	R	W	W	R
4	R	W	R	N	W	W	W	R	W	R	N	R	R	R	R	W	R
5	W	R	W	R	W	W	R	R	R	W	W	R	R	R	R	R	R
6	W	W	W	W	W	R	W	R	W	W	R	W	R	W	R	W	R
7	R	R	W	W	W	W	R	R	W	R	R	W	W	R	R	R	R
8	R	R	W	R	R	W	N	R	N	W	W	W	W	W	W	N	R
9	W	W	W	N	W	W	N	R	W	R	R	W	R	R	R	W	W
10	N	R	W	N	R	N	N	R	R	R	W	W	N	N	W	W	R
11	R	W	W	N	R	R	W	W	R	R	R	W	R	R	R	R	R
12	N	N	W	N	N	N	W	W	W	N	N	N	N	N	N	N	N
13	W	W	W	R	N	W	W	R	W	R	W	W	R	W	W	W	N
14	N	N	W	N	W	W	N	R	N	N	N	W	R	N	R	R	R
15	W	W	W	W	N	N	W	R	W	N	N	R	R	R	W	R	R
16	W	R	W	R	N	W	R	R	W	R	R	R	R	R	R	R	R
17	R	R	R	W	R	W	R	R	R	R	R	R	R	R	R	R	R
18	R	R	N	R	R	N	N	W	R	N	W	R	R	R	R	R	N
19	N	W	W	N	W	W	W	R	W	N	N	W	W	N	W	N	R
20	W	N	W	N	N	W	W	R	W	N	N	W	R	R	W	R	R

R = Correct **W = Incorrect** **N = Never Seen**

Table 2-1.
Computer Science GRE
Evaluation Results, cont'd.

Assigned Student ID Number

Question Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
21	N	N	W	N	W	W	W	W	W	N	N	N	W	N	W	W	R
22	N	R	W	N	N	N	N	R	W	W	N	W	W	W	W	W	R
23	N	N	W	N	R	W	W	R	W	R	N	W	R	N	R	R	R
24	N	R	R	N	W	N	W	W	R	N	N	R	R	W	W	W	R
25	W	W	R	W	R	W	R	R	R	N	N	W	N	W	R	R	R
26	N	W	W	N	W	N	W	R	W	R	W	W	N	N	W	W	R
27	N	N	W	N	N	N	N	W	W	N	N	N	W	N	N	W	R
28	R	R	W	R	R	R	W	R	R	R	W	R	R	N	R	R	R
29	R	N	W	R	N	N	N	R	R	N	N	W	R	N	N	W	R
30	N	N	W	R	W	N	N	R	W	R	N	N	W	N	W	W	R
31	R	R	R	R	N	R	N	R	W	R	W	W	R	W	W	W	R
32	N	N	R	N	N	N	N	R	N	N	W	W	R	W	W	W	N
33	R	N	N	W	W	W	W	R	W	W	W	W	R	N	W	W	N
34	R	N	N	W	W	W	W	R	W	W	W	R	R	N	R	N	R
35	N	N	W	R	W	N	N	R	W	R	N	N	R	R	W	W	R
36	N	W	W	N	N	N	N	R	W	R	R	W	R	W	W	W	R
37	W	W	W	R	N	N	N	R	W	R	N	R	N	N	N	W	N
38	N	N	N	N	R	N	W	R	R	N	N	N	N	N	N	N	N
39	R	N	N	N	N	N	W	R	R	N	N	W	W	W	N	N	R

R = Correct
Total R = 248

W = Incorrect
Total W = 226

N = Never Seen
Total N = 188

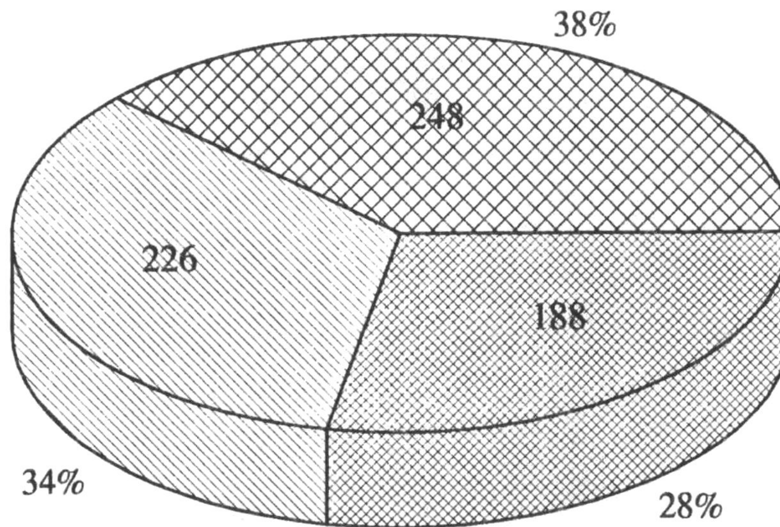
incorrect, and 28% of the material had never been seen before.

To be exact, 248 answers were correct, 226 were incorrect, and 188 were unknown meaning the material was unfamiliar. Please view Figure 2-1 for a table representation of the approximate percentages. These figures are rather surprising and disappointing.

The CAU undergraduates answered 156 correct, 164 incorrect, and 148 unknown. The respective percentages of 33%, 35%, and 32% may be seen in Figure 2-2. The graduates did perform better than the undergraduates. Figure 2-3 lists the graduate students' performance as a whole. The five graduates together scored 100 correct, 71 incorrect, and 33 unknown. The respective percentages are 49%, 35%, 16%. The two graduate students who attended CAU as undergraduates scored 48 correct, 20 incorrect, and 15 unknown. The percentages for these scores are 58%, 24%, and 18%, respectively, and are shown in Figure 2-4. The three graduates who attended colleges other than CAU scored 52 correct, 51 incorrect, and 18 unknown. Figure 2-5 gives the percentages for these scores of 43% correct, 42% incorrect, and 15% unknown.

These scores prove our students as a whole do not perform very well on the Computer Science GRE. As a result of this discovery, something has to be done to improve the scores of our students, and I am the person who wanted to make a difference to raise our test scores and performance.

Figure 2-1.
Computer Science GRE
Evaluation Results
for All Students



Correct # Incorrect # Never Seen

**Figure 2-2.
Computer Science GRE
Evaluation Results
for CAU Undergraduate Students**

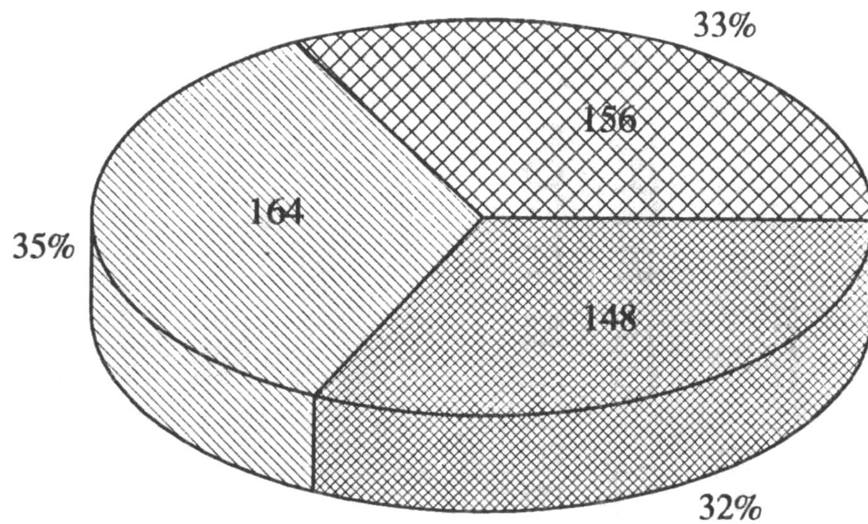


Figure 2-3.
Computer Science GRE
Evaluation Results
for CAU Graduate Students

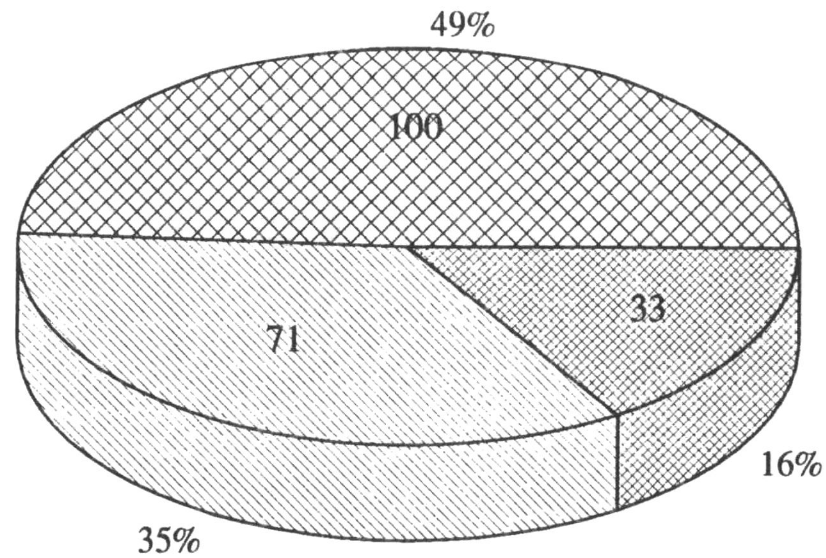


Figure 2-4.
Computer Science GRE
Evaluation Results
for CAU Graduate Students from CAU

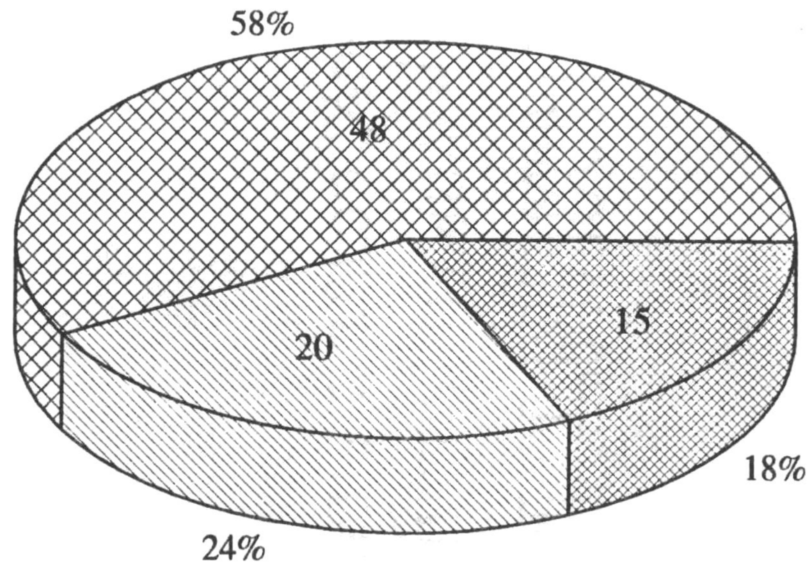
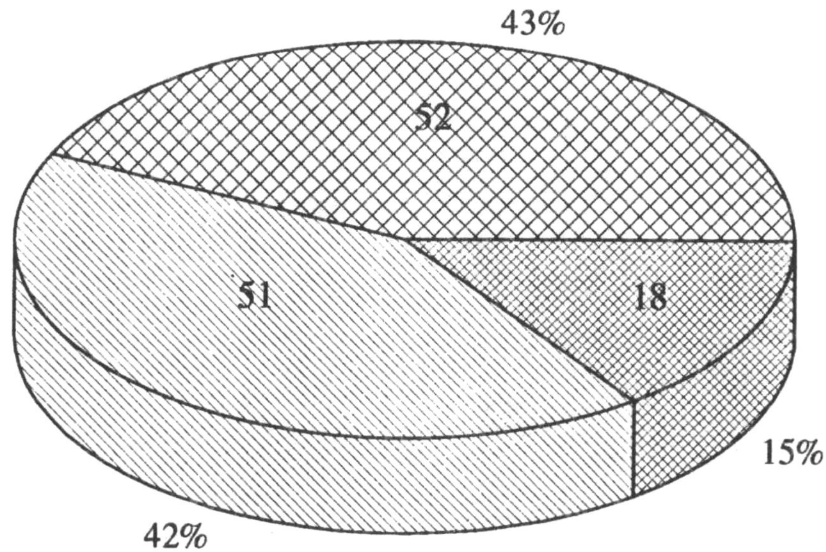


Figure 2-5.
Computer Science GRE
Evaluation Results
for CAU Graduate Students from Other Schools



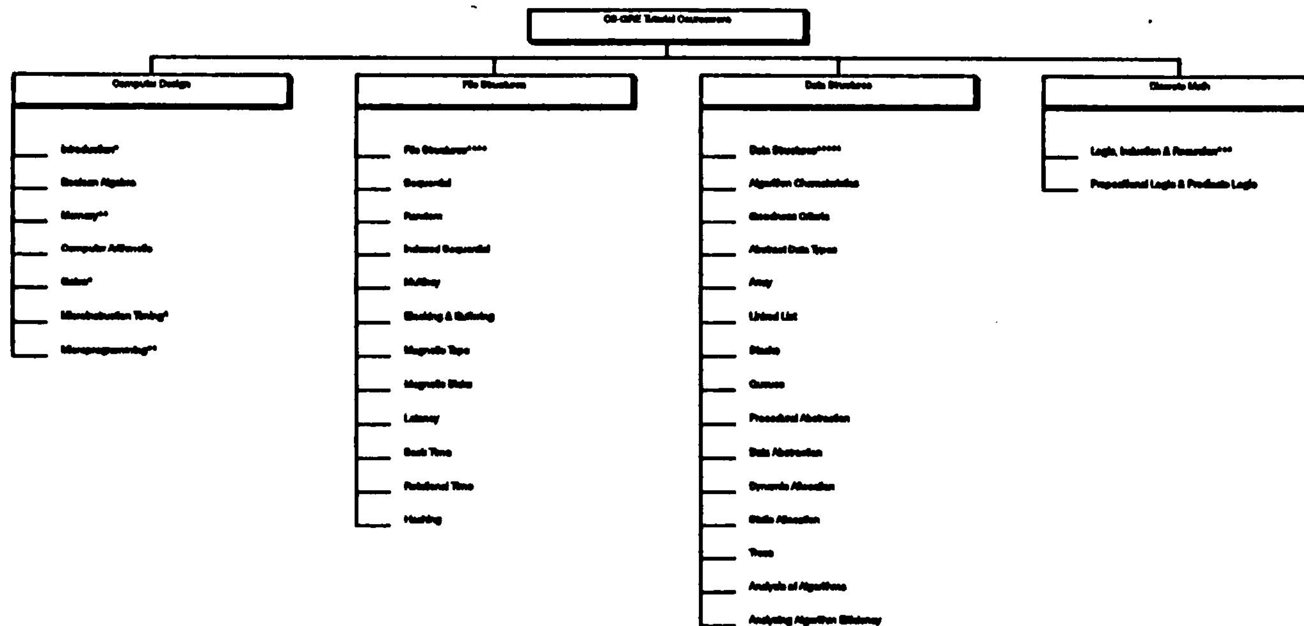
☒ # Correct ▧ # Incorrect ▩ # Never Seen

The poor performance may be attributed to the exam covering material which is not being reviewed thoroughly within a classroom environment. The incorrect answers and the unfamiliar material are evidence the exam analyzes material which is either difficult or foreign to the students. In order for the students to perform well, this unfamiliar and difficult material must be thoroughly reviewed during preparation for the Computer Science GRE. The CS-GRE tutorial courseware is developed to cover the unfamiliar subject matter and to improve students' scores.

After reviewing the GRE, we identified six courses from which most of the information originates. These six courses include Data Structures & Algorithm Analysis, Computer Organization, File Structures, Operating Systems & Architecture, Pascal, and Discrete Mathematics. Five courses are reviewed in the package within four modules. Pascal is the sixth course not yet included; but, it is being developed as a fifth module. The four modules implemented include Computer Design, File Structures, Data Structures, and Discrete Mathematics.

The structure diagram for the modules may be seen in Figure 2-6. Each topic provides narrative information within a window. One asterisk (*) indicates a diagram accompanies the narrative. Two asterisks (**) indicate animation or a demonstration is shown. Three asterisks (***) show where a future diagram may be included in the next version. Four

Figure 2-6.
CS-GRE Tutorial Courseware Structure Diagram



asterisks (****) indicate where future animation or video may be demonstrated. Five asterisks (*****) show where future sound may be added.

The next chapter on Program Design reviews the material included in the tutorial courseware. All four modules, Computer Design, File Structures, Data Structures, and Discrete Math, are reviewed.

CHAPTER 3

PROGRAM DESIGN

This chapter discusses the design philosophy used in creating the tutorial courseware and gives a detailed description of the software design features. The material presented familiarizes the reader with Computer Organization, Operating Systems & Architecture, File Structures, Data Structures & Algorithm Analysis, and Discrete Mathematics courses taught within a university's computer science degree program.

The CS-GRE tutorial is divided into five main headings: Computer Design, File Structures, Data Structures, Discrete Math, and References. The software is designed using a hierarchical menu structure which includes submenus that can be nested up to six levels deep. This organizational structure within the tutorial software is used to present an overview of computer science in a clear, concise manner with related topics grouped together.

The software provides the user a general overview of computer science. The package lays a basic foundation to prepare a student for the Computer Science GRE Subject Exam

and increase our students' test scores.

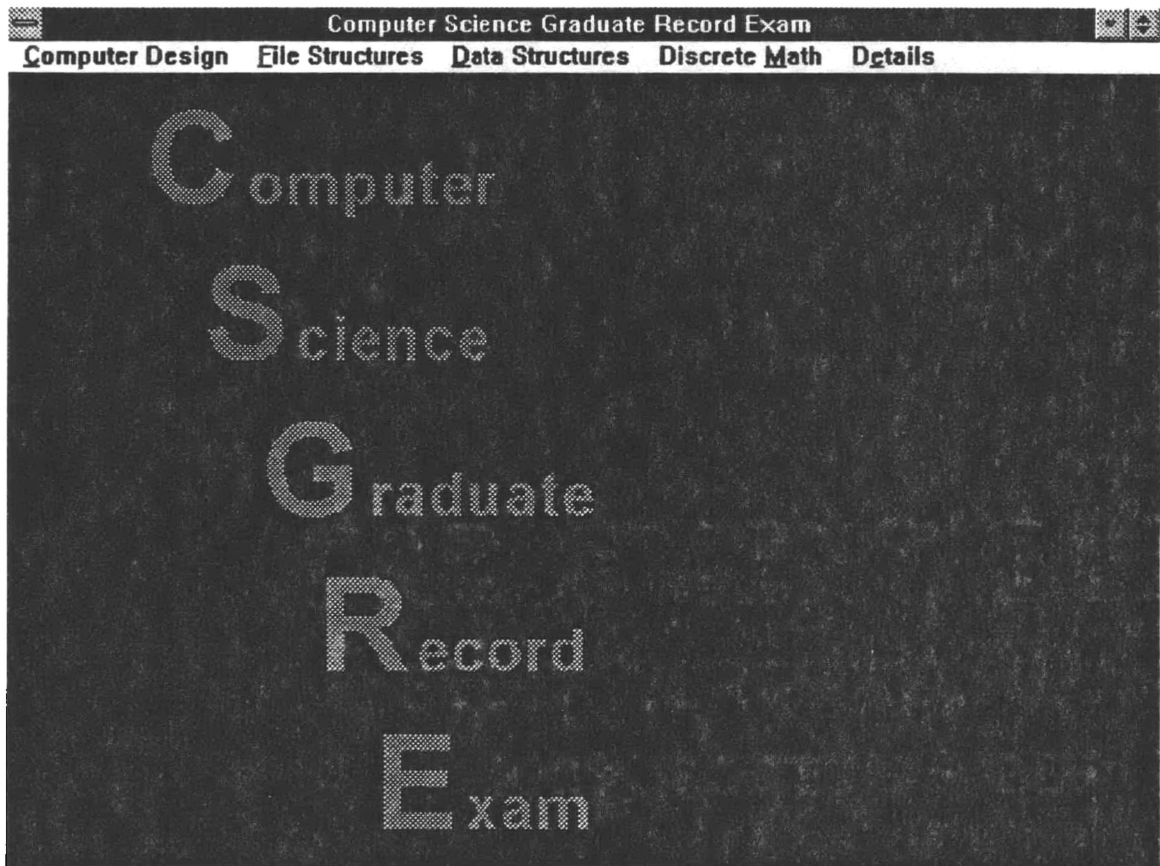
In addition, the tutorial is intuitively designed to be simple and easy to use. The user may use either the mouse or the keyboard to review the tutorial. The menu items' windows may appear by either selecting the desired item with the mouse, or typing in a specific underlined letter from the menu items' title as a hotkey.

Visual Basic was chosen as the application programming tool because 1) our objective is to develop low-cost software for the largest installed base which is the PC market; 2) we want the software to run under Microsoft Windows since it is such a common computing environment; 3) Visual Basic is the fastest and easiest way to develop Windows applications; and 4) Visual Basic is the Applications Programmer Interface Language for all future Windows releases including Windows NT and greater.

The structure of the screens displayed in the tutorial is described.

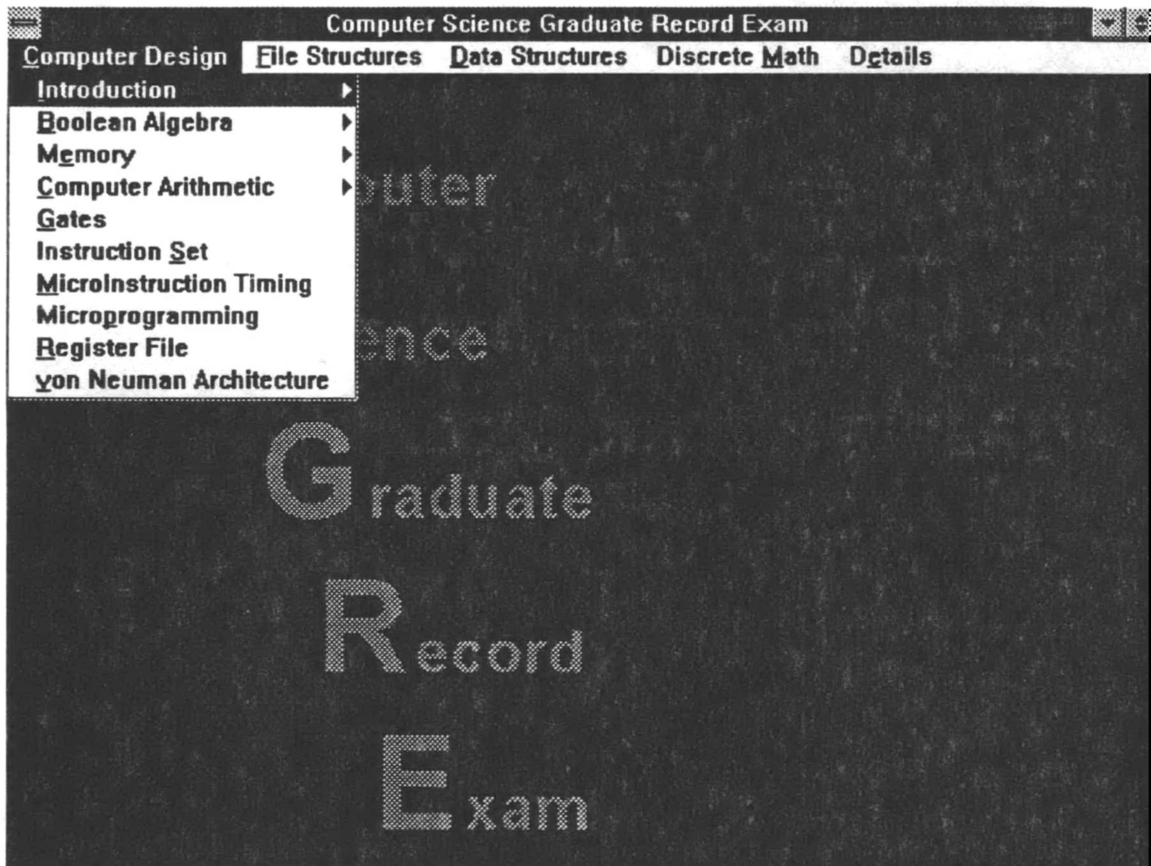
Main Menu

The Main Menu is displayed in Figure 3-1. The menu headings with the CS-GRE background screen are shown.



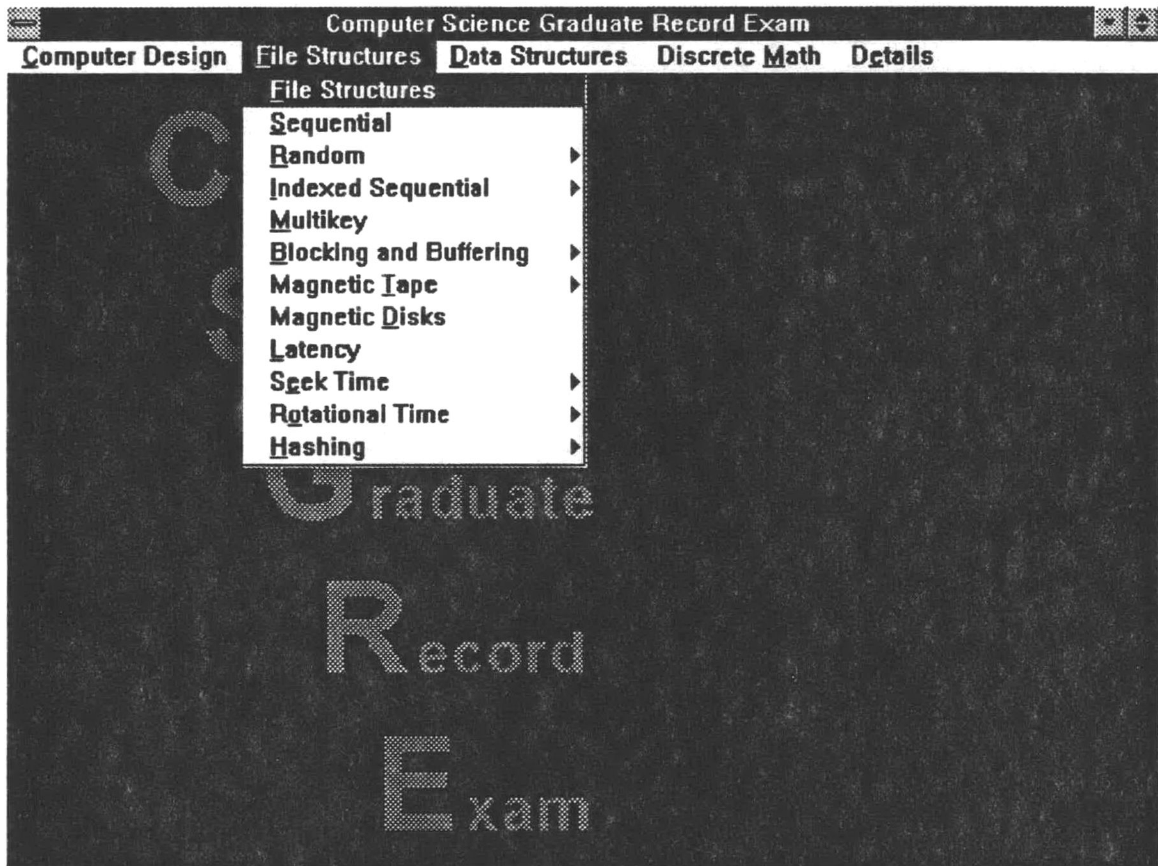
Computer Design

The Computer Design section includes the following menu items as shown in Figure 3-2. Several items are thoroughly discussed within this document.



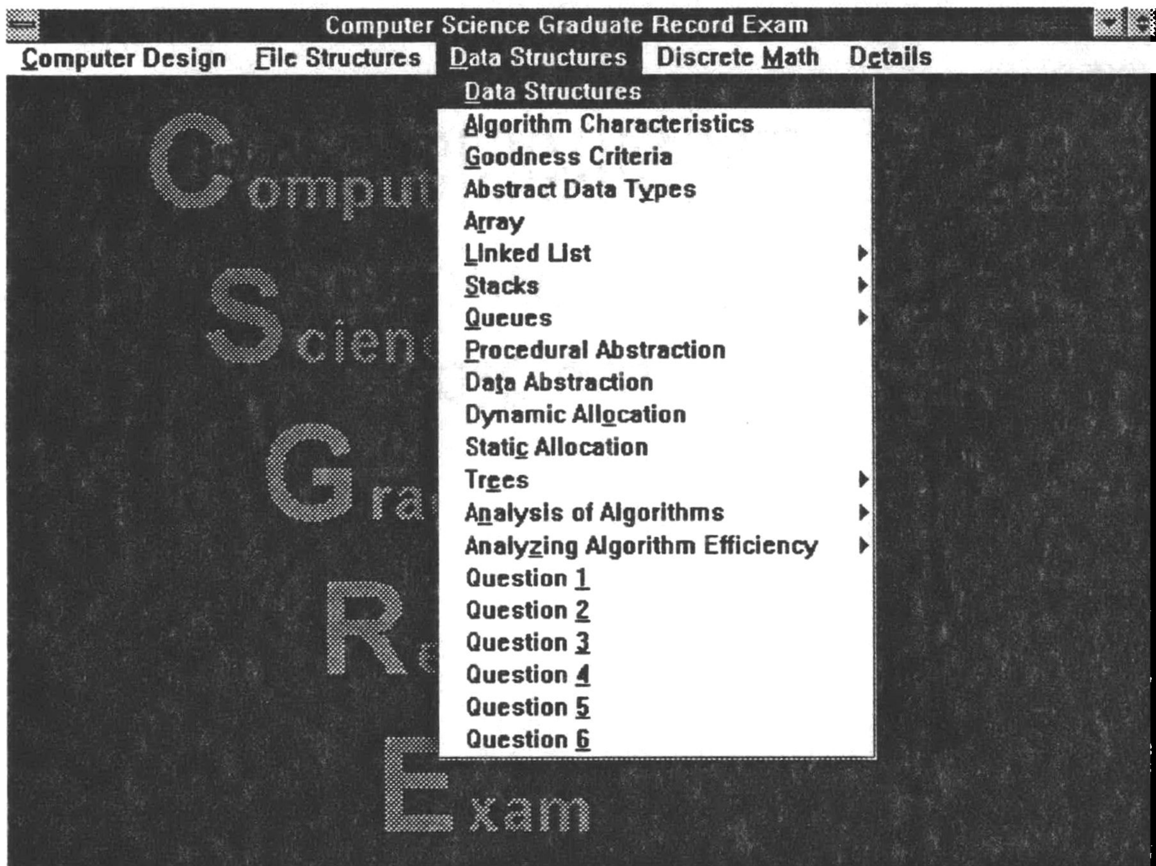
File Structures

The File Structures section is shown in Figure 3-3 and discussed below in section B.



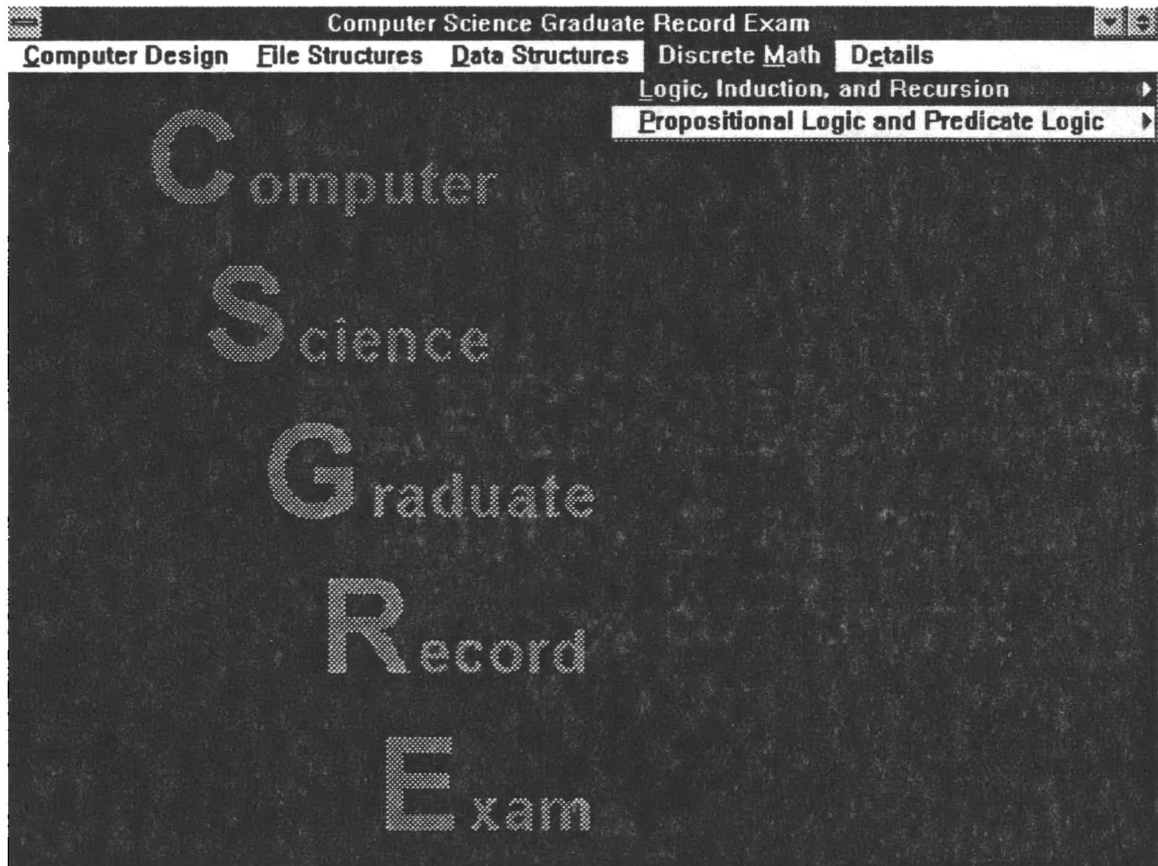
Data Structures

The Data Structures menu in Figure 3-4 illustrates the following topics within the menu that pertain to this subject.



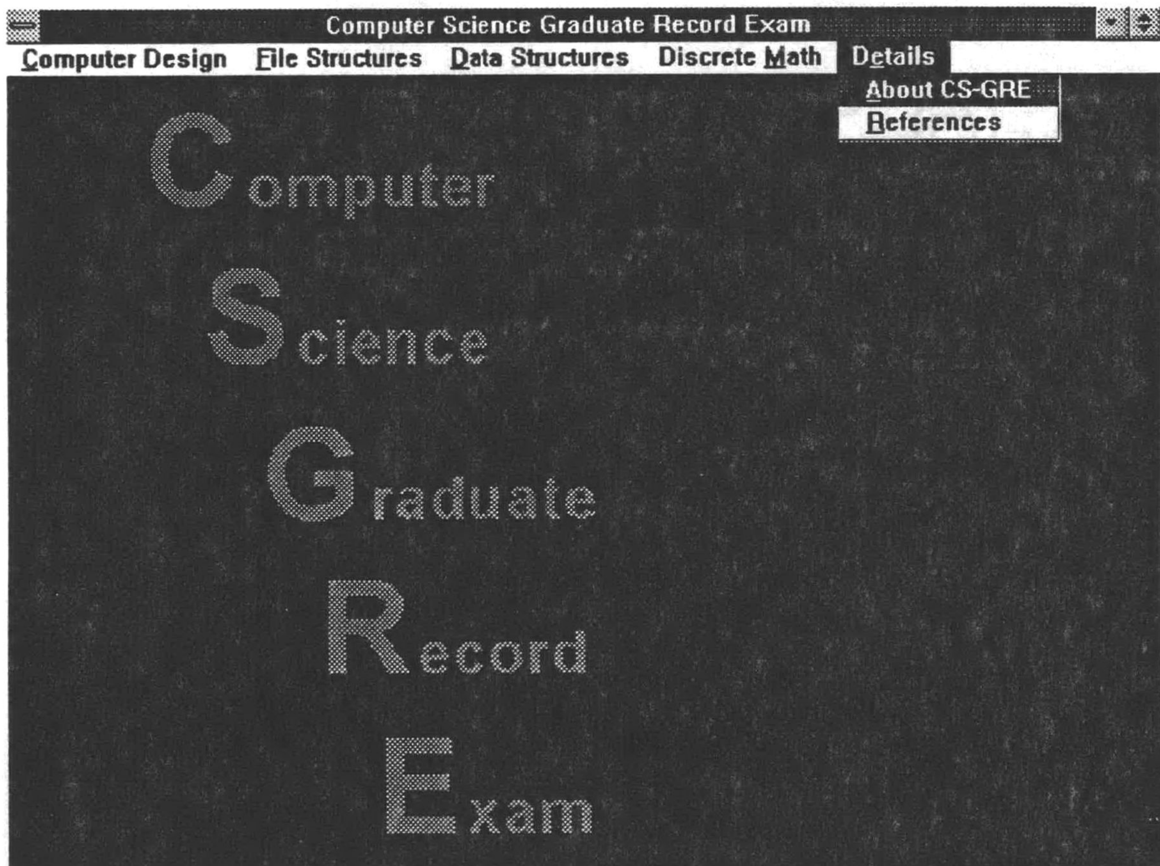
Discrete Math

In Figure 3-5, this Discrete Math menu shows the two subjects into which the topic is divided. Logic, induction, and recursion are discussed in the first topic. Propositional logic and predicate logic represent the second topic.



Details

The Details menu in Figure 3-6 presents the author and references sections. The tutorial author is given in About CS-GRE. The references section lists all actual references used to gather the information in this tutorial.



A. Computer Design

This section presents an overview of the von Neumann machine, modern computers, the central processing unit, memory systems, microinstruction sequencing, and Boolean algebra. The modern computers and memory systems discussion includes diagrams which enable the reader to understand the organization of a computer, several memory types, and different cache schemes. Several questions and answers are given to demonstrate a variety of caching strategies. The discussion gives the reader a general understanding of computer organization and operating system architecture.

A.1. Introduction

The four basic functions of a computer are: data processing, data storage, data movement, and control. A computer includes a set of instructions that when used together as a group allows the user to perform any conceivable data-processing task. This allows the use of computers in many fields whether the user plans to develop applications, perform research, design models, or manipulate records. Therefore, computers include an ample number of instructions to move data to and from memory; perform arithmetic and logic instructions; provide instructions for checking status information which enables the computer to render decision-making capabilities; and, perform input and output operations. General-purpose computers contain high-speed primary memory,

auxiliary memory, a central processing unit (CPU), and an input/output (I/O) system.

A.2. IAS Computer

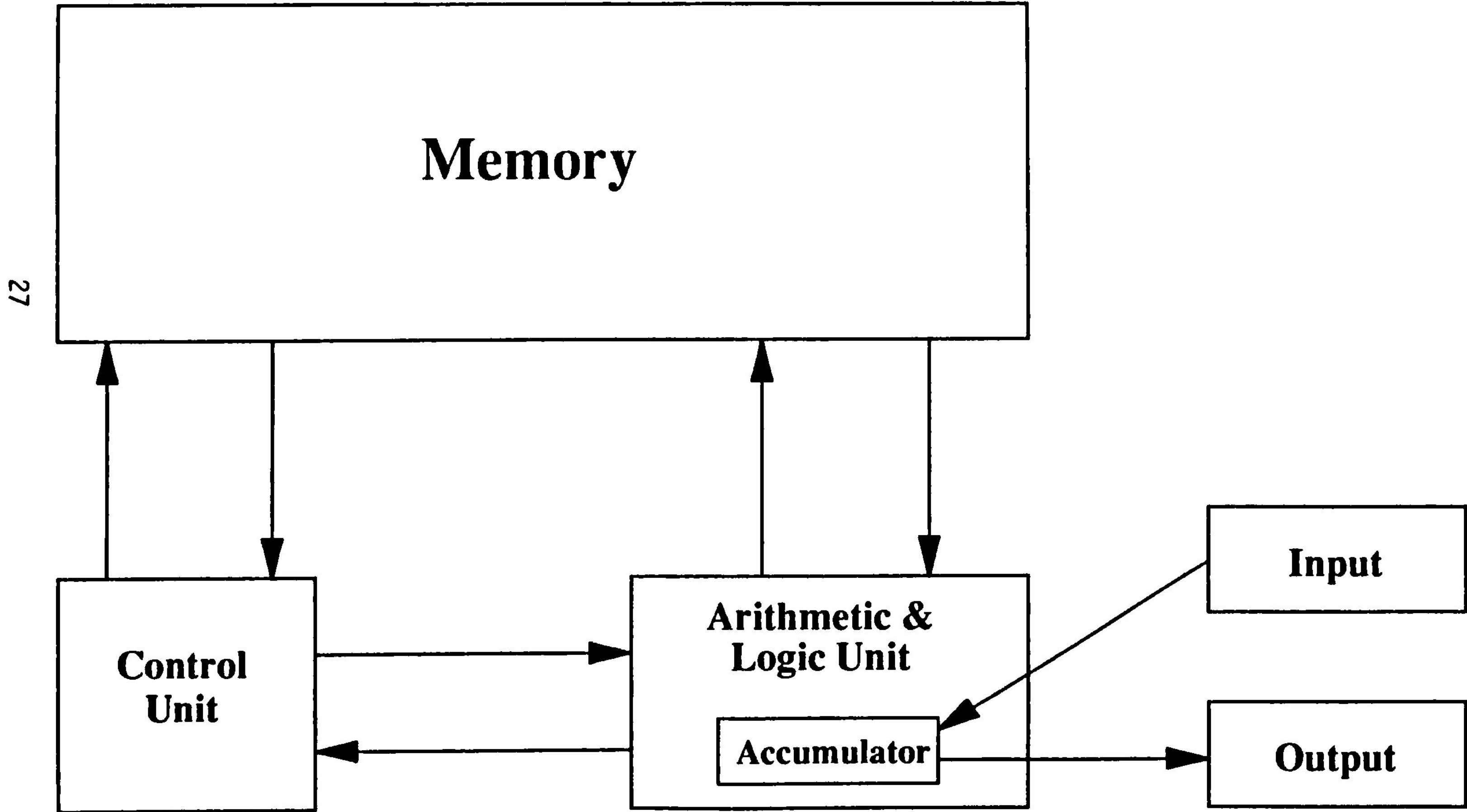
John von Neumann built the Princeton's Institute of Advanced Studies (IAS) computer modeled after the Electronic Discrete Variable Automatic Computer (EDVAC) in the early 1950's. The IAS computer is referred to as a von Neumann machine and is still the basis for digital computers today. An important feature is that the program and data are represented in digital form in the computer's memory instead of the antiquated system of using large numbers of switches and cables to program a computer.

The IAS computer in Figure 3-7 contains five parts as follows: the memory, the ALU, the program control unit, and the input and output equipment. Properties of a von Neumann architecture include the storage of data and instructions in a single read-write memory;and, the contents of this memory are addressable by location, without regard to the type of data contained therein. Execution occurs in a sequential fashion from one instruction to the next, unless explicitly modified.

A.3. Modern Computers

There are four main structural components to a computer: the central processing unit (CPU), main memory, input/output (I/O), and the system interconnections. Figure 3-8 presents a basic computer's organization. The CPU controls the

FIGURE 3-7.
THE ORIGINAL VON NEUMAN MACHINE



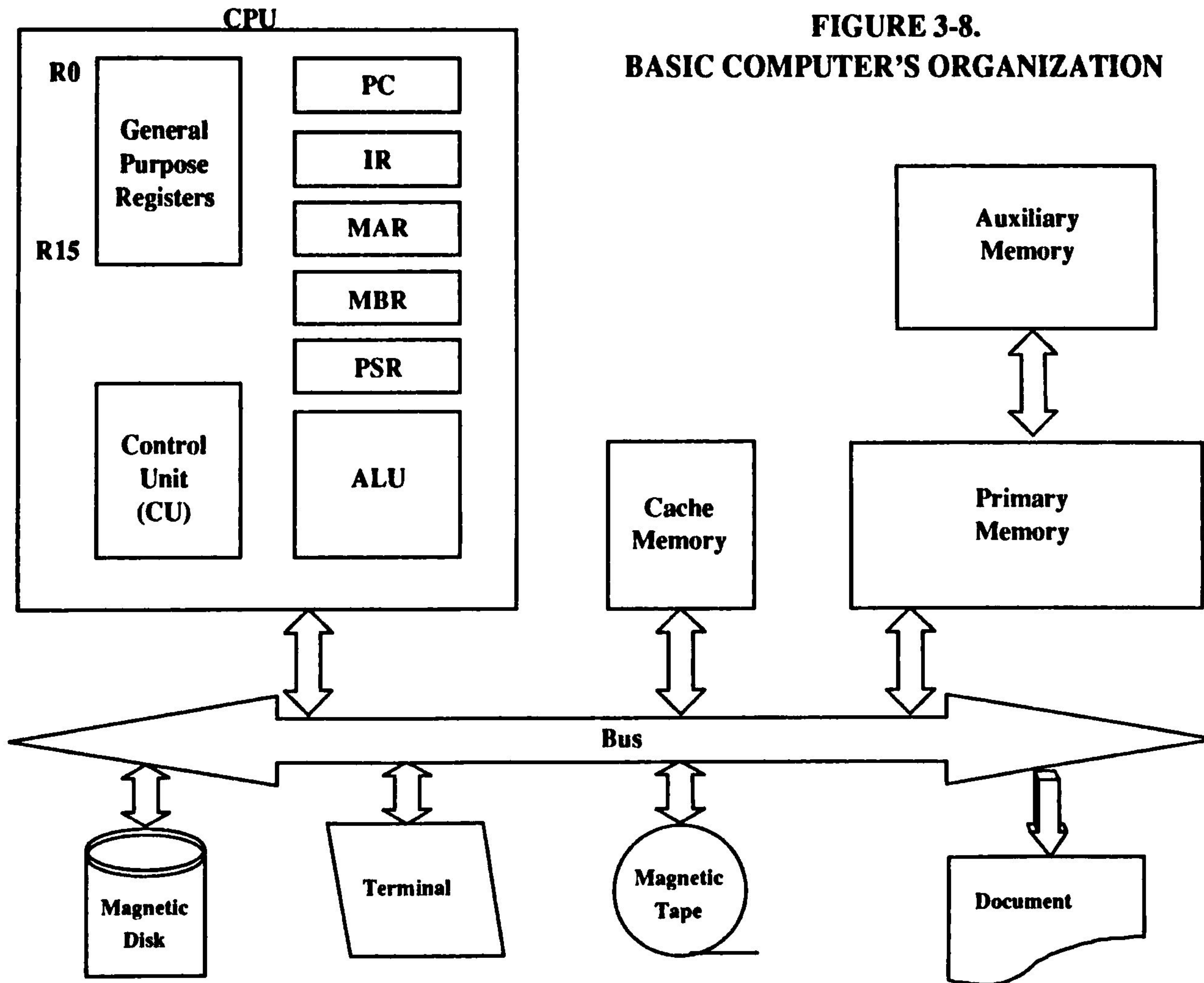


FIGURE 3-8.
BASIC COMPUTER'S ORGANIZATION

operation of the computer and performs its data processing functions. Main memory stores data. Input/output moves data between the computer and its external environment. The system interconnection provides some mechanism which supplies communication among the CPU, main memory, and I/O.

A.4. CPU

The CPU or central processing unit shown in Figure 3-8 is the section of a computer which handles the interpretation and execution of the programming instructions. The CPU consists of four major structural units: the ALU, the control unit, registers, and the CPU interconnections. The arithmetic and logic unit (ALU) performs the computer's data processing functions. The control unit controls the operation of the CPU. Registers provide storage internal to the CPU. CPU interconnections provide communication between the control unit, ALU, and registers. The CPU contains the arithmetic and logical unit (ALU), control unit (CU), instruction register (IR), memory address register (MAR), memory buffer register (MBR), program counter (PC), general purpose registers, buses, and program status register (PSR).

A.4.a. ALU

The ALU contains electronic circuits which receive and operate on the data. The ALU exercises the basic arithmetic operations of addition, subtraction, shift operation,and logical operations, i.e. AND, OR, and COMPLEMENT. The ALU typically consists of a binary adder and binary shifter. The

adder inputs two binary numbers and produces a binary sum for output. The shifter changes the position of the bits either to the left or to the right. The shifter and the adder can be used together to execute multiplication. Complimentary addition is used to perform subtraction. Division is achieved by repeated subtraction and shifting.

A.4.b. Control Unit

The control unit (CU) manages the flow of data between all other devices. It also translates the operations represented by the program instructions. Furthermore, the CU sends signals to the devices which perform these operations. The CU directs both the flow of instructions and data to and from memory. The flow of both the instructions and data to memory is termed the store cycle; and, the flow of both from memory to various devices is defined as the fetch cycle or read cycle.

Before program execution begins, the PC must receive the memory address of the first instruction to be executed. The CU then continues program execution, or rather evaluates instruction execution cycles. An instruction execution cycle begins with the read cycle of an instruction. The cycle ends when a result is obtained. As the instruction is moved or copied from primary memory, the CU interprets the instruction. During this interpretation, the CU transmits the appropriate signals to the various devices necessary for processing the instruction. Also, the CU sends the necessary data to be used

in the instruction to the ALU. Upon completion of the operation, the CU directs the result from the ALU into the appropriate memory cell or processor register.

A.4.b.1. Processor Registers

Instructions for moving data to and from memory are essential. Most information in a computer is stored in memory while the computations are done in the CPU and stored in the processor registers. The system must have the capability of moving data between memory and the processor registers if the system is to be fully effective and useful.

A CPU instruction cycle performs several different operations. The cycle first fetches the next instruction from memory and places it in the instruction register. The program counter changes to hold the address of the next instruction to be executed. The instruction is decoded. Data is fetched from memory if necessary. The instruction is executed, and the results are stored. Then, the CU returns to the first step and repeats the process until all instructions have been executed.

A.4.b.2. IR, MAR, & MBR

The instruction register (IR) is the register within the CPU which holds the current instruction. The memory address register (MAR) holds an address of an instruction or data to be read or written; while, the memory buffer register (MBR) stores the instruction or data.

A.4.c. Program Counter

The program counter (PC) is designated as a special register which contains the address of the next instruction to be executed. As the instruction is retrieved from memory and placed in a special register within the CPU, the PC is updated. The PC is incremented by a number equivalent to the length of the instruction in bytes. The number of bytes required to store the instruction in memory is the length of the instruction.

Since memory is a linear array, the PC normally stores the address of the next sequential instruction to be executed. An exception arises when the current instruction is a branch or transfer of control instruction. In this situation, the CPU discerns the instruction to be executed is not the next sequential instruction. Therefore, the PC is updated to store the branch address for the next instruction to be executed.

A.4.d. General Purpose Registers

General purpose registers supply local, high-speed storage for the processor. Registers function as discrete storage cells of some fixed length. Registers receive, hold, and transfer data as well as store addresses, but their primary use is to hold data temporarily. The advantage to using registers is that data can be processed more quickly than by accessing main memory.

A.4.e. Program Status Registers

Program status registers (PSRs) contain status information which report the current state of the program in

execution. There are one or more PSRs within the CPU. PSRs provide information about the results obtained from instruction execution by using condition codes.

Almost every instruction execution records some information in the condition code. The condition code is just one type of status information. On the basis of this information, the programmer decides which actions to take next. Condition codes supply the following information: the result is zero, the result is negative, the result is positive, the result is out-of-range (overflow), and whether the arithmetic operations use "carry-out" or "borrow-into."

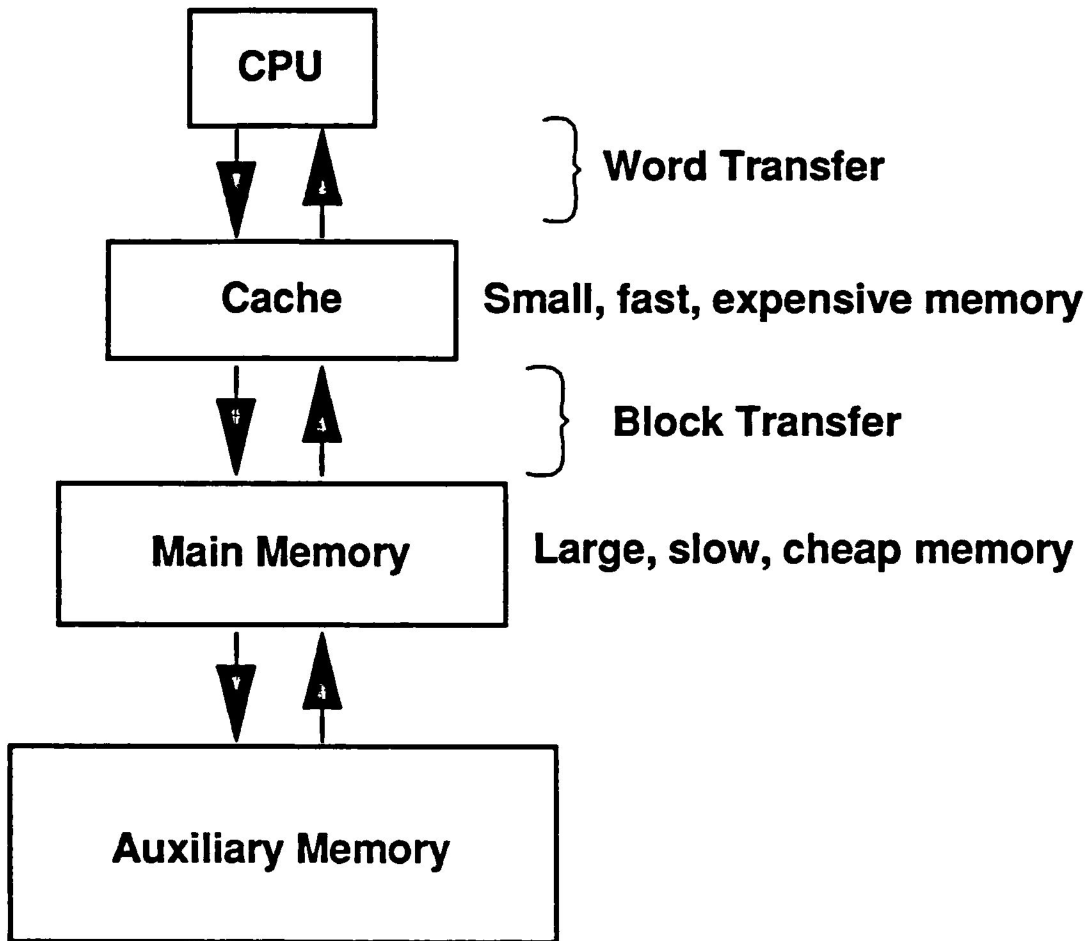
A.5. Memory Systems

This section presents an overview of memory systems including memory hierarchy, locality of reference, address space, primary memory, auxiliary memory, virtual memory, and cache memory. Cache memory includes a discussion on associative cache, direct-mapped cache, and set-associative cache. Five questions are given which represent several caching strategies.

A.5.a. Memory Hierarchy

Whereas a virtual memory system shifts data between auxiliary memory and primary memory, cache organization moves data between primary memory and the CPU. Thus, each memory type involves a different level in the memory hierarchy system. Figure 3-9 displays the memory hierarchy. Within the computer memory hierarchy, programs and data are first stored

Figure 3-9.
Memory Hierarchy



in auxiliary memory. The cache holds those most heavily used parts of the program and data, while the auxiliary memory stores those parts the CPU is not presently handling. As a result, the CPU commands direct access to both cache and primary memory but not to auxiliary memory.

A.5.b. Locality of Reference

Locality of reference means references to memory at any given time interval tend to be confined within a few localized memory cells. Locality of reference occurs since a typical computer program flows sequentially from top to bottom, encountering frequent program loops and procedure calls. Upon a program loop's execution, the CPU repeatedly refers to the set of instructions which form the loop. Every time a procedure is called, its own set of instructions must be moved from memory. The main idea is that when a word is referenced, it is brought from the large, slow memory into the cache. Therefore, the next time it is needed, it will be retrieved quickly. Loops and procedures then localize references to memory. References to data stored in primary memory also tend to be localized.

A.5.c. Address Space

During the execution of the program, each address the CPU references undergoes address mapping. This mapping transforms a virtual address into a physical address. The virtual address only demonstrates meaning in reference to the page's domain, whereas the physical address functions as the memory

cell's actual address. A programmer evaluates a virtual address. The set of all virtual addresses completes the address space. An address in primary memory designates a location or physical address. The entire set of such locations specifies memory space. Therefore, the address space is the set of addresses programs generate as they reference instructions and data. The memory space is characterized by the actual primary memory locations directly addressable for processing.

A.5.d. Primary Memory

Primary memory or main memory consists of two state components which can represent two measurable, distinct states. The binary numbering system of 0 and 1 is used to represent its two states. The binary digit symbol 0 represents one state, while 1 represents the other. Portions of a program or data are only brought into primary memory as they are needed for program execution. Primary memory performs at high speed and forms two organizational features: 1) each memory cell is the same size and, 2) each cell is uniquely referenced by its very own numeric address. A memory cell contains an address to indicate the cell's relative position in reference to some known position. The contents of the cell represent a numeric value or an alphanumeric character stored in a particular memory cell. These contents may be changed or used in an operation.

A.5.e. Auxiliary Memory

Auxiliary memory devices are used for backup storage. The most common auxiliary memory devices are magnetic disks, magnetic drums, and magnetic tapes. This particular kind of memory does not communicate directly with the CPU. Instead, information is moved to primary memory from which the CPU obtains the necessary instructions and data. Please review a basic computer's organization in Figure 3-8. Only programs and data the processor currently uses reside in primary memory. All other information is stored in auxiliary memory and is then transferred to primary memory on demand.

Figure 3-10 illustrates a memory model where the memory is represented by an array M of size N. The illustration shows the employment of addresses to reference memory cells. Each element within the array pertains to one memory cell. The cell size, N, varies according to the specifications the computer manufacturer chooses. The specifications must meet the needs of the program application for which the computer is to be used. Each index addresses a particular memory cell in array M. Any memory cell may be referenced when the index is switched to a specific value. With this method, the memory cell allows the placement of new data or the copying of old data.

A.5.f. Virtual Memory

The idea of executing a program while only a portion of the program remains in the primary memory is referred to as virtual memory. Virtual memory streamlines memory space since

FIGURE 3-10.
MEMORY REPRESENTED BY ARRAY M OF SIZE N

Memory cell	Address each cell equals the index of the cell	To obtain contents
M0	0	M(0)
M1	1	M(1)
M2	2	M(2)
Mi	i	M(i)
M N-1	N-1	M (N-1)

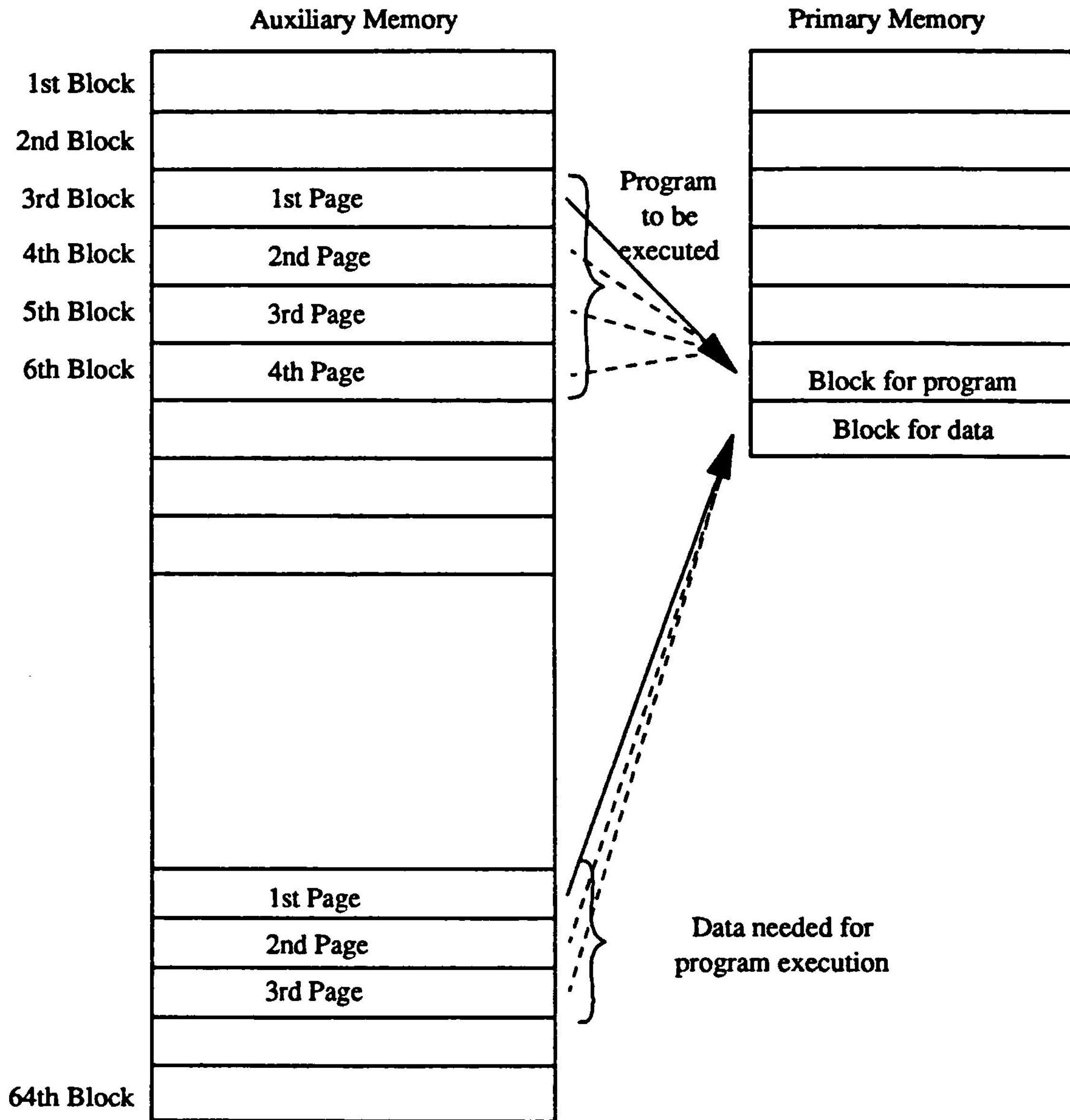
the user constructs a program with memory space equal to the combined sizes of the auxiliary and primary memory. The virtual memory system contains both hardware and software necessary to implement and manage virtual memory.

When a program is submitted for execution, the virtual memory system divides the program into equal parts or pages. Please review Figure 3-11 for the moving of pages between auxiliary and primary memory diagram. The pages are transferred between the primary and auxiliary memories. When program execution begins, the program's first page is moved into the first available primary memory block. A block of memory is a group of contiguous memory cells equivalent to the size of a page. During the program's execution, the virtual system transfers the next page to the same block of the primary memory from the auxiliary memory whenever the program's next page is required. The same method is followed for the program's necessary data. Primary memory contains separate blocks of information for both the program and its data.

A.5.g. Cache Memory

Cache memory is a fast memory where the active portions of the program and data are placed to reduce the average memory access time. As a result, the program's total execution time is reduced. Cache memory is situated between the CPU and the primary memory. Figure 3-8 shows the location of cache memory. This kind of memory possesses an access time

FIGURE 3-11.
THE MOVING OF PAGES BETWEEN AUXILIARY AND PRIMARY MEMORY



less than the primary memory access time by an average factor of five to ten. Leading as the fastest memory device in the memory hierarchy and approaching the speed of CPU devices, the use of cache memory has many advantages.

The fundamental idea of cache organization is to keep the most frequently accessed instructions and data in the fastest memory. Although the cache remains just a small fraction of primary memory's size, many memory requests are transmitted to the fast cache memory due to locality of reference.

The cache is examined whenever the CPU needs to access memory. If the CPU discovers the necessary instructions or data in the cache, the instructions or data are passed to the necessary device which made the request. However, if the instructions are not found, primary memory will then be accessed. A block of instructions or data containing the requested information is now transferred to the cache memory. This process ensures most future references to memory will find the required information within the fast cache memory. The hit ratio is determined by the percentage of hits of the memory references finding the requested information within the cache.

A.5.g.1. Different Cache Organizations

There are three different types of cache organizations: associative cache, direct-mapped cache, and set-associative cache. Associative cache and direct-mapped cache specify two different kinds of organization, while set-associative cache

is a combination of the other two. For all three types, the memory is assumed to be 2^m bytes. Please see Figure 3-12. The memory is divided into sequential blocks of b bytes equaling a total of $2^m/b$ blocks. Each block contains an address representing some multiple of b. The block size, b, is usually a power of two.

The associative and direct-mapped caches have their own advantages and disadvantages. The direct-mapped cache is less complex, cheaper to build, and has a faster access time because the appropriate slot being found by indexing into the cache uses a portion of the address as the index. The associative cache supplies a higher hit ratio for any given number of slots.

A.5.g.1a. Associative Cache.

The associative cache consists of a number of slots or lines. Each slot or line contains one block and its block number along with a bit telling whether or not that slot is currently in use. In an associative cache, the order of the entries is random. When the system is reset, all the valid bits are set to zero which indicates there are not any valid cache entries. Figure 3-12 shows primary memory with 2^{24} bytes partitioned into 2^{22} 4-byte blocks. The primary memory diagram contains the actual value of words which are used in an executing program. The values 137, 52, 1410, and 635 are random numbers chosen to represent the word values.

Figure 3-13 provides a diagram of an associative cache.

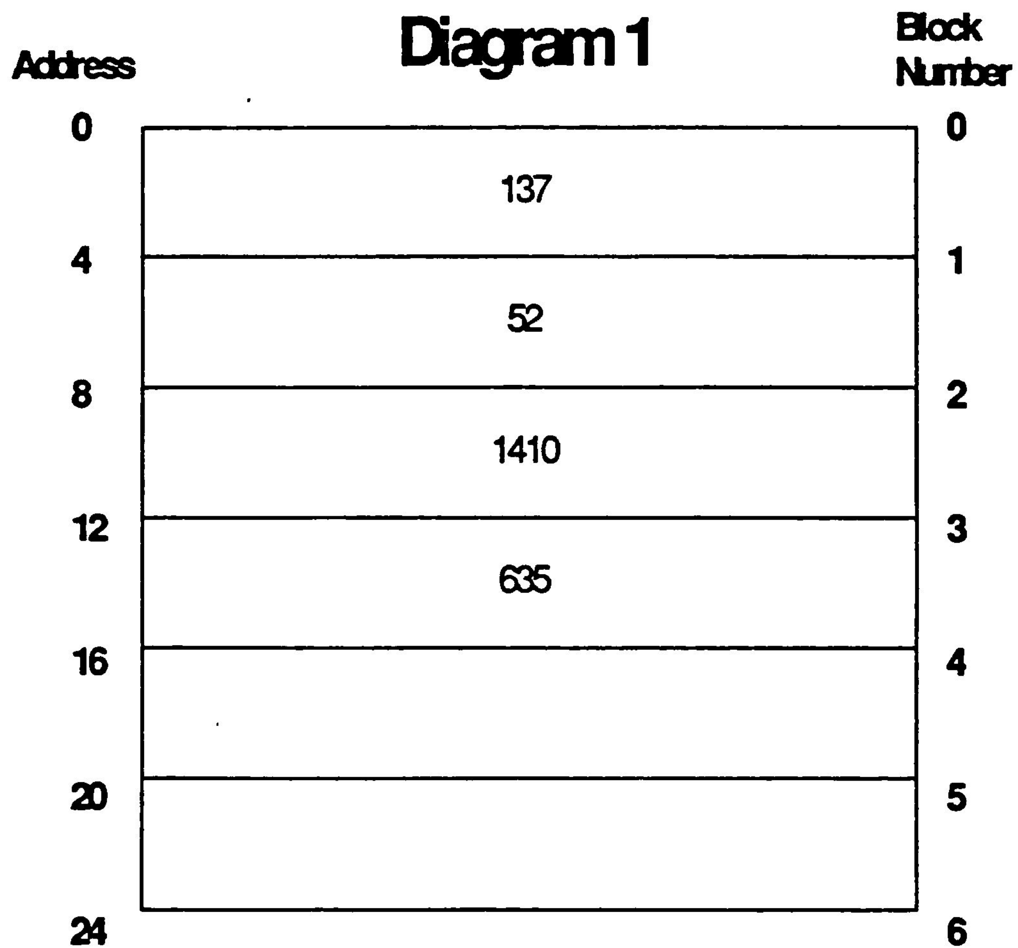
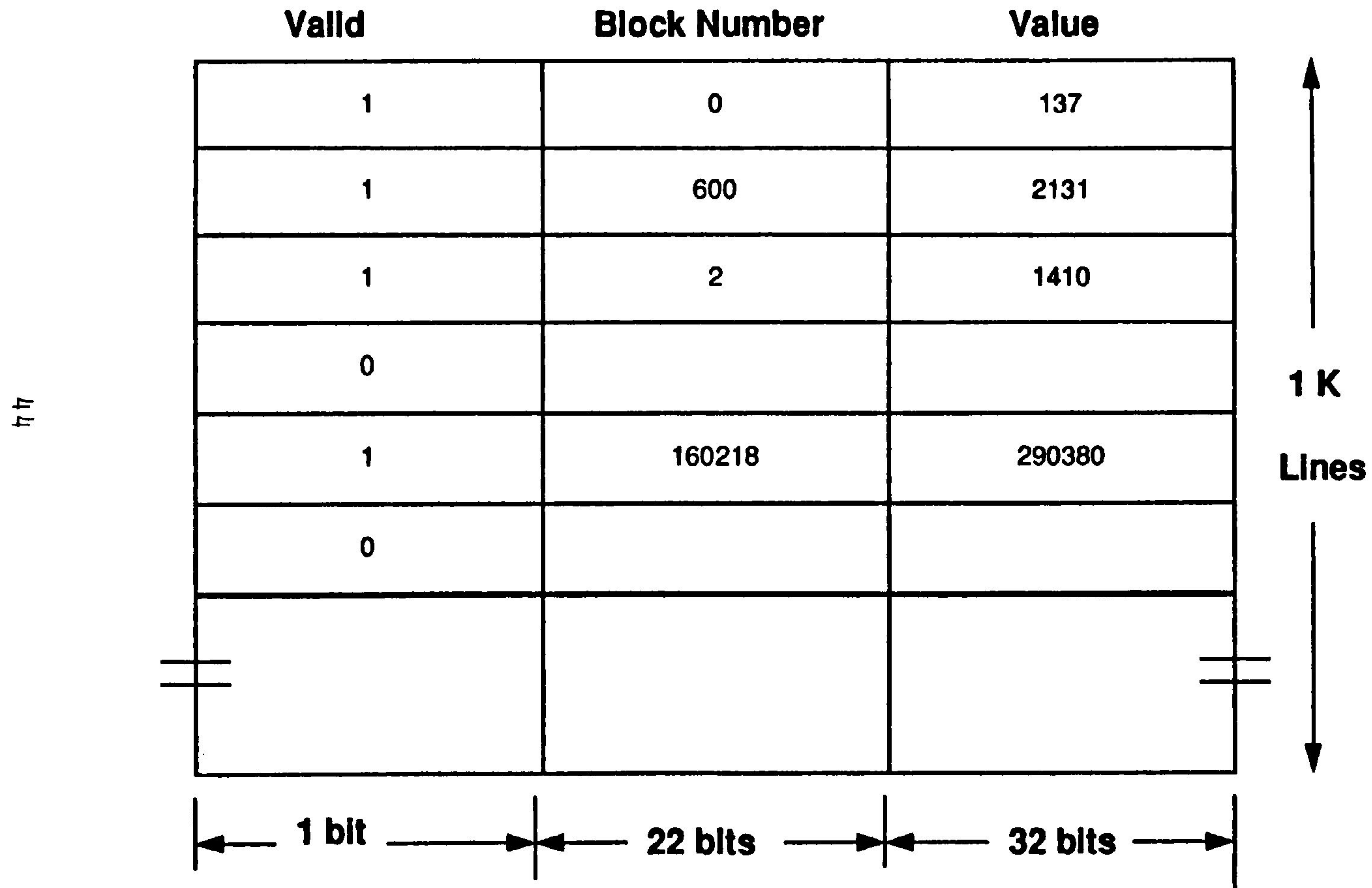


FIGURE 3-12. MEMORY WITH 4-BYTE BLOCKS.

Diagram 2

FIGURE 3-13. **Associative Cache with 1024 Lines**



Valid	Block Number	Value
1	0	137
1	600	2131
1	2	1410
0		
1	160218	290380
0		

If the first program instruction references the 32-bit word at address 0, the microprogram will check all entries of the associative cache in search of an occupied entry containing block number 0. When the cache controller fails to find the entry, it will send a bus request to fetch word 0 from the primary memory. The word 0 is fetched from primary memory at address 0, and the controller then generates a valid entry for block number 0 which will contain the contents of word 0 within the cache. The associative cache diagram contains all words which have been referenced for execution of the program's instructions. The block numbers 0, 600, 2, and 160218 are random block numbers chosen to reference the program's instructions. The values 137, 2131, 1410, and 290380 are randomly chosen values for the words used to execute the program. If a word is needed again, it is then taken from the cache which eliminates the need for a bus operation.

When more cache entries are marked as occupied or valid, the entire program and its data will eventually appear in the cache. Since the program will be running at high speed at this particular point, there will not be any need to make any memory references over the bus. At some specific point, the cache becomes full; and, an old entry will have to be removed to create space for a new one.

The associative cache is distinguished from other types since each slot contains a block number and its entry. When

a memory address is presented, the microprogram computes the associative block number and finds the block number in the cache. To prevent a linear search from being done, the associative cache uses special hardware which compares every entry to a given block number simultaneously as opposed to exercising a microprogram loop. This hardware causes the associative cache to be quite expensive.

A.5.g.1b. Direct-Mapped Cache

Conversely, the direct-mapped cache was created to reduce the cost accompanying associative cache memory. Direct-mapped cache prevents a search from being done by moving each block in a slot whose slot number can be easily calculated from the block number. The slot number is the block number modulo the number of slots. Figure 3-14 shows a pictorial representation of this example. With 4-byte word blocks and 1024 slots, the slot number for the word at address A is (A/4) modulo 1024. So the words at 0, 4096, 8192, etc. would map onto slot 0, while the words at 4, 4100, 8196, etc. would map onto slot 1.

This eliminates the problem of searching every slot number, but direct-mapped caches must reveal which of the many words mapped into a given slot are currently occupying that particular slot location. One may be able to tell by placing part of the address in the cache within the tag field. The tag field is the part of the address which cannot be computed from the slot number.

Let us consider another example, suppose you have an

Diagram 3

Slot	Valid	Tag	Value	Addresses that use this slot
0	1	2	12130	0, 4096, 8192, 12288, ...
1	1	1	170	4, 4100, 8196, 12292, ...
2	1	3	2142	8, 4104, 8200, 12296, ...
3	0			12, 4108, 8204, 12300, ...
4	0			16, 4112, 8208, 12304, ...
5	0			20, 4116, 8212, 12308, ...
...				
1023	1			4092, 8188, 12284, 16380

FIGURE 3-14. **A Direct-Mapped Cache with 1024 Slots of 4 bytes each**

instruction at address 8192 which moves the word at 4100 to 12296. Within Figure 3-14, 1024 blocks have been created using the total block numbers modulo the cache size. The block number corresponding to 8192 is computed by dividing 8192 by 4 which equals 2048. The 12 bit binary representation for 2048 in binary form is 100000000000. The slot number is computed by taking 2048 modulo 1024 which equals 0. This is the same as using the low-order 10 bits of 2048. The upper 12 bits contain a 2, which becomes the tag. The block number for 4100 is computed by dividing 4100 by 4 which equals 1025. The 12 bit binary representation for 1025 is 010000000001. The slot number is 1 since 1025 modulo 1024 equals 1; and, the low-order 10 bits contain 1. The upper 12 bits of 1025 contain a 1; the tag then equals 1. The block number for 12296 is calculated by dividing 12296 by 4 which equals 3074. The 12 bit binary representation for 3074 is 110000000010. The slot number equals 2 since 3074 modulo 1024 is 2; and, the low-order 10 bits holds a 2. The upper 12 bits equal 3 which becomes the tag value. Figure 3-14 displays the cache after the evaluation of all three addresses.

Please view Figure 3-15 to fully understand how the address is divided. The low-order two bits are always set to 0 since the cache works with whole blocks; and, these bits are multiples of the block size. In this example, the block size is 4 bytes; the slot number is 10 bits; and, the tag is 12 bits. For maximum efficiency, hardware should be built which

Diagram 4

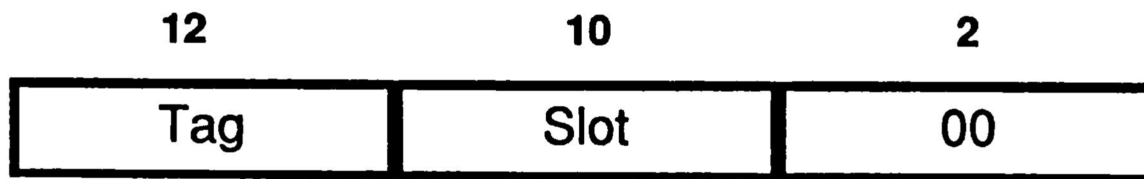


FIGURE 3-15.

Calculation of the slot and tag from a 24-bit address.

extracts the slot number and tag from any memory address.

Additional problems will occur since multiple blocks map into the same cache slot. If two words both have addresses which map into the same slot, the word which was processed last would occupy the cache while the word processed first would be discarded. The direct-mapped cache performance is degraded if many words are being mapped to the same slot.

A.5.g.1c. Set Associative Cache

Set associative cache provides a solution to problems which arise from associative and direct-mapped cache. It offers a direct-mapped cache with multiple entries per slot. The set associative cache combines associative and direct-mapped caches. Please view Figure 3-16 for a pictorial representation. If the number of slots is reduced to 1, all the cache entries will be in the same slot. Therefore, they would have to be distinguished entirely by their tags since they all map into the same address. This distinction is similar to an associative cache. If n=1, then that resembles a direct-mapped cache with one entry per slot.

A.5.g.2. An Explanation of the Cache Questions & Answers

The questions selected provide an illustration of different caching strategies. There are five questions within the tutorial which are actual questions taken from a previously given Computer Science GRE.

The first question asks for the major disadvantage of unpaged caches. An unpaged cache or read-ahead cache links

Diagram 5

Slot	Entry 0			Entry 1 Valid	Tag	Value		Entry n-1 Valid	Tag	Value
0										
1										
2										
3										
4										
5										

FIGURE 3-16. **A Set Associative Cache with n Entries per Slot.**

disk domains, or fields, with the start address of the read request and proceeds for a particular length. The major disadvantage of an unpaged cache is cache domains, or fields, are permitted to hold repetitious data. A cache domain is always loaded beginning with the first address of the read request.

Yet, the recently entered data may be required for the following read requests; and, subsequent read requests may begin with data that is partially located in the cache domain. A problem arises when the request cannot be completed since only part of the data for the request is in the cache. Therefore, another cache domain must be filled from the disk which may very well lead to the storage of redundant data in the cache.

Unpaged or read-ahead caching demands a more complex algorithm for handling input and output, especially when a write request is submitted. The correct cache domain has to be found to update the file; and, then the cache domain containing unnecessary repetitious data must be removed to resolve the problem.

The second question pertains to cached and interleaved memory models. Cached and interleaved memories are methods of increasing the speed of memory access time between CPUs and slower RAM. The question asks which memory models improve performance the most, or are best suited, for which particular programs. The choices include: 1) cached memory is best

suited for small loops; 2) interleaved memory is best suited for small loops; 3) interleaved memory is best suited for large sequential code; and, 4) cached memory is best suited for large sequential code.

The answer involves choices 1 and 3. Caches are small in relation to the size of memory and a small loop would fit within a cache to decrease the accesses to slower RAM. Interleaved memory accesses RAM in parallel sequential memory blocks. Branching occurs seldom with large sequential code; therefore, each sequential piece read from slow RAM is fully utilized.

On the contrary, choices 2 and 4 are false. A small loop would cause frequent branching; and, interleaved memory would reaccess RAM without using all the previously read information from the prior read request. Large sequential code permits the access of new addresses which results in more RAM read requests and the decrease of cache performance.

Third, a memory system with the following parameters is evaluated: the cache access time, T_c, is 100 nanoseconds,and the main memory access time, T_m, is 1200 nanoseconds. If you desire effective or average memory access time less than 20% higher than cache access time, the hit ratio for the cache needs to be at least 98%.

The following formula provides an effective system access time, T_s: $T_s = HT_c + (1-H)T_m$. H is the hit ratio expressed from .80 to .95 up to 100. Therefore, 100H + 1200(1-H) needs

to be less than or equal to T_c + (.20 x T_c) which equals 120 nanoseconds, or 20% more than the cache access time. Furthermore, 100H + 1200 - 1200H $\leq$ 120 and H $\geq$ (1100 $\div$ 1080) which equals .98.

The fourth question concerns cache and main memory. Assume the cache and main memory access times are 100 and 1200 nanoseconds. Some market survey denotes the average cost per bit of cache memory, C_c, is .0002 cents per bit, and the average cost per bit of main memory, C_m, equals .00001 cents per bit. The cost of a 1 megabyte memory system rounded to the nearest 10 dollars using a cache hit ratio of at least 98% would be $230.

The hit ratio is assumed to be 98%. The optimal cache memory size is not possible to obtain. The size of the cache other than the hit ratio is affected by several other factors including the application's environment, the work load, and the design considerations involving the chip and board area.

Generally, cache sizes vary from 1K to 128K for a 1 megabyte, MB, system depending upon the desired hit ratios. Suppose the following experimental results involving the hit ratio versus the cache memory size for a 1 MB memory system occur for the application and work load being considered:

Cache size per 1MB memory	Application hit ratio
1K	.75
32K	.80
64K	.85
96K	.90
128K	.95
196K	.96
256K	.98

512K	.99

According to the table, the desired 98% hit ratio requires 256K of cache memory. Therefore, the cost of a 1MB system follows:

$$
\begin{aligned}
C &= \text{cost of main memory} + \text{cost of cache memory} \\
&= (2^{20} \times 8 \text{ bits} \times .00001 \text{ cents/bit}) + \\
&\quad (2^{18} \times 8 \text{ bits}) \times (.0002 \text{ cents/bit}) \\
&= \$(.00001 \times 2^{21}) + \$(.0002 \times 2^{20}) \\
&= \$20.97 + \$209.72 \\
&= \$230.69
\end{aligned}
$$

Coincidentally, cache equals the same amount as main memory.

The fifth question asks for the hit ratio of a cache if a system performs memory access at 30 nanoseconds with the cache and 150 nanoseconds without it. Suppose the cache uses a 20 nanosecond memory. The following formula calculates the memory access, M, using the hit ratio, H, the cache speed, C, and the ram speed, R: $M = H \times C + (1-H) \times R$.

$$
\begin{aligned}
&30 = H \times 20 + (1-H) \times 150. \\
&30 = 20H + 150 - 150H = -130H + 150 \\
&H = 130 \div 120 = 26 \div 24 = 13 \div 12 = \\
&\ 92\% \text{ approximately.}
\end{aligned}
$$

More questions will be added as future work.

A.6. Microinstruction Sequencing

A basic ALU cycle sets up the A and B ALU input latches, gives the ALU and shifter time to do their work, and stores the results. These steps must occur in that sequence. To achieve the correct event timing, we use a four-phase clock. A four-phase clock is a clock with four subcycles.

The key events during each of the four subcycles are as follows:

1. Load the next microinstruction to be executed into the MIR.
2. Gate the registers onto the A and B buses and capture them in the A and B latches.
3. Now that the inputs are stable, give the ALU and shifter time to produce a stable output and load the MAR if required.
4. Now that the shifter output is stable, store the C bus in the scratchpad and load the MBR, if either is required.

A.7. Boolean Algebra (Switching Algebra)

In 1854, George Boole introduced a symbolic notation to handle symbolic statements that employ a binary value of either true or false. This symbolic notation was adopted by Claude Shannon to analyze logic functions and has since been known as Boolean algebra or switching algebra.

A Boolean algebra is a closed algebraic system containing a set K of two or more elements and two binary operators '+' (OR) and '&' (AND); for, every X and Y in set K, X&Y belongs to K and X+Y belongs to K. In addition, the following postulates must be satisfied:

P1: Existence of 1 and 0
a) X + 0 = X
b) X & 1 = X

P2: Commutativity
a) X + Y = Y + X

P3: Associativity
a) X + (Y + Z) = (X + Y) + Z
b) X & (Y&Z) = (X&Y)&Z

P4: Distributivity
a) X + (Y&Z) = (X + Y) & (X + Z)
b) X & (Y+Z) = X&Y + X&Z

P5: Complement
a) X + X' = 1
b) X & X' = 0

Summary

$X + 0 = X \qquad X + 1 = 1$

$X \& 0 = 0 \qquad X \& 1 = X$

$0' = 1, \quad 1' = 0, \quad X'' = X$

Where & denotes AND, + denotes OR and ' denotes a complement.

Two expressions are said to be equivalent if one can be replaced by the other. The "dual" of an expression is obtained by replacing each "+" in the expression by "&", each "&" by "+", each 1 by 0, and each 0 by 1. The principle of duality states that if an equation is valid in Boolean algebra, its dual is also valid.

The following theorems are useful in manipulating Boolean functions. They are traditionally used for converting Boolean functions from one form to another, deriving canonical forms, and minimizing, or reducing the complexity of, Boolean functions.

T1: Idempotency

a) $X + X = X$

b) $X \& X = X$

T2: Properties of 1 and 0

a) $X + 1 = 1$

b) $X \& 0 = 0$

T3: Absorption

a) $X + XY = X$

b) $X \& (X + Y) = X$

T4: Absorption

a) $X + X'Y = X + Y$

b) $X \& (X' + Y) = X \& Y$

DeMorgan's Laws

a) $(X + Y)' = X' \& Y'$

b) $(X \& Y)' = X' + Y'$

Consensus
a) $XY + X'Z + YZ = XY + X'Z$
b) $(X + Y) \& (X' + Z) \& (Y + Z) = (X + Y) \& (X' + Z)$

B. File Structures

File structures specify the forms in which files may be structured. These structures are processed and even organized in different ways. File processing refers to the manner in which records are processed or stored in an external file. In addition, file organization pertains to the data structures associated with organizing the data. Four common file organizations include sequential, random, indexed sequential, and multikey.

B.1. Sequential File Organization

Sequential file organization characterizes the simplest type. Sequential grouping employs the least complex process for organizing files. Files are written consecutively in sequence from beginning to end and must be accessed in the same manner. The retrieval of files adopts a LIFO (last-in-first-out) method.

B.2. Random File Organization

Random file organization implies a predictable relationship between the key used to identify an individual record and that record's location in an external file. A relative file illustrates a common implementation of random file organization while being available in most high-level programming languages. Once a key-position relationship is

established, the position of the record in the file is specified as a record number relative to the position of the record from the beginning of the file. Each address is computed by using the following formula: the record's address = (the relative record number x the fixed record length) + the beginning of the file.

B.3. Indexed Sequential File Organization

Indexed sequential file organization combines sequential access and ordering with the capabilities of random access. An indexed sequential file contains two parts. The first part stores a collection of records in contiguous locations within blocks in a relative file. The record's order is according to a key field. The second part holds an index to the file of ordered records. The index contains a hierarchical structure of record keys and relative block numbers. The blocks of records in the file are not necessarily stored in sequential order. The index indicates the order in which the blocks should be obtained to achieve sequential order by record keys. Indexed sequential file organization provides sequential access to records by the primary key order field. Random access is supplied, as well, to an individual record by the same key. Indexed sequential file organization comprises two types: the cylinder and surface indexing method, and the index and data blocks method.

B.4. Multikey File Organization

Multikey file organization permits access to a data file

by several different key fields. Hence, multiple key fields are recognized to retrieve a particular file. The advantage simply lies in the file's access not being restricted to only one key field.

Multiple key-file organization includes two different types. The first being inverted file organization which accesses data from a file by secondary keys. The access is achieved by using a directory of all possible attributes for each secondary key field; and, the primary key or address of all records containing those attributes. The second type is multilist file organization which uses a file to link data records with identical secondary key values to organize a multilist file. A multilist file is divided into two areas: a directory and a data record area.

The directory contains an inverted list for each secondary key field to be used in accessing the data records in the data record area. The directory also stores all possible attributes for each secondary key field and a pointer to a linked list of all records storing the attribute. The data records are organized to provide random access to the records.

B.5. Blocking

The smallest amount of data which can be read from or written to a secondary storage device at a time occupies a block. By blocking several components into one block, several components can be accessed at one time. This increases

efficiency by decreasing the amount of physical accesses to the file and lessening the execution time of the program accessing the file.

The amount of data transferred to or from the file during access forms the size of one block. The size cannot exceed the amount of available main memory.

B.6. Buffering

A buffer functions as a location in memory to serve as an intermediary between I/O devices and main memory. Double buffering is possible when both the I/O operations performed by the I/O channel and the processing operations generated by the CPU overlap in time.

Implementations are I/O bound when access time to input one buffer from the file is longer than process time. The total time to process the file cannot be reduced without reducing the file's access time. Processor-bound demonstrates processing time is longer than access time.

B.7. File Storage Devices

Magnetic tape represents a sequential storage device in which blocks of data are stored serially along the length of the tape and can only be accessed consecutively. Data are recorded in 9 tracks aligned parallel to the edges of the tape. Magnetic tapes are used for sequentially organized files only.

The recording density of the tape is the number of bytes of data to be stored per inch. A block of data is sometimes

called a physical record, and may contain one or more logical records. These physical records or blocks are separated by interblock gaps (IBGs) in which the tape can start and stop between I/O requests.

The following formulas involve accessing specific information from magnetic tape:

Space Calculations:

Block Length (bytes) = Logical Record Length x Blocking Factor

Block Length (inches) = Block Length (bytes) / Density

Number of Blocks = Number of Records / Blocking Factor

Block Length = Block Length (bytes) / Density

Tape Length = (Number of Blocks x Block Length) + (Number of IBGs x IBG Length)

Accessing Time and Tape Calculations:

Time to Read 1 Block = Start Time + Stop Time + Block Length (inches) / Transfer Speed

Time to Read 1 Tape = (Time to Read 1 Block) x (Number of Blocks)

Single Buffering: Total Time for File = (Time to Read 1 Block + Processing Time for 1 Block) x Number of Blocks

Double Buffering: Total Time for File = (Input Time x Number of Blocks) + Processing Time for 1 Block

B.8. Timing of Access Methods

Brief definitions of additional terms will ensue. These terms are associated with several formulas given later. The

transfer rate is the uniform rate at which all tracks on the same disk pass under the read/write head. Rotational delay or latency pertains to the time required for the beginning of the accessed block to rotate around to the read/write head. Transfer time regards the time required for the entire block to pass under the read/write head. Minimum seek time concerns the time needed to move the access arm to an adjacent track. Maximum seek time refers to the time it takes to move from the outermost or innermost track to the farthest track.

Average seek time relates to the average of both the maximum and minimum seek times. Seek time establishes the most significant factor of the access time, while the transfer rate is the least significant factor.

Rotational time pinpoints the time it takes for a disk pack to make one revolution. Rotational delay or latency operates between zero delay and the rotational time. The following formulas involve accessing specific information from a fixed disk:

Access times:

Average Access Time per Block = Seek + Latency + Transfer

For Fixed Disks with One Head per Track, the Average Access Time = Latency + Transfer

Hard Disk Capacity:

(If Using Cylinders)

Readable Radius = (Outer Diameter - Inner Diameter) / 2

Tracks/Surface = (Readable Radius / Spacing Between Tracks) + 1

Bits/Track = Density x Circumference of Smallest Track = Density x Inner Diameter x Pi

Bits/Pack = Bits/Track x Tracks/Surface x Surfaces/Pack

(If Using Sectors)

Bits/Sector = (Bits/Track) / (Sectors/Track)

Bytes/Sector = (Bits/Sector) / (Bits/Byte)

Bits/Track (Sectoring) = Bits/Byte x Bytes/Sector x Sectors/Track

Bits/Pack = Bits/Track x Tracks/Surface x Surfaces/Pack

Transfer Rate = Bytes/Revolution x [(Revolutions/Minute) / (60 Seconds/Minute)]

B.9. Record Keys

On the other hand, with direct file organization there exists a predictable relationship between the key used to identify an individual record and the record's absolute address in the external file. A relative file with space for N records contains positions with relative record numbers 0, 1, ..N-1 where the ith number has the relative record number I - 1. The key-position relationship must be a predictable relationship to ensure direct access to the record is possible once the record is stored in the relative file.

B.10. Hashing

Hashing is the application of a function to the key of a particular record resulting in mapping a range of possible key values into a smaller range of relative addresses. The

function randomly selects a relative address for a specific key value without regard to the physical sequence of records in the file. A collision occurs when the hashing function for two different keys results in the same relative address.

Load factor concerns the relationship between file space and the number of keys. Load factor also may be referred to as packing density or packing factor. Load factor is equivalent to the number of key values per the number of file positions. The smaller the load factor, the less chance there is of collision.

The methods of hashing involve prime-number division remainder, digit extraction, folding, radix conversion, mid-square, quotient reduction, and remainder reduction. These methods directly reference records in a table by performing arithmetic transformations on keys into table addresses. A perfect hashing function provides an one-to-one mapping from a key value into a specific position.

The prime-number division remainder refers to the division of the key value by a number N. The remainder of the division yields a number in the range 0 to N - 1.

Digit extraction involves the analysis of key values to determine which digit positions of the key are more evenly distributed. The more evenly distributed digit positions are assigned from right to left. The digit values in the chosen digit positions are extracted; and, the resulting integer is used as the relative address.

Folding entails splitting the key value into two or more parts. The results are summed or subjected to arithmetic and/or logical operations as if each part were an integer. If the resulting address contains more digits than the highest address in the file, the excess high-order digits are truncated.

Radix conversion simply interprets the key value as having a different base or radix. The value is then converted into a decimal number.

The mid-square hashing method extracts the middle n digits from the key value. The value is squared to form a relative address.

The hashing of a key is equivalent to the key length and the associated value of the key's first character and the associated value of the key's last character. The associated value is based on the frequency of use of each letter in either the first or last position of the word.

B.10.a. Collision-Handling Techniques

Two classes of collision-handling techniques exist: open addressing and separate chaining. Open-addressing hashing methods entail the storage of N records in a table of size M>N and rely on empty places in the table to help with collision resolution. Separate chaining hashing methods handle the situation when two keys hash to the same address. The least complex method involves building for each table address a linked list of the records whose keys hash to that address.

Within a linked list, the keys are kept which hash to the same table position. Therefore, it would be advantageous to keep the keys in order.

B.10.a.1. Linear Probing

Linear probing is the simplest open-addressing method. Collisions occur either when hashing to an occupied place in the table or when the record's key is not the same as the search key. If a collision arises, probe the next position in the table which means comparing the key in the record there against the search key. Three possible consequences will happen. The search will successfully terminate if the keys match. The search unsuccessfully terminates if there is not any available record. Otherwise, continue probing the next position until discovery of either the search key or an empty table position is reached.

Linear probing definitely works since when searching for a particular key, every key hashing to the same table address is examined. However, problems arise when the table slowly becomes full and insertion of a key with one hash value tremendously increases the search times for keys with other hash values. As a result, clustering occurs and makes linear probing operate extremely slow for nearly full tables.

B.10.a.2. Double Hashing

Double hashing provides the solution. Double hashing concerns the process of hashing synonyms to an overflow area using a second hashing function. If the hashed overflow area

is not available, then linear probing is used. The same strategy of linear probing is employed. The main difference exists where a second hash function obtains a fixed increment to use for the 'probe' sequence instead of evaluating each subsequent entry coming after a collided position.

B.10.b. Hashing vs. Binary Search Trees

In general, hashing is preferred to binary search tree structures because of its simplicity and provision for very fast searching times when enough space is available for a large table. However, binary tree structures provide an advantage over hashing since they are dynamic, capable of giving guaranteed worst-case performance, and support a wider range of operations. The dynamic feature plays an important role since no advance information on the number of insertions is needed. The worst-case performance quality is essential because with hashing, all entries could hash to the same place even while using the best hashing method.

B.11. Trees

Binary trees may be considered as both file organization structures and data structures, too. Before a discussion of binary trees is begun, let me thoroughly explain the definition of trees. Trees may be clarified as abstract objects or a nonempty collection of vertices and edges satisfying specific requirements. A vertex or node acts as a simple object possessing a name and carrying other associated information. An edge provides the connection which lies

between two vertices or nodes. A tree's path concerns a list of distinct vertices where subsequent vertices hold a connection by edges in the tree. One node in the tree is assigned as the root which implies the defining property of a tree; there is always precisely one path between the root and each of the other nodes in the tree. Assume all edges point either away from or toward the root, while the root itself is located at the very top.

In addition, every node, excluding the root, has only one node above it which is referred to as its parent; and, the nodes located immediately below that particular node are called its children. Nodes without any children are referred to as leaves or terminal nodes. Nodes generating at least one child are defined as nonterminal nodes. Nonterminal nodes are internal nodes, and terminal nodes are external nodes. A subtree consists of any given node in a tree together with all its children. A set of trees refers to a forest. The level contains the number of nodes on the path from the node to the root without including the node. A tree's path length includes a sum of all levels of the nodes in the tree or the sum of all lengths of the paths from each node to the root.

Multiway trees concern each node having a particular number of children which appear in a particular order. Binary trees are the simplest type of multiway trees. The distinguishing property of a binary tree involves the possession of two types of nodes: external nodes without any

children and internal nodes with only two children. The child on the left is the left child; and, the child on the right is the right child.

The height of a tree consists of the number of levels of nodes contained in the tree. The number of nodes, N, in the tree is derived from the height as follows: $N = 2h - 1$. The minimum height of a binary tree may be defined in the following formula: $h \leq \lceil \lg(N + 1) \rceil$.

A height balanced binary search tree is a balanced tree in which the height is minimum for the number of nodes. An AVL tree is a non-empty tree T being height balanced when it holds the following properties:

1. TL is the root of the left subtree (left child) of T
2. TR is the right subtree of T
3. TL and TR are height balanced
4. hL is the height of TL
5. hR is the height of TR
6. $|^3hL - hR^3| \leq 1$

Multiway or m-way search trees also include B-trees, B* trees, and B+ trees. A B-tree holds the property of being an m-way search tree which is either empty or the height ≥ 1; the root node has at least two children implying it contains at least one value; and, all nonterminal nodes other than the root node store at least $\lceil m/2 \rceil$ children directing at least $\lceil m/2 \rceil - 1$ values. B* trees involve a modification to the B-tree structure such that every node in the tree is at least two-thirds full as opposed to half full. B+ trees resemble the B-tree except all data or record addresses appear only in

the leaf nodes. All the B+ tree interior nodes contain only subtree pointers to other nodes.

C. Data Structures

Data structures are a collection of data elements whose organization is characterized by accessing operations to store and retrieve the individual data elements. Important to note, data structures display three features. The first feature being the structure can be split into its respective component elements. Second, the grouping of the elements affects how each element will be accessed. Third, the grouping of the elements and the manner in which they are retrieved can be enclosed.

C.1. Array

The most elementary data structure implemented is the array. An array contains a fixed number of data items which are stored continuously and are accessible by an index. The user must declare the size of an array at the beginning of the program. This static declaration must be made whether or not all the declared space is utilized.

C.2. Linked List

The next elementary data structure generated is the linked list. A linked list holds a set of items which are organized sequentially. Each item represents a part of a node containing a link to the next node. Every node must have a link. To initialize the list, the head points to the first item. To link the list, move the nil pointer to the next

node. Nodes must be deleted to free up space to prevent space from running out. The head always points to the first node.

Linked lists display several useful advantages over arrays. The linked list may grow and shrink in size. Its space is dynamically allocated. The linked list is more flexible because of its explicit way of ordering.

Different types of the list include circular linked list and doubly linked list. A circular linked list allows each element to point to the next element, and the last element points to the first element. A doubly linked list permits the simultaneous pointing of the links to the next and previous node.

C.3. Stacks

The stack data structure imposes restrictive access. This ordered list only permits all insertions and deletions to be performed at one end, which is called the top. This structure exhibits LIFO or last-in-first-out behavior.

Stacks satisfy several specifications. The stack elements depict a variety of types. The stack structure establishes the mechanism used to store the elements and determines their order of arrival into the stack.

Stack operations implement various procedures to perform processes directly on the stack structure. The operations include create, push, pop, empty, and full. Each procedure may have either a pre-condition, a post-condition, or both.

Create initiates the formation of the stack. Its post-condition is the stack exists and is empty afterwards. Push(x) forces the element x on the stack. The pre-condition of push is the stack is not full, and the post condition is (x) is placed on the stack. Pop(x) pulls an element x off the stack. The pre-condition is the stack is not empty, and the post-condition is (x) is removed from the stack. The empty:boolean function returns the value, true or false, after determining whether or not the stack is empty. The post-condition for this function is if the stack does not contain any elements, then empty is true else empty is false. The full:boolean function answers the question as to whether or not the stack is full. This function's post-condition is if the stack has reached its maximum allowable size then the structure is full; finding the function, full, to be true else full is false. Clear removes all elements from the stack. Its post-condition simply maintains the stack is empty.

C.4. Queues

Queues achieve an ordered list in which all insertions take place at one end which is the rear and all deletions take place at the front. This particular data structure exhibits FIFO or first-in-first-out behavior. These operations include create queue, clear queue, enq, and deq. The create queue function initiates a queue and its structure. Clear queue deletes all elements from the queue. The pre-condition demands the queue contains an element, and the post condition

stresses the queue is empty. The function enq(queue, z) adds an element to the queue. Deq(queue, z) removes the element stored in the front from the queue. Stacks and queues are both forms of linked lists.

C.5. Trees

Trees are also data structures which represent hierarchical structures having an one-to-many relationship among its components. The vertex or node remains a simple object possibly bearing a name and carrying other associated information. An edge acts as an arc which connects a pair of related vertices. A path represents a sequence of nodes such that there is an edge between each pair of nodes.

The first property of a tree defines the structure as an acyclic, connected graph such that there is exactly one path between the root and each of the other nodes in the tree. There are not any cycles present.

The second property lies in the root being the unique first node bearing no predecessors, but possibly producing many successors. There are not any nodes located before the root; yet, many nodes succeed the root. The relationship existing between a node and its successor is that of a parent and a child.

An ordered tree has a specified ordering of parent and siblings. A multiway tree is an ordered tree with a specific number of 'kids' or children at each node. A binary tree has two children at the most for each node; it is an ordered tree

since there is a left and right child. A binary tree is a directed acyclical graph with a degree of two; however, binary trees could be empty.

Additional tree properties concur there is one path connecting two nodes in a tree. A tree with N nodes contains N-1 edges. A binary tree with N internal nodes includes N+1 external nodes. Internal nodes store two children; however, external nodes do not have any children. The external path length with N internal nodes is 2N greater than the internal path length.

A relationship exists between forest trees and binary trees. Even though, a forest tree is a group of trees and a binary tree has two kids for each node, a binary tree can be a subset for a forest.

Tree traversal recognizes three different methods for traversing trees or accessing the nodes. Preorder traversal visits in order the root, left side, and right side. Inorder traversal reaches the left side first, the root second, and the right side last. Postorder traversal meets the left side, right side, and then the root. Level-order traversal visits nodes as they appear on the page, down from top to bottom and from left to right. With level-order traversal, all nodes on each level appear together in order.

C.6. Measuring a Program's Performance

Above all, a program's performance is measured by the execution time and the amount of storage used. With the

execution time, does speed vary according to the type of input used? With storage, how much storage is required? To determine the performance of a program, measure the performance of the working version and analyze the expected performance of algorithms and data representations on which the program is based.

An evident rule about performance and efficiency is major performance gains almost always come from a better choice of data representation and algorithms, not from adjustments of an existing design. The 90-10 rule states more than 90% of execution time is spent executing less than 10% of your code. The critical 10% of the program is often difficult to determine. The critical 10% may be in the language system and not in the code you have actually written. The worst case answers the question, what is the longest time the algorithm may run if we choose data values forcing the algorithm to take the longest possible execution time? The average case determines for the expected range of possible data values, what is the average execution time we can expect?

C.7. Algorithm Characteristics

Characteristics of an algorithm include finiteness; definiteness where each step must be precisely defined; input where the algorithm has 0 or more inputs; output where an algorithm has 0 or more outputs; and, effectiveness involving an effective procedure and any algorithm which can be executed in a finite number of steps that can be performed by person or

machine. The main criteria for goodness is situated between execution speed and the amount of storage required.

Abstract data types describe data structures and its accompanying algorithms. This description is in terms of its operations as opposed to its actual implementation. Procedural abstraction pertains to the grouping of a complex sequence of actions into a single unit. The grouping of actions is considered a procedure. Data abstraction is the grouping of a complex sequence of data items into a single unit.

C.8. Allocation of Storage Space

Dynamic allocation demands the allocation of storage upon execution time or when the program is running. Space is not set aside beforehand, it is created as you need it or originated "dynamically." Static allocation requires storage to be set aside at compilation time before the actual program is run.

C.9. Program Performance Improvements

The problem size is N. The three classes of improvements are: 'order of,' 'constant factor,' and 'additive constant' improvements. 'Order of' improvements involve major changes where the change and efficiency is noticeable. The improvement gets better as the problem size increases from one data representation to another. The 'constant factor' improvement involves taking something from the loop which is dependent on problem size so the improved version is a

constant factor better than the original one. The amount of improvement does not depend on problem size. The 'additive constant' improvement is where the newer version of the improved program may always take exactly two fewer operations or use five fewer simple variables than the old.

C.10.An Explanation of the Questions & Answers

The questions selected demonstrate different data structure strategies. The tutorial user slightly comprehends the wide range of issues included within the data structures subject. There are six questions within the tutorial which are actual questions taken from a previous Computer Science GRE.

The first question relates the average time necessary to perform a successful sequential search for an element in an array, A, which is 1:n is calculated by using several steps. You find the average number of comparisons required for all successful searches and divide by n. The equation (1+2+3+4+...+n)/n demonstrates the previously defined process. The series 1+2+3+4+...+n equals n(n+1) / 2. Substituting into the first equation, n(n+1) / 2n equals (n+1)/2.

The second question asks for the correct relationship among several of the more common computing times for algorithms to be chosen from the following:

A. $O(\log n) < O(n*\log n) < O(n) < O(n^2) < O(2^n)$
B. $O(n) < O(\log n) < O(n*\log n) < O(2^n) < O(n^2)$
C. $O(n) < O(\log n) < O(n*\log n) < O(n^2) < O(2^n)$
D. $O(\log n) < O(n) < O(n*\log n) < O(n^2) < O(2^n)$

E. $O(\log n) < O(n*\log n) < O(n) < O(n^2) < O(2^n)$

The accurate choice is D where $n > n_0$. The tutorial user may plot graphs for values of n to prove the rate of growth of these specific time functions.

The third question concerns finding the best algorithm for calculating an arbitrary boolean function of N variables to produce a 1 value. The best algorithm requires exponential time. More constructively, you can write an algorithm which computes a truth table but requires 2^N rows.

The fourth question pertains to printing a sorted array of numbers after building a sorted binary insertion tree. A certain type of traversal must be done to print the sorted array of numbers. An in-order traversal works best since the binary insertion tree contains a number in the root, smaller numbers on the left, and larger numbers on the right. In-order means visit the left, the root, and then the right. The in-order tour will visit the smaller numbers, the number, and then the higher numbers to give a sorted order.

The fifth question discusses ordered groups of homogeneous elements insofar as a stack is similar to an array. The stack contents are homogeneous data items with the same type and the same requirements for memory. If a stack holds items with different types, it would be extremely difficult to exercise the same stack functions to access and manipulate the items. The array type contains elements of the same data type, i.e. an integer, character, etc. Queues also may be implemented using either a stack or an array since this

similarity does exists.

The last question concerns heap sort and merge sort algorithms yielding approximately the same worst-case and average-case running time behavior in O(n log n). When you evaluate the number of moves in an n-element array, constructing the heap requires at most (n/2)log n, while (n-1)log n moves are needed to sort the array.

Therefore, both the average and worst cases for the heap sort are (3/2)n log n which is an order of n log n. The merge sort for the same data requires n log n moves corresponding to as many passes involving a pointer manipulation over the list of n elements. There will be anywhere from log n to n log n comparisons with little effect on overall running time.

D. Discrete Mathematics (Set Theory)

This section reviews the concepts upon which set theory, theoretical tools, or discrete mathematics is based. The terminology upon which the foundation lies is thoroughly reviewed. Logic, induction, recursion, the AND statement, the OR statement, the compound statement, quantifiers, predicates, expression interpretation, free variables, validity, axioms, theorems, propositional logic, deductions, and predicate logic are principles discussed within the section which form the basic Discrete Mathematics foundation.

D.1. Logic, Induction, & Recursion

In reference to set theory, certain terms must be understood before the discussion of theoretical applications

begins. Logic may be explained as an organized, precise method of thinking used in program verification to prove the output of a given computer program will always comply with certain predetermined conditions. Induction is a proof technique with wide application in computer science. Recursion may be interpreted as repetition with a termination condition closely related to mathematical induction and is important in algorithms and their analysis.

D.1.a. The AND Statement

A statement or proposition refers to a sentence that is either true or false. A common connective belongs to the word 'and.' The symbol for this particular connective is represented by $\wedge$. The statement A $\wedge$ B is referable to the conjunction of 'A and B.' The effects of conjunction may be summarized by a truth table. In each row of the truth table, truth values are assigned to the statement letters. The resulting truth value for the compound statement is then shown.

D.1.b. The OR Statement

Similarly, another connective lies in the form of the word 'or.' The statement 'A or B' pertains to the disjunction of 'A and B'. The symbol for 'or' belongs to V.

D.1.c. The Compound Statement

Statements 'A $\wedge$ B' may be combined in the form 'if A, then B', symbolized by A $\rightarrow$ B, or 'A implies B'. The connective here is implication, and it conveys the meaning the

truth of A causes the truth of B. In the compound statement $A \rightarrow B$, A is called the antecedent and B the consequent.

The equivalence connective, $A \leftrightarrow B$ symbolizes the statement $(A \rightarrow B) \land (B \rightarrow A)$, or if A then B and if B then A. Binary connectives join two statements together to produce a second statement. The negation connective, A', represents a unary connective and is read 'not A', 'it is false that A', or 'it is not true that A.' This does not mean A' always has a truth value of false, but its truth value is opposite to the truth value of A.

A statement whose truth values are always true is a tautology. A statement whose truth values are always false is a contradiction. When a compound statement of the form $P \leftrightarrow Q$ is a tautology, the truth values of P and Q agree for every row of the truth table. In this case, P and Q are equivalent statements.

D.1.d. Quantifiers

Expressions containing variables can be made into statements by adding quantifiers. Quantifiers are phrases such as 'for every' or 'for each' or 'for some' relating in some sense how many objects have a certain property.

The universal quantifier is symbolized by $\forall$ and is read 'for all', 'for each', 'for every', or 'for any.' Thus, $\forall(x)$, $x>0$ relates to the expression 'for every x, x is greater than zero.' In order to determine the truth value, you must know the domain of objects in which you are interpreting the

expression, that is, the collection of objects from which x may be chosen. If the domain of interpretation consists of the positive integers, the expression has the truth value true because every possible value for x has the required property of being greater than zero. It the domain of interpretation consists of all the integers, the expression has the truth value false.

The existential quantifier is symbolized by ∃, and is read 'there exists one', 'for at least one', or 'for some one.' Hence, $\exists(x), x>0$ refers to the expression 'there exists an x such that x is greater than zero.' The truth value of this expression depends upon the interpretation. If the domain of interpretation contains a positive number, the expression has the value true; otherwise, it holds the value false.

D.1.e. Predicates

Symbols such as P in the expressions $(\forall x)P(x)$ and $(\exists x)P(x)$ are named predicates. Specifically, they are termed unary predicates since they involve one variable and are interpreted as properties of single objects. Predicates are distinguished by being either binary or n-ary with two variables or n variables, respectively. These variables are interpreted as properties of either two objects at a time or n objects at a time. Additional quantifiers may be added to expressions with n-ary predicates.

D.1.f. Interpretation of an Expression

An interpretation for an expression involving quantifiers includes: 1) a collection of objects, designated as the domain of the interpretation, which must include at least one object; 2) an assignment of a property of the objects in the domain to each predicate in the expression; and, 3) an assignment of a particular object in the domain to each constant symbol in the expression. Expressions are structured together by employing connectives.

Parentheses and brackets identify the scope of a quantifier. The scope correlates to the section of the expression to which the quantifier applies. Scope resembles the order of precedence of connectives since the truth value of the expression in any particular interpretation would be affected if the scope of a quantifier is misunderstood. This misunderstanding arises whether or not the expression has a truth value.

D.1.g. Free Variables

If there is an occurrence of a variable which does not fall within the scope of a quantifier involving that variable, then it is a free variable. An expression with free variables will not, in general, inherit a truth value in a given interpretation. Rather, the expression will be true for some choices of values of the free variable and false for others. An expression involving quantifiers generates a statement only if it does not contain any free variables.

D.1.h. Validity

For unquantified statements, a tautology remains true for all rows of the truth table. The analogy to tautology for quantified statements is validity. A statement is valid if it is true in all possible interpretations. The validity of a valid statement must be derived from the form of the statement itself, since validity is independent of any particular interpretation.

D.1.i. Axioms & Theorems

Axioms are statements not needing to be proved. An axiom should therefore pinpoint a statement whose 'truth' is self-evident. An axiom proposes a tautology if it involves quantifiers or a valid statement.

In addition to axioms, formal systems may contain rules of inference. A rule of inference demonstrates a convention allowing a new statement of a certain form to be inferred, or deduced, from one to two other statements of a certain form. A sequence of statements in which each statement is either an axiom, or the result of applying one of the rules of inference to earlier statements in the sequence, is directed as a proof sequence. A theorem composes the last entry in such a sequence; and, the sequence of statements is the proof of the theorem.

The following example shows a typical proof of a theorem:

s1 an axiom
s2 an axiom
s3 inferred from s1 and s2 by a rule of inference
s4 an axiom
s5 inferred from s4 by a rule of inference
s6 inferred from s3 and s5 by a rule of inference

Statement s6 serves as the theorem which is the last statement in the sequence. The entire sequence forms its proof. The other statements could function as theorems; however, the proof sequence would discontinue at that particular point. Axioms should deduce tautologies or valid statements, and there should be as few axioms and rules of inference as possible.

D.2. Propositional Logic

Propositional logic concerns two separate formal systems involving one for statements without quantifiers and one for statements with quantifiers. The former case with unquantified statements is designated as propositional logic, statement logic, or propositional calculus. In this system, a 'true' statement implies a tautology. Hence, the chosen axioms and rules of inference need to prove all tautologies, and only tautologies, as theorems.

In the following example, the statements perform as axioms, where P, Q, and R are compound statements:

1. $P \rightarrow (Q \rightarrow P)$
2. $(P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$
3. $(Q' \rightarrow P') \rightarrow (P \rightarrow Q)$

Each statement can be implemented as a tautology. There is only one rule of inference which can be deduced from statements P and $P \rightarrow Q$. Statement Q can be inferred. This rule of inference shows modus ponens meaning 'method of assertion.' Since P, Q, and R can be compound statements, each axiom given above derives a statement form, or schema,

for an infinite number of statements. Thus $(A \rightarrow B) \rightarrow ((C \land D) \rightarrow (A \rightarrow B))$ sets up an axiom since it fits axiom schema 1, where P is the statement $A \rightarrow B$ and Q is the statement $C \land D$.

The system of axioms and one rule of inference where it achieves the desired solution proves every tautology is a theorem (i.e., has a proof) and vice versa. This property is described by saying our formal system is complete and correct. Completeness is fulfilled when everything that should be a theorem is. Correctness is attained when there is not anything that is a theorem which should not be.

Shortcuts in proof sequences are allowed by using already proved theorems. Once theorem T has been proven as a theorem, then T can serve as a statement in another proof sequence. Since T has its own proof sequence, T is utilized as a substitution into the proof sequence we are constructing.

D.2.a. Deductions

Deductions arise when you desire to prove statements of the form $P \rightarrow Q$ as theorems where P and Q are compound statements. P is denoted the hypothesis of the theorem, while Q is the conclusion. If $P \rightarrow Q$ is a theorem, it must propose a tautology. Whenever P is true, Q must be true, too. Intuitively, we think of being able to deduce Q from P. Formally, we define a deduction of Q from P as a sequence of statements ending with Q where each statement is either an axiom, the statement P, or is derivable from earlier statements by the rules of inference.

Actually, this is a proof of a theorem, where we allow P as an axiom. It can be shown $P \rightarrow Q$ is indeed a theorem if and only if (iff) Q is deducible from P. Our technique for proving theorems of the form $P \rightarrow Q$ is therefore to include the hypothesis as one of the statements in the sequence and to conclude the sequence with Q.

Valid arguments entail an argument is presented by a series of statements P1, P2, ...Pn followed by a conclusion Q. The argument represents a valid argument if the conclusion is a logical deduction of the conjunction $P1 \land P2 \land \ldots \land Pn$, or better yet, if $P1 \land P2 \land \ldots \land Pn \rightarrow Q$ is a theorem.

D.3. Predicate Logic

Predicate logic defines the formal logic system which allows quantified as well as unquantified statements. The logic system is also referred to as predicate calculus. Within this system, 'true' means valid or true in all possible interpretations. The goal is the axioms and rules of inference allow the proof of all valid statements, and only valid statements, as theorems.

E. Summary

This chapter explained the design philosophy behind the tutorial courseware and reviewed subjects from which most Computer Science GRE questions originate. The computer design section discussed the von Neumann machine, modern computers, central processing unit, memory systems, microinstruction sequencing, and Boolean algebra. The file structures section

covered different file organizations, blocking, buffering, file storage devices, file access methods, record keys, hashing, collision-handling techniques, linear probing, double hashing, and trees. The section on data structures provided information on the array, linked list, stacks, queues, trees, program performance measurements, algorithm characteristics, storage space allocation, and program performance improvements. Within the discrete mathematics section, logic, induction, recursion, propositional logic, and predicate logic are reviewed subjects. The overall discussion gave the reader a general understanding of computer organization, operating system architecture, file structures, data structures, and discrete mathematics.

CHAPTER 4

CONCLUSIONS & FUTURE WORK

The CS-GRE Tutorial Courseware version 1.1 laid the groundwork for a tutorial software package. The courseware seemingly prepares students for the Computer Science Graduate Record Examination. However, the sample size of 17 tested students is too small to make effective conclusions about the performance of graduates versus undergraduates. Yet, in general, the students scored less than 50% of the questions correct. Therefore, the tutorial courseware justifies all the work done.

Graduates performed better than the undergraduates; yet, they have seen more material than undergraduates. This shows undergraduates are expected to know the information on the exam. Therefore, we need software to supplement the undergraduate education at CAU for students. The material seen on the Computer Science GRE is taught to graduates and should be taught to undergraduates, as well.

Unfortunately, new students did not have the opportunity to review the software since the CAU Computer Science Department moved into a new building. The move disrupted the

student laboratory. The personal computers were not working for some time.

The next version will include a more attractive human interface design and more animation. Several changes are recommended below to allow easier human interaction. The changes involve using pop-up windows; standard Windows' windows; backtrack, forward, and history buttons; and, scroll bars. A self-testing capability can be included to permit the user to practice taking the exam and to obtain immediate test results.

Pop-up windows can be used to display the information in a simpler, yet efficient manner. These windows are different from the original version in that the windows can mostly cover the entire screen area. Figure 4-1 gives an example of the CS-GRE Tutorial Courseware window which can be implemented in the future. The tutorial courseware modules are listed. Figure 4-2 shows how the window appears after the Computer Design button is selected. The Computer Design menu of topics is displayed. Figure 4-3 shows the window after the selection for the Introduction button is made from the list of Computer Design topics. The topics under the Introduction selection are listed. Figure 4-4 gives the menu of topics falling under Boolean Algebra after the Boolean Algebra button is selected from the Computer Design menu as shown in Figure 4-2.

The window can occupy a particular area of the screen depending on that topic's nested level. They are standard

Figure 4-1.

CS-GRE Tutorial Courseware

Contents	Search	Forward	Back	Glossary	History

Contents for CS-GRE Tutorial Courseware

Computer Design

Data Structures

File Structures

Discrete Math

References

Figure 4-2.

CS-GRE Tutorial Courseware

Contents | Search | Forward | Back | Glossary | History

Computer Design

Introduction

Boolean Algebra

Memory

Computer Arithmetic

Gates

Microinstruction Timing

Microprogramming

von Neumann Architecture

Figure 4-3.

CS-GRE Tutorial Courseware

Contents	Search	Forward	Back	Glossary	History

Introduction

Overview	Functions
Economics	Structure
Price/Performance	Computer Family
Definitions	Hierarchy
Architecture	Central Processing Unit (CPU)
Organization	Control Unit

Figure 4-4.

CS-GRE Tutorial Courseware

Contents | Search | Forward | Back | Glossary | History

Boolean Algebra

Definitions

History

Postulates

Theorems

Microsoft Windows' windows. The window may be used like any other window in that the size can be either minimized or maximized, the window may be closed or opened, or several windows may be open on the screen at once. The text within the window can possibly be displayed in a different color according to the nested level on which the selected topic exists. The user can be given the option of selecting topics in one color from a list, while the nested pertaining choices are in a different list in another color.

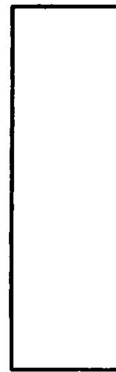
Scroll bars along the right side of the window can be added. Backtrack and forward buttons can be exercised to promote ease of use for thoroughly reviewing the material. The backtrack button will review information that has previously been covered in order from the last selection to the first. After the user finishes backtracking, the forward button will permit the user to review the window information selections from the current window to the last window selection made before backtracking. A history feature can be employed in case the user wants a list of the topics that have been selected for review within that particular session. A last-in-first-out stack mechanism is implemented to keep a record of the selections made by the user. This mechanism permits the operation of the backtrack, forward, and history buttons.

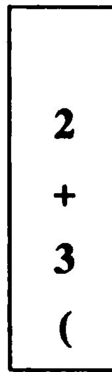
The presented information requires more animation through pictorials and demonstrations to break the monotony of reading

to prevent the user from becoming bored. Animations of different data structures performing various actual operations may be implemented. Array operations on vectors and their elements may be exhibited. Different tree order traversals may be displayed on different tree types. The creation, deletion, and building of linked lists may be shown. The linking of queue nodes may be seen after the user views their creation, removal, and other queue operations i.e. adding and deleting data elements. Stack operations of push and pop may be displayed so the user comprehends the stack functions. See Figure 4-5 for an example of a demonstration on stack operations. After the stack is created, the equation (3+2)+(5*6) is evaluated. The demonstration displays the stack functions: create, push, and pop.

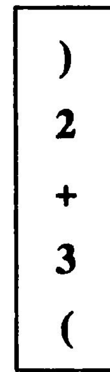
Different colors may be applied to these data elements within the various data structure animations so the user fully understands the distinction between which elements are created, added, removed, deleted, and operated upon. The user can select which data structure demo they wish to view; the operations that should be performed; whether the moved data should be stored on the screen, hidden, discarded, or filed; the maximum number of elements within some limit to be used; the kind of elements to be applied in an operation whether numeric, alphanumeric, or alphabet characters; and, the colors from a given list representing a particular operation on a data element and the element itself.

Figure 4-5.
Evaluation of Equation (3+2)+(5*6) Using a Stack

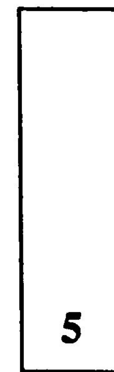

Creation of Stack



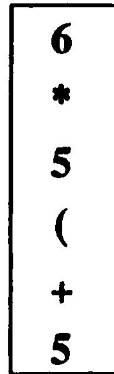
(3+2 equation elements pushed on stack



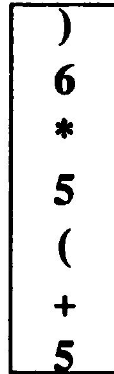
(3+2) popped off stack since ')' encountered



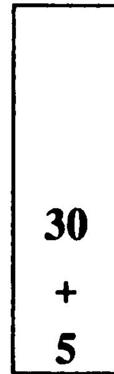
(3+2) is evaluated; 5 is pushed on stack



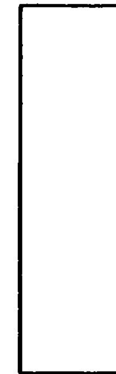
+(5*6 equation elements pushed on stack, too

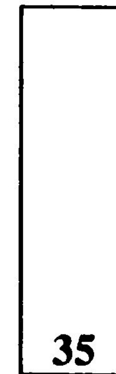


(5*6) popped off stack since ')' encountered



(5*6) is evaluated; 30 is pushed on stack


30+5 popped off stack; 30+5 is evaluated



35 is pushed on stack

Array vector elements may be represented by different shades. The user should be prompted to select the number of vectors and the number of elements to be used in each vector. Also, the type of elements within each vector should be specified.

Likewise, a program may be implemented to give a pictorial representation of the difference between static and dynamic allocations. With static allocation, one color may be used to show how space is set aside at the beginning of the program's compilation time. In reference to dynamic allocation, a different color may be used to show how storage space is created during the program's execution.

Further demonstrations may be given which show actual file manipulations for various file organizations. Examples may be given to obtain certain file information by performing detailed calculations. The retrieval and storage of file elements and files may be employed within the tutorial as pictorials. We intend to provide animation sequences of magnetic tape, floppy disks, hard drives, etc. rotating and spinning as information is stored, retrieved, or accessed.

Moreover, the relationships between the tutorial courseware and an actual Computer Science GRE may be explained. The tutorial presents information that is actually reviewed in the GRE. It thoroughly discusses many keyword definitions. A glossary can be included within the tutorial to further define highlighted terms and text. To correctly

answer many of the GRE questions, the test taker must have a thorough understanding of the terminology behind the computer science jargon evaluated in the exam.

The user may achieve an even better understanding if many more examples are included. Detailed explanations should be given along with the examples. More information should be added to the courseware for each subject that is already discussed in version 1.1. Additional courses may be analyzed to comprise an even more extensive tutorial. Pascal will be the fifth module. It is presently being constructed to provide insight on the programming language used as a teaching tool.

A self-testing capability may be in the next version. This would serve as an on-line GRE enabling a user to test oneself. The scores for the exam would be available immediately after the test was given. The on-line Computer Science GRE would be old exams which were given several years ago. The user will choose from five on-line exams. The same number of questions from a standard Computer Science GRE will be asked. There are 80 questions on the Computer Science GRE. Four hundred questions can be added to the data bank from which the user may study so we may obtain a larger sample size. There shall not be any time limit the first time the user practices taking the exam; however, the standard time limit will be followed the second time the user takes the same test within that particular session.

The following appendix is provided for the reader's information. Appendix A contains an executable copy of the CS-GRE Tutorial Courseware.

APPENDIX A

THE CS-GRE TUTORIAL COURSEWARE

SELECTED BIBLIOGRAPHY

Epp, Susan. Discrete Mathematics with Applications. New York: Prentice-Hall, Inc., 1986.

Gersting, Judith L. Math Structures for Computer Science. New York: Prentice-Hall, Inc., 1988.

GRE Computer Science. New Jersey: Educational Testing Service, 1989.

Lehmkuhl, Nonna Kliss. An Introduction to VAX Assembly Language Programming. St. Paul: West Publishing Company, 1987.

Sedgewick, Robert. Algorithms. New York: Addison-Wesley Publishing Company, Inc., 1988.

Tanenbaum, Andrew S. Operating Systems Design & Implementation. New Jersey: Prentice-Hall, Inc., 1987.

________. Structured Computer Organization. New Jersey: Prentice-Hall, Inc., 1990.