Von Neumann Computers

Rudolf Eigenmann
Purdue University
eigenman@purdue.edu

David J. Lilja
University of Minnesota
lilja@ee.umn.edu

January 30, 1998

1 Introduction

The term *von Neumann computer* has two common meanings. Its strictest definition refers to a specific type of computer organization, or "architecture," in which instructions and data are stored together in a common memory. This type of architecture is distinguished from the "Harvard" architecture in which separate memories are used to store instructions and data. The term "von Neumann computer" also is used colloquially to refer in general to computers that execute a single sequence of instructions, which operate on a single stream of data values. That is, colloquially, von Neumann computers are the typical computers available today.

There is some controversy among historians of technology about the true origins of many of the fundamental concepts in a von Neumann computer. Thus, since John von Neumann brought many of these concepts to fruition in a computer built at the Princeton Institute for Advanced Study (see Figure 1), many people in the field of computer science and engineering prefer to use the term "Princeton" computer instead of "von Neumann" computer. The intention of this terminology is to acknowledge the important concepts introduced by many other individuals while not over-emphasizing von Neumann's contributions. Recognizing that many people in addition to von Neumann contributed to the fundamental ideas embodied in this widely adopted computer architecture, this article nevertheless uses the colloquial version of the term von Neumann computer to refer to any computer with the fundamental characteristics described in Section 3. The term "Princeton architecture" is then used to distinguish between computers with the split (Harvard) and unified (Princeton) memory organizations.

History

The von Neumann computer concept was developed in the 1940s when the first electronic computers were built. Nearly all modern computers are based on this *stored program* scheme, in which both

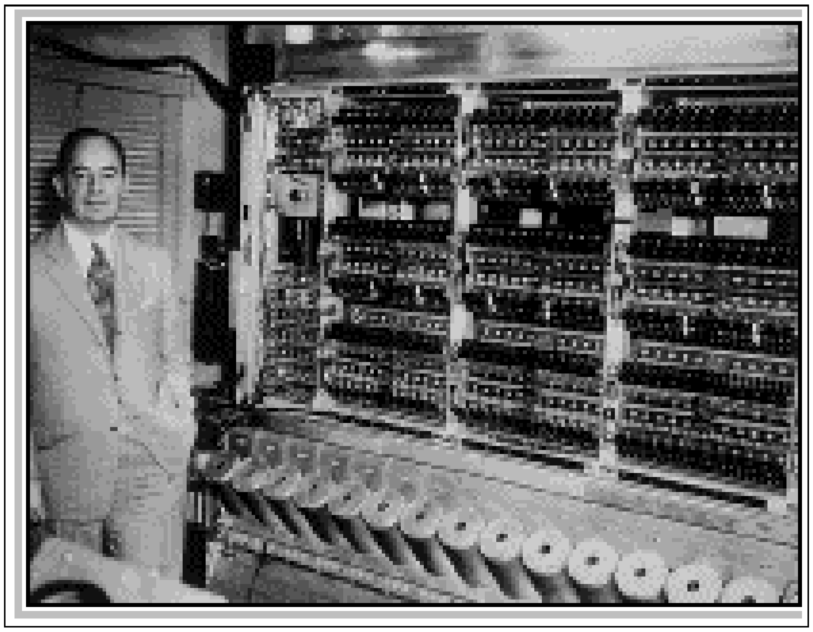

Figure 1: John von Neumann in front of the computer he built at the Institute for Advanced Study in Princeton (Courtesy of the Archives of the Institute for Advanced Study).

machine instructions and program data are stored in computer memory in the same manner. After the 1940s the computer industry began a rapid development with the speed and cost of computer systems improving by a factor of two every two years. Amazingly, this trend has continued, in principle, through today.

Computer applications initially served the needs of the military. They soon found their way into the commercial market, however, where they revolutionized every business they encountered. The development of microprocessors brought the von Neumann computer onto the desks of secretaries, the counters of sales clerks, the office tables of homes, and into small appliances and children's games. Accompanying organizations were created to support the computer era in various ways. Notable among these are the many computer science and engineering departments established at universities and two professional societies, the IEEE Computer Society and the Association for Computing Machinery (ACM).

The von Neumann Computer Architecture

The heart of the von Neumann computer architecture is the Central Processing Unit (CPU), consisting of the control unit and the ALU (Arithmetic and Logic Unit). The CPU interacts with a memory and an input/output (I/O) subsystem and executes a stream of instructions (the computer program) that process the data stored in memory and perform I/O operations. The key concept of the von Neumann architecture is that data and instructions are stored in the memory system in exactly the same way. Thus, the memory content is defined entirely by how it is interpreted. This is essential, for example, for a program *compiler* that translates a user-understandable programming language into the instruction stream understood by the machine. The output of the compiler is ordinary data. However, these data can then be executed by the CPU as instructions.

A variety of instructions can be executed for moving and modifying data, and for controlling which instructions to execute next. The collection of instructions is called the *instruction set*, and, together with the resources needed for their execution, the *instruction set architecture (ISA)*. The instruction execution is driven by a periodic *clock signal*. Although several substeps have to be performed for the execution of each instruction, sophisticated CPU implementation technologies exist that can overlap these steps such that, ideally, one instruction can be executed per clock cycle. Clock rates of today's processors are in the range of 200 to 600 Mhz allowing up to 600 million basic operations (such as adding two numbers or copying a data item to a storage location) to be performed per second.

With the continuing progress in technology, CPU speeds have increased rapidly. As a result, the limiting factors for the overall speed of a computer system are the much slower I/O operations and the memory system since the speed of these components have improved at a slower rate than CPU technology. *Caches* are an important means for improving the average speed of memory systems by keeping the most frequently used data in a fast memory that is close to the processor. Another factor hampering CPU speed increases is the inherently sequential nature of the von Neumann instruction execution. Methods of executing several instructions simultaneously are being developed in the form of parallel processing architectures.

Types of von Neumann Computers Today

Today, the von Neumann scheme is the basic architecture of most computers appearing in many forms, including supercomputers, workstations, personal computers, and laptops.

Supercomputers The term supercomputer has been used to refer to the fastest computer available at any given time. Supercomputers use the fastest hardware technology available. For example, when the Cray-1 computer was introduced in 1976, it achieved a clock rate of 80 MHz, which was much faster than clock rates in conventional electronics technology at that time. In addition, its *vector operations* could process an array of data as one instruction, leading to significant speed increases in applications that exhibited certain regular characteristics. Such characteristics often can be found in science and engineering applications, which became the primary application domain of supercomputers. Several supercomputer generations following the Cray-1 system maintained a large performance lead over their competitors, which were primarily the machines based on fast microprocessors. Developers sought to increase the speed further by developing *parallel computer architectures*, which can process data using several processors concurrently. However, due to the fast progress in microprocessor technology, the speed advantage of supercomputers reduced enough that customers were no longer willing to pay the significantly higher prices. By the mid 1990s, most of the former supercomputer vendors merged with microprocessor manufacturers.

Workstations Workstations are relatively powerful systems that are used primarily by one person. They usually fit on or under an engineer's desk. Workstations were an alternative to *mainframes* and *minicomputers*, which served a number of users and were placed in a computer center or in a department's computer room, respectively. When introduced, workstations were substantially more powerful than personal computers (PCs), due to their faster processor technology, greater amounts of memory, and expensive peripheral devices. Typically, workstations are connected to a powerful network that allows communication with other computers and the use of remote resources, such as large storage devices and high-speed compute servers. Through this network, the computers and their peripheral devices can be accessed by several users, in which case one may use the term *server* instead of *workstation.* Workstations are typically used by scientists and engineers who run compute-intensive applications. The predominant workstation operating system is the *UNIX* system (see also *UNIX).*

Similar to the development of the supercomputer market, workstations experienced increasing difficulties in maintaining their user communities against the overpowering market of PCs, which offer an inexpensive and almost infinite range of utilities and conveniences. Although the large installed base of workstation infrastructures cannot be replaced as easily as supercomputers could, the advantages of PCs over workstation environments in beginning to have an impact. For example, some experts see a trend of replacing the workstation operating system *UNIX* with Microsoft's *Windows NT.*

Personal Computers, PCs Personal computers had existed several years before the announcement of the "IBM PC" in 1981. PCs started out as economical computer systems for small business applications and home use since their price range allowed for fewer peripheral devices than typical workstations. Initially they were desk-top, single-user systems with no network support. Although announced and manufactured by IBM, PCs included a processor from Intel and an operating system from Microsoft. The huge market that PCs have found have made the prices even more competitive and have made it possible to add peripheral devices and network support that are typical of workstation setups. As a result, their application range has become huge. Parallel and network-connected PCs are now becoming commonly available and are competing with one of the last bastions in the supercomputer realm. Newest generations of PC operating systems, such as Windows NT, now include multiuser and multitasking capabilities, offering the support that used to be associated with UNIX-based machines.

Laptops Computers that are light and small enough to carry from place to place began to appear in the mid-1970s in the form of pocket calculators with programming capabilities. Laptop computers are advanced versions of this concept. Today they include capabilities that are no different from mid-size PCs. Low-power devices, flat high-resolution color displays, miniature disks, and CD-ROM technology make laptop computers powerful, portable additions, or even alternatives, to fixed office

PCs. Connections with the main office computers are typically provided through plug-in network connectors when in the office, or through modem connections, possibly via portable phones.

Applications

Computer applications have emerged in every conceivable area. They have penetrated equally into commercial, engineering, science, home, and hobby activities. Thanks to Internet connections (see *Network Computing*), computers can be setup in practically any location on our planet and applications can be used and controlled remotely.

Computer applications serve numerous purposes. They provide convenience (e.g., composing a letter); they allow information to be retrieved (from the Internet or from local databases); they support online record keeping and decision making (e.g., inventory control and automatic orders); they control peripheral devices (e.g., the control of assembly lines or robot devices); and they process signals (e.g., audio, video, radar, or signals from outer space). In addition, one can create experiments "in the computer" by computing and simulating the exact behavior of the experiment's substances. This area of computer applications will be described in more detail in Section 6.

There are virtually no limits to computer applications. However, in practice, computer speeds, the development costs for computer applications, and the accuracy with which a problem in the real world can be represented and modeled in the computer, creates bounds. One of the hardest limitations is that of software development costs. Measured productivity rates for new software are very low (e.g., a few programming lines per day, if one factors in the entire software development process). The search for more advanced ways of specifying and coding an application in a computer is ongoing and is perhaps the greatest challenge for the future of all types of computers.

2 Historical Perspectives

2.1 Evolution of the von Neumann Computer

Computer Technology Before the Electronic Computer

Ideas of an analytical machine to solve computing problems date back to Charles Babbage around 1830, with simple pegged-cylinder automata dating back even significantly further [20]. Babbage described four logical units for his machine concept: memory, input/output, arithmetic units, and a decision mechanism based on computation results. The latter is a fundamental concept that distinguishes a computer from its simple sequencer predecessors. While Babbage's machine had to be constructed from mechanical building blocks, it took almost 100 years before his ideas were realized with more advanced technology such as electromechanical relays (e.g., the Bell Labs Model 1 in 1940) and vacuum tubes (ENIAC in 1946).

The Birth of Electronic Computers

ENIAC, the Electronic Numerical Integrator And Computer, is considered to be the first modern, electronic computer. It was built from 1944 through 1946 at the University of Pennsylvania's Moore School of Electrical Engineering [24]. The leading designers were John Presper Eckert Jr. and John William Mauchly. ENIAC included some 18,000 vacuum tubes and 1,500 relays. Addition and subtraction were performed with 20 accumulators. There also was a multiplier, a divider, and square root unit. Input and output was given in the form of punch cards. An electronic memory was available for storing tabular functions and numerical constants. Temporary data produced and needed during computation could be stored in the accumulators or punched out and later reintroduced.

The designers expected that a problem would be run many times before the machine had to be reprogrammed. As a result, programs were "hardwired" in the form of switches located on the faces of the various units. This expectation, and the technological simplicity driven by War-time needs, kept the designers from implementing the more advanced concept of storing the instructions in memory. However, in the view of some historians, the designers of ENIAC originated the *stored-program* idea, which now is often attributed to John von Neumann.

Von Neumann's Contribution

John von Neumann was born in Hungary in 1903. He taught at the University of Berlin before moving to the United States in 1930. A chemical engineer and mathematician by training, his well-respected work in the U.S.A., which was centered around physics and applied mathematics, made him an important consultant to various U.S. government agencies. He became interested in electronic devices to speedup the computations of problems he faced for projects in Los Alamos during World War II. Von Neumann learned about ENIAC in 1944 and became a consultant to its design team. His primary interest in this project was the logical structure and mathematical description of the new technology. This interest was in some contrast to the engineering view of Eckert and Mauchly whose goal was to establish a strong commercial base for the electronic computer.

The Development of EDVAC, a follow-up project to ENIAC, began during the time that von Neumann, Eckert, and Mauchly were actively collaborating. At this time, substantial differences in viewpoints began to emerge. In 1945, von Neumann wrote the paper "First Draft of a Report on the EDVAC," which was the first written description of what has become to be called the von Neumann stored-program computer concept [2, 10]. The EDVAC, as designed by the University of Pennsylvania Moore School staff, differed substantially from this design, evidencing the diverging viewpoints. As a result, von Neumann engaged in the design of a machine of his own at the Institute for Advanced Study at Princeton University, referred to as the IAS computer. This work has caused the terms *von Neumann architecture* and *Princeton architecture* to become essentially synonymous.

The Stored-Program Concept

Given the prior technology of the Babbage machine and ENIAC, the direct innovation of the von Neumann concept was that programs no longer needed to be encoded by setting mechanical switch arrays. Instead, instructions could be placed in memory in the same way as data [10]. It is this equivalence of data and instructions that represents the real revolution of the von Neumann idea.

One advantage of the stored program concept that the designers envisioned was that instructions now could be changed quickly which enabled the computer to perform many different jobs in a short time. However, the storage equivalence between data and instructions allows an even greater advantage: programs can now be generated by other programs. Examples of such program-generating programs include compilers, linkers, and loaders, which are the common tools of a modern software environment. These tools automate the tasks of software development that previously had to be performed manually. In enabling such tools, the foundation was laid for the modern programming system of today's computers.

Of comparably less significance was the issue of "self-modifying code." Conceivably, programs can change their own instructions as they execute. Although it is possible to write programs that perform amazing actions in this way, self-modifying code is now considered a characteristic of bad software design.

2.2 History of Applications

While from a 1990s perspective it is evident that every computer generation created new applications that exceeded the highest expectations, this potential was not foreseeable at the beginning of the computer age. The driving applications for ENIAC, EDVAC, and the IAS computer were primarily those of military relevance. These included the calculation of ballistic tables, weather prediction, atomic energy calculations, cosmic ray studies, thermal ignition, random number studies, and the design of wind-tunnels.

Although the ENIAC designers, Eckert and Mauchly, recognized the importance of a strong industrial base, actually creating this base was difficult. Initially, the Army not only funded the development of the new technology, but it also sponsored customers to use it. As in many other disciplines, applications in research and government agencies preceded commercial applications. The introduction of computers in the late 1940s started a decade of initial installations and exploration by commercial companies. An important machine at that time was the IBM 604, available in 1948, which was similar to ENIAC's design. It included 1,400 vacuum tubes and could perform 60 program steps (see [3] for a description of early computer installations). Computer customers in this era were manufacturers of aircraft and electronic components, large banks, and insurance companies. In the 1950s, the new computer technology was not yet of great value to other types of businesses.

In the second half of the 1960s and the 1970s, computers began to be widely adopted by businesses. An important computer in this period was the IBM System 360, which substantially dominated its competitors (namely Burroughs, Control Data, General Electric, Honeywell, NCR, RCA, and Sperry Rand). A notable competitor in the late 1960s was Control Data Corp. with its CDC 6600 and successors. CDC achieved a 5% market share by focusing on applications in science and engineering. A new company, Digital Equipment Corporation, was founded at this time and gained a large market share with its PDP8 minicomputer, which was priced well below the IBM/360. Applications in this period included accounting, inventory control, retail, banking, insurance, and diverse areas of manufacturing.

A massive use of computers followed in the 1980s and early 1990s, affecting almost all manufacturing and service sectors. Computers became cheaper, faster, and more reliable. Peripheral devices, such as disks and terminals, made the interaction with the computer more convenient and allowed the storage and retrieval of large volumes of data. The many existing applications then could be performed online rather than in batch mode. This capability then enabled new applications, such as decision-support systems. For example, daily online access to financial performance figures of a company could be obtained, and computers supported the tasks of financial modeling and planning, sales, marketing, and human resource management. In retail, real-time inventory control emerged, OCR (Optical Character Recognition) became important, and the Universal Product Code (UPC) was developed. A further enabler of the fast dissemination of the new technology was the microcomputer. However, it was not taken seriously by commercial enterprises until IBM introduced its first Personal Computer (PC) in 1981. This initiated a shift of computer applications from mainframes (see also *Mainframes*) to PCs. While this shift happened for business and commercial applications first, the trend is still ongoing for scientific and engineering applications, which were once the clear domain of mainframe high-performance computers.

In the last decade of the millennium, computers have started to penetrate every aspect of life. Microprocessors serve as control units of small and large appliances of every kind. Personal computers are found in most households of modern countries, and they are companions for business and leisure travelers world-wide. The Internet has enabled “mobile computing”. Such travel computers started out as important tools for sales representatives, giving them access to home databases, electronic mail, and the World-Wide Web (see *Network Computing*). These developments of the computer industry and its applications were led by the U.S.A., although Europe and Japan followed with only a few years delay [18, 4, 6]. It reasonably can be assumed that in other countries similar developments are happening or will happen.

2.3 Factors Contributing to the Success of the von Neumann Computer

Progress in Hardware Technology and Computer Architecture

Progress in electronics technology is the basic enabler for the revolution of the von Neumann machine. This progress was initiated during World War II when there were enormous advances in the development of electronics. While the vacuum tube was a first step, orders of magnitude improvement in computing speeds, miniaturization, and power consumption have been achieved with the transistor and with integrated circuits. The improvements in computer speeds and the cost of electronic components in the past five decades amount to approximately a factor of 2 every 2 years.

These numbers are even more remarkable if we consider that the source of this information is a 20-year review of information processing, made in 1988 [29], in which trends that were predicted 20 years earlier were indeed confirmed. Furthermore, even if we include 1998 data points[1], the somewhat simplistic, linear predictions of 1968 are still true in principle. A few caveats are necessary, however. For example, the peak performance of 1 TeraOPS has been reported for a parallel processor architecture, where the performance of the individual processors are approximately 3 orders of magnitude less. Hence, to maintain the previous rate of performance improvement, computer systems must use a mix of raw hardware speed and architectural innovations. One could argue that, in fact, the rate of performance increase of individual processors has slowed down significantly over the past few years.

In addition to the basic hardware components, significant progress has been made in combining these elements into powerful computer architectures. In part, these innovations were driven by the rapid miniaturization of the fundamental components. For example, it became possible to place a growing number of processor components onto one chip, although determining the most effective mix for these functional units is an ongoing problem. Furthermore, the question of how to best serve the software systems that harness the processors has become of paramount importance. In all this progress, the basic stored-program concept has remained the same, although its specific realization in processors, memory modules, peripheral devices, and interconnections have changed significantly.

Progress in Software Technology

The ENIAC computer was programmed with switch arrays on its front panels. Today, software costs dominate hardware costs by far. This change from almost ignorance of the software problem to making it a number one priority may be considered more important than the progress in hardware technology. Nevertheless, enormous advances in software technology have been made over the past five decades. Computer languages have been developed that allow a problem to be coded in a user-oriented manner (known as *high-level languages*). Powerful translators (see also *Program Compilers*) have been developed that can transform these languages into the efficient low-level machine code understood by the processing units.

Operating systems have been created that make it possible to use a computer system in a convenient, interactive way. Operating systems also offer the programmer a rich application program interface, which permits and coordinates a wide range of calls to existing software modules (called *libraries*)

[1]Cost per logic element: \$8/1MB RAM = 10^{-6} \$/logic element (assuming 1 logic element per memory cell); Fastest reported computer: 1 TeraOPS = 10^{12} instructions/second)

that perform commonly needed functions. Examples are functions that write to a disk file, prompt the user to select from a command menu, visualize a data structure as a 3-D graph, or solve a system of linear equations. While basic functions are usually part of the operating system itself, less commonly-used ones can be found in an ever-growing range of available library packages (see also *UNIX*).

At the highest software layer, full applications have been developed to perform an increasing range of tasks. Many applications are parameterizable so that they can be adapted to new problems and to user preferences. For example, a chemist may find a standard application package that performs the simulation of a new substance. The application may be purchased commercially or even may be freely available, although free applications typically come without support (see also *Public Domain Software*). Obtaining good support is crucial for many application users since a thorough knowledge of the application is necessary to determine if it can be adapted to the problem at hand. If not, then the expensive development of a new application may become necessary. As computer applications become more sophisticated, their development costs grow enormously. This cost represents a significant limit to the seemingly unbounded opportunities for computer-based problem solving, as discussed in Section 6.

Computer Science and Engineering

Despite his very practical achievements, John von Neumann devoted most his efforts to developing the fundamental concepts and logical underpinnings of the new electronic computers. He made many important contributions, not only in terms of computer architecture, but also in software principles. He developed *flow diagramming* techniques and computer algorithms for diverse mathematical problems. His vision becomes evident in his early discussions of parallel processing concepts, techniques that deal with fast computation but slow input/output, algorithms for solving partial differential equations, and errors introduced by finite computer precision [1].

While von Neumann's work represents a substantial initial contribution to the new discipline of computer science and engineering, many others have also influenced its evolution. For example, a very notable contribution has been made by Donald E. Knuth in *The Art of Computer Programming* [16], which represents a conscious effort to place computer programming on a foundation of mathematical principles and theorems. This type of work has led to the acceptance of computer science and engineering by the academic community, which is important since this acceptance adds legitimacy to the field and causes a systematic search for innovations.

Since the design of ENIAC and the IAS computer, there has been a growing trend to deal with software issues more than hardware issues. This shift has been caused, in part, by the steady increase in software costs, but it also indicates a tendency to move discussions from the immediate practical problems that need to be engineered to more theoretical, formal considerations. Even five decades after Mauchly and Eckert's dispute with von Neumann, the issue of how theoretical or practical computer science should be is still under debate. Historians date the beginning of an actual Computer Science, defined to be the "systematic study of computers and information processing", to the late 1950s. However, more important is the fact that systematic methods for describing both hardware and software have indeed emerged and have led to the support of the new computer age by the academic community.

Professional Societies

Substantial support for a discipline also comes from its associated professional organizations. Two such organizations were founded shortly after the ENIAC computer became operational. These are the IEEE Computer Society, founded in 1946, and the Association for Computing Machinery (ACM), founded in 1947. Both organizations support the community by sponsoring workshops, conferences, technical committees, and special interest groups; by establishing distinguished lecturer programs and committees that give recommendations regarding university curricula; and by publishing professional journals [23].

Standardization

Standards help promote a technology by substantially reducing development costs for machine and component interfaces and learning costs for users who have to interact with the machines. A number of computer-related standards have emerged. Some are conscious efforts to set standards, while others have emerged as de-facto standards, or as a result of all but one offerer leaving the market.

Explicit international standards are administered by the International Standards Organization (ISO). They cover areas such as information encoding, programming languages, documentation, networking, computer graphics, microprocessor systems, peripheral devices, interconnections, and many aspects of computer applications. An example of a de-facto standard is the *UNIX* operating system, which has emerged as the system of choice for workstation and high-speed computers. A standard resulting from all but one offerer leaving the market is the PC with its DOS/Windows user interface. It has emerged as the most widely-used business and home computer, dominating its initial competitors.

Standard methods for measuring computer systems performance are also important because they allow the comparison of different systems using the same measuring stick. A notable effort has been made by the Standard Performance Evaluation Corporation, SPEC. *SPEC benchmarks* are available for most workstation and PC systems to compare computation rates based on a range of application programs. New benchmarks for measuring graphics, network, and high-performance computers also are being developed.

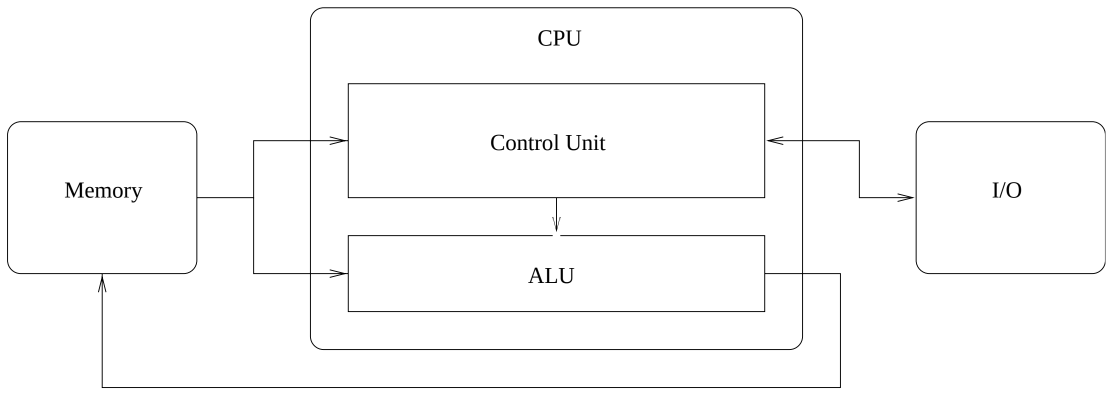

Figure 2: The basic components of a computer with a von Neumann architecture are the memory, which stores both instructions and data, the Central Processing Unit (CPU), which actually executes the instructions, and the Input/Output (I/O) devices, which provide an interface to the outside world.

3 Organization and Operation of the von Neumann Architecture

As shown in Figure 2, the heart of a computer system with a von Neumann architecture is the **CPU**. This component fetches (i.e., reads) instructions and data from the main memory and coordinates the complete execution of each instruction. It is typically organized into two separate subunits: the Arithmetic and Logic Unit (ALU), and the control unit. The ALU combines and transforms data using arithmetic operations, such as addition, subtraction, multiplication, and division, and logical operations, such as bit-wise negation, AND, and OR. The control unit interprets the instructions fetched from the memory and coordinates the operation of the entire system. It determines the order in which instructions are executed and provides all of the electrical signals necessary to control the operation of the ALU and the interfaces to the other system components.

The **memory** is a collection of storage cells, each of which can be in one of two different states. One state represents a value of “0”, and the other state represents a value of “1.” By distinguishing these two different logical states, each cell is capable of storing a single *binary digit*, or *bit*, of information. These bit storage cells are logically organized into *words*, each of which is b bits wide. Each word is assigned a unique *address* in the range $[0, ..., N-1]$.

The CPU identifies the word that it wants either to read or write by storing its unique address in a special *memory address register* (MAR). (A *register* temporarily stores a value within the CPU.) The memory responds to a read request by reading the value stored at the requested address and passing it to the CPU via the CPU-memory data bus. The value then is temporarily stored in the *memory buffer register* (MBR) (also sometimes called the *memory data register*) before it is used by the control unit or ALU. For a write operation, the CPU stores the value it wishes to write into the MBR and the corresponding address in the MAR. The memory then copies the value from the MBR into the address pointed to by the MAR.

Finally, the **input/output** (I/O) devices interface the computer system with the outside world. These devices allow programs and data to be entered into the system and provide a means for the system to control some type of output device. Each I/O *port* has a unique address to which the CPU can either read or write a value. From the CPU's point of view, an I/O device is accessed in a manner very similar to the way it accesses memory. In fact, in some systems the hardware makes it appear to the CPU that the I/O devices are actually memory locations. This configuration, in which the CPU sees no distinction between memory and I/O devices, is referred to as *memory-mapped I/O*. In this case, no separate I/O instructions are necessary.

3.1 Key Features

Given this basic organization, processors with a von Neumann architecture generally share several key features that distinguish them from simple preprogrammed (or *hardwired*) controllers. First, instructions and data are both stored in the same main memory. As a result, instructions are not distinguished from data. Similarly, different types of data, such as a floating-point value, an integer value, or a character code, are all indistinguishable. The meaning of a particular bit pattern stored in the memory is determined entirely by how the CPU interprets it. An interesting consequence of this feature is that the same data stored at a given memory location can be interpreted at different times as either an instruction or as data. For example, when a compiler executes, it reads the source code of a program written in a high-level language, such as Fortran or Cobol, and converts it to a sequence of instructions that can be executed by the CPU. The output of the compiler is stored in memory like any other type of data. However, the CPU can now execute the compiler output data simply by interpreting them as instructions. Thus, the same values stored in memory are treated as data by the compiler, but are subsequently treated as executable instructions by the CPU.

Another consequence of this concept is that each instruction must specify how it interprets the data on which it operates. Thus, for instance, a von Neumann architecture will have one set of arithmetic instructions for operating on integer values and another set for operating on floating-point values.

The second key feature is that memory is accessed by name (i.e., address) independent of the bit pattern stored at each address. Because of this feature, values stored in memory can be interpreted as addresses as well as data or instructions. Thus, programs can manipulate addresses using the same set of instructions that the CPU uses to manipulate data. This flexibility of how values in memory are interpreted allows very complex, dynamically changing patterns to be generated by the CPU to access any variety of data structure regardless of the type of value being read or written. Various addressing modes are discussed further in Section 3.2.

Finally, another key concept of the von Neumann scheme is that the order in which a program executes its instructions is sequential, unless that order is explicitly altered. A special register in the CPU called the *program counter* (PC) contains the address of the next instruction in memory to be executed. After each instruction is executed, the value in the PC is incremented to point to the next instruction in the sequence to be executed. This sequential execution order can be changed by the program itself using *branch* instructions, which store a new value into the PC register. Alternatively, special hardware can sense some external event, such as an interrupt, and load a new value into the PC to cause the CPU to begin executing a new sequence of instructions. While this concept of performing one operation at a time greatly simplifies the writing of programs and the design and implementation of the CPU, it also limits the potential performance of this architecture. Alternative *parallel architectures* that can execute multiple instructions simultaneously are discussed in Section 5.

3.2 Instruction Types

A processor's *instruction set* is the collection of all the instructions that can be executed. The individual instructions can be classified into three basic types: data movement, data transformation, and program control. *Data movement* instructions simply move data between registers or memory locations, or between I/O devices and the CPU. Data movement instructions are actually somewhat misnamed since most move operations are nondestructive. That is, the data are not actually moved but, instead, are copied from one location to another. Nevertheless, common usage continues to refer to these operations as data movement instructions. *Data transformation* instructions take one or more data values as input and perform some operation on them, such as an addition, a logical OR, or some other arithmetic or logical operation, to produce a new value. Finally, *program control* instructions can alter the flow of instruction execution from its normal sequential order by loading a new value into the PC. This change in the instruction execution order can be done conditionally on the results of previous instructions.

In addition to these three basic instruction types, more recent processors have added instructions that can be broadly classified as *system control* instructions. These types of instructions generally are not necessary for the correct operation of the CPU but, instead, are used to improve its performance. For example, some CPUs have implemented *prefetch* instructions that can begin reading a location in memory even before it is needed [27]. A variety of other system control instructions also can be supported by the system.

Each instruction must explicitly or implicitly specify the following information [12]:

1. The operation to be performed, which is encoded in the *op-code*.
2. The location of the *operands*, which are the input data on which to perform the operation.
3. The *destination* location, which is where the result of the operation will be stored.
4. The next instruction to be executed.

All instructions must explicitly specify the op-code, although not all instructions will need to specify both source and destination operations. The *addressing mode* used by an instruction specifies the location of the source and destination operands, which may be, for example, registers, memory addresses, or I/O ports. With the *implicit* addressing mode, the instruction assumes that the operation is in a predetermined location. This mode is commonly used to access certain internal registers. The *immediate* addressing mode is used to access a constant data value that has been encoded as part of the instruction itself. The *direct* addressing mode, in contrast, uses a constant value encoded in the instruction as the address of either a register or a location in memory.

With *indirect addressing*, the value encoded in the instruction is the address of a register or memory location that contains the actual address of the desired operand. This addressing mode is commonly used to manipulate pointers, which are addresses stored in memory. Finally, *indexing* is an addressing mode that can be used to efficiently scan through regular data structures, such as arrays. With this mode, the address of the desired operand is found by adding a value in an index register to a given base address. Thus, subsequent elements in an array, for instance, can be accessed simply by incrementing the value stored in the index register. While these are the basic addressing modes, a variety of combinations of these modes have been implemented in different processors [11, 19].

Both data transformation and data movement instructions implicitly assume that the next instruction to be executed is the next instruction in the program sequence. Program control instructions, such as branches and jumps, on the other hand, must explicitly specify the address of the next instruction to be executed. Note that conditional branch instructions actually specify two addresses. The *target address* of the branch, which is the address of the instruction the program should begin executing if the branch outcome is *taken*, is explicitly specified. If the branch is *not taken*, however, it is implicitly specified that the next instruction in sequential order should be executed.

The *Instruction Set Architecture* (ISA) of a processor is the combination of all the different types of instructions it can execute plus the resources accessible to the instructions, such as the registers, the functional units, the memory, and the I/O devices. The ISA gives each type of processor its unique "personality" since it determines the programmer's view of *what* the processor can do. In contrast, the *implementation* of the processor determines *how* the ISA actually performs the desired actions. As a result, it is entirely possible to have several different implementations of an ISA, each of which can have different performance characteristics.

3.3 Instruction Execution

Executing instructions is a two-step process. First, the next instruction to be executed, which is the one whose address is in the program counter (PC), is fetched from the memory and stored in the *Instruction Register* (IR) in the CPU. The CPU then executes the instruction to produce the desired result. This *fetch-execute* cycle, which is called an *instruction cycle*, is then repeated for each instruction in the program.

In fact, the execution of an instruction is slightly more complex than is indicated by this simple fetch-execute cycle. The interpretation of each instruction actually requires the execution of several smaller substeps called *microoperations*. The microoperations performed for a typical instruction execution cycle are described in the following steps:

1. Fetch an instruction from memory at the address pointed to by the Program Counter (PC). Store this instruction in the IR.
2. Increment the value stored in the PC to point to the next instruction in the sequence of instructions to be executed.
3. Decode the instruction in the IR to determine the operation to be performed and the addressing modes of the operands.
4. Calculate any address values needed to determine the locations of the source operands and the address of the destination.
5. Read the values of the source operands.
6. Perform the operation specified by the op-code.
7. Store the results at the destination location.
8. Go to Step 1 to repeat this entire process for the next instruction.

Notice that not all of these microoperations need to be performed for all types of instructions. For instance, a conditional branch instruction does not produce a value to be stored at a destination address. Instead, it will load the address of the next instruction to be executed (i.e., the *branch target* address) into the PC if the branch is to be *taken*. Otherwise, if the branch is *not taken*, the PC is not changed and executing this instruction has no effect. Similarly, an instruction that has all of its operands available in registers will not need to calculate the addresses of its source operands.

The time at which each microoperation can execute is coordinated by a periodic signal called the CPU's *clock*. Each microoperation requires one clock period to execute. The time required to execute the slowest of these microoperations determines the minimum period of this clock, which is referred to as the the CPU's *cycle time*. The reciprocal of this time is the CPU's *clock rate*. The minimum possible value of the cycle time is determined by the electronic circuit technology used to implement the CPU. Typical clock rates in today's CPUs are 200 to 300 MHz, which corresponds to a cycle time of 3.3 to 5 ns. The fastest CPUs, as of the time of this writing, are reported at 600 MHz.

An instruction that requires all seven of these microoperations to be executed will take seven clock cycles to complete from the time it is fetched to the time its final result is stored in the destination location. Thus, the combination of the number of microoperations to be executed for each instruction, the mix of instructions executed by a program, and the cycle time determine the overall performance of the CPU.

A technique for improving performance takes advantage of the fact that, if subsequent instructions are independent of each other, the microoperations for the different instructions can be executed simultaneously. This overlapping of instructions, which is called *pipelining*, allows a new instruction to begin executing each CPU cycle without waiting for the completion of the previous instructions. Of course, if an instruction is dependent on a value that will be produced by an instruction still executing, the dependent instruction cannot begin executing until the first instruction has produced the needed result. While pipelining can improve the performance of a CPU, it also adds substantial complexity to its design and implementation.

If the depth of the instruction pipeline is n, then up to n independent instructions can be in various phases of execution simultaneously. As a result, the time required to execute all of the instructions in a program can be reduced by at most a factor of n. Dependences between instructions reduce the actual speedup to something less than this theoretical maximum, although several "tricks" can be used to minimize the performance impact of dependences in pipelined processors [8, 14]. The possible depth of a pipeline is determined by the amount of work to be performed in each microoperation in an instruction's execution cycle and by the circuit technology used to implement the CPU.

4 Memory Access Bottleneck

While the basic computer organization proposed by von Neumann is widely used, the separation of the memory and the CPU also has led to one of its fundamental performance limitations, specifically, the delay to access memory. Due to the differences in technologies used to implement CPUs and memory devices and to the improvements in CPU architecture and organization, such as very deep pipelining, the cycle time of CPUs has reduced at a rate much faster than the time required to access memory. As a result, a significant imbalance between the potential performance of the CPU and the memory has developed. Since the overall performance of the system is limited by its slowest component, this imbalance presents an important performance bottleneck. This limitation often has been referred to as the *von Neumann bottleneck* [9].

4.1 Latency and Bandwidth

Memory performance can be characterized using the parameters *latency* and *bandwidth*. Memory latency is defined to be the time that elapses from the initiation of a request by the CPU to the memory subsystem until that request is satisfied. For example, the read latency is the time required from when the CPU issues a read request until the value is available for use by the CPU. The bandwidth, on the other hand, is the amount of data that can be transferred per unit time from the memory to the processor. It is typically measured in *bits per second*. While the description of the basic organization in Section 3 implies that only a single word is transferred from the memory to the CPU per request, it is relatively simple to increase the memory bandwidth by increasing the width of the data bus between the CPU and the memory. That is, instead of transferring only a single word from the memory to the CPU per request, multiple words can be transferred, thereby scaling-up the memory bandwidth proportionally. For example, in a CPU with a 64-bit word size, the 8 bytes (1 byte = 8 bits) that constitute a single word could be transferred from the memory to the CPU as 8 single-byte chunks in 8 separate cycles. Alternatively, the memory bandwidth could be increased by a factor of 8 if all eight bytes are transferred in a single cycle. In high-performance systems, it would not be unusual to transfer 128 to 256 bits (2 to 4 64-bit words) per cycle.

Another approach for improving the memory bandwidth is to split the memory into two separate systems, one for storing data, and the other for storing instructions. This type of computer organization is referred to as a *Harvard architecture* (see *Harvard architecture*). It was developed by a research group at Harvard University at roughly the same time as von Neumann's group developed the Princeton architecture. The primary advantage of the Harvard architecture is that it provides two separate paths between the processor and the memory. This separation allows both an instruction and a data value to be transferred simultaneously from the memory to the processor. The ability to access both instructions and data simultaneously is especially important to achieving high performance in pipelined CPUs because one instruction can be fetching its operands from memory at the same time a new instruction is being fetched from memory.

4.2 Memory Hierarchy

While memory bandwidth can be increased simply by increasing the size and number of buses between the memory and the CPU, reducing memory latency is much more difficult. Latency is ultimately limited by the propagation time of the signals connecting the processor and the memory, which is guaranteed to be less than the speed of light. Since this is a fundamental physical limitation, computer designers have resorted to using a variety of techniques that take advantage of the characteristics of executing programs to tolerate or hide memory latency. The most common of these techniques is the use of *caches* in a *memory hierarchy* [22].

The ideal memory system would be one with zero latency and infinite storage capacity and bandwidth. Unfortunately, latency and cost are inversely related. Thus, fast (i.e., low latency) memory systems are expensive, while large memory systems are relatively slow. Given this cost-performance tension, the goal of a computer designer is to construct a memory system that appears to have the performance of the fastest memory components with the approximate cost per bit of the least expensive memory components. This goal has been approached by designing a hierarchical memory system that temporarily copies the contents of a memory location when it is first accessed from the large, slow memory into a small, fast memory called a *cache* that is near the processor.

In this hierarchy of memory, the CPU sees the full latency of the main memory, plus the delay introduced by the cache, the first time a memory location is accessed. However, subsequent references to that address will find the value already in the cache. This situation is referred to as a *cache hit*. In this case, the memory delay is reduced to the time required to access the small, fast cache itself, which is considerably less than the time required to access the main memory. A reference that does not find the desired address in the cache is called a *cache miss*. A miss causes the desired address to be copied into the cache for future references. Of course, since the cache is substantially smaller than the main memory, values that were previously copied into the cache may have to be evicted from the cache to make room for more recently referenced addresses.

The average time required for the CPU to access memory with this two-level hierarchy can be determined by partitioning all memory accesses into either cache hits or cache misses. The time required to read an address on a hit is t_h. On a miss, however, time t_h is required to determine that the desired address is not in the cache. An additional time of t_m is then required to copy the value into the cache and to transfer it to the CPU. Furthermore, let h be the *hit ratio*, which is the fraction of all of the memory references issued by a program that hit in the cache. Then the *miss ratio* is $m = 1 - h$, and the average memory access time is

$$\bar{t}_{mem} = ht_h + m(t_h + t_m) = (1 - m)t_h + m(t_h + t_m) = t_h + mt_m. \tag{1}$$

This equation shows that, when the miss ratio is small, the average memory access time approaches the time required to access the cache, t_h, rather than the relatively long time required to access the main memory, t_m.

The average cost per bit of this hierarchical memory system is easily found to be

$$\bar{c}_{mem} = \frac{c_c s_c + c_m s_m}{s_c + s_m} \tag{2}$$

where c_c and c_m are the respective costs per bit, and s_c and s_m are the respective sizes in bits, of the cache and memory. Note that, as the size of the memory is made much larger than the size of the cache, that is, $s_m >> s_c$, the average cost per bit of this memory system approaches the average cost per bit of the main memory, c_m/s_m. Thus, this type of memory hierarchy approximates the computer designer's goal of providing a memory system whose average access time is close to that of the fastest memory components with a cost that approaches that of the least expensive components.

Of course, the caveat when using a cache is that the miss ratio must be sufficiently small or, conversely, the hit ratio must be sufficiently large. Fortunately, application programs tend to exhibit *locality* in the memory addresses they reference. *Spatial locality* refers to the fact that programs tend to reference a small range of addresses in any given time period. Programs also tend to repeatedly access the same small set of memory locations within a short period of time, a characteristic referred to as *temporal locality*. This program behavior allows a relatively small cache to capture most of a program's *working set* of memory addresses at any given time so that hit ratios of 95 to 99 percent are not uncommon. While these high hit ratios may seem surprising, they are a direct consequence of the way programs are written to run on a von Neumann architecture. In particular, instructions are typically executed sequentially, and vectors or arrays of data are often accessed in sequential memory order, both of which lead to high spatial locality. Furthermore, most programs contain many loops that are executed a large number of times, which causes high temporal locality.

4.3 Cache Coherence

Most current computer systems use a combination of both Harvard and Princeton architectures in their memory hierarchies [9]. A Harvard architecture is used on-chip for the cache portion of the hierarchy while the off-chip main memory uses a Princeton architecture with a single connection to the separate caches in the CPU. While this approach allows for the simultaneous access of instructions and data from their respective caches, it also introduces a potential problem in which there can be inconsistent values for the same address stored in the different caches and the main memory. This potential inconsistency is referred to as the *cache coherence* problem. In a computer system with a single CPU, the cache coherence problem stems from the fact that all executable programs begin their lives as output data from a compiler or an assembler.

To understand this problem, consider a system that has a *writeback* data cache and a separate instruction cache. A writeback cache is one in which a new value written to the cache is not written back to the main memory until the cache is full. The word is then evicted from the cache to make room for a newly referenced word. At that point, the latest value in the cache is written back to the main memory. Until the writeback takes place, however, the value in the cache for that specific address is different from the value stored in the main memory. These two copies of the same address are said to be *incoherent* or *inconsistent*. Under normal operation, this inconsistency is not a problem since the CPU first looks in the cache for a copy of the address it is reading. Since the copy in the cache is the most current value that has been stored in that address, it does not matter to the CPU that the value stored in memory is inconsistent.

A problem can arise, however, when a program is compiled and then executed. Since the output of any program is treated as data, the output of the compiler, which is simply the executable program, is stored in the data cache. If this newly compiled program is then immediately executed, the CPU will begin fetching the instructions from the instruction cache. Not finding the desired addresses in its instruction cache, it fetches the instructions from the main memory. However, the instructions to be executed are actually still sitting in the data cache. As a result, the CPU attempts to execute whatever happened to be stored in memory at the indicated address, which is not the first instruction of the program just compiled. While there are many solutions to this coherence problem, it is still a problem that has caused difficulties in recent computer systems [9](pp. 262-264) and that is critical to the correct execution of programs in parallel computing systems [25, 17].

5 Alternatives to the von Neumann Architecture

Beyond the memory bottleneck, the performance of computer systems based on the von Neumann architecture is limited by this architecture's "one instruction at a time" execution paradigm. Executing multiple instructions simultaneously using pipelining can improve performance by exploiting parallelism among instructions. However, performance is still limited by the *decode bottleneck* [7] since only one instruction can be decoded for execution in each cycle. To allow more parallelism to be exploited, multiple operations must be simultaneously decoded for execution.

The sequence of instructions decoded and executed by the CPU is referred to as an *instruction stream*. Similarly, a *data stream* is the corresponding sequence of operands specified by those instructions. Using these definitions, Flynn [7] proposed the following taxonomy for parallel computing systems:

- SISD–Single Instruction stream, Single Data stream.
- SIMD–Single Instruction stream, Multiple Data stream.
- MISD–Multiple Instruction stream, Single Data stream.
- MIMD–Multiple Instruction stream, Multiple Data stream.

An SISD system is a traditional processor architecture that executes one sequence of instructions. In an SIMD system, however, an instruction specifies a single operation that is performed on several different data values simultaneously. For example, the basic operand in an SIMD machine may be an array. In this case, an element-by-element addition of one array to another would require a single addition instruction whose operands are two complete arrays of the same size. If the arrays consist of n rows and m columns, nm total additions would be performed simultaneously. Because of their ability to efficiently operate on large arrays, SIMD processors often are referred to as *array processors* and are frequently used in image-processing types of applications.

In an MISD processor, each individual element in the data stream passes through multiple instruction execution units. These execution units may combine several data streams into a single stream (by adding them together, for instance), or an execution unit may transform a single stream of data (performing a square root operation on each element, for instance). The operations performed and the flow of the data streams are often fixed, however, limiting the range of applications for which this type of system would be useful. MISD processors often are referred to as *systolic arrays* and typically are used to execute a fixed algorithm, such as a digital filter, on a continuous stream of input data.

MIMD systems often are considered to be the "true" parallel computer systems. *Message-passing* parallel computer systems are essentially independent SISD processors that can communicate with each other by sending messages over a specialized communication network. Each processor maintains its own independent address space so any sharing of data must be explicitly specified by the application programmer.

In *shared-memory* parallel systems, on the other hand, a single address space is common to all of the processors. Sharing of data is then accomplished simply by having the processors access the same address in memory. In the implementation of a shared-memory system, the memory may be located in one central unit, or it may be physically distributed among the processors. Logically, however, the hardware and the operating system software maintain a single, unified address space that is equally accessible to all of the processors. For performance reasons, each of the processors typically has its own private data cache. However, these caches can lead to a coherence problem similar to that discussed in Section 4.3, since several processors could have a private copy of a memory location in their data caches when the address is written by another processor. A variety of hardware and software solutions have been proposed for solving this shared-memory coherence problem [25, 17].

While these parallel architectures have shown excellent potential for improving the performance of computer systems, they are still limited by their requirement that only independent instructions can be executed concurrently. For example, if a programmer or a compiler is unable to verify that two instructions or two tasks are never dependent upon one another, they must conservatively be assumed to be dependent. This assumption then forces the parallel computer system to execute them sequentially.

However, several recently proposed *speculative* parallel architectures [21, 26, 28, 13, 5] would, in this case, aggressively assume that the instructions or tasks are not dependent and would begin executing them in parallel. Simultaneous with this execution, the processors would check predetermined conditions to ensure that the independence assumption was correct when the tasks are actually executed. If the speculation was wrong, the processors must *rollback* their processing to a nonspeculative point in the instruction execution stream. The tasks then must be re-executed sequentially. A considerable performance enhancement is possible, however, when the speculation is determined to be correct. Obviously, there must be a careful trade-off between the cost of rolling-back the computation and the probability of being wrong.

6 Current Applications of von Neumann Computers

This section gives a list of computer application areas and describes the significance and limits of problem solving with the computer. The basic steps in creating an application also are outlined. The main focus is on problem solving in science and engineering, which is often referred to as the Computational Science and Engineering (CSE) area. This area provided the first applications of early computers. Despite its rapid growth, today, computer applications in non-CSE fields are commercially even more important (see also *Microcomputer Applications, Office Automation, Databases, Transaction Processing,* and *Hobby Computing*).

Computational Science and Engineering (CSE) includes a wide range of applications that allow scientists and engineers to perform experiments "in the computer." CSE applications typically find solutions to complex mathematical formulas, which involves operations on large sets of numbers. This is called numerical computing or, colloquially, *number crunching.*

Numerical Application Areas

The following list outlines several important CSE applications and the problems they solve.

Computational chemistry is an important computer user area (see also *Chemistry Computing*). Chemical reactions and properties of substances can be studied and simulated at the molecular and quantum levels (the latter accounts for the inner forces of atoms) allowing, for instance, the synthesis of drugs, the design of lubricants, and the study of reactions in a combustion engine.

Computational biology is similar to computational chemistry, except that biochemical processes are modeled for purposes such as protein studies and syntheses, and genetic sequence analysis.

Quantum physics is being modeled computationally for the study of superconductivity, particle collisions, cosmology, and astrophysics (see also *Physics Computing*).

Structural mechanics is an important area for the synthesis, analysis, and testing of mechanical components and structures. Mechanical properties of engines or airplane hulls can be determined, and forces and deformations in a car crash can be studied.

Materials science aims at the understanding of material and its properties at the molecular and atomic level. Insights into the behavior of superconductors and semiconductors, as well as the microscopic properties of cast metal, can be obtained.

Computational electromagnetics is used for studying fields and currents in antennas, radars, microwave ovens, and many other electrical devices.

Computational fluid dynamics (CFD) simulates the flow of gases and fluids for studying an ever-growing range of topics, such as the aerodynamics of airplanes, cars, boats and buildings; the characteristics of turbines; the properties of combustion processes; atmospheric effects; and the processes in rocket motors and guns.

Climate and environmental modeling applications simulate the global climate and the behavior of oceans; provide short-term weather forecasts; find answers to early events in the ice age; and study the distribution of atmospheric pollutants. (see also *Environmental Science Computing*).

Ecosystem modeling applications study the change of land cover, such as vegetation and animal habitats, and land use.

Geophysical modeling and **seismic processing** programs investigate the earth's interior for locating oil, gas, and water reservoirs, and for studying the earth's global behavior.

Electronic device simulation investigates properties of the very building blocks that make processor chips. It plays a crucial role in advancing basic computer technology.

Image processing applications are found in medical tomography, filtering of camera, satellite, and sensor data, surface rendering, and image interpretation. In general, **digital signal processing** (DSP) methods are used for the analysis, filtering, and conversion of camera, acoustic, and radar signals.

Non-numerical and Hybrid Applications

Classical scientific and engineering applications involve numerical methods while an increasing range of new applications involve non-numerical algorithms or hybrid solutions. For example, image processing may involve both numerical low-level filters and non-numerical methods for the identification of objects. Discrete event simulation involves non-numerical algorithms, but may be combined with numerical simulations of individual events. Decentralized Command Control is a term used in military applications but applies equally to industrial and scientific settings. It involves the gathering of information from diverse, geographically distributed sources, methods for reasoning about these data, decision-making support, and tools to steer the distributed processes as needed.

The Decentralized Command Control area makes obvious the trend in CSE applications toward increasingly complex solutions. As compute power increases, computer methods for analysis, simulation, and synthesis are developed in all conceivable fields. Simulators of different application areas can be combined to create an even more powerful application. In doing so, resources and input/output devices may be used world-wide and reactions to global changes can be computed. Another example of such multi-disciplinary methods is found in *Robotics*. This field involves the processing of sensory data, the simulation and prediction of the behavior of diverse kinds of visible objects, decision-making methods for proper responses, and the coordination of commands to put these responses into action. A third example of an interdisciplinary and increasingly complex application is the simulation of nuclear reactor systems. While chemical processes must be simulated to capture the behavior inside a reactor, the reactor system as a whole involves diverse thermodynamic processes that require CFD methods.

6.1 Significance and limits of computational problem solving?

Virtually Unlimited Experiments "in the Computer"

Many areas of science and all areas of engineering need experimentation. Computational methods allow the scientist and engineer to perform experiments in virtual instead of in physical space. This allows one to overcome many limits that are associated with our reality.

The following are examples of such limits.

- Laws set many important limits to experiments. One example is experimentation with hazardous material. While strict limits are set that, for example, control the release of lethal substances into the atmosphere, the computational engineer can explore chemical reactions in all conceivable settings. As a result, hazards may be characterized more quantitatively, and accident scenarios may be explored.

- Certain experiments may be permitted by law, but ethical rules prevent the scientist from doing excessive exploration. Experiments with animals fall into this category. The computational scientist can overcome these limits and, for example, design drugs that are more reliably tested.
- Physical limits set the most obvious constraints to experiments in real space. The computational engineer, however, can easily "switch off gravity" or construct a device that is larger than our entire planet.
- Financial limits prohibit many experiments. Crashing one or several new cars for safety tests is very expensive. Accurate crash test simulation tools therefore are among the important investments of car manufacturers.
- Exploring processes that take extremely long or short time spans is difficult. Just as one cannot wait 1000 years to observe a material's aging process, an engineer's instruments may not be fast enough to record events in the picosecond range. Simulations can easily stretch and compress time scales.
- Other experiments may not be feasible because of human limitations. A human observer may not record events with sufficient accuracy; situations may be too complex to grasp; and real experiments may require inappropriate human interfaces. Computer tools can provide remedies in all of these areas.

Limits on Pushing the Limits

While there are virtually unbounded opportunities for computational problem solving, there are several factors that set limits. These include computer speeds, application development costs, and the accuracy of simulation models.

The fastest computer speeds reported today are on the order of one trillion operations per second (or 1 *TeraOP*). This is more than a 1000-fold performance improvement over the average PC. In a recent initiative to replace nuclear explosion experiments by computer simulations, the necessary computational power for this task was estimated to be approximately 1 Quadrillion operations per second (or 1 *PetaOPS*). Simulating a complete nuclear explosion would be the most advanced computational problem ever solved. The fact that it would take compute resources that are a thousand times higher than the current cutting-edge technology gives an indication of the complexity of computations that are tractable today and what may become possible in the future.

The effort and cost for developing a new computer application program represents a second major hurdle in the computational race. Whereas the design of hardware was the major problem during the IAS computer's era, software costs have since exceeded hardware costs by several factors. As applications evolve and become increasingly complex, the development effort increases drastically and offsets the progress made in software technology. Developing flexible applications so that they can be adapted to new problems is even more costly. However, such flexible applications are very important because not being able to adapt an existing application to a new problem may lead to prohibitive development costs.

Most software is written in standard programming languages, such as Fortran, C, or C++. The number of lines written per day by a programmer is in the single digits if one includes all costs from the problem specification to the software maintenance phase. Thus, the investment in a program that is 100,000 lines long, which is a relatively small size for an "interesting" application, may reach several million dollars. There are hopes to lower these costs with *problem solving environments* (PSE). PSEs attempt to provide user-oriented program development facilities that allow the specification of a problem at a much higher level than current programming languages. For example, the physicist would enter physics equations and the chemist a chemical formula. However, the current state of technology is still far from this goal (see also *Specification Languages*). Future progress will depend critically on how well these software issues can be solved.

A third major limitation in computational problem solving is the accuracy of computational models with which reality is described, approximated, and coded in a computer program. There are several reasons that accuracy can be limited. First, even if the physical phenomena can be described precisely with exact mathematics (e.g., applying fundamental laws of physics), computers will solve these equations in a *discretized space* rather than in a continuum. The accuracy of the solution depends on how fine-grained this discretization is made. The smaller the grain size, the better the accuracy, but also the more compute-intensive the problem becomes. This trade-off limits the accuracy for a given problem size and available compute power. Second, one typically cannot rely only on fundamental laws of physics, but instead must use less complex models that describe the behavior at a more abstract level. These abstractions are less detailed and, hence, less accurate than the underlying phenomena. Third, coding the models as computer programs introduces additional inaccuracy since one may need to derive linear equations from non-linear models, or the programmer may choose approximate algorithms that are faster, have already been developed, or have proven more reliable than the exact ones.

6.2 Steps from the Original Problem to its Computation by a von Neumann Machine

A typical scenario for developing a scientific or engineering computer application is as follows. First, a model is developed to describe in precise terms the phenomenon to be computed. For example, to investigate the temperature distribution in a car engine block, the engineer will describe mathematically the temperature flow in the material, given certain initial temperatures and the shape of the engine parts. To contain the complexity within reasonable limits, the engineer will make simplifying assumptions. Such assumptions could be that the material is homogeneous, the geometry is simple, and the initial temperatures are well known. An important class of model equations are partial differential equations (or PDEs). The PDE at hand may describe that, in any time interval, the temperature flow between two adjacent points in the car engine is some coefficient times the temperature difference since the beginning of the time interval. In actuality, the PDE describes this situation for only one point in space and time. The mathematical solution of the PDE needs to be developed such that the temperature behavior of the entire body over the desired time period can be determined. To do this precisely is mathematically complex and intractable for non-trivial geometries and surface temperatures.

The idea behind the computer solution is to split the engine block into a finite number of intervals (called a *grid* or *mesh*) and divide the time period into small steps. The computation then steps through time, updating the temperature at each grid point from its neighbor points (called the *stencil*) as described by the PDE. The fact that this is done on a finite interval instead of on the point described by the PDE makes it an approximation. The finer the grid space the more accurate the approximation becomes, so that building grids with the right spacing is an important and difficult issue. Ideally, grids are dense where the values being computed are expected to change significantly (e.g., in corners of the engine block) and sparse in "uninteresting" areas.

This computation is typically represented as operations on large matrices. Computer algorithms that manipulate such matrices and the corresponding large systems of equations are important. Of particular importance are linear algebra methods because they are well understood and there exist many algorithms for their solution. Many numerical methods are known to solve problems such as systems of linear and non-linear equations, linear least squares, eigenvalue problems, interpolation, integration, differentiation, ordinary and partial differential equations, and Fourier transforms. Such algorithms often are available in the form of software libraries, which application designers will use to the maximum extent possible.

Building applications from libraries alone is not sufficient. Additional software modules need to be developed to perform input and output operations, to orchestrate the library calls, to arrange data in the form necessary for the library calls, and to implement methods that are not found in libraries or for which library algorithms are not accurate or fast enough. Developing this additional code can significantly increase the software costs.

Fortran is the classical language for CSE applications. Although it is continuously being updated (Fortran77, Fortran90, Fortran95) and incorporates many features of modern programming languages, there is a trend to express new CSE applications in C and C++. In addition to these standard languages, there are many dialects that allow the programmer to exploit key features of specific machines. For example, there are several Fortran dialects that provide elements for exploiting parallel machine architectures.

Programming languages are translated by a compiler into the low-level machine code (see *Program Compilers*). The degree of sophistication of such a compiler can be an important consideration for the programmer. For example, Fortran compilers have been developed that can take advantage of parallel computer architectures by performing *automatic program parallelization*. Even for single processors, the degree of optimization that compilers are capable of performing can differ substantially between applications. The consequence is that the performance of applications on today's von Neumann computers can vary greatly.

7 Conclusions

The fundamental ideas embodied in the traditional von Neumann architecture have proven to be amazingly robust. Enhancements and extensions to these ideas have led to tremendous improvements in the performance of computer systems over the past 50 years. Today, however, many computer researchers feel that future improvements in computer system performance will require the extensive use of new, innovative techniques, such as parallel [15] and speculative execution. In addition, complementing software technology needs to be developed that can lower the development costs of an ever-increasing range of potential applications. Nevertheless, it is likely that the underlying organizations of future computer systems will continue to be based on the concepts proposed by von Neumann and his contemporaries.

References

[1] William Aspray. John von neumann's contributions to computing and computer science. *Annals of the History of Computing*, 11(3):189–195, 1989.

[2] William Aspray. *John von Neumann and the Origins of Modern Computing.* The MIT Press, Cambridge, Mass., 1990.

[3] Paul Ceruzzi. Electronics technology and computer science, 1940–1975: A coecolution. *Annals of the History of Computing*, 10(4):257–275, 1989.

[4] James Cortada. Commercial applications of the digital computer in american corporations, 1945-1995. *IEEE Annals of the History of Computing*, 18(2):19–29, 1996.

[5] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. *International Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, 1995.

[6] Boelie Elzen and Donald MacKenzie. The social limits of speed: The development and use of supercomputers. *IEEE Annals of the History of Computing*, 16(1):46–61, 1994.

[7] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[8] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett Publishers, Boston, MA, 1995.

[9] James M. Feldman and Charles T. Retter. *Computer Architecture: A Designer's Text Based on a Generic RISC.* McGraw-Hill, Inc., New York, 1994.

[10] M. D. Godfrey and D. F. Hendry. The computer as von Neumann planned it. *IEEE Annals of the History of Computing*, 15(1):11–21, 1993.

[11] John P. Hayes. *Computer Organization and Design (Second Edition).* McGraw-Hill, Inc., New York, 1988.

[12] Vincent P. Heuring and Harry F. Jordan. *Computer Systems Design and Architecture.* Addison Wesley Longman, Menlo Park, CA, 1997.

[13] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. *International Symposium on Computer Architecture*, pages 136–145, 1992.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann, San Mateo, CA, 1995.

[15] Kai Hwang. *Advanced Comptuer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., New York, NY, 1993.

[16] Donald E. Knuth. *The Art of Computer Programming. Vol 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.

[17] David J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[18] Emerson W. Pugh and William Aspray. Creating the computer industry. *IEEE Annals of the History of Computing*, 18(2):7–17, 1996.

[19] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, San Mateo, CA, 1994.

[20] Brian Randell. The origins of computer programming. *IEEE Annals of the History of Computing*, 16(4):6–15, 1994.

[21] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *International Symposium on Computer Architecture*, pages 414–425, 1995.

[22] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[23] Merlin G. Smith. IEEE Computer Society: Four decades of service. *IEEE Computer*, 1991.

[24] Nancy Stern. *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers.* Digital Press, Bedford, Mass., 1981.

[25] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[26] Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. *International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, 1996.

[27] Steven VanderWiel and David J. Lilja. When caches are not enough: Data prefetching techniques. *IEEE Computer*, 30(7):23–30, July 1997.

[28] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, 1991.

[29] Heinz Zemanek. Another look into the future of information processing, 20 years later. *Annals of the History of Computing*, 12(4):253–260, 1990.