CONCEPTS

IN

COMPUTING

Dr. Ken Hoganson
Copyright 2005
Preface

Concepts in Computing provides an overview of the discipline of computer science, and explains the related and sibling disciplines; specifically Applied Computer Science, which is an emerging sub-discipline within computer science. The first chapter explains the relationships between the computing disciplines and engineering and mathematics, along with a discussion of the expectations of graduate students in the sciences. The majority of the text provides an overview of the key concepts and fundamental “great ideas” and innovations of computer science.

One of the purposes behind the development of this textbook is the need to facilitate the entry into and success of students studying the computer science discipline. A second purpose is to generate excitement, curiosity, and enthusiasm for learning more about the science of computing.

Students who are coming to computing from disparate disciplines, generally lack an overall appreciation of what computer science is, and how it has evolved from older disciplines, as well as how it has spawned new disciplines and connections with other fields. This textbook is intended to address this conceptual knowledge gap, while also acclimatizing new students to the demands and expectations of graduate work in computer science.

This text addresses a fundamental issue: How to engage students and generate enthusiasm and curiosity about their future studies in computer science. The excitement and profile of the high-tech computing fields have diminished somewhat of late, as the science and discipline has been maturing as part of the natural evolution of disciplines in the sciences:

1. At first, an emerging discipline is a mere off-shoot and sideshow of the mature and established parent discipline.
2. Then as a critical mass of important applications and foundations are discovered, the discipline continues to build followers and excitement. Funding and economic opportunities are ripe.

3. The critical mass of activity reach full-flower, and applications and technologies translate into products that change our everyday life, and reach into and change other disciplines of learning and study.

4. The rate of change in the discipline slowly begins to diminish, as the foundations of the science have been discovered, tested, and verified. Further changes tend to involve applications and specialized or splinter disciplines, rather than fundamental theory. Employment opportunities peak, as the science/technology is integrated into the economy.

5. Maturation – Perhaps computer science is somewhere between level 4 above and level 5. The bubble of opportunities and excitement in basic computer science is growing modestly, as excitement begins to involve applications of the science and specialization rather than new fundamentals.

All students considering the study of computer science are already familiar with the machines and are expert in using them for many tasks both significant and mundane: students are often exposed to computers in preschool. Yet students typically do not truly understand what the machines really are all about, and cannot therefore make informed decisions about whether they would like to invest their education in the computing discipline.

This text is intended to address both these issues with information and exciting ideas intended to draw the audience into the excitement of discovery and the desire to learn and study more. The approach to the multiplicity of topics is necessarily high-level, and is not a full and in-depth presentation in any single area. The overview presented here will set the foundation for future successful learning in each area, while “whetting the appetite”.

4
The text is appropriate for a first introductory course in computing at the undergraduate level, perhaps a CS-0 course that precedes the first courses in programming at many universities. This textbook is also appropriate as a supplemental textbook in a first course in computer programming, where programming is the primary learning objective of the course, as in a CS-1 course of a two-course sequence (CS1-CS2). The textbook is also particularly valuable as a supplement or independent study textbook for graduate students in computer science, who are approaching the discipline from another field, and lack the formal preparation of an undergraduate degree in computer science. In that role, the textbook fills in many gaps and makes connections in student knowledge about the discipline in strengthening the student’s base of knowledge for further study.

This introductory survey textbook requires only a modest understanding of mathematics, and saves many formal presentations for later study. Nor is a sophisticated knowledge of digital computer electronics or logic required as a prerequisite. Specific chapters will address these knowledge areas and fill-in the basics in each area.
TABLE OF CONTENTS

Preface

Chapter 1: What is computer science
   What is applied CS
   Expectations of graduate CS student scholars

Chapter 2: The computer system and basic processing function

Chapter 3: Programming

Chapter 4: The “Spiral” or “Evolutionary” Model of Software Development

Chapter 5: Complexity and algorithm analysis

Chapter 6: software engineering

Chapter 7: electrical circuits and transistors and Boolean Algebra
   7.1 Boolean Algebra
   7.2 De’Morgan’s Theorem

Chapter 8: machine architecture
Chapter 9: Cache

Chapter 10: Language Translation

Chapter 11: Parallel Computation

Chapter 12: Computability and Finite State Machines

Chapter 13: Artificial Intelligence

Possible future appendices:

Appendix: Introductory Programming

Appendix: Academic Integrity and Standards

Appendix: Number Systems

Appendix: Boolean Logic

Appendix: Flowcharts

Appendix: UML

Index
Chapter 1: What is computer science?

Computer science is a scientific and engineering discipline of study, which investigates all aspects of computing with computing machines. Compared to the natural sciences like physics and biology, computer science is different in that it investigates an artificial human-created world. In a natural science, the knowledge in the discipline progresses through the familiar “hypothesis and experiment” cycle, to acquire and validate new knowledge. In computer science, the discipline advances through proof of theory and demonstration, with experiments used to confirm theory after an appropriate device, software or technique has been created. Often, computer simulation is used to confirm the value of new ideas, techniques and theory, particularly when building a prototype for testing may require millions of dollars. Confirmation of theory through simulation is weaker than through experiment with the natural world, as the simulation itself is a creation that may contain errors or the assumptions and biases of the scientist, while true experiments that test against uncaring nature are truly impartial.

Computer science also shares much with mathematics, as mathematics also studies a world of logic, but the logic of numbers. Mathematics investigates concepts that are not explicitly part the physical world, but are applied in a conceptual universe of thought and the logic of numbers. Perhaps not coincidentally, the rules and laws of mathematics can be used to model the physical world with great effect, assuming that the mathematical model is correctly designed. Or perhaps put another way: we can model the world and its behavior using mathematics, and the predictions based on our mathematics hold true in the “real” and physical world. Computers and software also obey logical rules that are a part of the mathematical universe.
Computer science has much in common with engineering disciplines: a study of how to build and construct useful things in a particular domain. For computer science the construction domain is computing machines, logic, and programming.

The roots of computer science lie in both mathematics and electronics engineering, both of which spawned offshoot disciplines: computer science growing from mathematics focusing on theoretical concepts and software, and computer engineering diverging from electronics engineering, emphasizing the design and fabrication of computers. A more recent discipline has emerged from computer science called software engineering, which attempts to study how to build software with the certainty and predictability of the traditional engineering disciplines. Currently, much of software design remains more an applied art or craft than engineering, but the goal is to improve the accuracy of the software design process and the reliability of the resulting software, and to reduce the time required to complete a project.

![Computer Science and Related Disciplines](image)

Computer science and the success of computing hardware and software have spawned a number of even more recent inter-disciplinary fields:

- Information Systems: the application of computing technology to organizations and business.
• Geographical Information Systems: the application of computing technology for representing geographical information, used heavily by cities and governmental organizations.

• Bio-Informatics: the application of computing technology to biological systems, one area is representing genes and DNA.

• Bio-Medical Systems: the application of computing technology in medical domains, patient care and histories, for use by medical practitioners.

• Information Technology: the study of the management and maintenance of computing systems, including networking, operating systems, hardware, and software.

• Information Security and Assurance: an emerging field investigating the vulnerabilities and weaknesses of computing systems (both hardware and software) to threats, and how to insulate and protect systems and reduce their vulnerability.

There are various sub-disciplines within computer science itself, ranging from mathematically based, to engineering-oriented, to applied. There are areas called graph theory and formal languages, which are very mathematical. These cross-over knowledge areas are part of computer science because they have important applications in that discipline and contributed to the theoretical foundation of computer science, but otherwise would reside happily within mathematics. The study of computer architecture and high-performance computing is a sub-field within computer science that also exists in and overlaps with computer engineering. The study of database theory and systems is an application of computer hardware and software, which exists only because computers exist. Other fields in a constantly-expanding list include:

• Computer architecture
• Computer networking
• Formal languages
• Data structures
• Compilers
• Programming languages
• Graphics
• Artificial intelligence
• Database
• Software engineering
• Human-computing interaction

In summary:
• Computer Science: the study of the “science” of computing
• Computer Science: the engineering of applications and systems involving computing machinery
• Computer Science: a study of complexity and methods to manage complex systems.
• Computer Science: the study of systems (HW & SW), methods, applications, and approaches to managing information.
• A combination of all of the above.
What is applied computer science?

The computer science discipline is maturing, while continuing to develop specializations, and to spawn new disciplines and specializations. This fragmentation is a natural feature of a successful maturing discipline. The following figure illustrates the fragmented and overlapping nature of emerging computing disciplines.

Applied computer science focuses on areas within computer science that have direct and common-place application. This includes theoretical concepts, but does not emphasize theory. Applied computer science is a selected sub-set of overall computer science topics and knowledge-areas.
Expectations of Graduate Students in Computer Science

Graduate computer science students are expected to be able to perform at a high level in terms of tenacity and dedication in acquiring inherently complex knowledge. In addition, a competent analytical and mathematical ability is also required. Students must have mature and effective study skills.

Perhaps the most important requirement for computer science students is the ability to independently solve problems. This requires first, the ability to analyze and understand a problem, which may itself require extensive study and research. Then, the student must be able to break-down a problem into components, and develop a plan to solve the problem. In computer science, this process generally involves the synthesis of a design, software, or system.

The requirement that students master problem-solving skills is not unique to the computer science discipline, as problem solving is one of the hallmarks of the human experience, and is one of the abilities that separate us from animals. But the intensity and purity with which computer scientists operate in the world of problem solving has no parallel with the other disciplines except mathematics.

Applied areas in computer science are often mathematically less intense than in theoretical computer science areas, but can just as easily be as complex or more as well. Complexity is used in the computing field not as a synonym for difficulty, but with a more precise meaning. The sum of the number of things: components, modules, variables plus the number of their interactions and combinations, plus the way in which the complexity of a system grows as the system scales in size, defines the complexity of a system in the computer science arena.
For instance, a system that consists of a pail, a shovel, and a pile of sand is a simple system of only three components. But the number of ways those pieces can be combined, plus the number of ways they can be used together, plus the number of things that can be constructed with them, multiplies the complexity of the complete system many times over: The shovel can be used to fill the bucket to make sand-castle towers, full buckets make tall towers, partial buckets make smaller towers. The shovel can dig walls and motes to surround a sand castle. There are many ways to arrange towers and walls in a sand castle: square, rectangle, triangle, five-sided or more, nested castles within a castle, etc. The study of the many possible physical design layout alternatives, is the study of the TOPOLOGY of the system.

The number of steps required to make each design varies, and different arrangements of the steps can be more or less efficient. The study of instructions or recipes for building sand castles is very much like the study of program construction and algorithms.

**SAND CASTLE ANALOGY:**

Consider building a sand-castle consisting of four adjacent towers of full-(pail)-height.

**Algorithm 1:**

1. Create a level space for the castle
2. Repeat 4 times:
   i. Fill the bucket from the pile of sand
   ii. Upend the bucket to create a tower. Position the towers in a square pattern
3. repeat 4 times:
   i. Excavate a mote along one side of the tower

Algorithm 2:
1. Create a level space for the castle
2. Repeat 4 times:
   a. Excavate a mote along one side of the future castle, placing the excavated sand in a bucket.
   b. When the bucket is full, upend it to create a tower. Position the towers in a square pattern.

Clearly, Algorithm 2 requires less work, because there are fewer shovel-fulls to dig, since digging the mote is combined with filling the bucket for the towers. Children at play on the beach rarely consider the order of operations in their construction, unless they have built many sand castles and have learned from their experience.

Software systems in particular can become extraordinarily complex, with thousands and millions of components, and interactions that far exceed the capability of the human mind to encompass at one time. One of the preeminent themes of software engineering is the management of software complexity, and software design processes which compartmentalize and minimize the apparent complexity, leading to a faster construction with fewer errors, and a more complete project.
An analytical area within computer science called Algorithm Analysis, analyzes computer algorithms, to determine their performance and complexity, in relation to the number of data items that are to be processed.

Artificial intelligence includes a study of interesting and novel solutions/algorithms to a variety of difficult problems, which are often those that are complex in terms of the number of analysis steps to be processed by the computer running the software.

It should be apparent then to the reader, that computer science students must be interested in solving problems, thinking about solving problems, analyzing and comparing alternative solutions/approaches/models/algorithms for problems, and also must have the tenacity and persistence required to continue to work on difficult problems, even when that might require a fresh start or new approach to a problem.
Chapter 2: The Computer System and Basic Instruction Processing Function

Examining a computer system from a high-level point-of-view reveals that a computer has six basic components or categories of components:

- **CPU**: The Central Processing Unit. In common usage, this term is often used to describe the box or enclosure within which the computer components reside. But to a computer scientist, the Central Processing Unit is the processor chip that can interpret and execute the instructions in a computer program (software).

- **Main Memory**: This is working storage for programs and information, which is used while the compute is turned on and running. Main memory is generally not permanent or fixed storage, its contents are wiped clean when the machine is power-down.

- **Secondary Storage**: This term covers a variety of types of devices to store and retrieve data, information, and software programs. Devices range in speed, amount of storage, and cost. These can include hard-drives, floppy drives, ram-drives, CDs, etc.

- **I/O Devices**: This is a category of devices used to provide input to the machine, or display output for the user or to communicate with other computers. Devices in this category include: monitor, keyboard, mouse, network card, modem, camera, printers, scanners, etc.

- **Bus**: an electrical highway that is used to connect the components. The bus is multiple wires, so that many bits can be communicated between devices at the same time. It is typical for a computer system to have two or more buses.

- **Operating System**: This is the software program that makes the computing hardware usable. The operating system includes low
level software for controlling the hardware devices, as well as software for managing programs and the resources in the computer system. The operating system includes software that provides a user interface into the system – typically a Graphical User Interface (GUI) though other types of interfaces are possible. The operating system also generally provides a programming interface: a way for programmers to utilized portions of the operating system routines in developing software. Most of the operating system is software, which resides in secondary storage and is loaded into the computer’s memory when the computer boots-up. Usually, small portion of system code is stored in hardware in a chip, and may be called a Basic Input-Output System (BIOS), which contains instructions for loading and starting the operating system.

The following figure illustrates a high-level diagram of a simple computing system.

2.2 The Central Processing Unit (CPU)

The Central Processing Unit (CPU) is the part of the computer system that contains the logic used to execute or process instructions, which then cause the computer do work. The CPU is a single chip that is the master of all the other devices in the system and any secondary processors.

The CPU chip itself is quite small, the size of a fingernail or smaller. Because it is enveloped in a plastic or ceramic package, the CPU chip, when handled, appears larger, perhaps an inch by an inch in size, give-or-
take. The ceramic enclosure around the CPU chip protects the fragile CPU, connects input and output wires to pins on the chip for easy connection to the rest of the computer system, and is involved in transferring waste heat away from the chip. Depending on how fast the CPU operates, many chips can generate sufficient heat to cause internal failures unless the excess heat is dealt-with in some way.

Inside the CPU everything is stored as numbers, represented inside the computer with binary digits. The binary number system has only two digits: zero and one. This two-digit system turns out to be convenient to build and manufacture using modern digital electronics technologies. A binary representation of the number eighteen for instance, looks like this in binary:

\[ 10010 \]

All information that we manipulate with computers, including names, pictures and music, must be translated at some point by the computer and computer software, into simple binary representations. This translation can occur at a number of different times using a small number of ways to represent data and information in binary digits (0,1) called bits. The CPU operates on values represented as binary digits, and in fact, has no “knowledge” about the meaning of the binary numbers or what they represent in our world. It simple-mindedly manipulates the data represented by binary numbers as it is told to do so by its programs, which are also translated or converted into binary representations before the computer can work with them. The computer itself has no intelligence of its own, and all of its abilities are simply the result of capturing the intelligence and logic of its makers (both hardware designers and software programmers).

A simple Central Processing Unit contains three basic components. Modern advanced processors blur these distinct components and include additional performance-enhancing features. The three primary components of a simple CPU are:
1. **Registers**: a set of temporary storage locations for numbers while the CPU is working with them. If the intention is to add two numbers together, usually, each number would first be loaded from memory into a register, prior to adding process. Then, the result of the addition might be stored temporarily in a register, prior to being stored back to the computer’s main memory.

2. **Arithmetic Logic Unit (ALU)**: this is the logic that can do operations with binary numbers. In addition to basic math functions like addition, subtraction, multiplication and division, the ALU can also manipulate numbers in other ways and compare numbers together for equality and inequalities (greater than, less than).

3. **Control Unit**: This component is the logic that is written into the hardware chip that determines the sequence of events needed to process instructions. Things like: how to decode an instruction; and how to move data from one register to the ALU; where to put results from the ALU; and which instruction should be processed next, etc., are all functions that are encoded in the CPU chip as part of the control logic or control unit of the processor.

**2.3 Computer Instruction**

The computer is a machine that processes instructions, one at a time, over and over. Each single instruction has only a small effect, but the computer can process instructions at such a high rate of speed (millions or billions of instructions per second) that the computer can perform a tremendous quantity of computing work in a short period of time. At the lowest level, computer instructions are represented as numbers stored in using the binary number system. The computer’s Central Processing Unit (CPU) must examine each instruction that is to be processed, in order to determine what function must be performed (i.e. math operation or other), and what data will be manipulated. This is called decoding the instruction.
The following is an instruction that adds the contents of two registers together. This representation is an example of what an assembly language instruction looks like. Assembly language is a low level programming language.

**ADR R1 R2**

The assembly language code can be directly translated (by an assembler program) into machine code which the computer can process:

```
00 010000 0001 0010
```

The computer can process the machine code instruction, which is hard for humans to work with. Hence, the creation of assembly language and other higher level programming languages.

Machine code and assembly language is organized in an instruction format, with fields that indicate the operation to be performed (Op Code or Op), and operands that the instruction will operate on (R1 and R2 are registers).

```
2          6                 4               4            = 16 bits
```

```
|   00 |   Op |  R1 |  R2 |
```

R1 ← R1 Op R2

In the figure above, there are fields for the operation code (Op) and two operands (registers R1 and R2).
2.4 Fetch/Decode/Execute/Interrupt Cycle

The computer is a digital and electronic machine that processes instructions. Both the instructions themselves and the logic to process them should be considered “captured” human intelligence and logic and incorporated into a machine. The logic needed to process computer instructions can be quite complex. As an introduction, the following explains the cycle that the CPU repeats millions or billions of times per second in executing instructions.

1. **Instruction Fetch**: The next instruction to be process by the CPU must be fetched from the memory into the CPU, where it is stored in a registers expressly designed to hold instructions. Fetching an instruction will generally require a single cycle. On some systems with very large instruction formats, a number of processor/bus cycles may be required to fetch an instruction, depending on the width of the instruction and the width of the bus, measured in bits. The logic to do an instruction fetch is part of the control unit of the CPU.

2. **Instruction Decode**: Determine what the instruction is supposed to do, in particular, what operation or manipulation will be performed on the data, and what operands (data) are required before the instruction can execute. Usually, operands will be required to be loaded into registers in the CPU.

3. **Operand Fetch**: Operands that are not already stored in CPU registers may be loaded from memory into registers. If multiple operands are required for the instruction, some computer systems may require multiple fetches from memory to registers. How many operands are allowed, and whether they must already be in registers or can be in memory, are key design points in building processor chips. The control unit has the logic needed to fetch operands from memory and store them in registers.
4. **Instruction Execution:** After all operands are ready, and the what operation is to be performed has been determined, in this phase of the instruction execution cycle, the CPU Control Unit instructs the Arithmetic Logic Unit (ALU) to execute the operation on the operands.

5. **Check for Interrupts:** The last phase in the cycle has the CPU pausing to check before executing the next instruction, for signals requesting the CPU’s attention. Other devices, events, or inputs may require processing by the CPU, forcing the CPU to interrupt the current program it is executing, to do other things. When a CPU “services” and interrupt, it first saves its place in its current processing, then switches to running other programs and instructions to service the interrupt. Then, after the needed processing is complete, the CPU returns to the “saved place” in its processing and picks up where it left off.
2.4 Simple Computer Instructions

As an example, consider the simple computing function of adding two numbers together. The numbers to be added (operands) are stored in the computer’s memory. The result that is computed must also be stored back into memory. For the CPU to do the addition, the operands must first be copied from memory into registers.

    LOAD R1 Num1
    LOAD R2 Num2
    ADD    R1  R2
    STOR  R1  Result

The initial State of the computer system prior to executing any of these instructions is as follows:
    Register 1 empty
    Register 2 empty
    Number 1 = 4
    Number 2 = 5
    Result empty
The current “state” of the computer system can be conveniently displayed in a table of the following form, that shows both the contents of the CPU’s registers, and the contents of memory:

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Contents</td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After executing the first instruction: **LOAD R1 Num1**

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Contents</td>
</tr>
<tr>
<td>R1</td>
<td>4</td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After executing the second instruction: LOAD R2 Num2

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Contents</td>
</tr>
<tr>
<td>R1</td>
<td>4</td>
</tr>
<tr>
<td>R2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After executing the third instruction: ADD R1 R2

Note that the sum overwrites the temporary storage in R1.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Contents</td>
</tr>
<tr>
<td>R1</td>
<td>9</td>
</tr>
<tr>
<td>R2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After executing the fourth instruction: **STOR R1 Result**

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Contents</td>
</tr>
<tr>
<td>R1</td>
<td>9</td>
</tr>
<tr>
<td>R2</td>
<td>5</td>
</tr>
<tr>
<td>Result</td>
<td></td>
</tr>
</tbody>
</table>

This example illustrates the number of instructions are required to accomplish a modest amount of work. Some operations may require substantially more instructions than this simple addition example. The power of the computer is that it can process billions of instructions per second, each doing only a small amount of the task, but so many instructions can be completed in a small amount of time, that the computer can accomplish much.

The previous example also illustrates low level programming functions in an assembly language programming. Programming at this level is painstaking and tedious, so we have developed high(er)-level languages. In high-level languages, a single instruction can be written that accomplishes the work of the four assembly instructions in the example:
LOAD R1 Num1
LOAD R2 Num2
ADD R1 R2
STOR R1 Result

Can be accomplished in a high-level language with a single instruction or programming statement:

Result = Num1 + Num2
2.5 COMPUTER ARCHITECTURE LAYERS

A computer system can be viewed as being composed of a set of layers of functionality. This conceptual point-of-view of examining complex systems in layers, is used in many areas of computing, from networking protocols, to operating systems, to application program development. It is just one way to apply the divide-and-conquer approach to problem solving that allows us to look at small portion of a large and complex system, and then understand and design each portion individually.

In computer architecture, each layer is constructed on top of the layer before it, and each layer then becomes a foundation for the layers on top of it. The complexity of each layer below is abstracted for the layers built above it, so that the complexity and details can be "black-boxed" or hidden. Each layer uses the constructions of the previous layers as building-blocks to support the new layer’s construction.

This approach is similar to that used in software design, and the point is the same: managing complexity. The following layers create a computing system.

1. Transistors and Gates: constructing small devices called Logic Gates from transistors.
2. Simple Devices: building simple, elemental devices using gates as the construction components.
3. System Devices: building more complex devices (like registers, comparators, and memory) from
5. Instruction Set: specifying the CPU internal architecture and capabilities with the instruction set. The instruction set specifies the programmer’s interface to the hardware, both machine code and assembly language.
7. Operating Systems: provides basic functionality to the device, a user interface to the computer system, and a programmer’s interface to the system.

8. Distributed, N-Tier, Client/Server and Parallel systems: interconnecting many computing systems together to work cooperatively.

The understanding of each of the layers of a computer system and the interdependencies of the layers is the foundation for understanding and comparing different computer system designs. Note that there are architectural designs at multiple levels in a computer:

- Architecture of the CPU
- Architecture of the Computer System
- Architecture of the Operating System
- Architecture of the Computing Applications
- Architecture of the interconnections between one computing system and many others.

Each of these different layers represents a different focus of study, with separate courses and research tracks in each area.
Chapter 3: Programming

This section is not an introduction to a specific programming language, or a tutorial on how to program with coverage of syntax, but is instead, an overview and summary of computer programming concepts. Each different programming language has its own set of keywords for specific functions and operations, and has its own syntax and grammar for arranging programming statements (instructions) and multiple statements together. This overview presents common ideas that are universal to computer programming.

PROGRAMMING LOGIC STRUCTURES

There are three fundamental programming logic structures, but with variations for some, and seemingly infinite ways to combine these elemental logic structures.

**Sequence:** Simply the idea of executing instructions in order, starting with the first instruction. This simple concept is actually implemented in the Central Processing Unit’s hardware, which is set to bring in the next instruction after the current one.

**Selection:** Choices between alternative instructions and groups of instructions are implemented with IF statements. The IF statement will check or test some condition to see if it is currently true, and then execute one or another blocks of programming statements. The logic is:

```
IF (some condition is true)
  THEN do something
  ELSE do something else
```

IF statements can be connected together, in order to make more complex logic structures. Specialized form of IF statements have been developed to
replace often used structures with multiple IF statements. These are often
called CASE or SWITCH statements, and designed for use when there are
more than just two alternatives (THEN do something, ELSE do something
else).

**Iteration:** Repetition or repeating a set of instructions is called iteration.
Much of the power of computing comes from the ability to repetitively
execute simple instructions many times very quickly, thereby
accomplishing significant work. Iteration is also called “looping”.

Iteration is implemented with two general types of statements:

```
WHILE [some condition remains true]
  Some set of statements
```

An alternative structure changes where the testing of the condition is
located relative to the repeated statements:

```
REPEAT
  Some set of statements
UNTIL [some condition becomes true]
```

These fundamental logic structures have been utilized since the dawn of
computing with little change.
STRUCTURING PROGRAMS WITH MODULES

Computer programs can quickly grow in size to exceed the ability of the human mind to grasp in entirety at one moment. Using the “Divide and Conquer” strategy, we break large programs down into smaller pieces, where each piece may be simple enough to design and implement easily and reliably. This mechanism for dealing with complexity is critical to our ability to reliably design complex and sophisticated software in a reasonable amount of time.

The approach and process to building software in components and pieces has continued to evolve as our understanding of programming has grown, leading to new programming languages and new programming paradigms. Object-Oriented Programming is a relatively new approach, now widely accepted, which pushes the idea of construction by components (now called objects) to new, useful, and efficient levels.

It should be emphasized that the use of modules to structure a software program is for the convenience of the human programmer, and is not an artifact of the digital computer itself. But as software has grown in size and complexity, the need for programmers and software designers and developers to work as a team has grown as well. The ability to break-down a design into individual components which can be constructed in parts by different developers and then assembled into a whole, has been critical to the success of modern software application development. The process of architecting a software design focuses on the best way to structure a large software system consisting of components, and on determining and specifying the ways that those software pieces will communicate with and interact with each other.

Each programming language has its own syntax that directs how the programmer is to specify software components, modules and objects. The general modularization concepts that have evolved will be presented in roughly their historical order of evolution:
MODULE, SUBROUTINE, PROCEDURE

Here, software is grouped into blocks of programming statements, and identified with a name or label. The blocks of code that comprise the module or subroutine can be complex and may include many loops and if statements. Transferring the CPU’s execution of statements from the current location, to the statements in a module or subroutine, is a PROCEDURE CALL. After the CPU has completed the instructions in the subroutine, the flow of execution RETURNs from the subroutine back to the previous location. A module or subroutine may also CALL another module or subroutine, so the design if the flow of program execution between modules may become complex. Different programming languages use different keywords to define modules or subroutines, but a general example follows:

PROCEDURE [name] BEGIN
    Some set
    of programming
    statements
    RETURN
END PROCEDURE

The instruction that causes the CPU to execute the statements in the module might look like the following:

Some instructions before the subroutine call
[subroutine name]
some instructions executed after completing the call
After the subroutine or module has been defined, it is sufficient to simply refer to the module by its name, and the flow of control will transfer to the instructions in that module.

**RECURSION**

Recursion is a technique that can be used to solve some problems elegantly using repetitive procedure calls. Rather than calling different program modules, a module can call itself. Recursive modules and subroutines must be designed with some care, in order to allow the program to eventually complete and not run forever repetitively calling itself.

Classic programming languages like COBOL, FORTRAN, BASIC, C, and PASCAL implemented versions of the preceding logic structures and concepts.

**OBJECT ORIENTED PROGRAMMING**

The development of OBJECTs and Object-Oriented Programming is a relatively recent development and advance in computing programming. The idea grew out of the software engineering concepts of the 1970s and 1980s. Scientists were learning that when dealing with software and hardware devices for storing and communicating data, that it made sense to have one set of program modules for talking with that device, and to force all programs and components to work through those modules when accessing the device. In this way, a standard INTERFACE to the device or data structure was defined, which could then be used by any programmer and accessed by any module. By using a standard interface, errors when different implementations were defined from different modules, are avoided.

The first conceptual objects took this idea and made it formal. An object was defined as the data structure or device to be accessed, AND the defined interface modules that access it. In modern programming terminology, we call the interface modules or functions METHODS.
Scientists realized that this concept lead to other powerful higher-level programming concepts.

- Defining an object, allows the idea of creating more than one copy of an object, each with a different label or handle for accessing.

- Similar objects that do the same thing but operate on different types of data, can be organized into CLASSES. This allows objects to be created from one source programming definition, that can operate on different “things” (data, other objects, structures of various types).

- Classes can be designed in a hierarchy, where each class is a variation of a common parent type, and INHERITS various attributes from the common parent class, modified with specific unique attributes of its own.

These and other advanced programming concepts added a new level of power and sophistication to our toolbox of programming techniques, but not without a cost. Defining a program as a set of modules or components simplified the design of each component, but added another level of complexity: the architecture of the overall design of the modules. The power of Object-Oriented programming comes at a similar price: increased complexity in the relationships between objects, methods, classes, and inheritance between classes and derived classes.

Object-Oriented Programming in a specific language is beyond the scope of this book. Modern programming languages like JAVA, C++ and C# implement the object-oriented programming paradigm.
Chapter 4: The “Spiral” or “Evolutionary” model of software development.

Various approaches to developing software have been proposed, investigated, used, and adopted as standards. There is one tried-and-true approach that is particularly valuable for students, or anyone building a software system in a new application area, or tools that are new to the programmer: the “Spiral” of “Evolutionary” model.

Other approaches to managing the process of developing software, presume that there are few new technical problems never before having been faced, that will require research and the development of a unique solution. That is, most software development methods concentrate on managing a process where discerning and developing the user’s requirements and specifications is a primary problem, along with managing the complexity of a large software system with many components, and multiple programmers or engineers working together.

This leads to the observation that there are fundamentally two different problem types in software development: The first assumes that the technical issues are solvable and known, while the structure of the software and management of the work process are the difficult challenges: technically writing the code is not a challenge. The second problem type is when the issue of how to solve a problem using a programming language for a computer system, is the primary challenge.

Students of computer science who are learning new problem solving techniques, perhaps in new application domains, and perhaps with new programming languages are more often faced with primary challenges of the second type, with larger group projects that enter the domain of the first type. Professional programmers or software engineers with years of experience are generally faced only with problem types of the first domain.

When the how to solve a problem with computers and software is the issue, the process to be followed and overall structure of the resulting software, is generally not apparent at the beginning of the software development cycle. The programmer does not have an experience base
that can be used to project into the future how the work will progress. When this is the case, it is often useful to develop code using a cyclic model of analysis and development that “drills-down” to a final product in repeated cycles of stages yielding ever-increasing complexity and completeness. Each preceding cycle of stages can be considered as the development of a prototype, which is then used as the foundation which guides the development of the next level of system development and completeness.

This design approach is particularly useful in obtaining user requirements for a new system. When there is no existing software system to compare to and use as a basis for improvements, users who are not information technology savvy, often have difficulty in thinking about and defining how the application will be used in their work process. The development of a user interface prototype is often a useful first iteration in the evolutionary design process, in helping the eventual users understand and convey how the system should perform.

This approach may not always be as efficient as other software engineering approaches. A system prototype developed at a given level, will occasionally have to be significantly altered, upon investigation of the details and complexity in the next deeper design, as compared to an approach that builds the entire system with knowledge about the system known before-hand. But that is the problem: in new areas, with new tools, the critical fore-knowledge of the future system and structure is missing.

This chapter develops a useful problem-solving approach for new domains and new tools. The presentation of formal accepted software-engineering techniques is left for courses dedicated to the study of formal software engineering.
4.1 The “Spiral” of Evolutionary Approach to Program Design

This software design method has practitioners design software in iterative stages and possibly as separate components, toward building a completed project. The design and development process is broken down into the following phases, which is repeated in each cycle.

- **Analysis** of the requirements and problem to be solved.
- **Design** of the next iterative refinement of the current prototype.
- **Coding** writing the design in a programming language.
- **Testing and Debugging** multiple iterations of testing, diagnose problems, revise code, retest.

A project can evolve from a very simple structure (could be a shell or skeleton), where one piece at a time is added to the structure. Or, the project could begin with a prototype user interface, and gradually evolve. This breaks down the complexity of a large project into smaller "bites".

Often, in order to be able to implement/test/debug each piece must be able stand alone, at least temporarily during this iteration of the cycle. The use of "dummy" code as place holders for later development is useful.

Before the creation by parts commences, an overall problem analysis and partial design is needed. My version of the evolutionary or spiral model of software design differs somewhat from the originators version:

1. Overall Problem Analysis
2. High-level design of a general skeleton or structure.
3. Selection of a piece to work on
   1. Analysis of one piece of the overall problem
   2. Design for that one piece
   3. Code
   4. Test/Debug
4. Back to selection of the next piece to work on and/or high-level design.

As the implementation continues, it is possible that modifications to the overall design or project skeleton will be needed. This design philosophy
assumes that high level changes will be minimal - a complete redesign could be time consuming. If this method has a weakness, that is it.
Chapter 5: Complexity and Algorithm Analysis

Concept: it is useful to compare the performance of algorithms and computer architecture features, as measured against the resources required, and operations required. Generally, we are interested in how the resources or steps required, grows as the number of items involved grows. For instance, if we are searching through a list of items or records in a database, or sorting the same, or building a multiprocessor system, we need to be able to quantify the behavior of the system as it grows in size. It is typical for data storage requirements of a system to grow significantly throughout the lifetime of the system.

We consider a modest growth to be one where the number of steps required to run an algorithm, grows linearly with the growth of the number of items involved. More expensive algorithms or architectures growth patterns show an exponential growth in work or resources required, as the items involved grow linearly. A preferred solution might to have the work or resources growing in line with the Log₂ of the number of items involve.

Often, it is most important to understand the magnitude of the growth in work or resources (whether it is linear, exponential, logarithmic) rather than an exact equation or model that describes the growth perfectly. We use a notation that represents the ORDER of magnitude of the growth in work and/or resources as the size grows, which is called “Big O” or \( O \). For instance, an algorithm where the work grows as the square of the number of items or elements (n), is noted as \( O n^2 \).

Common growth models ranked in terms of the growth of work or resources required, as the number of items (N) grows from least to most (best to worst):

- \( O \) logn (logarithmic, generally log base 2)
- \( O \) n (linear)
- \( O \) n logn (logarithmic)
- $O(n^2)$ (Exponential)
- $O(n^3)$ (Exponential)

The following graph illustrates the different Orders. In this graph, the number of resources required or steps required in an algorithm is the vertical axis, while the number of items ($n$) involved is the horizontal axis. Both are represented on a logarithmic scale.
Linear growth in work/resources with a growth in n, is shown in the blue line at a 45-degree angle.

Logarithmic growth is shown as the red curved line, where the steps/resources required grows slowly as the number of items involved grows.

The steep exponential growth graphs on the left illustrate undesirable situations, where the work and resources required increase much more quickly than the number of items (n). Systems with these growth characteristics do not scale well.
Chapter 6: Software Engineering

The Need for Software Engineering

Software Engineering is the study of how to use a defined process to create software. It turns out that developing software is difficult for humans: psychology tells us that we can keep in our minds 7 - 3 different concepts at the same time. Software is much more complex than that: it is not possible to hold all the details of an entire program in our minds at the same time. This has serious consequences for our ability to develop correct programs in a reasonable time.

Our humanistic approach to this kind of problem is to use a “Divide and Conquer” strategy, to break down the problem into a set of manageable pieces. Then, each piece can be solved more easily as a separate construction, and the pieces then assembled to make the whole. This concept of developing a program in pieces (functions, modules, objects, classes), has a drawback, which is that it introduces new complexity in the interfaces and interactions between the separate components.

This problem-space is made more complex for large projects that require multiple concurrent software developers working as a team. Each developer must know how his component(s) are intended to integrate into the whole, and the interfaces and interactions between components must be defined and agreed-upon by the developers. In this way, a software engineer will have confidence that the components he/she is responsible for developing will actually “fit” with the rest of the project and work properly in the completed software application.

Software engineering then, is the management of complexity by following a defined process so that the decomposition of a problem into manageable pieces being engineered by different individuals, will result in the assembly into a correctly working solution. Software engineering deals with the communication and interactions between the software components, and the communication and interaction between a group of programmers developing the software solution.
Extreme programming (XP)

There are other approaches to developing software, one is called Extreme Programming (XP). Extreme programming shares ideas in common with the evolutionary or spiral design process. Rather than the formal regulated set of steps involved with formal Software Engineering, Extreme Programming focuses on producing and testing prototype solutions quickly, and extensive testing and communication with the end user. Extreme programming runs greater risk of pursuing dead-ends and having to back-up and rebuild as compared to traditional Software Engineering. But the costs of making a change to system requirements is much less than traditional Software Engineering approach, which starts with “frozen” specifications at the beginning. Traditional Software Engineering assumes that the entire project can be correctly anticipated prior to actually writing code, where as Extreme Programming is more tolerate of unknowns and evolving ideas and goals.

Overview of Software Engineering

Analysis, Specification, Development, Testing Cycles

An introduction to the discipline.

Under Development.
Chapter 7: Electric Circuits and Transistors and Boolean Algebra

Take from any number of sources, specifically including my lab manual and 3510 material.

- Transistors as a switch
- Boolean Representation
- Boolean Manipulations
- Simple storage device

The Transistor:

A transistor is a simple device that provides an important function: a control voltage input can control an output current. This allows small voltages to be amplified to control larger voltages and currents. Transistors allow the construction of feedback circuits where outputs feedback into inputs. Feedback circuits allow changes over time to be measured and controlled. The development of the transistor was one of the crucial technological developments of the 20th century, allowing tiny solid-state devices to be fabricated in the millions on a single tiny chip. Solid-state devices replaced vacuum tubes which have a filament much like a light bulb, and also like a light bulb, burn out over time.

Using the transistor as a switch is perhaps the transistor’s simplest function. A control voltage when applied to a transistor, can enable the flow of current through a circuit.

Note that the output voltage changes exactly with the control voltage. When a control voltage is present (a “1” in the table), a voltage at the output is detected. Essentially, the transistor is controlling the flow of electricity between the voltage source (V+), and the output.
The flow of the electricity controlling the transistor flows through a resistor (the jagged line), and the ground at the bottom of the figure. The resistor limits the amount of current that can flow through the resistor, as large currents can destroy the device. Similarly, there may be resistors in the circuit beyond the output (not shown in the figure) to limit the flow of current through that portion of the circuit.

The real power of the transistor is realized when more than one are used together to implement more complex logic. The following figure illustrates two arrangements using two transistors.
The figure on the left shows two transistors connected together in series. The series circuit forces the current to flow from the voltage source (V+) through both transistors (A & B) to reach the output and ground.

The table below the circuit illustrates the logic that is implemented by the transistors. The output "sees" a voltage ONLY when BOTH of the transistors are activated. This circuit implements a logic operation, AND.

The output voltage is present when inputs to A and B are both present:

\[
\text{OUT} = \text{A AND B}
\]

This circuit is called an AND gate.

The circuit on the right of the figure, shows two transistors connected together in parallel. Electricity can flow from the voltage source (V+) to ground through either of the transistors. The output "sees" a voltage when either input A or B, receive a voltage.
The table below the circuit shows that this arrangement of transistors implements a logical OR: the output is a 1, if either or both inputs are 1.

\[ \text{Out} = \text{A OR B} \]
This circuit builds an OR gate.

There are technical problems involved in this straight-forward way to build an AND or OR gate, which will be discussed later. The following alternative construction also creates useful logic circuits.

The following figure again illustrates two transistors wired together in circuits, both serial and parallel. Note the difference in the arrangements: the current-limiting resistor and the output have been moved to “ahead” of the transistors (closer to the voltage source V+).
The figure on the left, shows two transistors connected together in series. The series circuit forces the current to flow from the voltage source \((V+)\) through both transistors (A & B) to reach the output and ground.

The table below the circuit illustrates the logic that is implemented by the transistors in series. The output always “sees” a voltage EXCEPT when both of the transistors are activated.

This function is exactly opposite of that of the previous figure. Because the output is located “ahead” of the transistors, close to the voltage source \((V+)\). The only time the output does not see a voltage, is when BOTH transistors are turned on, creating an electrical pathway through each transistor from the voltage course to the ground. This alternative pathway through the transistors, is low resistance, and the electricity prefers this easier pathway from source to ground. Essentially, the output value is “shorted-out”.

The table shows that this circuit implements a logic operation that is the exact opposite of the AND, and is called a NOT AND or NAND. The output voltage is always present except when inputs to both A and B are present:

\[
\text{OUT} = A \text{ NAND } B
\]

The circuit on the right of the above figure, shows two transistors connected together in parallel with the output and resistor “ahead” of the transistors. Electricity can flow from the voltage source \((V+)\) to ground through either of the transistors. The output “sees” a voltage when neither input A nor B, receive a voltage. If either transistor is turned on (sees and input voltage), they create an alternative pathway between the voltage source and the ground, essentially “shorting out” the output so that it sees no significant voltage.

The table below the circuit shows that this arrangement of transistors implements the exact opposite of a logical OR, a NOT OR or NOR. The output is a 1, only when neither of the inputs are 1 and “turned-on”.

\[
\text{Out} = A \text{ NOR } B
\]
The following summary figure shows a set of symbols for each of these logical operations and gates, along with an algebraic representation as a function, and a truth table that illustrates the logic.

<table>
<thead>
<tr>
<th>Name</th>
<th>Graphic Symbol</th>
<th>Algebraic Function</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td><img src="image" alt="AND Symbol" /></td>
<td>( F = A \cdot B ) or ( F = AB )</td>
<td>( \begin{array}{c</td>
</tr>
<tr>
<td>OR</td>
<td><img src="image" alt="OR Symbol" /></td>
<td>( F = A + B )</td>
<td>( \begin{array}{c</td>
</tr>
<tr>
<td>NOT</td>
<td><img src="image" alt="NOT Symbol" /></td>
<td>( F = \overline{A} ) or ( F = A' )</td>
<td>( \begin{array}{c</td>
</tr>
<tr>
<td>NAND</td>
<td><img src="image" alt="NAND Symbol" /></td>
<td>( F = (AB) )</td>
<td>( \begin{array}{c</td>
</tr>
<tr>
<td>NOR</td>
<td><img src="image" alt="NOR Symbol" /></td>
<td>( F = (A + B) )</td>
<td>( \begin{array}{c</td>
</tr>
</tbody>
</table>

51
We can use transistors to build AND, OR, NAND, NOR, and Invertors. The Invertor (NOT) circuit simply inverts of complements the input to produce the opposite value as its output. It is a single transistor wired as a switch, but with the output and resistor located “ahead” of the transistor and close to the voltage source (V+).

In practice, there are technical reasons to prefer the use of NAND and NOR gates as opposed to AND, OR, and NOT gates. Manufacturing is simplified with NAND/NOR gates because either of those can be used to implement any other logic functions. The NAND and NOR gates are called a “Complete Set of Operations”, because either the NAND or NOR can be used to build any of the other logic gates. Manufacturing a single gate type on an integrated circuit simplifies its construction, making it cheaper to build and produce. There is also a problem with the straightforward construction of the AND gate circuit as illustrated above. A certain amount of control voltage will pass through the transistor, potentially generating a false logic. Some call this the “transistor bleed-through” effect. This problem is more pronounced if the gate is constructed with more inputs and transistors than just the two inputs and transistors illustrated in the figure.
7.1 Boolean Algebra

Conveniently, it turns out that there is an easy way to convert between AND/OR circuits and NAND/NOR circuits, one method is using Boolean Algebra. Boolean Algebra is named after its discoverer, mathematician George Boole.

In Boolean Algebra, a Boolean Variable can be in just one of two possible states: if can be either 0 or 1. Boolean variables are represented with capital letters of the alphabet.

The AND operation is represented in a Boolean equation with the multiplication symbol (this is called ‘overloading’ of multiplication operator – giving it two or more possible meanings, depending on the context.) The OR operation is represented with the addition symbol +.

A ANDed with B can be represented as A*B (usually noted as AB)

A ORed with B can be represented as A + B.

The next set of figures shows how the NAND and NOR gates are a Complete Set of Operations. The circuit diagrams show how multiple NAND or NOR gates can be assembled together to build any of the other logic gates. Even though more gates are required using this method, in many applications, the trade-off pays off against the ability to manufacture a chip with a single gate type on the chip.
Invertor

AND

OR

Figure A.2 The Use of NAND Gates

Invertor

OR

AND

Figure A.3 The Use of NOR Gates
7.2 De’Morgan’s Theorem

One of the most useful principles in Boolean algebra is De’Morgan’s Theorem. De’Morgan’s Theorem provides an easy way to convert a circuit between AND gates and NOR gates, and also converts conveniently between OR gates and NAND gates.

In the following algebraic equations, NOT terms or Inverted terms are represented with a horizontal line over the terms:

\[ \bar{A}B = \bar{A} + \bar{B} \]
\[ A + B = \bar{A}B \]

On the first line above, the left hand term is a NOT AND or NAND or A and B. A NAND B is equivalent to the ORing of NOT A with NOT B. Similarly, the second line shows how a NOR gate (A NOR B) is equivalent to NOT A ANDed with NOT B. The following table proves the equivalence through a process called perfect induction – listing the values for all combinations of 0 and 1 and demonstrating that the outcomes are exactly equal. The notation in the table uses single quotation marks (‘) to indicate the complemented terms.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
<th>A’</th>
<th>B’</th>
<th>A’ AND B’</th>
<th>A’ NAND B’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
The columns of particular interest are the column for A ORed with B (third column) and the last column A’ NAND B’. Reading the table from left to right, shows how the last column is derived. Demonstrating that these two columns are exactly the same for all combinations of inputs A and B, proves the validity of De’Morgan’s Theorem.

De’Morgan’s Theorem provides a way to convert between OR gates and NAND gates: an OR can be converted to a NAND if the inputs are complemented:

\[
A \text{ OR } B \equiv A' \text{ NAND } B'
\]

In the same fashion, an AND can be converted to a NOR, if the inputs are complemented.

\[
A \text{ AND } B \equiv A' \text{ NOR } B'
\]
Chapter 8. Machine Architecture

This section is under development.
Chapter 9: Cache

Historically, memory performance has not kept pace with processor performance, creating a disparity between two critical components of a computer system. Or, put another way, the cost in making memory perform at a certain level is much higher than the cost of making a processor that runs at that same speed. This disparity in performance potentially creates a performance bottleneck that limits the overall performance of a computer system to that of the memory system. In addition, the Von-Neuman computer system architecture exacerbates this disparity, by requiring multiple memory accesses for many instructions (perhaps an operand must be fetched from memory along with the instruction itself that will operate on it).

The use of pipelines as a technique that increases the number or of instructions which can completed in a given time period, further multiplies this problem. Increasing the number of instructions and operands that can be processed further increases the memory/bus bandwidth required to keep up with the demands of the processor. Here, bandwidth is used to in its common computing meaning, being the communication requirement in terms of bits per second, rather than its more traditional communications usage as a range of radio frequencies.

Clearly, computer system performance has in fact, continued to increase as the performance of processors also continues to increase, implying that there must be a practical solution to this performance disparity that ameliorates the performance bottleneck effects of this speed disparity. That solution is caching.

Caching is a technique that reduces the data transfer demands of the processor on the memory and bus, by holding the instructions and data values currently being used, in a small, relatively expensive, and very-high-speed memory called a cache. The cache is fast enough to keep up with the demands of high-speed processors, so accessing data and instructions in the cache does not slow down the processor. It is only when the processor requests access to memory that is NOT currently in the
temporary high-speed cache, that a request for data or instructions must go out on the bus to the system memory. The idea is that most of the time, the processor can get what it needs from the high-speed cache, and only occasionally it will have to slow down to load a new block of data or instructions from memory into the cache. Following that loading of a block of data or instructions into the cache, the processor can then proceed again at high speed.

Caching is truly an effective mechanism that has turned out to be highly effective. It has been applied to a number of different problems in computing where a disparity in speed of access exists:

- Caching blocks of memory to solve the disparity between the performance of the CPU and memory. This cache is located on the processor side of the bus, so that memory requests that are satisfied by the cache do not need to use the bus, freeing that resource for other uses.
- Caching hard drive data to mitigate the performance disparity between the hard-drive and main memory.
- Caching blocks of memory that are in use by multiple processors in a multi-processor or multi-computer parallel system, to mitigate the time-to-access (called latency) between memory local that is local to one processor, and is non-local to other processors and is accessed by distant processors at a higher latency than the access time from the local processor.
- Caching blocks of data in multiple processor system in order to reduce the bandwidth requirements on the interconnection network between the processors. Reducing the bandwidth requirements of the processors on the bus/memory system allows more processors to access a shared bus or interconnection network before the bandwidth demands overload the interconnection network bandwidth capability.
• Caching blocks of data in a communication system like the internet, to mitigate the effects of communication latency for data (or web pages) that are held remote from the requesting processor, and to reduce the overall bandwidth demands on a system with hundreds, thousands, or millions of computers.

• Caching information about running programs in a multiple-user system, where the operating system must share the CPU between many users and between many running programs. Caching is used to reduce the amount of time required to switch from one running program (called a process) to another process. The switching is called a “context switch”, and this special-purpose cache for this purpose might be called a register set or register file.

It turns out that caching in practice is highly successful, and its universal and wide application to many different problems where there is a performance disparity, or a need to reduce interconnect bandwidth, leads to the description of cache as the “universal computing band-aid”.

It is worth considering the principles behind caching and considering why it works so well. The foundation observations behind caching are the locality principles. These have to do with the patterns of access in memory locations over time by the processor.

**Temporal Locality**

The most recently accessed memory locations are more likely to be accessed again in the immediate future, than are less recently accessed memory locations.

**Spatial Locality**

Blocks of memory that are near to the most recently access memory block, are more likely to be accessed in the future than are blocks that are farther away.
What characteristics of programs when they are being executed yield these observations?

- Programs contain loops which repeat same instructions in the same areas of code (both locality principles)
- Programs are sequential: the next instruction is the most likely to be needed instruction (spacial)

Memory is cached in blocks – anywhere from .5 to 2K bytes per block is common. The two locality principles imply that it makes sense to store the memory blocks that are most likely to be accessed memory in the future in expensive high speed memory. This will yield a high probability that many accesses will be satisfied by the cache, yielding on average a faster memory response time.

These two principles say that the most recently accessed memory block is very likely to be accessed again, so it pays off to store it in the high speed cache. Less likely to be accessed blocks will be stored in slower memory, and perhaps even on secondary storage (disk) in a virtual memory system. Some percentage of memory accesses will be satisfied very quickly (low latency) in the cache, while the remainder will require access to the slower memory. The average rate of access will be between the low latency cache and the high latency memory. The greater the percentage that can be handled by the cache, the lower the average memory access latency.

The computer system has memory storage components that respond at different speeds, building inherent performance disparities. A memory latency hierarchy exists in computer systems:

- cache
- memory
- hard disk
- networked storage
backup tape storage

Cache improves the average performance of a system:

- The accesses or requests that are satisfied by the cache are termed “hits” in the cache. This percentage of fast accesses occur at the speed of the cache.

- The accesses or requests that are not satisfied by the cache (have to go out to memory or other storage) are termed “cache misses”. This percentage of requests for memory or memory access occur at the slower latency of the system memory.

All requests for memory are satisfied by either the cache or the system memory. This leads to a simple way to model the performance improvements that results from the use of cache.

\[ \text{Prop(hit)} + \text{Prob(miss)} = 1 \]

Average system request access performance then is:

\[ \text{Prob(hit)} \times \text{Time(cache)} + (1-\text{Prob(hit)}) \times \text{Time(miss)} \]

An example:

- Cache access time = 5 nanoseconds
- Memory access time = 50 nanosecond
- Cache hit rate = 90% (0.9)

**Average Latency** =

\[ \text{Probability(hit)} \times \text{Time(cache)} + (1-\text{Probability(hit)}) \times \text{Time(miss)} \]

\[ 0.9(5) + 0.1(50) \]

\[ = 4.5 + 5 = 9.5 \text{ns} \]
Observe that the average system latency with cache at 9.5ns is much better than the performance of the memory alone (50ns).

An obvious question is to wonder if high cache hit rates are reasonable in the real world? Experience has demonstrated that the answer to this question is yes! In fact, in many cases cache hit rates in the high 90s are common. The following calculations illustrate some average latencies with varying cache hit rates:

<table>
<thead>
<tr>
<th>Hit Rate</th>
<th>Latency Calculation</th>
<th>Result (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>0.50(5) + 0.50(50)</td>
<td>2.50 + 25 = 27.50ns</td>
</tr>
<tr>
<td>90%</td>
<td>0.90(5) + 0.10(50)</td>
<td>4.5 + 5 = 9.5ns</td>
</tr>
<tr>
<td>95%</td>
<td>0.95(5) + 0.05(50)</td>
<td>4.75 + 2.5 = 7.25ns</td>
</tr>
<tr>
<td>99%</td>
<td>0.99(5) + 0.01(50)</td>
<td>4.95 + 0.5 = 5.45ns</td>
</tr>
</tbody>
</table>

The effectiveness of caching also depends on the performance disparity between the cache and the memory. When there are large differences, there are large payoffs to using caching:

**Cache access time = 5 ns, Memory access time = 50 ns**

Cache hit rate = 95% (0.95)

Ave Mem Latency 0.95(5) + 0.05(50) = 4.75 + 2.5 = 7.25ns

*Large disparity: 50ns down to 7.25ns == 85.5% improvement!*

**Cache access time = 5 ns, Memory access time = 10 ns**

Cache hit rate = 95% (0.95)

Ave Mem Latency 0.95(5) + 0.05(10) = 4.75 + 0.5 = 5.25ns

*Smaller disparity: 10ns down to 5.25ns == 47.5% improvement.*

Other issues with caching must wait for future exploration:

- What about cache writes?
• How to manage the blocks in cache (block replacement)?
• How to find the desired memory block in the cache?

Caveats:
• The calculations show above are somewhat simplified:
  – Memory speed is time to read a single value
  – What about blocks of memory? – these take longer load times
  – Cache Writes? - write to cache
  – Finding a cache block, or freeing a spot for the new block
• The “miss penalty” could be much worse than our simplified analysis.

This section will end with an example of a Fully Associative Cache. Fully Associative Cache allows blocks to be stored anywhere in the cache. This is very flexible, but means we will have to do a comparison to find the particular memory block desired. Part of the address bits will be used as a Tag in order to identify the desired cache block.

Example: A computer system has

- 128 MB (million bytes) of Memory ($2^{27}$ bytes),
- 1MB of cache memory ($2^{20}$ bytes),
- 1KB cache and memory block size ($2^{10}$ bytes),

From these specifics, we can deduce some information about the cache and memory:

• The number of blocks in the cache = $2^{20} / 2^{10} = 2^{10} = $ one thousand or 1K blocks.
• The fraction of memory that can be held in cache = 1MB/128MB = 0.78%.
• Since a block can be stored anywhere in the cache, a method of identifying the specific cache block is needed. In Fully Associative Cache and memory address is divided into two fields, a TAG to identify the block and an offset (or internal block address) to specify the location desired within the cache. ALL of the cache block Tags must be compared with the tag bits from the address. This is reasonably expensive hardware which constrains the fully-associative method to be restricted to small caches.

The following figure illustrates the process of accessing a cache block from a 27-bit address. Since the cache block is 1KB \(2^{10}\) in size, 10 bits in the address are required to locate the byte within the block (the Block Offset). The remaining bits are used as the Tag bits to identify the desired block.

This diagram illustrates a parallel comparison of Tags of all 1000 blocks in the cache, with the Tag bits from the address. This would be a large and expensive associative cache.
Chapter 10: Language Translation

At their fundamental level, computers “understand” only numbers, and binary numbers at that. All other high-level representations of information must be translated into the simple binary representation system. A program that has been converted into binary code that the computer can run is called “machine code”.

Programming in machine code is inherently hard for humans, as machine code instructions are closely related to the internal design and construction of the processor itself. A machine code instruction is composed of a set of fields (like a record in a database), where each field has a specific use, and specific binary number codes in those fields represent different operations and events. For instance, the following illustrates a machine instruction format for a simple computer system:

```
2  6  4  4 = 16 bits
```

```
00 Op R1 R2
```

```
R1 ← R1 Op R2
```

Above the format is the number of bits allowed in each field. This instruction format has four fields, and the total number of bits in the instruction is 16.

The leftmost field indicates the specific instruction format: this is the layout for instructions of type 00. Since two bits are allowed for this field, there are four basic instruction types for this computer ($2^2=4$).

Counting from the left, the second field is used to indicate the operation code, or OpCode, which is, the specific operation the computer is to perform, like add, subtract, etc. Since there are six bits allowed for this field, there are ($2^6=64$) different instructions possible of type 00.

The third field from the left is 4 bits in size, allowing 16 combinations. This field represents the operand that the instruction will work with. For this type of instruction, the operands are all numbers stored in registers in
the CPU. Other formats will allow operands that are stored in memory which are referenced with an address. The fourth field is also a register operand.

The representation directly beneath the figure is a logical representation of what this instruction type does: It takes two operands stored in registers (R1 and R2) and combines them/modifies them in some way as specified by the operation code, with the result then being stored back into the same register and replacing, the first operand (R1).

Each field is stored as a binary (unsigned) value, so if the operation to be performed is to add the two operands together. For instance, the 6-bit OpCode for adding two registers together might be 010000.

Specifying, in binary, the operand registers is simple:

\[
\begin{align*}
R1 &= 0001 \text{ (binary value of 1 in 4 bits)} \\
R2 &= 0010
\end{align*}
\]

So the entire 16-bit instruction can be represented as bits for each of the four fields:

\[
00 \; 010000 \; 0001 \; 0010
\]

Working in binary is particularly difficult for humans, as it is tiresome, and we are very prone to errors at this level, so assembly language was developed to make the programming process easier. Assembly language itself did not add additional capabilities or features to the computer's machine code, it simply substituted a more human-friendly set of symbols or acronyms. For instance,

\[
\text{ADR R1 R2}
\]

is an assembly-language representation of the same machine-code instruction that was considered earlier. ADR means to add two registers together, while R1 and R2 specify the registers to be added. It is understood that for this particular instruction format (00), the result of the operation is always replaces the first operand by storing the result in the location of the first operand. Other more complex formats might have a
separate field for specifying where the result is to be stored. A program to translate assembly language into its proper machine code is called an assembler.

Later assembly languages began to evolve by offering improved features through a system of “macros”. Macros allow blocks or sections of often repeated assembly code to be abstracted, and represented with an abbreviation. The programmer could write ‘macro1:’ in place of the block of code containing many instructions. When translating the assembly language to machine code, the assembler will literally substitute the predefined block of instructions for each instance of the macro label, and then translate each instruction in that block.

Assembly language generates 1:1 machine code instructions when translated: one machine code instruction for each assembly instruction. As computing evolved, the need for more powerful and higher-level programming languages that are easier to work with became apparent. In high-level programming languages, a single instruction might be used in place of a hand-full or many machine-language instructions that are often used together in the same repetitive pattern. For instance, the instruction

\[
\text{Sum} = \text{Number1} + \text{Number2}
\]

might represent the four machine-language instructions to load the operands (Number1, Number2) from the computer’s memory into two registers (two machine instructions), add them together (one machine instruction), and then store the result back to memory at the location called Sum (one machine instruction).

Languages like C, C++, Cobol, Fortran, Pascal are all compiled languages, where the program that is run to translate the high-level program instruction to machine code is called a compiler. The compiler will run through the set of instructions written in the high-level language and will create a new machine-code program. The high-level program instructions cannot be run by the computer directly, but the new machine-code program that results from the translation by the compiler can be executed by the computer. The translation (compiling) need be done only once, and
the machine-code program that results may be executed as many times as desired. However, if the high-level language program is modified, the program must be re-compiled to create a new executable machine-code program.

An alternative translation mechanism for high-level languages is called an interpreter. An interpreter does not produce a machine code program to be run. Instead, the interpreter translates one instruction at a time, and the passes that translated instruction to the processor to be executed. Interpreted computer languages are traditionally slower than compiled language programs because each instruction must be translated while the program is being run, and instructions that are repeated may have to be retranslated each time. Interpreted languages include BASIC and VisualBASIC.

A modern take on interpretation has evolved for the Java language and Internet, where an interpreter/run-time-environment is installed on each computer that will run Java. The interpreter/run-time-environment is specific to the processor and machine code that it will run on. Then, any Java program can be run on any machine anywhere, independent of the machine code and processor type. This run-anywhere flexibility is very powerful and useful, and has contributed to Java’s acceptance as a modern programming language.
Chapter 11: Parallel Computation

Regardless how much faster each generation of computer chip design increases computing power, there is always a need for more. We seem to have an insatiable need for additional need to do more computing in less time. Increasing computing power enables the solution in a timely fashion, of problems that would previously be impracticable to solve. Increasing computing power makes it possible to enhance programs with additional features and capabilities that enhance the human-computer interface – like voice recognition interfaces, 3-D graphics, virtual-reality interfaces, and intelligent interfaces.,

Each generation of computer chip design improves performance by miniaturizing the transistors on the chip. Small transistors means that more can be packed in the same space, but also smaller transistors require less energy to operate. Transistors that need less energy to run (often revealed in the voltages necessary to run the devices – 5V, 3.3V, 2.2V, 1.1V, 0.7V) generate less heat as wasted energy, allowing a denser packing of transistors without temperatures reaching the failure point. Smaller transistors that operate on less power switch faster, and with shorter connections between transistors, allow for chips to operate at higher clock speeds.

But regardless of how the physicists and engineers improve our computing devices with each generation (early chips ran at 4000 cycles per second and less, current chips run at 3,000,000+ cycles per second), there is a need for more computing power. An alternative approach to achieving higher power is through parallel processing, where multiple processors are combined to complete a workload in less time.

Parallel computing systems also offer the possibility of a higher degree of fault tolerance. If a single processor in a system of multiple redundant processors fails, the entire system can continue to function, but at reduced power. Redundancy can be applied to many aspect of a computing system,
allowing systems with multiple processors, buses, disk drives, memories, caches, network interfaces, etc.

The following example illustrates the basic foundation concept behind parallel computing. In this example, there are four processors available for concurrent execution of this process. The process also happens to be dividable into ten discrete pieces, at most four of which can be executed in different processors concurrently. The process has some set-up work at the beginning that can only be executed on a single processor. Similarly, there is some work at the conclusion of the process where results are collated and consolidated which must be executed on a single processor.

**An example parallel process of time 10:**

- **S** - Serial or non-parallel portion
- **A** - All A parts can be executed concurrently
- **B** - All B parts can be executed concurrently

- All A parts must be completed prior to executing the B parts

**Executed on a single processor:**

```
S A A A A B B B B S
```

**Executed in parallel on 4 processors:**

```
A B  
A B  
S A B S
A B  
```

In the previous example, it is apparent that the process is composed of ten blocks of processing, each taking the same time duration. When executed on a single processor in serial, it takes ten time units to complete the work. But when taking advantage of the four processors available, then running
the same process on the parallel-processing multi-processor machine, the work is completed in only 4 time units.

It is useful to measure the performance improvement of a parallel machine. One measure used is called *SPEEDUP*, which is the ratio of the serial time divided by the parallel time.

\[
\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{10}{4} = 2.5
\]

\[
\text{Efficiency} = \frac{\text{Speedup}}{\text{processors}} = \frac{2.5}{4} = 62.5\%
\]

Another performance measure is *EFFICIENCY*, which gives an idea of how much of the available computing power is applied to the problem, as opposed to sitting idle. Efficiency is the ratio of the speedup obtained on a particular system, over the number of processors or processing elements on the system.

These concepts about parallel processing were first observed by Gene Amdahl, and formulated a different way, are known as Amdahl’s law. Amdahl’s formulation focuses on the fraction or percentage of the workload that can be done in parallel, and the percentage that must be completed in serial. These two fractions add up to 100%. The parallel percentage is executed on the parallel processors and will take proportionately less time to complete. The ratio of the serial workload divided by the parallel workload (composed of the parallel and serial fractions) defines speedup.

In the following formulation, the parallel fraction of the work is represented as alpha, and the serial percentage is then 1-alpha.
Amdahl’s law is not the final word on parallel processing speedup. It is a simple model that incorporates a number of implied assumptions. These assumptions mean that it is a correct and complete model only for that reduced environment. Specifically, the following assumptions are implied, which have represented opportunities for investigators to expand our knowledge of parallel processing beyond this baseline model:

- The workload consists of a single process to be executed.
- The process runs at just one constant Degree of Parallelism (always uses exactly n parallel processors for the portion that can run in parallel).
- The process and work to be completed is constant, and will not be scaled-up to take advantage of are larger number of available processors.
- Parallelism exists at a single level (Amdahl’s model is at the processor level). Recent work has identified parallelism at five distinct levels, and the incorporation of multiple simultaneous levels of parallelism yields greater speedups.

\[
Speedup = \frac{SerialTime}{ParallelTime} = \frac{1}{(1-\alpha) + \frac{\alpha}{n}}
\]

where

\[\alpha = \text{fraction of work that can be done in parallel}\]
\[1-\alpha = \text{the fraction of work that must be done in serial}\]
\[n = \text{number of processors}\]
Chapter 12: Computability and Finite State Machines

Computability is the study of problem types, primarily from a formal and mathematical point of view, and includes formal methods for representing problems. Of interest are the problem types that have no solution. Also of interest are the how resources required scale with the problem size.

Computability also includes formal models of computing, including finite automaton, push-down automaton, Turning machines, regular languages, and finite state machines.

Finite State Machines will be examined in this chapter as an example of this fascinating area of study. Finite state machines can be presented casually without an extensive formal mathematical presentation (reserved for later computing classes). Finite State Machines also have application in a number of areas in computing including computer architecture, data communications, and embedded systems. Thus, the student of computing is likely to encounter this conceptual model of computing in other areas of study, where it is used as a tool for defining and solving problems.

A Finite State Machine is a model for representing computing as inputs causing the current state to change to a new change. This is a graphical model, where states are represented as circles, and the inputs and the changes to new states represented by arrows which connect the states.

The following diagram is a very simple finite state machine. It has just two states, which represent two ideas:

- The most recent input to the machine is a zero.
- The most recent input to the machine is a one.

The inputs to the machine are clearly either zero or one, no other inputs are possible.
If the machine is in state $q_0$, and receives a “1” as its input, it will transition to state $q_1$. If the machine is in state $q_1$, and receives a “0” as its input, it will transition to state $q_0$.

If the machine is in state $q_0$, and receives a “0” as its input, it will stay in the same state $q_0$. If the machine is in state $q_1$, and receives a “1” as its input, it will stay in the same state $q_1$.

So regardless of the sequence of inputs of 0 or 1, the machine will always be in the state that indicates the last input received.

A more complex machine can be created which recognizes a sequence of inputs of zero and one. As an example consider a machine that “looks for” the pattern of inputs “101”:
This machine has four states, $q_0$, $q_1$, $q_2$, and $q_f$. State $q_f$ is the “final" state, that indicates that the looked for sequence of zeros and ones has been “seen” in the input stream.

From state $q_0$, the only way to get to state $q_f$ is through a sequence $1, 0, 1$. We can assign a “meaning’ to each of the states, in terms of how much of the intended sequence has been observed at this point:

State $q_0$: have received zeros so far.

State $q_1$: have received the first “1” in the sequence.

State $q_2$: have received a “10” so far.

State $q_f$: received the complete sequence of “101”.

76
When inputs are received out of sequence, they cause the machine to fall-back to other states. For instance, when in state $q_2$, the machine is “looking for” a final “1”. If it receives a “0” instead, it has now received two zeros in a row “00”, which is not a part of the desired sequence. The machine falls back to $q_0$, and must repeat the process of looking for the first 1 in an uninterrupted sequence of 101. The reader should examine each state, and consider each of the two possible inputs zero or one, to confirm that the machine is transitioning to the next correct state.

Finite State Machines are interesting because they can represent many useful machines. A classic example is a soda machine, that collects nickels, dimes, and quarters until it receives the cost of a soda, say $0.60, at which point it “recognizes” that a drink has been paid for, and dispenses one to the customer. This machine has more than two inputs, it has three: nickels, dimes, and quarters. Multiple combinations of inputs can total to $0.60: six dimes, two quarters and a dime, etc., so there are multiple ways to get to the “final” soda-dispensing state. A final complication for this machine, is that the customer could put in more money than exactly $0.60, say three quarters. This machine will need to give change as well as dispense soda as its outputs. The student is encouraged to play with this idea to build a Finite State Machine for this problem.

Finite State Machines (FSM) can be used in building computer systems at more than one level. Wherever a sequence of inputs is intended to cause specific outcomes, a FSM may be a useful design tool. FSMs are used to build simple digital logic circuits, where the FSM is a specification for what the hardware is intended to do, and a hardware implementation of the FSM is then constructed. FSMs can be useful in data communications and networking, where bits are being transmitted in a sequence, and parts of the bit stream transmit an address for the destination, the data, and other control information. FSMs can scan the incoming stream of bits for particular control sequences of bits that are used to signal the receiving device.
Section 13: Artificial Intelligence

Artificial intelligence is the investigation into building computing systems that appear to be intelligent. The study of artificial intelligence pre-dates the modern electronic computer, with early thinkers investigating the possibility of mechanical thinking machines. We don’t truly understand what intelligence is, but we hope we know it when we see it. To build an artificial intelligence then, is a clouded goal that we cannot see clearly. Since we do not truly understand the goal of achieving artificial intelligence, a more pragmatic goal has been accepted: Systems that exhibit behavior that mimics human decision making, and systems that behave rationally can be said to be artificially intelligent.

What is intelligence? We do not understand human intelligence: Does it arise autonomously from assembling a sufficient number of neurons – a sort of “the whole is greater than the sum of the parts”. If so, then building computers with sufficient processing capability to match the complexity of the human brain may result in generating an artificially intelligent computer.

Or perhaps human intelligent awareness is a process – a change of state over time. Consciousness generates detectable patterns of electrical waves, as neurons fire and communicate in a cyclic pattern, within the human brain. Perhaps establishing a similar cyclic pattern of repetitive waves of communication can create a system that becomes intelligent. As we think to ourselves in our language, we are formulating ideas and concepts that evolve over time. Perhaps a sufficiently complex system with multiple levels of feedback and sufficient memory to hold our current “state” will generate intelligent behavior.

We have long had mathematical models and systems for dealing with logic and logical problem solving. We can build systems that can process according to rules of logic. Then, by representing facts and relationships using symbols that can be entered into a computer’s memory, the computer can manipulate the symbols according to the rules of logic also entered
into the machine. Computers have had this type of capability for decades, but it is not considered to be proof of intelligence. The computer does not understand what the symbols mean, nor the meaning of any conclusions that it arrives at. The computer in this system is simply manipulating digital data according to a fixed set of rules.

The complexity and our lack of understanding of what intelligence is, has long befuddled investigators predicting intelligent machines always 10 years in the future. In fact, researchers in the 50s expected intelligent computers in a decade, in the ‘60s, they expected it in the ‘70s, in the ‘70s, it was predicted for the ‘80s. In the ‘80s, Japan began a national effort to create the fifth generation of computers – artificially intelligent, intending to leap-frog the competition and other nations. That major effort obtained some worthwhile results, and contributed to our understanding of artificial intelligence, but failed to create a new generation of intelligent computers, and so failed in its strategic purpose. [Perhaps Japan’s leadership in domestic robotics will achieve the strategic goal in the 2010s.]

The remainder of this chapter is a high-level overview of the various fields of study within Artificial Intelligence.
13.1 Problem Solving and Complex Algorithms

One early approach to A.I. centered around finding clever and efficient ways to solve very complex problems. Particularly problems that have no clear solution, or problems sufficiently complex as to require an excessive amount of time (perhaps years of processing time) to solve. Typically, these problems are manifested as a problem with no mathematical solution, but the number of possible solutions are too large for even a fast computer to examine them all in a search for the best solution.

A classic example is the traveling salesman problem. Here, a salesman must travel to visit clients in many cities, each separated geographically by a different number of miles. The problem or goal is to find the best order of cities to visit, which minimizes the number of miles traveled. The problem is easy enough to state, but for more than just a few cities, the problem becomes too large to be solvable in reasonable time using the “brute force” method of looking at all possible solutions and ranking them. This is because the number of possible solutions that would have to be examined increases at an exponential rate. With two cities there is just two alternatives. With three cities there are six, and four cities there are more:

<table>
<thead>
<tr>
<th>Cities:</th>
<th>1&lt;sup&gt;st&lt;/sup&gt;</th>
<th>2&lt;sup&gt;nd&lt;/sup&gt;</th>
<th>3&lt;sup&gt;rd&lt;/sup&gt;</th>
<th>4&lt;sup&gt;th&lt;/sup&gt;</th>
<th># of routes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two cities</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td>2 routes</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td>6 routes</td>
</tr>
<tr>
<td>A &amp; C</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Three cities</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, B, &amp; C</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>C</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Four Cities</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>24 routes</td>
</tr>
</tbody>
</table>

80
The relationship between the number of cities to be visited, and the number of possible routes to be examined is $n$ Factorial ($n!$), where $n$ is the number of cities to be visited. Informally, $n$ Factorial is the product of

<table>
<thead>
<tr>
<th>A, B, C, &amp; D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>D</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>
n terms, each of which is one less than the previous, i.e.: 3! = 3 X 2 X 1 = 6. 4! = 4 X 3 X 2 X 1 = 24.

\( n! \), for even larger numbers grows extremely large:

<table>
<thead>
<tr>
<th>N</th>
<th>N!</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
</tr>
<tr>
<td>8</td>
<td>4030</td>
</tr>
<tr>
<td>10</td>
<td>3,628,800</td>
</tr>
<tr>
<td>20</td>
<td>2,432,902,008,176,640,000</td>
</tr>
</tbody>
</table>

Clearly, even a modest number of cities (20) generate too many possible routes to exhaustively check (brute force) every possible solution to find the best one in a reasonable period of time.

The brute force method of checking every possible solution won’t do. The study of the clever ways to skip some of the possible solutions from needing to be checked, is part of this approach to A.I. Other approaches to these intractable problems are not intended to obtain the exact best solution, but will at least return a good solution but in a reasonable amount of time.

### 13.2 Game Playing

Computer game playing has long been a forum for testing our ability to create computer programs that behave intelligently. Early approaches to game playing involved quantifying the game as a discrete problem that can be represented on a computer, and then let the computer explore possible solutions. Simple games like tic-tac-toe and checkers have been solved.

More complex games like chess, are approached with a combination of brute-force searching for solutions, combined with ways to eliminate
sections of possible solutions from needing to be checked, along with observation of human chess players recognizing patterns in the arrangements of the pieces on the board, that lead to likely avenues to explore and negative outcomes not worth exploring.

Complex games like chess, experience the same kinds of combinatorial explosion in possible solutions (i.e. factorial and other) as the traveling salesman problem. The best computer players are now competitive with the very best human player, and will soon be superior. As an example, consider a game that might have 10 game pieces for each opponent. Each piece can move to any of 10 different locations on the game board. For the first player, there are 10 X 10 or 100 possible moves. For each of those possible moves, the opponent (who has 10 pieces each of which can move to 10 different locations) may make 100 possible moves as well. That leads to 100 * 100 possible situations after each player takes a single turn. A third turn will generate $100^3$ possible outcome situations. Granted, in a real game some moves may eliminate game pieces or some moves may be illegal, etc, which does cut down on the number of moves to explore.

There are War Game enthusiasts who re-fight historical military battles, involving sometimes as many as 100 or more game pieces (representing military units) for each side. Some game pieces may be able to move to any of 100 or more locations. Each military unit also has multiple actions it can take in addition to simply moving (attacking, rearming, reorganizing, etc). Clearly then, there are games that are even more complex (in terms of possible solutions to explore) than chess. Humans can deal with these types of games in a variety of ways without the brute-force power of computers, many of which are being mimicking in computer game playing. Some of these methods involve specialized knowledge about the situation (military strategy and tactics), while others involve ways to reason about uncertainty, and others involve finding good solutions without guaranteeing finding the best solution.
13.3 Machine Learning

If a computer program can reliably defeat the best human opponents in a complex game like chess, has it achieved artificial intelligence? Early researchers thought that accomplishment might signal the creation of artificial intelligence. But modern researchers do not believe this is artificial intelligence, but merely human intelligence in creating the rules and programming that allows the computer to play well. Human intelligence has been “captured” and installed in a computer program so that it plays the game the way the human specifies it should. Perhaps if the computer could learn to play chess on its own at the Grand Master level through machine learning, then one might maintain that the ability to self-learn how to play a complex game might signal intelligence. Machine Learning is the investigation into how a computer and program can learn.

How do humans learn? Educational experts will argue that there are a variety of styles of learning that people use, each person generally favoring one method over another.

A general approach to machine learning requires:

- An initial set of rules, or knowledge about the environment the machine will operate within.
- The ability to observe the environment and identify and detect objects in that environment.
- The ability to manipulate that environment.
- The ability to record new facts through exploration of the environment and changes that occur.
- The ability to generate and add new rules or learning to its existing set of knowledge about the environment and how to manipulate it.
13.4 Expert Systems

Expert systems are one of the first commercialized applications of A.I. The idea is to capture the expert knowledge of a human expert as a set of rules stored in a database of knowledge base. That stored expert knowledge is then used by the expert systems engine to answer questions, diagnose problems, and guide others in decision making. Expert systems have been applied in many areas, from diagnosis of disease, capturing the expertise of a head chef, to guiding technicians through the repair of a computer or other equipment.

An expert system consists of an expert systems engine and a knowledge base. The engine can be developed using conventional programming languages, or a commercial system can be purchased. Then the knowledge of the expert is captured and recorded in the knowledge base as a set of rules with some data. The process of capturing the knowledge of an expert as rules requires expertise in working with the expert systems engine, depending on the sophistication of the engine. A simple system will require the formulation and entering of the rules, and then organizing the rules into a structure that determines which rule is active under specific circumstances. This can be a bit tricky and may use non-standard programming control structures.

13.5 Neural Networks

Neural Networks attempt to build logical structures that function the way neurons in the brain function, with many connections between neurons. Neural networks can be implemented entirely in software, as a simulation of a neural network, allowing the network to be changed and rewired as needed. Alternatively, a neural network can be implemented with hardware and processing tailored for this function.

The simple function of neurons is to take in electrical signals as inputs, and output electrical signals as outputs. Neurons have a threshold level of
input required before firing the output. The programming or training of a neural network consists of the pattern of connections between the neurons coupled with the training of the threshold level of the neurons.

Neural networks require a training period and process where responses that are correct are re-enforced, and incorrect responses are attenuated.

Neural networks have been applied in a number of domains. Since the neural network is analogous to our “wet-ware” neural networks in our brains, one might surmise that neural networks offer the best opportunity to achieve artificial intelligence. That remains to be seen, the largest impediment to progress is simply the cost and difficulty of building systems with a sufficiently large number of neurons. The human brain has trillions of neurons.

Neural networks are excellent at recognizing patterns (as is the human brain), and so are useful in computer vision, voice recognition, and etc.

13.6 Fuzzy Logic

Fuzzy logic is reasoning with probabilities. Humans can deal with uncertainty quite well in our decision making, while logic and rule based systems often stumble with things and relationships are known only with some probability.

Human ability to use fuzzy logic reasoning is one of the ways that we use to deal with the complexity of problems where a combinatorial explosion excludes direct analysis for a solution. For instance in the game of chess, humans experts become adept at understanding the patterns of relationships of pieces on the board. Humans do not explore every possible move many turns in the future, but instead guide the pattern of the pieces on the board toward a pattern that is favorable to their side. In making these kinds of decisions, humans will use reasoning about the probable moves of their opponent, since future moves are unknown. In fact, expert human chess players have used this pattern recognition ability to defeat the best machine chess players, by learning the patterns of
responses that the computer will make in various situations, and taking advantage of that knowledge to control the course of the game.

Basically, every fact stored in a computer using fuzzy logic is associated with a probability of it being true (which could be 100%), but the rules and relationships that the computer uses to reason about the environment also accommodates probabilities of the inputs or premises of a rule, and then generates a probability of the conclusion or outcome of a rule being true.
Appendix B: Academic Integrity and Standards

Sourcing: define plagiarism, copying.
Collusion on exams or homework, explain limits etc.
Under Development!