ENCYCLOPEDIA OF

Physical Science and Technology

THIRD EDITION

Computer Hardware

AP

Cluster Computing

Thomas Sterling
California Institute of Technology and NASA Jet Propulsion Laboratory

I. Introduction
II. A Taxonomy of Cluster Computing
III. A Brief History of Cluster Computing
IV. Cluster Hardware Components
V. Cluster Software Components
VI. Summary and Conclusions

GLOSSARY

Cluster A computing system comprising an ensemble of separate computers (e.g., servers, workstations) integrated by means of an interconnection network cooperating in the coordinated execution of a shared workload.

Commodity cluster A cluster consisting of computer nodes and network components that are readily available COTS (commercial off-the-shelf) systems and that contain no special-purpose components unique to the system or a given vendor product.

Beowulf-class system A commodity cluster implemented using mass-market PCs and COTS network technology for low-cost parallel computing.

Constellation A cluster for which there are fewer SMP nodes than there are processors per node.

Message passing A model and methodology of parallel processing that organizes a computation in separate concurrent and cooperating tasks coordinated by means of the exchange of data packets.

CLUSTER COMPUTING is a class of parallel computer structure that relies on cooperative ensembles of independent computers integrated by means of interconnection networks to provide a coordinated system capable of processing a single workload. Cluster computing systems achieve high performance through the simultaneous application of multiple computers within the ensemble to a given task, processing the task in a fraction of the time it would ordinarily take a single computer to perform the same work. Cluster computing represents the most rapidly growing field within the domain of parallel computing due to its property of exceptional performance/price. Unlike other parallel computer system architectures, the core computing elements, referred to as *nodes*, are not custom designed for high performance and parallel processing but are derived from systems developed for the industrial, commercial, or commodity market sectors and applications. Benefiting from the superior cost effectiveness of the mass production and distribution of their *COTS* (commercial off-the-shelf) computing nodes, cluster systems exhibit order-of-magnitude cost advantage with respect to their custom-designed parallel computer counterparts delivering the same sustained performance for a wide range of (but not all) computing tasks.

I. INTRODUCTION

Cluster computing provides a number of advantages with respect to conventional custom-made parallel computers for achieving performance greater than that typical of uniprocessors. As a consequence, the emergence of clusters has greatly extended the availability of high-performance processing to a much broader community and advanced its impact through new opportunities in science, technology, industry, medical, commercial, finance, defense, and education among other sectors of computational application. Included among the most significant advantages exhibited by cluster computing are the following:

- *Performance scalability.* Clustering of computer nodes provides the means of assembling larger systems than is practical for custom parallel systems, as these themselves can become nodes of clusters. Many of the entries on the Top 500 list of the world's most powerful computers are clusters and the most powerful general-purpose computer under construction in the United States (DOE ASCI) is a cluster to be completed in 2003.
- *Performance to cost.* Clustering of mass-produced computer systems yields the cost advantage of a market much wider than that limited to the high-performance computing community. An order of magnitude price-performance advantage with respect to custom-designed parallel computers is achieved for many applications.
- *Flexibility of configuration.* The organization of cluster systems is determined by the topology of their interconnection networks, which can be determined at time of installation and easily modified. Depending on the requirements of the user applications, various system configurations can be implemented to optimize for data flow bandwidth and latency.
- *Ease of upgrade.* Old components may be replaced or new elements added to an original cluster to incrementally improve system operation while retaining much of the initial investment in hardware and software.
- *Architecture convergence.* Cluster computing offers a single general strategy to the implementation and application of parallel high-performance systems independent of specific hardware vendors and their product decisions. Users of clusters can build software application systems with confidence that such systems will be available to support them in the long term.
- *Technology tracking.* Clusters provide the most rapid path to integrating the latest technology for high-performance computing, because advances in device technology are usually first incorporated in mass market computers suitable for clustering.
- *High availability.* Clusters provide multiple redundant identical resources that, if managed correctly, can provide continued system operation through graceful degradation even as individual components fail.

Cluster computing systems are comprised of a hierarchy of hardware and software component subsystems. Cluster hardware is the ensemble of compute nodes responsible for performing the workload processing and the communications network interconnecting the nodes. The support software includes programming tools and system resource management tools. Clusters can be employed in a number of ways. The master–slave methodology employs a number of slaved compute nodes to perform separate tasks or transactions as directed by one or more master nodes. Many workloads in the commercial sector are of this form. But each task is essentially independent, and while the cluster does achieve enhanced throughput over a single processor system, there is no coordination among slave nodes, except perhaps in their access of shared secondary storage subsystems. The more interesting aspect of cluster computing is in support of coordinated and interacting tasks, a form of parallel computing, where a single job is partitioned into a number of concurrent tasks that must cooperate among themselves. It is this form of cluster computing and the necessary hardware and software systems that support it that are discussed in the remainder of this article.

II. A TAXONOMY OF CLUSTER COMPUTING

Cluster computing is an important class of the broader domain of parallel computer architecture that employs a combination of technology capability and subsystem replication to achieve high performance. Parallel computer architectures partition the total work to be performed into many smaller coordinated and cooperating tasks and distribute these tasks among the available replicated processing resources. The order in which the tasks are performed and the degree of concurrency among them are determined in part by their interrelationships, precedence constraints, type and granularity of parallelism exploited, and number of computing resources applied to the combined tasks to be conducted in concert. A major division of parallel computer architecture classes, which includes cluster computing, includes the following primary (but not exhaustive) types listed in order of their level of internal communication coupling measured in terms of bandwidth (communication throughput) and latency (delay in transfer of data). This taxonomy is illustrated in Figure 1. Such a delineation is, by necessity, somewhat idealized because many actual parallel computers may incorporate multiple forms of parallel structure in their specific architecture. Also, the terminology below reflects current general usage but the specific terms below have varied in their definition over time (e.g., "MPP" originally was applied to fine-grain SIMD computers, but now is used to describe large MIMD computers).

1. *Vector processing.* The basis of the classical supercomputer (e.g., Cray 1), this fine-grain architecture pipelines memory accesses and numeric operations through one or more multistage arithmetic units supervised by a single controller.

2. *Systolic.* Usually employed for special-purpose computing (e.g., digital signal and image processing), systolic systems employ a structure of logic units and physical communication channels that reflect the computational organization of the application algorithm control and data flow paths.

3. *SIMD.* This *Single instruction stream, multiple data stream* or *SIMD* family employs many fine- to medium-grain arithmetic/logic units (more than tens of thousands), each associated with a given memory block (e.g., Maspar-2, TMC CM-5). Under the management of a single system-wide controller, all units perform the same operation on their independent data each cycle.

4. *MPP.* This *multiple instruction stream, multiple data stream* or *MIMD* class of parallel computer integrates many (from a few to several thousand) CPUs (central processing units) with independent instruction streams and flow control coordinating through a high-bandwidth, low-latency internal communication network. Memory blocks associated with each CPU may be independent of the others (e.g., Intel Paragon, TMC CM-5), shared among all CPUs without cache coherency (e.g., CRI T3E), shared in SMPs (symmetric multiprocessors) with uniform access times and cache coherence (e.g., SGI Oracle), or shared in DSMs (distributed shared memory) with nonuniform memory access times (e.g., HP Exemplar, SGI Origin).

5. *Cluster computing.* Integrates stand-alone computers devised for mainstream processing tasks through local-area (LAN) or system-area (SAN) interconnection networks and employed as a singly administered computing resource (e.g., Beowulf, NOW, Compaq SC, IBM SP-2).

6. *Distributed Internet computing.* Employs wide-area networks (WANs) including the Internet to coordinate multiple separate computing systems (possibly thousands of kilometers apart) under independent administrative control in the execution of a single parallel task or workload. Previously known as *metacomputing* and including the family of GRID management methods, this emergent strategy harnesses existing installed computing resources to achieve very high performance and, when exploiting otherwise unused cycles, superior price/performance.

Cluster computing may be distinguished among a number of subclasses that are differentiated in terms of the source of their computing nodes, interconnection networks, and dominant level of parallelism. A partial classification of the domain of cluster computing includes commodity clusters (including Beowulf-class systems), proprietary clusters, open clusters or workstation farms, super clusters, and constellations. This terminology is emergent, subjective, open to debate, and in rapid transition. Nonetheless, it is representative of current usage and practice in the cluster community.

A definition of commodity clusters developed by consensus is borrowed from the recent literature and reflects their important attribute; that they comprise components

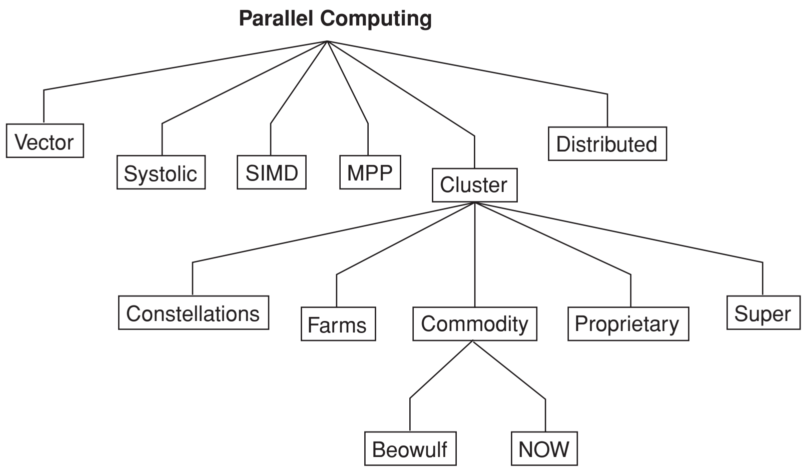


FIGURE 1 Taxonomy of cluster computing.

that are entirely off-the-shelf, i.e., already developed and available for mainstream computing:

A *commodity cluster* is a local computing system comprising a set of independent computers and a network interconnecting them. A cluster is *local* in that all of its component subsystems are supervised within a single administrative domain, usually residing in a single room and managed as a single computer system. The constituent *computer nodes* are commercial-off-the-shelf, are capable of full independent operation as is, and are of a type ordinarily employed individually for stand-alone mainstream workloads and applications. The nodes may incorporate a single microprocessor or multiple microprocessors in a symmetric multiprocessor (SMP) configuration. The *interconnection network* employs COTS LAN or SAN technology that may be a hierarchy of or multiple separate network structures. A cluster network is dedicated to the integration of the cluster compute nodes and is separate from the cluster's external (worldly) environment. A cluster may be employed in many modes including but not limited to high capability or sustained performance on a single problem, high capacity or throughput on a job or process workload, high availability through redundancy of nodes, or high bandwidth through multiplicity of disks and disk access or I/O channels.

Beowulf-class systems are commodity clusters employing personal computers (PCs) or small SMPs of PCs as their nodes and using COTS LANs or SANs to provide node interconnection. A Beowulf-class cluster is hosted by an open source Unix-like operating system such as Linux. A Windows-Beowulf system runs the mass-market widely distributed Microsoft Windows operating systems instead of Unix.

Proprietary clusters incorporate one or more components that are custom-designed to give superior system characteristics for product differentiation through employing COTS components for the rest of the cluster system. Most frequently proprietary clusters have incorporated custom-designed networks for tighter system coupling (e.g., IBM SP-2). These networks may not be procured separately (unbundled) by customers or by OEMs for inclusion in clusters comprising other than the specific manufacturer's products.

Workstation farms or open clusters are collections of previously installed personal computing stations and group shared servers, loosely coupled by means of one or more LANs for access to common resources, that, although primarily employed for separate and independent operation, are occasionally used in concert to process single coordinated distributed tasks. Workstation farms provide superior performance/price over even other cluster types in that they exploit previously paid-for but otherwise unused computing cycles. Because their interconnection network is shared for other purposes and not optimized for parallel computation, these open clusters are best employed for weakly interacting distributed workloads. Software tools such as Condor facilitate their use while incurring minimum intrusion to normal service.

Super clusters are clusters of clusters. Principally found within academic, laboratory, or industrial organizations that employ multiple clusters for different departments or groups, super clusters are established by means of WANs integrating the disparate clusters into a single more loosely coupled computing confederation.

Constellations reflect a different balance of parallelism than conventional commodity clusters. Instead of the primary source of parallelism being derived from the number of nodes in the cluster, it is a product of the number of processors in each SMP node. To be precise, a constellation is a cluster in which there are more processors per SMP node than there are nodes in the cluster. While the nodes of a constellation must be COTS, its global interconnection network can be of a custom design.

Of these, commodity clusters have emerged as the most prevalent and rapidly growing segment of cluster computing systems and are the primary focus of this article.

III. A BRIEF HISTORY OF CLUSTER COMPUTING

Cluster computing originated within a few years of the inauguration of the modern electronic stored-program digital computer. SAGE was a cluster system built for NORAD under an Air Force contract by IBM in the 1950s based on the MIT Whirlwind computer architecture. Using vacuum tube and core memory technologies, SAGE consisted of a number of separate stand-alone systems cooperating to manage early warning detection of hostile airborne intrusion of the North American continent. Early commercial applications of clusters employed paired loosely coupled computers with one performing user jobs while the other managed various input/output devices.

Breakthroughs in enabling technologies occurred in the late 1970s, both in hardware and software, that were to have a significant long-term effect on future cluster computing. The first generations of microprocessors were designed with the initial development of VLSI technology and by the end of the decade the first workstations and personal computers were being marketed. The advent of Ethernet provided the first widely used LAN technology, creating an industry standard for a modest cost multidrop interconnection medium and data transport layer. Also at this time, the multitasking Unix operating system was created at AT&T Bell Labs and extended with virtual memory and network interfaces at UC Berkeley. Unix was adopted in its various commercial and public domain forms by the scientific and technical computing community as the principal environment for a wide range of computing system classes from scientific workstations to supercomputers.

During the decade of the 1980s, increased interest in the potential of cluster computing was marked by important experiments in research and industry. A collection of 160 interconnected Apollo workstations was employed as a cluster to perform certain computational tasks by the NSA. Digital Equipment Corporation developed a system comprising interconnected VAX 11/750s, coining the term *cluster* in the process. In the area of software, task management tools for employing workstation farms were developed, most notably the Condor software package from the University of Wisconsin. The computer science research community explored different strategies for parallel processing during this period. From this early work came the communicating sequential processes model more commonly referred to as the *message-passing model*, which has come to dominate much of cluster computing today.

An important milestone in the practical application of the message passing model was the development of PVM (parallel virtual machine), a library of linkable functions that could allow routines running on separate but networked computers to exchange data and coordinate their operation. PVM, developed by Oak Ridge National Laboratory, Emory University, and University of Tennessee, was the first major open distributed software system to be employed across different platforms. By the beginning of the 1990s, a number of sites were experimenting with clusters of workstations. At the NASA Lewis Research Center, a small cluster of IBM workstations was used to simulate the steady-state behavior of jet aircraft engines in 1992. The NOW (Network of Workstations) project at UC Berkeley began operation of the first of several clusters there in 1993 that led to the first cluster to be entered on the Top 500 list of the world's most powerful computers. Also in 1993, one of the first commercial SANs, Myrinet, was introduced for commodity clusters, delivering improvements in bandwidth and latency an order of magnitude better than the Fast Ethernet LAN most widely used for the purpose at that time.

The first Beowulf-class PC cluster was developed at NASA's Goddard Space Flight Center in 1994 using early releases of the Linux operating system and PVM running on 16 Intel 100-MHz 80486-based PCs connected by dual 10-Mbps Ethernet LANs. The Beowulf project developed the necessary Ethernet driver software for Linux and additional low-level cluster management tools and demonstrated the performance and cost effectiveness of Beowulf systems for real-world scientific applications. That year, based on experience with many other message-passing software systems, the parallel computing community set out to provide a uniform set of message-passing semantics and syntax and adopted the first MPI standard. MPI has become the dominant parallel computing programming standard and is supported by virtually all MPP and cluster system vendors. Workstation clusters running the Sun Microsystems Solaris operating system and NCSA's PC cluster running the Microsoft NT operating system were being used for real-world applications.

In 1996, the Los Alamos National Laboratory and the California Institute of Technology with the NASA Jet Propulsion Laboratory independently demonstrated sustained performance of more than 1-Gflops for Beowulf systems costing under $50,000 and was awarded the Gordon Bell Prize for price/performance for this accomplishment. By 1997 Beowulf-class systems of more than 100 nodes had demonstrated sustained performance of greater than 10 Gflops with a Los Alamos system making the Top 500 list. By the end of the decade, 28 clusters were on the Top 500 list with a best performance of more than 500 Gflops. In 2000, both DOE and NSF announced awards to Compaq to implement their largest computing facilities, both clusters of 30 and 6 Tflops, respectively.

IV. CLUSTER HARDWARE COMPONENTS

Cluster computing in general and commodity clusters in particular are made possible by the existence of cost-effective hardware components developed for mainstream computing markets. The capability of a cluster is determined to first order by the performance and storage capacity of its processing nodes and the bandwidth and latency of its interconnection network. Both cluster node and cluster network technologies evolved during the 1990s and now exhibit gains of more than two orders of magnitude in performance, memory capacity, disk storage, and network bandwidth and a reduction of better than a factor of 10 in network latency. During the same period, the performance-to-cost ratio of node technology has improved by approximately 1000. In this section, the basic elements of the cluster node hardware and the alternatives available for interconnection networks are briefly described.

A. Cluster Node Hardware

The processing node of a cluster incorporates all of the facilities and functionality necessary to perform a complete computation. Nodes are most often structured either as uniprocessor systems or as SMPs although some clusters, especially constellations, have incorporated nodes that were distributed shared memory (DSM) systems. Nodes are distinguished by the architecture of the microprocessors employed, the number and organization of the microprocessors, the capacities of the primary and secondary storage, and the internal interconnect logic structure. The nodes of commodity clusters marketed primarily for mainstream computing environments must also incorporate standard interfaces to external devices that ensure interoperability with myriad components developed by third-party vendors. The use of the high-bandwidth interface allows clusters to be configured with little or no change to the node subsystem, minimizing any additional costs incurred on a per-node basis. The key elements of a node are briefly discussed below. It must be understood that this technology is evolving rapidly and that the specific devices that are provided as examples are likely to be upgraded in operational characteristics or to be replaced altogether in the near future.

1. *Central processing unit.* The CPU is a single VLSI integrated circuit microprocessor, possibly merged on an MCM (multichip module) with one or more cache chips. The CPU executes sequences of binary instructions operating on binary data, usually of 32- or 64-bit length. While many instructions are performed on internal data stored in registers, acquiring new data from the memory system is an important aspect of microprocessor operation, requiring one or more high-speed cache memories to minimize the average load/store access times. Both 32-bit and 64-bit architectures are used in clusters with the most popular based on the 32-bit Intel X86 family and the highest performance clusters based on the 64-bit Compaq Alpha family or IBM RS6000. The first Beowulf-class commodity clusters incorporated Intel 80486 microprocessors operating at 100 MHz. Today, descendents of this chip including the Intel Pentium III and the AMD K7 Athelon have clock rates in excess of 1 GHz. The CPU connects to an internal memory bus for high-speed data transfers between memory and CPU and to an external I/O bus that provides interfaces to secondary storage and networking control modules.

2. *Main memory.* Stores the working data and program instructions to be processed by the CPU. It is a part of a larger memory hierarchy that includes high-speed cache memories closer to the CPU and high-density persistent mass storage from which it acquires its initial data and stores its final results. For the last two decades, main memory has been dominated by DRAM technology, closely packed arrays of switched capacitive cells embedded on silicon wafers. DRAM chips containing 256 Mbits of data are available with gigabit chips to become common place in the near future. Typical cluster nodes support main memory capacities between 64 Mbytes and 1 Gbytes although large SMP or DSM nodes provide more. DRAM has undergone significant advances in recent years providing more rapid throughput as well as higher density, reducing if not closing the bottleneck between CPU and its main memory.

3. *Secondary storage.* Comprises a set of devices that provides persistent storage of a large amount of data. Secondary storage serves several purposes as a function of the usage of the data it contains. It provides all of the functions, both user applications and operating system tools, that govern the operation and computation of the CPU. It provides the data sets on which the user tasks are to operate and is the primary repository for the final results of user computations. It maintains configuration data concerning the setup and operational parameters of the computing node as well as information concerning the rest of the cluster devices and their relational roles. Because most memory systems support the virtual memory abstraction, providing a logical memory many times larger than the actual physical main memory installed, secondary storage temporarily holds those segments of the logical address space and associated data that do not fit in the existing physical main memory. Unlike main memory, data stored on secondary storage devices are retained, even when system power is disrupted. This nonvolatile property allows data to be archived indefinitely. The primary component type providing secondary storage is the venerable hard disk with its early genesis in the late 1950s based on magnetic storage (like a cassette tape) of one or more disks rotating on a single spindle at high speed and accessed by a magnetic detection head moved radically in and out across the disk surface, reminiscent of the arm of an old record turntable. Modern disk drives provide many tens of gigabytes at moderate cost and access times on the order of a few milliseconds. Other technologies are employed to provide more specialized forms of secondary storage, particularly for data portability and safe permanent archival storage. CD-ROMs developed from the original digital musical recording media provide approximately 600 Mbytes of storage at less than $1 a disk and read–write capability is now becoming commonplace, although this is of less importance to cluster systems. The long-lived and relatively diminutive floppy disk holding a mere 1.4 Mbytes is still employed, even on clusters, primarily for initial installation, configuration, and boot up.

4. *External interfaces.* Serve three important roles related to the operation and management of clusters. They provide direct user interactive access and control, they permit application data input and results to be conveyed with devices outside the system, and they connect to the cluster interconnection network and thereby to other nodes in the cluster. While there are many different types of interfaces (just look at the number of sockets on the back of a typical PC), PCI is universal from PCs to mainframes, connecting

the CPU to a plethora of interface control devices. The PCI bus has four different configurations employing 32- or 64-bit connections running at 33- or 66-MHz clock speed and a peak data throughput of 4 Gbps. The majority of network interface controllers (NIC) are compatible with one or more of these PCI forms. In the future, it is likely that a new external interface standard, Infiniband, will eventually replace PCI to deliver higher throughput and lower latency between the CPU and external devices.

B. Cluster Network Hardware

A model, but not necessarily the only possible model, of parallel processing with cluster systems involves each of the cluster nodes performing one or more tasks on local data and then exchanging the computed results with other nodes within the cluster. Networks make this possible. They provide physical channels between nodes by which data are transported and logical protocols that govern the flow and interpretation of the transferred data. Networks are employed in a broad range of integrated systems from the Internet spanning the globe requiring possibly as much as a hundred milliseconds for a message packet to reach its destination to a data bus internal to a computer integrating its various components supporting data transfers in 100 nanosec or less, a ratio of a million in network latency. Networks for commodity clusters fall in between with the initial use of Ethernet exhibiting on average approximately 100 nanosec latency falling in the middle (logarithmically speaking).

Network technology determines the potential value of cluster computing. Its principal properties are bandwidth, latency, scale, and cost. Bandwidth imposes an upper bound on the amount of data that can be transferred in unit time (e.g., Mbps, Gbps). Latency is the amount of time it takes for a message packet to transit the diameter of a system measured in microseconds. Cost is usually considered as the percentage of the total price of the hardware system. Scale is the largest number of nodes that a network can connect effectively. Together, they establish a cluster's capability, applicability, and user accessibility. Different applications exhibit varying global data access patterns that may be suitable for some networks rather than others. Higher bandwidth networks ordinarily will have greater generality of application than those networks of lower bandwidth. Similarly, for applications using short messages or involving frequent global synchronization, lower latency networks will be more general purpose than high-latency networks. But superior behavioral properties often come at additional cost that may preclude their use in many environments, where cost is a significant factor in the choice to employ clusters in the first place. Thus the selection of a specific network is dependent on how the cluster is to be used and by whom.

A cluster network includes NICs that connect the cluster node to the network, transport layer links to carry the data, and switches that route the data through the network. NICs move data from message buffers filled by the node processor to signal packets sent out to the transport layer performing a number of translation functions on the data in the process. The data links may comprise one or more parallel channels and may be implemented with metal coaxial cable or optical fiber (advanced development of free-space optical networks is under way). Switches accept messages at their multiple input ports, determine their required routing, switch as many as possible simultaneously sending them out the appropriate output ports, and arbitrating where contention for shared resources (ports, channels) occurs. The earliest Beowulf-class systems used low-cost hubs, rather than the more expensive switches, but these permitted only one transfer to occur at a time on the entire network. Switches deliver much closer to the peak bi-section bandwidth of the network as they isolate separate disjoint paths from each other.

Together these network components can be structured to form a number of different topologies. Most simple among these and used frequently for small clusters is the star configuration using a single switch of degree *n* (the number of separate ports) connecting *n* nodes. Larger systems can be formed with a hierarchy of switches to form a tree structure. The scale of tree-based clusters is limited by the bi-section bandwidth of the root node of the tree topology. More complex network structures permit the implementation of larger systems. Among them is the CLOS network (also referred to as the fat-tree) that overcomes the deficiency of the tree topology by providing multiple channels in parallel, balanced to keep the cross-section bandwidth equal at each level in the tree. Mesh and toroidal topologies provide scalable bandwidth and locality of interconnect with fixed degree nodes but may experience relatively high latency across the system diameter. Variations on these and other network topologies are possible and depend on requirements of a given system. A few of the most widely used network technologies used in commodity clusters are described next.

1. *Ethernet* is the most widely used network for clusters, even today, although devised as a LAN and originated in the late 1970s. Its success is due in part to its repeated reinvention, which takes advantage of technology advances while meeting expanding requirements. The 10-Mbps Ethernet that was first used in Beowulf clusters in the early 1990s superceded early Ethernet at 3 Mbps. Fast Ethernet provided 100 Mbps and with low-cost switches is the mainstay of small low-cost Beowulf-class systems.

Gigabit Ethernet, as the name implies, provides a peak bandwidth of approximately 1 Gbps. But its per-node cost remains high and it suffers from the relatively long latencies of its predecessors.

2. *Myrinet* was one of the first networks to be developed expressly for the SAN and cluster market. With a cost of approximately $1600 per node, Myrinet was initially reserved for the more expensive workstation clusters. But with its superior latency properties of 20 μsec or less, it permitted some classes of more tightly coupled applications to run efficiently that would perform poorly on Ethernet-based clusters. More recently, reduced pricing has expanded its suitability to lower cost systems and has proven very popular.

3. *VIA* is a recent advance in cluster network technology involving improvements in both hardware and software to further reduce data communication latency. Typically, message packets are copied from the user application space into the operating system space or vice versa. VIA (virtual interface architecture) employs a zero-copy protocol, avoiding the O/S intermediate stage and moving the packets directly between the network transport layer and the application. Giganet's cLAN and Compaq's Server net II both implement the VIA standard, delivering best case latencies well below 10 μsec.

4. *SCI* was perhaps the first SAN to achieve IEEE standardization and has very good bandwidth and latency characteristics. Existing implementations provide between 3.2- and 8-Gbps peak bandwidth with best latencies below 4 μsec. The SCI standard includes protocol for support of distributed shared memory operation. However, most clusters employing SCI use PCI-compatible network control cards (e.g., Dolphin) that cannot support cross-node cache coherence. Nonetheless, even in distributed memory clusters, it provides an effective network infrastructure.

5. *Infiniband* is the next-generation interconnection technology to extend the capabilities of SANs. Although not yet available, an industrial consortium of major computer technology (hardware and software) manufacturers has developed and released an extensive specification that will lead first to reference implementations, and eventually to widely distributed products. Bandwidths up to 12 Gbps (employing optical channels) and latencies approaching 1 μsec will become possible with Infiniband, which replaces previous I/O buses (e.g., PCI) and migrates the network interconnect closer to the memory bus of the compute node.

V. CLUSTER SOFTWARE COMPONENTS

The earliest use of commodity clusters involved little more software than the original node operating system and basic support for a network interface protocol such as sockets in Unix. Application programmers running a single problem on a small, dedicated cluster would hand craft the parallel program and painstakingly install the code and necessary data individually on every node of the cluster system. Good results were obtained for real-world problems on Beowulf-class systems and other such clusters, motivating continued advances in cluster hardware and methodology. Today, with commodity clusters contending for dominance of the high-performance computer arena, such primitive frontier techniques can no longer be justified and, indeed, would present a serious obstacle to wider usage of commodity clusters. During the intervening period, significant advances in software support tools have been developed for cluster computing. These are in the two critical areas of programming environments and resource management tools. Together, they provide the foundation for the development of sophisticated and robust cluster system environments for industrial, commercial, and scientific application.

The environments and tools described below engage the system as a global ensemble, treating its processing nodes as a set of compute and storage resources to be managed, allocated, and programmed. But each node is itself a complete and self-sustaining logical as well as physical entity, hosting its own environment: the node operating system. While some experimental clusters incorporate custom operating systems derived expressly for use within the cluster context, the vast majority of commodity clusters employ nodes hosting conventional operating systems. Many operating systems have been used in support of clusters. The IBM AIX operating system used on their SP-2 and Compaq True64 used on their Alpha-based SC series are two examples of vendor software migrated to use with clusters. However, the dominant operating systems employed with commodity clusters are Linux and Microsoft Windows. Linux emerged as the software of choice as a result of the Beowulf Project, which implemented the first clusters using Linux and running real-world science and technical applications. Linux gained prominence because of its Unix-like structure, which was consistent with the technical computing community's environments from scientific workstations to supercomputers and because of its free open source code policy. Microsoft Windows, the world's single most widely used operating system, has been favored in business and commerce environments for clusters using ISV applications software developed for Windows such as distributed transaction processing. Windows has also been used effectively for technical computing clusters at NSCA and Cornell Theory Center. Both IBM and Compaq as an alternative cluster node operating system to their proprietary software have adopted Linux.

A. Programming Environments

Parallel programming of clusters involves a sequence of steps that transforms a set of application requirements into a set of cooperating concurrent processes and data sets. Although actual programming styles may vary significantly among practitioners, a representative methodology may be the following process:

1. Capture the application in a set of ordered routines.
2. Partition the global data into separate approximately equal regions.
3. Define tasks to be performed on each data partition.
4. Determine precedence constraints between tasks of different regions.
5. In the programming language of choice, write the sequence of statements that encodes the tasks to be performed.
6. In the global communication medium of choice, set up synchronization conditions that will govern guarded program execution.
7. Devise procedures for exchanging necessary intermediate results among concurrent tasks.
8. Create a minimalist test data set and debug compile and runtime program errors.
9. Monitor program behavior and optimize code for best performance.
10. Partition real-world data set.
11. On selected cluster, allocate nodes to data partitions.
12. Install data and tasks on designated nodes.
13. Initiate execution and acquire result values.

These steps are rarely performed in such rigid lock-step manner but all of the actions described must be accomplished prior to successful completion of executing a real parallel problem on a cluster. The effectiveness achieved in programming a cluster is difficult to measure (although some metrics have been devised to this end). Nonetheless, the ease of parallel programming is strongly influenced by the execution model assumed and the tools available to assist in the process.

Many models have been conceived in the last two decades (or more) to provide a conceptual framework for parallel program execution and programming. These have been strongly influenced by the assumptions of the characteristics of the underlying parallel computer. Pipelined supercomputers used vector models, SIMD machines used fine-grain data parallel programming, SMP systems used coarse-grain multiple threads with shared memory synchronization (e.g., open MP), and large MPPs used single-program, multiple data stream (SPMD) style (e.g., HPF) with either put/get shared memory primitives or message passing for interprocessor cooperation. Because of the relatively long global latencies and constrained network bandwidth characteristic of clusters, the programming paradigms of widest usage have been the master–slave model for embarrassingly parallel job streams of independent tasks (e.g., transaction processing, web search engines) and the message-passing model for cooperating interrelated processes. Where clusters consist of SMP nodes, hybrid models are sometimes used employing message passing between nodes and multiple threads within the nodes.

Efficient programming practices demand effective programming environments that incorporate a set of sophisticated tools to support the steps listed above. A partial list of the desired tools based on a message-passing approach might include these:

- A core language and compiler (e.g., C, Fortran)
- A language sensitive editor (e.g., Emacs)
- A linkable message-passing library [e.g., MPICH (*http://www-unix.mcs.anl.gov/mpi/mpich*), LAM (*http://www.mpi.nd.edu/lam*)]
- Numeric libraries [e.g., Scalapack (*http://www.netlib.org/scalapack*)]
- Debuggers [e.g., gdb, Totalview (*http://www.etnus.com*)]
- Performance profilers [e.g., jumpshot (*http://www-unix.mcs.anl.gov/mpi/mpich*), XPVM (*http://epm.ornl.gov/pvm*)]
- Loaders and process distribution [e.g., Scyld (*http://www. scyld.com*), Rocks (*http://slic01.sdsc.edu*), vasystemimager (systemimager.org), OSCAR (*http://openclustergroup.org*), etc.]
- Schedulers [e.g., LSF (*http://www.platform.com*), PBS (*http://www.openpbs.org*), Condor (*http://www.cs.wisc.edu/condor*), etc.]

Such environments and tools are in a state of flux with a combination of free open-source and commercial offerings in continuous development yielding constant improvements in functionality, performance, and reliability. But new tools that provide more complete support, especially for parallel debugging, are still required. One important trend is toward the development of PSEs or problem-solving environments that target specific application domains and provide the programmer with a high-level framework within which to cast the problem. Other programming styles for clusters such as HPF, BSP, Split-C, and UPC are also being pursued and applied by some communities, although it is unclear which if any of these will become dominant.

One challenge that complicates programming clusters is the use of nodes comprising more than one processor in an SMP configuration. Such nodes view their local memory as common, sharing the name space within the node through hardware support of cache coherence. In principle, this allows computing within each node to employ a threaded shared memory model rather than the message-passing model. One would expect that such mixed-mode programming could yield superior performance. Surprisingly, this is not the mainstream practice. The majority of users of clusters of SMP nodes program intranode operation with message-passing operations such as those provided by MPI as they do internode processing. One widely used SMP programming methodology is Open MP, which provides the added constructs necessary for parallel programming on a shared memory multiprocessor. Some programmers, striving to take advantage of both clusters and their SMP nodes, employ a hybrid programming methodology consisting of both MPI and Open MP. This has not become common practice but is expected to grow in usage, in spite of its difficulties.

B. Resource Management Software

The management of cluster computers includes many responsibilities from initial assembly and software installation to possible dynamic load balancing of user application modules. Originally, users of moderate-scale low-cost Beowulf-class performed most of these chores manually, ignoring some as unnecessary for dedicated use. But modern clusters supporting multiple users and a range of application and workload types on systems scaled to hundreds of gigaflops require sophisticated environments and tools to manage the plethora of system resources. Programming tools, while still in transition, have achieved a level of community-wide standardization. Such is not the case for resource management tools. These are still in a state of experimentation although there is general consensus on the basic requirements. The principal capabilities needed for commercial grade resource management include the following:

- *Assembly, installation, and configuration.* Setting up a cluster, whether assembled on site by staff or vendor provided, can benefit from a set of low-level tools that organize the task of installing the large suite of software and configuring the large number of system parameters. Maintaining consistency across all of the nodes can be facilitated by routines that search and validate all copies and their version numbers.
- *Scheduling and allocation.* Loading a parallel application program on to a cluster shared by other users and jobs requires software tools that determine which resources will be employed to perform what jobs and when. Far more complicated than on a conventional uniprocessor, cluster scheduling involves space sharing where a system is physically partitioned into multiple subsystems to run as many jobs.
- *System administration.* The management of user accounts, job queues, security, backups, mass storage, log journaling, operator interface, user shells, and other housekeeping activities are essential elements of a commercial-grade computing system but impose added burden due to the multiplicity of computing resources and the diverse ways in which they may be used. PBS is an example of one software system that brings much of this capability to cluster computing.
- *Monitoring and diagnosis.* The complex state of a cluster, its operational status, tasks being performed, and its performance are all constantly changing. Operator tools are required to continuously monitor the behavior of the many components comprising a cluster system and quickly diagnosing hardware or software failures when they occur. A number of such tool sets have been developed by many cluster installations although no single suite has been adopted by the community as a whole.
- *Parallel mass storage.* Almost all computations require access to secondary storage including both local and remote disk drives for support of file systems. Commercial applications frequently use their own in-house distributed software for file management optimized around the specific needs of the application. Examples of general-purpose parallel file systems used for clusters include PPFS (*http://www-pablo.cs.uiac.edu/Project/PPFS/PPFSII/PPFSIIOverview.htm*), PVFS (*http://parlweb.parl.Clemson.edu/pvfs*), and GPFS (*http://gfs.lcse.umn.edu*).
- *Reliability.* Checkpoint and restart support software allows large programs with long run times to survive transient or hard failures of system components. Individual organizations and some vendors have developed some support for this capability although a general solution is not widely available. More difficult is the detection of errors. Software fault tolerance is a field in which strides are being made but which is still largely experimental.

While substantial advances have been made during the early 2000s, continued research and development are required to produce a common cluster environment that satisfies the requirements of a broad user community and exhibits sufficient reliability to garner confidence in the robustness and therefore utility of commodity clusters for commercial and industrial grade processing. A number of efforts are under way to synthesize a number of tools into common frameworks including the Oscar, Grendel, and RWCP projects. More than one ISV offers collections of cluster middleware including PGI and Scyld. In the long term, one or more advanced programming models and their complementing runtime execution models will have to be developed before commodity clusters become significantly easier to program and perhaps more efficient as well for a broader range of application algorithms. The development of PSE for a range of widely used application classes may provide partial solution to the challenge of programming, at least for those special cases. PSE software presents a template to the user who then fills in the parameters with data relevant to the specific problem to be performed. This eliminates the need for detailed program development and reduces the time to solution as well as providing improved efficiency of operation. But for general-purpose cluster computing, significant improvements in ease of use may depend on next-generation parallel programming formalisms.

VI. SUMMARY AND CONCLUSIONS

Commodity cluster computing is growing rapidly both for high-end technical and scientific application domains and for business and commerce. The low cost, high flexibility, and rapid technology tracking are making this class of computing the platform of choice for many user domains requiring scalability and excellent price/performance. The extraordinary rate of growth in capability for commodity clusters in general and Beowulf-class systems in particular is anticipated to continue for at least the next 5 years. By 2005 to 2006, price/performance may reach \$0.10 per megaflops with systems as large as 50 teraflops operational at a few sites. There is a strong likelihood that Linux and Microsoft Windows will be the mainstream operating systems, with one or the other offered by virtually every system vendor. Both are also likely to incorporate advancements that directly enhance cluster scalability and efficiency by eliminating bottlenecks and reducing overhead. Network bandwidths of 10 Gbps will have become commonplace with network latency approaching 1 μsec through the implementation of Infiniband. MPI-2 will be the ubiquitous programming model for parallel applications and .NET or an equivalent tool set will be employed for loosely coupled workloads, primarily in the commercial sector. Packaging will become cheaper and more compact to reduce footpad and overall system cost. Finally, systems administration tools will have reached the sophistication of mainstream servers. In 10 years, the first petaflops-scale commodity clusters will have been installed.

SEE ALSO THE FOLLOWING ARTICLES

Computer Architecture • Computer Networks • Databases • Data Structures • Parallel Computing • Project Management Software

BIBLIOGRAPHY

Buyya, R. (1999). "High Performance Cluster Computing," Vol. 1, Prentice Hall, Upper Saddle River, NJ.

Gropp, W., Lusk, E., and Skjellum, A. (1999). "Using MPI: Portable Parallel Programming with the Message-Passing Interface," The MIT Press, Cambridge, MA.

Pfister, G. F. (1998). "In Search of Clusters," 2nd ed., Prentice Hall, Upper Saddle River, NJ.

Seifert, R. (1998). "Gigabit Ethernet: Technology and Applications for High-Speed LANs," Addison Wesley Longman, Reading, MA.

Sterling, T. L., Salmon, J., Becker, D. J., and Savarese, D. F. (1999). "How to Build a Beowulf: a Guide to the Implementation and Application of PC Clusters," The MIT Press, Cambridge, MA.

Compilers

Keith D. Cooper
Ken Kennedy
Linda Torczon
Rice University

I. Structure of a Compiler
II. Recognizing Valid Programs
III. Internal Representations
IV. Optimization
V. Code Generation
VI. Summary

GLOSSARY

Back end The final phase of compilation, where the program is translated from the compiler's intermediate representation into operations for the target machine.

Compiler A program that translates an executable program from one form to another.

Constant propagation An optimization that discovers, at compile time, expressions that must have known constant values, evaluates them, and replaces their run-time evaluation with the appropriate value.

Data-flow analysis A collection of techniques for reasoning, at compile time, about the flow of values at run-time.

Front end The initial stage of compilation, where the program is translated from the original programming language into the compiler's intermediate representation.

High-level transformations Transformations performed on an intermediate representation that is close to the source language in its level of abstraction.

Instruction selection The process of mapping the compiler's intermediate representation of the program into the target language produced by the compiler.

Lexical analysis That part of the compiler's front end that has the task of converting the input program from a stream of individual characters into a stream of words, or tokens, that are recognizable components of the source language. Lexical analysis recognizes words and assigns them to syntactic categories, much like parts of speech. The pass that implements lexical analysis is called a *scanner*.

List scheduling An algorithm for reordering the operations in a program to improve their execution speed. A list scheduler constructs a new version of the program by filling in its schedule, one cycle at a time. The scheduler must respect the flow of values in the original program and the operation latencies of the target machine.

Memory hierarchy management A collection of transformations that rewrite the program to change the order in which it accesses memory locations. On machines with cache memories, reordering the references can increase the extent to which values already in the cache are reused, and thus decrease the aggregate amount of tie spent waiting on values to be fetched from memory.

Optimizer The middle part of a compiler, it rewrites the program in an attempt to improve its *runtime* behavior. Optimizers usually consist of several distinct passes of analysis and transformation. The usual goal of an optimizer is to decrease the program's execution time; some optimizers try to create smaller programs as well.

Parallelization The task of determining which parts of the program are actually independent, and can therefore execute concurrently. Compilers that use these techniques usually treat them as high-level transformations, performing both analysis and transformations on a near-source representation of the program.

Programming languages Artificial (or formal) languages designed to let people specify algorithms and describe data structures, usually in a notation that is independent of the underlying target machine.

Regular expressions A mathematical notation for describing strings of characters. Efficient techniques can convert a collection of regular expressions into a scanner.

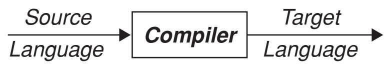
Semantic elaboration That part of a compiler's front end that checks semantic rules and produces an intermediate representation for the program. Often, semantic elaboration is coupled directly to the parser, where individual actions can be triggered as specific grammatical constructs are recognized.

Source-to-source translator A compiler that produces, as its target language, a programming language (rather than the native language of some target computer).

Syntax analysis That part of the compiler's front end that has the task of determining whether or not the input is actually a program in the source language. The parser consumes a stream of categorized words produced by the scanner and validates it against an internal model of the source language's grammatical structure (or syntax). The pass that implements syntax analysis is called a *parser.*

Vectorization A specialized form of parallelization that tries to expose computations suitable for SIMD, or vector, execution.

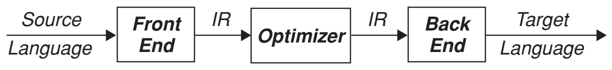
COMPUTER PROGRAMS are usually written in languages designed for specifying algorithms and data structures. We call such languages *programming languages.* On the other hand, the computer hardware executes operations, or *instructions*, that are much less abstract than the operations in the programming language, with set of available instructions varying from computer to computer. Before a program can execute, it must be translated from the programming language in which it is written into the machine language instructions for the computer on which it will run. The program that performs this translation is called a *compiler.*



Formally, a compiler is simply a program that takes as its input an executable program and produces as its output an equivalent program. For the purpose of discussing translation, we refer to the language of the input as the *source language* and the language of the output as the *target language.* The input program is typically written in some well-known programming language, such as Fortran, C, C++, Ada, Java, Scheme, or ML. The output program is rewritten into the set of native operations of a particular computer. (Some compilers use high-level languages as their target language. Such compilers are often called *source-to-source translators.*)

I. STRUCTURE OF A COMPILER

While compilers can be built as monolithic programs, more often, they are implemented as a series of distinct phases, generally organized into three major sections: a front end, an optional optimizer (or middle end), and a back end.



The *front end* analyzes the source program to determine whether or not it is well formed—that is, if it is a valid program in the source language. To accomplish this, it must first analyze each word in the program, classifying it into a syntactic category—a task called *lexical analysis.* It must then determine if the string of classified words fits into the grammatical structure of the language, a task called *syntax analysis.* In addition, it must check a variety of extrasyntactic rules to ensure that the source program has meaning, a process called *context-sensitive analysis.* Finally, it must construct an *internal representation* (IR) of the program for use by the later phases.

The *optimizer* takes the program, expressed in the compiler's IR, and produces an "improved" version of the program, usually expressed in the same IR. A program can be improved in different ways: it might be made to run faster; it might be made more compact in memory or secondary storage; or it might be transformed to consume less power. To produce the improved program, the optimizer must reason about how the program will behave when it executes. It can use this knowledge to simplify, specialize, and rearrange the elements of the internal form of the program. At each step in the process, it must preserve the meaning of the program, as expressed in its externally visible behavior.

The *back end* maps the program, expressed in the compiler's IR, into the target language. If the target is an instruction set for a particular computer, then the back end must account for that machine's finite resources and idiosyncratic behavior. This adds significant complexity to the task of code generation. The back end must select a set of operations to implement each construct in the program. It must decide where in the computer's memory each value will reside—a task complicated by the hierarchical memory systems of modern computers, and by the fact that the fastest locations, called registers, are extremely limited in number.[1] It must choose an execution order for the operations—one that preserves the proper flow of values between operations and avoids requiring the processor to wait for results. This may necessitate insertion of null operations to ensure that no operation begins to execute before its operands are ready. Many of the most challenging problems in compiler construction occur in the back end.

II. RECOGNIZING VALID PROGRAMS

Before the compiler can translate a source language program into a target machine program, it must determine whether or not the program is well formed—that is, whether the program is both grammatically correct and meaningful, according to the rules of the source language. This task—determining whether or not the program is valid—is the largest of the front end's tasks. If the front end accepts the program, it constructs a version of the program expressed in the compiler's internal representation. If the front end rejects the program, it should report the reasons back to the user. Providing useful diagnostic messages for erroneous programs is an essential part of the front end's work.

To address these issues, the typical front end is partitioned into three separate activities: lexical analysis, syntactic analysis, and context-sensitive analysis.

A. Lexical Analysis

A major difference between a programming language and a natural language lies in the mechanism that maps words into parts of speech. In most programming languages, each word has a unique part of speech, which the compiler can determine by examining its spelling. In a natural language, a single word can be mapped to several different parts of speech, depending on the context surrounding it. For example, the English words "fly" and "gloss" can be used as either noun or verb. The simpler rules used by programming languages permit the compiler to recognize and classify words without considering the grammatical context in which they appear.

To specify the spelling of words and their mapping into parts of speech, compiler writers use a formal notation called *regular expressions*. Regular expressions describe strings of symbols drawn from a finite alphabet, as well as ways to combine such strings into longer strings. First, any finite string of characters drawn from the alphabet is a regular expression. From these we can build longer strings by applying any of three rules. If r and s are regular expressions, then

1. $(r \mid s)$ is a regular expression, denoting a string that is either r or s
2. rs is a regular expression, denoting an occurrence of r followed immediately by an occurrence of s
3. r^* is a regular expression, denoting zero or more consecutive occurrences of r

Using regular expressions, we can compactly specify the spelling of fairly complex words. For example, since any specific word is a regular expression, we can specify the reserved words of a programming language, e.g., **define**, **do**, **if**, and **car**, simply by listing them. We can define a more complex construct, the counting numbers over the alphabet of digits, as $\mathbf{(1|2|3|4|5|6|7|8|9)\ (0|1|2|3|4|5|6|7|8|9)}^*$. (We read this as "a digit from 1 through 9 followed by zero or more digits from 0 through 9.") Notice that this regular expression forbids leading zeros.

We call the pass of the compiler that recognizes and classifies words the *scanner* or *lexical analyzer*. The scanner consumes a stream of characters and produces a stream of words; each annotated with its part of speech. In a modern compiler system, the scanner is automatically generated from a set of rules, specified by regular expressions. The scanners generated by this process incur small, constant cost per character, largely independent of the number of rules. For this reason, recognizers derived from regular expressions have found application in tools ranging from text editors to search engines to web-filtering software.

B. Syntactic Analysis

The job of the *syntax analyzer* is to read the stream of words produced by the scanner and decide whether or not that stream of words forms a *sentence* in the source language. To do this, the compiler needs a formal description of the source language, usually called a *grammar*. Informally, a grammar is just a collection of rules for deriving

[1](Possible cross reference to cache memory, registers, . . .)

sentences in some language. For example, the following grammar describes a class of simple English sentences:

1. Sentence → Subject Predicate period
2. Subject → noun
3. Subject → adjective noun
4. Predicate → verb noun
5. Predicate → verb adjective
6. Predicate → verb adjective noun

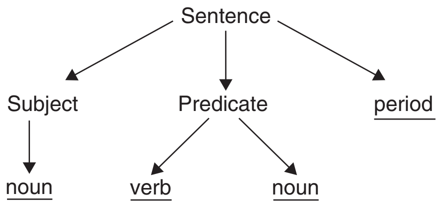
In these rules, underlined, lowercase symbols are *terminal symbols* for the grammar—that is, they are actual parts of speech for words that can appear in a valid sentence. The capitalized symbols are syntactic variables called *nonterminal symbols*. Nonterminals are distinguished by the fact that they appear on the left-hand side of one or more rules. Each nonterminal is defined by a set of rules. The nonterminals thus provide structure to the grammar. Each rule describes some aspect of an English sentence and an interpretation of the nonterminal that appears on its left-hand side. We read the first rule as "a Sentence is a Subject, followed by a Predicate, followed by a period." The next two rules establish two ways to construct a Subject. The final three rules give three options for building a Predicate.

Consider the sentence "Compilers are programs." It fits the simple grammar. To see this, first convert each word to its part of speech: "noun verb noun period." To derive a sentence with this structure, we can start with Sentence and use the rules to rewrite it into the desired sentence:

Grammatical Form	**Rule**
Sentence	—
→ Subject Predicate period	1
→ noun Predicate period	2
→ noun verb noun period	4

This derivation proves that any sentence with the structure "noun verb noun period" fits within the grammar's model of English. This includes "Compilers are programs" as well as "Tomatoes are horses."

We can depict this derivation graphically, as a *derivation tree*, sometimes called a *parse tree*.



The front end of a compiler does not need to derive sentences. Instead, it must solve the inverse problem—given a stream of words, construct its derivation tree. This process is called *parsing*. The parser uses a model of the source language's grammar to automatically construct a parse tree for the input program, if such a tree exists. If this process fails, the input program, as classified by the scanner, is not a sentence in the language described by the grammar.

Tools that construct an efficient parser from a grammar are widely available. These tools, called *parser generators*, automate most of the process of building the parser, and have simple interfaces to automatically derived scanners.

C. Context-Sensitive Analysis

The parser, alone, cannot ensure that the source program makes sense. Consider our English sentences: "Compilers are programs" and "Tomatoes are horses." Both fit the model embodied in our grammar for simple sentences; in fact, both have the same derivation. (Remember, the derivation operates on parts of speech, not the actual words.) However, the first sentence is a cogent comment on the nature of compilers while the second sentence is nonsense. The difference between them is not grammatical—it lies in properties of the words that the grammar cannot express. The verb "are" implies a relationship of equality or similarity. Compilers are, in fact, a specific kind of program. Tomatoes and horses are dissimilar enough that it is hard to accept them as equal.

Similar correctness issues arise in computer programs. Names and values have extra grammatical properties that must be respected. A correct program must use values in ways that are consistent with their definitions. Names can have distinct meanings in different regions of the program. The definition of a programming language must specify many properties that go beyond grammar; the compiler must enforce those rules.

Dealing with these issues is the third role of the front end. To succeed, it must perform *context-sensitive analysis*—sometimes called *semantic elaboration*. This analysis serves two purposes: it checks correctness beyond the level of syntax, as already discussed, and it discovers properties of the program that play an important role in constructing a proper intermediate representation of the code. This includes discovering the lifetime of each value and where those values can be stored in memory.

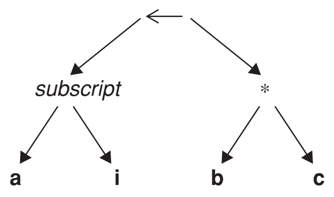
Specification-based techniques have not succeeded as well in context-sensitive analysis as they have in scanning and parsing. While formal methods for these problems have been developed, their adoption has been slowed by a number of practical problems. Thus, many modern compilers use simple ad hoc methods to perform context-sensitive analysis. In fact, most parser generator systems include substantial support for performing such ad hoc tasks.

III. INTERNAL REPRESENTATIONS

Once the compiler is broken into distinct phases, it needs an internal representation to transmit the program between them. This internal form becomes the definitive representation of the program—the compiler does not retain the original source program. Compilers use a variety of internal forms. The selection of a particular internal form is one of the critical design decisions that a compiler writer must make. Internal forms capture different properties of the program; thus, different forms are appropriate for different tasks. The two most common internal representations—the abstract syntax tree and three-address code—mimic the form of the program at different points in translation.

- The *abstract syntax tree* (AST) resembles the parse tree for the input program. It includes the important syntactic structure of the program while omitting any nonterminals that are not needed to understand that structure. Because of its ties to the source-language syntax, an AST retains concise representations for most of the abstractions in the source language. This makes it the IR of choice for analyses and transformations that are tied to source program structure, such as the high-level transformations discussed in Section VC.
- *Three-address code* resembles the assembly code of a typical microprocessor. It consists of a sequence of operations with an implicit order. Each operation has an operator, one or two input arguments, and a destination argument. A typical three-address code represents some of the relevant features of the target machine, including a realistic memory model, branches and labels for changing the flow of control, and a specified evaluation order for all the expressions. Because programs expressed in three-address code must provide an explicit implementation for all of the source language's abstractions, this kind of IR is well suited to analyses and transformations that attack the overhead of implementing those abstractions.

To see the difference between an AST and a three-address code, consider representing an assignment statement **a**[**i**] ← **b** ∗ **c** in each.Assume that **a** is a vector of 100 elements (numbered 0–99) and that **b** and **c** are scalars.



Abstract syntax tree

```
LOAD VAR       @b        ⇒ r1
LOAD VAR       @c        ⇒ r2
MULT           r1, r2    ⇒ r3
LOAD VAR       @i        ⇒ r4
MULTIMMED      r4, 4     ⇒ r5
ADDIMMED       r5, @a    ⇒ r6
STORE          r3        ⇒ r6
```

Three-address code

The AST for the assignment, shown on the left, captures the essence of the source-language statement. It is easy to see how a simple in-order treewalk could reprint the original assignment statement. However, it shows none of the details about how the assignment can be implemented. The three-address code for the assignment, shown on the right, loses any obvious connection to the source-language statement. It imposes an evaluation order on the statement: first **b**, then **c**, then **b** ∗ **c**, then **i**, then **a[i]** , and, finally, the assignment. It uses the notation@**b** to refer to **b**'s address in memory—a concept missing completely from the AST.

Many compilers use more than one IR. These compilers shift between representations so that they can use the most appropriate form in each stage of translation.

IV. OPTIMIZATION

A source language program can be mapped into assembly language in many different ways. For example, the expression **2** ∗ **x** can be implemented using multiplication, using addition (as **x** + **x**), or if **x** is an unsigned integer, using a logical shift operation. Different ways of implementing an operation can have different costs. Over an entire program, these cost differences can mount up.

Compiler optimization is the process by which the compiler rewrites the internal representation of a program into a form that yields a more efficient target-language program. The word "optimization" is a misnomer, since the compiler cannot guarantee optimality for the resulting code. In practice, optimizers apply a fixed sequence of analyses and transformations that the compiler writer believes will produce better code.

Optimizers attempt to improve the program by analyzing it and using the resulting knowledge to rewrite it. Historically, optimizers have primarily focused on making the program run faster. In some contexts, however, other properties of the program, such as the size of the compiled code, are equally important.

A. Static Analysis of Programs

To improve a program, the optimizer must rewrite the code (or its IR form) in a way that produces better a target language program. Before it can rewrite the code, however, the compiler must prove that the proposed transformation is *safe*—that is, the transformation does not change the results that the program computes—and likely to be *profitable*—that is, the resulting code will be an improvement over the existing code. To accomplish this, the compiler analyzes the program in an attempt to determine how it will behave when it runs. Because the compiler performs the analysis before the program runs, the analysis is considered a static analysis. In contrast, an analysis built into the running program would be a dynamic analysis.

Static analysis takes many forms, each one tailored to address a particular problem in the safety or profitability of optimization. One such problem is *constant propagation*—discovering variables whose run-time values can be determined at compile time. If the compiler can discover that **x** always has the value 2 at a particular point in the program, it can specialize the code that uses **x** to reflect that knowledge. In particular, if **x** has the value 2 in an expression **x** ∗ **y**, it can replace the expression with **2** ∗ **y** or with **y** + **y**, either of which may be faster than a generic multiply. (The former avoids referring to **x**, with a possible memory reference, while the latter also replaces the multiply operation with an addition, which may be cheaper.)

Analyses such as constant propagation are formulated as problems in *data-flow analysis*. Many data-flow problems have been developed for use in optimization. These include the following:

- *Finding redundancies*: An expression is *redundant* if, along any path that reaches it, that value has already been computed. If the compiler can prove that an expression is redundant, it can replace the expression with its previously computed value.
- *Live variable analysis*: A variable is *live* at some point in the code if there exists a path from that point to a use of its value. With the results of live variable analysis, the compiler can stop preserving a variable at the point where it stops being live.
- *Very-busy expressions*: An expression is *very busy* at some point if it will be used, along every path leaving that point before one of its inputs is redefined. Moving the expression to the point where it is very busy can eliminate unneeded copies of the expression, producing a smaller target program.

These problems are formulated as systems of equations. The structure of the equations is dictated by the control-flow relationships in the program. The solutions to the equations are found by using general solvers, analogous to Gaussian Elimination, or by using specialized algorithms that capitalize on properties of the program being analyzed.

B. Classic Scalar Transformations

To apply the results of analysis, the compiler must rewrite the code in a way that improves it. Hundreds of optimizing transformations have been proposed in the literature. These techniques address a wide range of problems that arise in the translation of programming languages. Examples include reducing the overhead introduced by source-language abstractions, managing hardware features such as memory, and general strategies that use properties such as commutativity and associativity to speed expression evaluation.

Among the transformations that you will find in a modern optimizer are as follows:

- *Useless code elimination*: The compiler may discover that some part of the source code is useless—either it cannot execute or its results are never used. In this case, the compiler can eliminate the useless operations. This may make the resulting code faster (because it no longer executes the operations). The resulting code should also be smaller. The presence of other transformations, such as constant propagation, inline substitution, and redundancy elimination, can create useless code.
- *Inline substitution*: In many cases, the compiler can replace an invocation of a procedure or function with a copy of the called procedure, after renaming variables to avoid conflicts and to enforce the parameter binding and mapping of the original call. This eliminates the overhead of the call itself. It may create opportunities to specialize the inlined code to the context that calls it. Because inline substitution can significantly increase the size of the program by duplicating code, it is most attractive when the called procedure is small, as often happens in object-oriented languages.
- *Constant folding*: If the compiler can determine, at compile time, the value of an expression in the program, it can eliminate the operations that evaluate the expression and replace any references to that expression directly with its value. This speeds up the program by avoiding computation. If the expression is used in a control flow decision, folding may lead to the elimination of larger code fragments.

To design an optimizer, the compiler writer must select a set of transformations to apply. For most transformations, several implementation techniques exist. Those distinct methods may require different kinds of analysis, may operate on different IRs, and may address different cases. Thus, the choice of specific algorithms for the transformations has an impact on how well the optimizer works.

Finally, individual transformations can create opportunities for further improvement; they can also foreclose such opportunities. Thus, selecting an order in which to apply the transformations has a strong impact on the optimizer's behavior.

C. High-Level Transformations

Some compilers apply aggressive transformations to reshape the code to perform well on a specific computer architecture. These transformations depend heavily on properties of the underlying architecture to achieve their improvement. Some of the problems that compilers attack in this way are as follows:

- *Vectorization*: A vector computer applies the same operation to different elements of the same array in parallel. To support vector hardware, compilers apply a series of transformations to expose loops that can be expressed as vector computations. Vectorization typically requires the compiler to determine that the execution of a particular statement in any iteration does not depend on the output of that statement in previous iterations.
- *Parallelization*: A parallel computer uses multiple processors to execute blocks of code concurrently. To make effective use of parallel computers, compilers transform the program to expose data parallelism—in which different processors can execute the same function on different portions of the data. Parallelization typically requires the compiler to determine that entire iterations of a loop are independent of one another. With independent iterations, the distinct processors can each execute an iteration without the need for costly interprocessor synchronization
- *Memory hierarchy management*: Every modern computer has a cache hierarchy, designed to reduce the average amount of time required to retrieve a value from memory. To improve the performance of programs, many compilers transform loop nests to improve data locality—i.e., they reorder memory references in a way that increases the likelihood that data elements will found in the cache when they are needed. To accomplish this, the compiler must transform the loops so that they repeatedly iterate over blocks of data that are small enough to fit in the cache.

All of these transformations require sophisticated analysis of array indices to understand the reference patterns in the program. This kind of analysis is typically performed on an AST-like representation, where the original structure of the array references is explicit and obvious. Most compilers that perform these optimizations carry out the analysis and transformation early in the compilation process. Because these transformations are tied, by implementation concerns, to a source-like representation of the code, they are often called *high-level transformations.*

V. CODE GENERATION

Once the front end and optimizer have produced the final IR program, the back end must translate it into the target language. The back end must find an efficient expression for each of the program's constructs in the target language. If the target language is assembly code for some processor, the generated program must meet all of the constraints imposed by that processor. For example, the computation must fit in the processor's register set. In addition, the computation must obey both the ordering constraints imposed by the flow of data in the program and those imposed by the low-level functioning of the processor. Finally, the entire computation must be expressed in operations found in the assembly language. The compiler's back end produces efficient, working, executable code. To accomplish this, it performs three critical functions: instruction selection, instruction scheduling, and register allocation.

A. Instruction Selection

The process of mapping IR operations into target machine operations is called *instruction selection.* Conceptually, selection is an exercise in pattern matching—relating one or more IR constructs into one or more machine operations.

If the details of the IR differ greatly from those of the target machine, instruction selection can be complex. For example, consider mapping a program represented by an AST onto a Pentium. To bridge the gap in abstraction between the source-like AST and the low-level computer, the instruction selector must fill in details such as how to implement a switch or case statement, which registers are saved on a procedure call, and how to address the ith element of an array. In situations where the abstraction levels of the machine and the IR differ greatly, or in situations where the processor has many different addressing modes, instruction selection plays a large role in determining the quality of the final code.

If, on the other hand, the details of the IR and the target machine are similar, the problem is much simpler. With a low-level, linear IR and a typical RISC microprocessor, much of the code might be handled with a one-to-one mapping of IR operations into machine operations. The low-level IR already introduces registers (albeit an unlimited number), expands abstractions like array indexing into their component operations, and includes the temporary names required to stitch binary operations into more complex expressions. In such situations, instruction selection is less important than high-quality optimization, instruction scheduling, and register allocation.

Instruction selectors are produced using three distinct approaches: *tree pattern matching, peephole optimization,* and *ad hoc, hand-coded solutions.*

- *Tree pattern matching* systems automatically generate a tree-matcher from a grammatical description of the tree. The productions in the grammar describe portions of the tree. Each production has an associated snippet of code and a cost for that snippet. The tree-matcher finds a derivation from the grammar that produces the tree and has the lowest cost. A postmatching pass uses the derivation to emit code, pasting together the snippets from the various productions.
- *Peephole-optimization systems* analyze and translate the IR, one small segment at a time. These matchers use a sliding window that they move over the code. At each step, the engine expands the operations into a detailed, low-level representation, simplifies the low-level code within the window, and compares the simplified, low-level IR against a library of patterns that represent actual machine operations. These matchers can work with either tree-like or linear IRs.
- *Ad hoc methods* are often used when the IR is similar in form and level of abstraction to the target machine code. Of course, this approach offers neither the local optimality of tree pattern matching nor the systematic simplification of peephole optimization. It also lacks the support provided by the tools that automate generation of both tree-matchers and peephole matchers. Still, compiler writers use this approach for simple situations.

B. Instruction Scheduling

On most computer systems, the speed of execution depends on the order in which operations are presented for execution. In general, an operation cannot execute until all of its operands are ready—they have been computed and stored in a location that the operation can access. If the code tries to execute an operation before its operands are available, some processors stall the later operation until the operand is ready. This delay slows program execution. Other processors let the operation execute. This strategy inevitably produces invalid or incorrect results.

Similarly, the compiled code might not make good use of all the functional units available on the target machine. Assume a machine that can execute one operation of type A and one operation of type B in parallel at each cycle, and that the compiled code contains the sequence AAABB-BAAABBB; then the processor can use just half of its potential. If the compiler can reorder the code into the sequence ABABABABABAB, then the computer might execute it in half the time—because at each step, it can execute an operation of type A and another of type B.

The goal of *instruction scheduling* is to reorder the operations in the target machine program to produce a faster running program. This task depends heavily on low-level details of the target machine. These details include the number of operations it can execute concurrently and their types, the amount of time it takes to execute each operation (which may be nonuniform), the structure of the register set, and the speed with which it can move data between registers and memory.

The instruction-scheduling problem is, in general, NP-complete; thus, it is likely that no efficient method can solve it optimally. To address this problem, compilers use greedy, heuristic scheduling methods. The most common of these methods is called *list scheduling*. It relies on a list of operations that are ready to execute in a given cycle. It repeatedly picks an operation from the list and places it into the developing schedule. It then updates the list of ready operations, and records when operations that depend on the just-scheduled operation will be ready. This method works on straight-line code (i.e., sequences of code that contain no branches). Many variations have been proposed; they differ primarily in the heuristics used to select operations from the ready list.

To improve the quality of scheduling, compilers use several variations on replication to create longer sequences of straight-line code. Profile-based techniques, such as *trace scheduling*, use information about the relative execution frequency of paths through the program to prioritize regions for scheduling. Loop-oriented techniques, such as *software pipelining*, focus on loop bodies under the implicit assumption that loop bodies execute more often than the code that surrounds them. Still other techniques use graphs that describe the structure of the program to select and prioritize regions for scheduling.

C. Register Allocation

Before the code can execute on the target computer, it must be rewritten so that it fits within the register set of the target machine. Most modern computers are designed around the idea that an operation will draw its operands from registers and will store its result in a register. Thus, if the computer can execute two operations on each cycle, and those operations each consume two values and produce another value, then the compiler must arrange to have four operands available in registers and another two registers available for results, in each cycle. To complicate matters further, processors typically have a small set of registers—32 or 64—and the ratio of registers per functional unit has been shrinking over the last decade.

Compilers address these issues by including a register allocator in the back end. Deferring the issue of register management until late in compilation permits the earlier phases largely ignore both the problem and its impact on code quality. Earlier phases in the compiler typically assume an unlimited set of registers. This lets them expose more opportunities for optimization, more opportunities to execute operations concurrently, and more opportunities for reordering. Once the compiler has explored those opportunities and decided which ones to take, the register allocator tries to fit the resulting program into the finite storage resources of the actual processor.

At each point in the code, the allocator must select the set of value that will reside in the processor's registers. It must rewrite the code to enforce those decisions—moving some values held in the unlimited, or virtual, register set into memory. When it moves some value into memory to make room in the register set—called *spilling* the value—it must insert code to store the value, along with code to retrieve the value before its next use. The most visible result of register allocation is the insertion of code to handle spilling. Thus, allocation usually produces a larger and slower program. However, the postallocation program can execute on the target machine, where the preallocation program may have used nonexistent resources.

The register allocation problem is NP-complete. Thus, compilers solve it using approximate, heuristic techniques. The most popular of these techniques operates via an analogy to graph coloring. The compiler constructs a graph that represents conflicts between values—two values conflict when they cannot occupy the same space—and tries to find a k-coloring for that conflict graph. If the compiler can find a k-coloring, for k equal to the number of registers on the processor, then it can translate the coloring into an assignment of those values into the k registers. If the compiler cannot find such a coloring, it spills one or more values. This modifies the code, simplifies the conflict graph, and creates a new coloring problem. Iterating on this process—building the graph, trying to color it, and spilling—produces a version of the program that can be colored and allocated. In practice, this process usually halts in two or three tries.

D. Final Assembly

The compiler may need to perform one final task. Before the code can execute, it must be expressed in the native language of the target machine. The result of instruction selection, instruction scheduling, and register allocation is a program that represents the target machine code. However, it may not have the requisite form of a target machine program. If this is the case, the compiler must either convert the program into the format from which it can be linked, loaded, and executed, or it must convert it into a form where some existing tool can do the job.

A common way of accomplishing this is to have the compiler generate assembly code for the target machine, and to rely on the system's assembler to convert the textual representation into an executable binary form. This removes much of the knowledge about the actual machine code from the compiler and lets many compilers for the machine share a single assembler.

VI. SUMMARY

A long-term goal of the compiler-building community has been to reduce the amount of effort required to produce a quality compiler. Automatic generation of scanners, of parsers, and of instruction selectors has succeeded; the systems derived from specifications are as good (or better) than handcrafted versions. In other areas, however, automation has not been effective. Adoption of specification-based tools for context-sensitive analysis has been slow. Few systems have tried to automatically generate optimizers, instruction schedulers, or register allocators—perhaps because of the intricate relationship between these tools, the IR, and the target machine's architecture.

A modern compiler brings together ideas from many parts of computer science and bends them toward the translation of a source language program into an efficient target language program. Inside a compiler, you will find practical applications of formal language theory and logic, greedy heuristic methods for solving NP-complete problems, and careful use of algorithms and data structures ranging from hash tables through union-find trees, alongside time-tested, ad hoc techniques. Compilers solve problems that include pattern recognition, unification, resource allocation, name–space managemnt, storage layout, set manipulation, and solving sets of simultaneous equations. Techniques developed for compiling programming languages have found application in areas that include text editors, operating systems, digital circuit design and layout, theorem provers, and web-filtering software.

SEE ALSO THE FOLLOWING ARTICLES

BASIC Programming Language • Cluster Computing • Computer Algorithms • Computer Architecture • Data Structures • Parallel Computing • Software Engineering

BIBLIOGRAPHY

Aho, A. V., Sethi, R., and Ullman, J. D. (1986).“Compilers: Principles, Tools, and Techniques,” Addison-Wesley, Reading, MA.

Cooper, K. D., and Torczon, L. (2002). “Engineering a Compiler,” Morgan-Kaufmann, San Francisco.

Fischer, C. N., and LeBlanc, R. J., Jr. (1991). “Crafting a Compiler with C,” Benjamin Cummings.

Morgan, C. R. (1998). “Building an Optimizing Compiler,” Digital Press.

Muchnick, S. S. (1997). “Advanced Compiler Design and Implementation,” Morgan-Kaufmann, San Francisco.

Wilhelm, R., and Maurer, D. (1995). “Compiler Design,” Addison-Wesley, Reading, MA.

Computer Architecture

Joe Grimes
California Polytechnic State University

I. Introduction
II. Historical Perspective
III. Levels of Abstraction
IV. Data Structures in a Computer System
V. Architecture Life Cycle
VI. Interrelation of Architecture Components
VII. Enhancing Architecture in the Trade-Off of Performance and Cost
VIII. Conclusions

GLOSSARY

Accumulator Usually a part of the arithmetic-logic unit (ALU) of a computer that is used for intermediate storage.

Address Name identifying a location where data or instructions may be stored.

Assembler Language translator program that converts assembly language instructions into conventional machine language.

Assembly language A computer language that is more easily understood than the language that is translated into in order for the computer to understand it.

Bus A set of lines shared by one or more components of a computer system. In many computers, a common bus interconnects the memory, processor, input, and output units. Because the bus is shared, only one unit may send information at a given time.

Cache Memory that is accessed more rapidly than main memory but less rapidly than a register. Today often found in processor chip with several levels of cache present in the computer.

Combinational circuit A digital circuit whose output value(s), at a given point in time, is dependent only on the input values at that time.

Compiler Language translator program that converts application program language instruction, such as Pascal or FORTRAN, into a lower level language.

Complex instruction set computer (CISC) A class of computers whose instruction set (conventional machine language instructions) was large, instructions were of multiple lengths, instructions had variable formats, instructions took variable lengths of time to execute, etc.

Computer architecture The study of the computer as viewed by the programmer.

Control That part of the processor that controls itself and the datapath of the processor.

Datapath The components of the datapath may include registers, arithmetic and logic unit (ALU), shifter, bus, and multiplexors. It also contains the buses necessary in order to transfer information from one of the components to another.

Debugger Tool used by a programmer to locate and correct an error in a computer program.

Exception An action in the computer that will ultimately result in the suspension of the execution of a program. The exception is caused by an activity outside of the program and is handled in such a way that the program can be resumed after the exception is handled.

Exception cycle That part of the instruction cycle that checks for exceptions and handles them if they are pending. The exception is handled by saving the suspended program's state and resumes execution of instructions in the exception handler routine (program).

Input Movement of data from an input device, such as a keyboard, to the processor or another unit of the computer system.

Interrupt Special control signal(s) that diverts the flow of instruction execution to the instruction in the location associated with the particular interrupt.

Memory A unit where information (data and instructions) may be stored.

Memory consistency The same data stored in different modules is correctly stored.

Memory cycle Operations required to store or retrieve information from memory.

Microprogram Program that interprets conventional machine language instructions that are interpreted directly by the hardware.

Operating system Set of programs that monitor and operate computer hardware, and simplify such tasks as input/output (I/O), editing, and program translation. Also serves as an interface with the user.

Output Movement of data from a unit of the computer system to an output device such as a printer.

Program counter Register in which the location of the current instruction is stored.

Reduced instructions set computer (RISC) This class of computer had features that were the opposite of those of the CISC computers.

Registers Memory locations that are internal to processor and that can be accessed quickly. Some registers are available for programmer while the processor only uses others internally.

Sequential circuit A digital circuit whose output value(s), at a given point in time, is dependent on the input values as well as the state of the circuit at that time. Memory units are sequential circuits.

Virtual memory Approach used to extend main memory by using a combination of main memory and secondary memory to store a program.

COMPUTER TECHNOLOGY has come a long way from the days when a 1940s computer, without stored program capability, required the space of a family room and living room in a large home, and when turned on, would dim the lights of all the homes in the community. Computer advances have made it possible for computers to be as much a part of our lives as our brains. Like our brains we often use computers without knowing that we are doing so. Today, a computer controls the digital alarm clock, controls the microwave, assists the checkout person in a store that stocks computer coded items, partially controls vehicles with sometimes in excess of fifty computer controllers, is used for computer-assisted instruction, and in general, substantially automates the modern work environment. A computer purchased today for a thousand dollars has more performance, functionality, and storage space than a computer purchased for several million dollars in the 1960s. These computer advances result from technological as well as design breakthroughs. Over the past 30–35 years, performance advances have averaged around 25% each year. Technological advances have been more consistent than those of computer architecture. In a sense, computer architecture had been in a state of hibernation until the mid-1980s because microprocessors of the 1970s were merely miniature clones of mainframes.

I. INTRODUCTION

Trends such as the use of vendor independent operating systems (UNIX), making company proprietary environments public, and a large reduction in assembly language programming have opened a window of opportunity for new architectures to be successful. Starting in the mid-1980s, these opportunities, along with quantitative analyses of how computers are used, a new type of architecture has reared its head. Because of this and as a result of improvements in compilers, enhancements of integrated circuit technology, and new architectural ideas, performance has more than doubled each year between mid-1980s and the present.

Most textbooks related to this subject have titles that encompass both the architecture and the organization of a computer system. There is a difference between computer architecture and computer organization with computer architecture referring to the programmer model (those aspects that have a direct impact on the logical execution of a program) and computer organization referencing the operational units and their interconnections that result in the architectural specification. Architectural issues relate to such things as the instruction set, data structures, memory addressing modes, interrupt capabilities, and I/O interfaces. Organizational features include those hardware capabilities unavailable to the programmer that pertain to the implementation of the architecture and the interconnection of the computer components.

The study of computer architecture amounts to analyzing two basic principles, performance and cost for a set of computer architecture alternatives that meet functionality requirements. This appears to be straightforward when considered at a general level, but when these topics are looked at in detail the water can become muddy. If a systematic approach is used the concepts can be straightforward. Fortunately, central processing units, memory hierarchy, I/O systems, etc., are layered, and these layers may be considered as levels of abstractions. Each level of abstraction has its own abstractions and objects. By studying aspects of computer science in this fashion, it is possible to censor out details that are irrelevant to the level of abstraction being considered and to concentrate on the task at hand.

An analogy to the following effect has been made: if cost and performance improvements in automobiles had kept up with computers, a car would cost ten cents and would allow travel between Los Angeles and New York in five minutes. If this pace is to continue, it is believed that computer design must be studied in a layered fashion using a systematic approach afforded by quantitative techniques. Also, it may be appropriate to adopt new models for studying computers that are different from that of the traditional Von Neumann model (Fig. 1) developed by John Von Neumann in 1940. This Von Neumann concept is still the basis of many computers today with many improvements such as miniaturization and other contributions to performance improvement. Therefore, this article has been written to present this approach and create the enthusiasm in the readers so that they will be biting at the bit to chip in with their own new ideas. This section strives to introduce necessary background material and many of the concepts that will be addressed in more depth throughout the rest of the article.

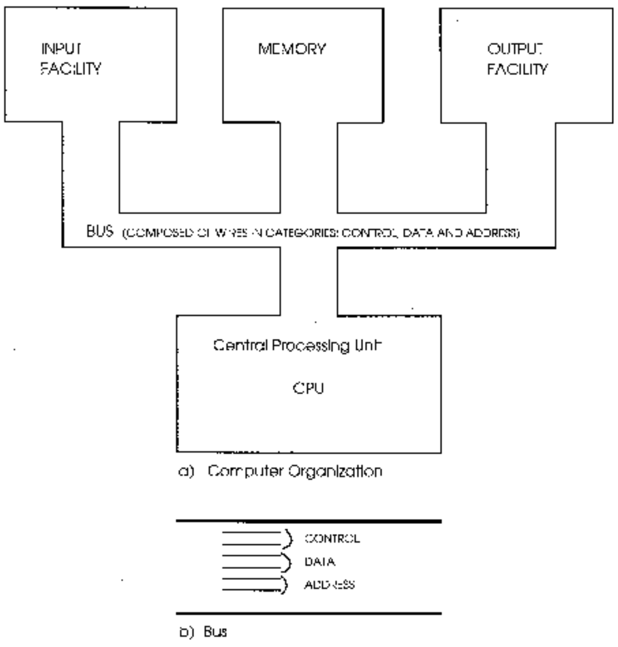


FIGURE 1 High-level Von Neumann computer organization.

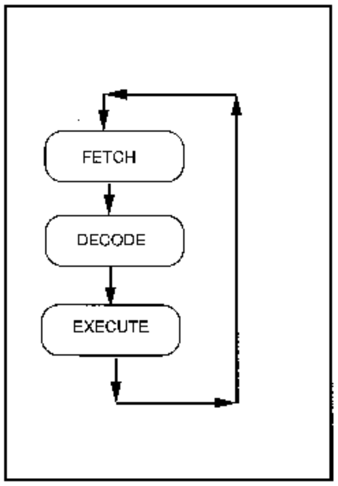


FIGURE 2 Block diagram of the instruction cycle for a computer.

It would be wonderful be able to come up with a good definition of a computer. It is impossible to give an all-encompassing definition, because a person working with computers continually refines his definition of it based on the increased understanding of its capabilities and enhancements that have occurred. A start at the definition might be Fig. 1, which gives a high-level view of the components of a Von Neumann computer with the five traditional components: datapath, control, memory, input, and output, with the former two sometimes combined and called the processor. Processors come in two flavors: imbedded processors that are internal to products such as radios, microwaves, automobiles, and standard processors that are part of a computer system. Every part of every computer can be classified under one of the five components. Computer architecture addresses how the programmer views these five components at the level of abstraction where they are programming and the computer organization addresses the implementation of the architecture in the inter- and intrarelationship of the components.

The fundamental responsibility of a computer is to execute a program, which consists of a set of instructions stored in the computer's memory. A computer operates by repeatedly performing an instruction cycle as depicted in a simple form in Fig. 2. The computer understands a simple set of instructions called an *instruction set* and cannot act on special cases that are not covered by the set of instructions. The computer would repeat this cycle over and over again for each instruction. This type of work is boring and best suited for a machine.

As mentioned previously, a computer has levels of abstractions. A person writing the instructions for the computer is a computer programmer. This view of the computer is at a high level without much detail about the way the computer operates. The person who would design a new path from the memory to the processor, or how the memory should handle the instruction cycle, would be viewing the computer at a much lower level of detail at the hardware level. An automobile would be a good analogy regarding levels of abstraction. The driver of the automobile views it at a much higher level of abstraction than the person designing any automobile system such as engine, steering, and brakes. A computer has the potential for performance improvements with performance inversely proportional to the time it takes to do a job. Performance is dependent on the architecture and organization of a computer, and should be traded off with cost.

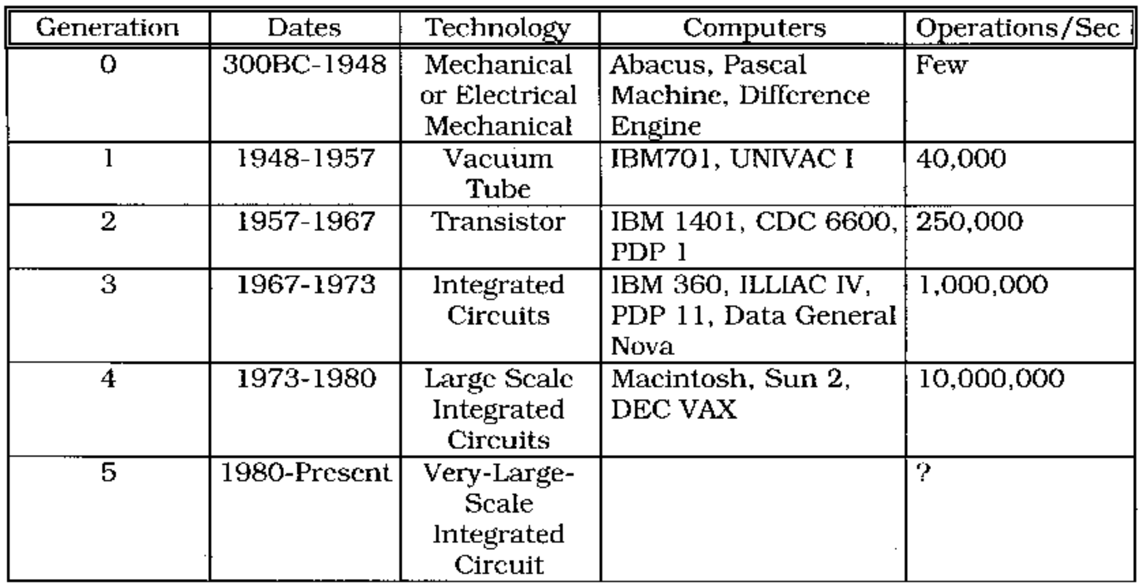
The task of the computer architect is to participate in the phases of the life cycle, described later, of architecture that satisfies computer-user functional requirements and yet are economically implementable using available technology. All of this is quite relative, as something that is too costly in one case may be perfectly reasonable in another case.

This article will deal with these and other aspects of the computer definition in more detail.

II. HISTORICAL PERSPECTIVE

Each writer on the topic of computer history has a different perspective to present. Their presentations vary in terms of dates of events and who was responsible for certain breakthroughs. Absolute dates are not as important in the study of computer architecture as a relative perspective of what has happened and how quickly it has occurred. One good way to gain a perspective on the milestones or developments in the computer industry is through the classification of computers by generation, with generations distinguished by the technology used in the implementation. Figure 3 presents the classical generation classification of computers by the technologies used. There is no solid agreement between historians on defining generations, especially the dates. The entry of operations per second is not meant to be absolute and is meant to give only a picture of the relative performance improvements. Although the generations concept gives an overview of the historical development of computers, it is worthwhile to look at the details further. There are two interesting ways to do this: development paths of contributors could be reviewed, or the evolution of the abstraction-layered computer could be traced. The abstraction-layered computer will be reviewed here because it is important to the evolution of computer architecture.

It is widely accepted that the first operational general purpose computer was built in the 1940s. Because of recent research and legal activities, there is some controversy over whether the credit for this should be given to John Atanasoff and his graduate student Berry for the



Generation	Dates	Technology	Computers	Operations/Sec
0	300BC-1948	Mechanical or Electrical Mechanical	Abacus, Pascal Machine, Difference Engine	Few
1	1948-1957	Vacuum Tube	IBM701, UNIVAC I	40,000
2	1957-1967	Transistor	IBM 1401, CDC 6600, PDP 1	250,000
3	1967-1973	Integrated Circuits	IBM 360, ILLIAC IV, PDP 11, Data General Nova	1,000,000
4	1973-1980	Large Scale Integrated Circuits	Macintosh, Sun 2, DEC VAX	10,000,000
5	1980-Present	Very-Large-Scale Integrated Circuit		?

FIGURE 3 Computer generations.

Atanasoff/Berry Computer built in the early 1940s at Iowa State University, or J. Presper Eckert and John Mauchly for their ENIAC (Electronic Numerical Integrator and Calculator) built in the mid-1940s at the Moore School of the University of Pennsylvania, because Mauchly had briefly visited Atanasoff before he built the ENIAC. Its basic principles were enunciated in a memorandum written by von Neumann in 1945 and, largely because of this widely circulated report, von Neumann's name alone has come to be associated with concept of the stored-program computer.

One of the remarkable features of the von Neumann report was its focus on the logical principles and organization of the computer rather than on the electrical and electronic technology required for its implementation. Thus the idea of architecture appears early in the modern history of the computer. The word "architecture" itself, however, appears to have been first used in the early 1960s by the designers of the IBM System/360 series of computers. They used this word to denote the logical structure and functional characteristics of the computer as seen by the programmer. Over the years since, the theory and practice of computer architecture has extended to include not only external characteristics but also the computer's internal organization and behavior. Thus, a computer is now commonly viewed as possessing both an "outer" and an "inner" architecture.

The original digital computers had two abstractions, digital logic and something comparable to the conventional machine level (programming in binary) of today, two levels of detail. Because the digital logic level was unstructured in design and unreliable, M. V. Wilkes, in the early 1950s, proposed an intermediate abstraction, microprogramming that would perform some of the hardware tasks. A few of these computers were constructed in the 1950s. When IBM introduced the 360 family (all 360 family computer had the same architecture, but the organization was different for each family member) of computers shortly after the publication *Datamation* made the clever prediction that the concept of microprogramming would never catch on, microprogramming caught on like wild fire and by early the 1970s most computers had the abstraction 2 present. Recently things have gone full circle with true RISC computers not having the microprogramming abstraction present.

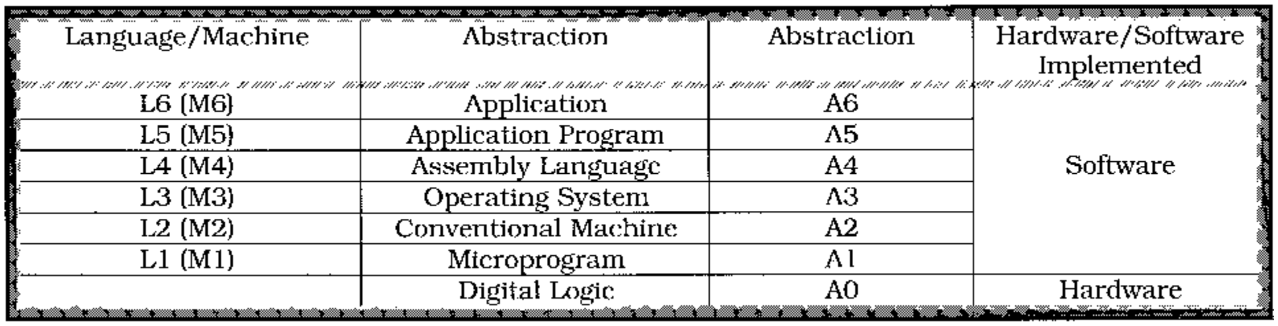
As mentioned previously, early computers had only two levels of abstraction. Each computer site had to develop their own program environment and it was a difficult as the computer manufacturer did not provide any software support. In the 1950s assemblers and compilers, supplied by computer vendors, were developed without operating systems. The 1950s programmer had to operate the computer and load the compiler or assembler along with the program into the computer. At this point there were three levels of abstraction: the digital logic, conventional machine language, and application (in the case where compilers were present).

Starting around 1960, operating systems were supplied by the computer manufacturer to automate some of the operator's tasks. The operating system was stored in the computer at all times and automatically loaded the required compiler or assembler from magnetic tape as needed. The programmer still had to supply the program and necessary control information (request the specific compiler) on punched cards. During this era, the programmer would place the program (deck of punched cards) in a tray, and later an operator would carry this tray with many other programs into the sacred computer room (room maintained in fixed temperature and humidity range with reduced dust level), run the programs, and return the results and programs to the individual programmers. Often the time between placing the cards in the tray and receiving the results may have been several hours (even days at the end of the university term). In the meantime, if the due date were close, the programmer would sweat blood hoping that the program would run properly. This type of operating system was called a batch system.

In future years, operating systems became more sophisticated. Time-sharing operating systems were developed that allowed multiple programmers to communicate directly with the computer from terminals at local or remote sites. Other capabilities added to the operating system included such things as new instructions called "operating system calls." In the case of the UNIX operating system, it interprets the "cat" instruction but it is capable of much more than that. As compilers became much more sophisticated and assembly language programming became less common, there was an evolution from the CISC to RISC computer, and as a result a reduction in need for the microprogramming abstraction.

III. LEVELS OF ABSTRACTION

A computer architect may try to be a Don Quixote of computer science and try to view all aspects of a computer system at one time, or can proceed in a manageable fashion and view it as a set of abstractions or layers. Contemporary computers may be viewed as having at least two abstractions, with the bottom abstraction considered by computer architects being the digital logic abstraction, abstraction 0. Below abstraction 0 it would be possible to consider the device abstraction or even below that, the physics abstraction with the theory of devices. The abstraction above digital logic on some computers is microprogramming, abstraction 1. Often digital logic and microprogramming



Language/Machine	Abstraction	Abstraction	Hardware/Software Implemented
L6 (M6)	Application	A6	
L5 (M5)	Application Program	A5	
L4 (M4)	Assembly Language	A4	Software
L3 (M3)	Operating System	A3	
L2 (M2)	Conventional Machine	A2	
L1 (M1)	Microprogram	A1	
	Digital Logic	A0	Hardware

FIGURE 4

abstractions are grouped together and called the control. On almost all computers there is a conventional machine language abstraction. Figure 4 displays the levels that are present on many modern computers. A true RISC computer would not have the microprogramming abstraction present. A programmer dealing with a particular abstraction has a distinct set of objects and operations present, and need not be concerned with the abstractions below or above. Because abstractions are interdependent, a computer designer must ultimately be concerned with all levels present but need only deal with one or two at any time.

The digital logic abstraction is present on all computers and it defines the language understood by the computer. This language (L1) is the microprogramming abstraction in the case of Fig. 4, where all the abstractions are present. These instructions are usually very simple, such as move data from one storage location to another, shift a number or compare two numbers, but they are laborious, boring (some people would disagree; computer science is an art and as with any area of art not all critics agree), and very subject to programming errors. These microinstructions are directly interpreted by the hardware (digital logic abstraction). At each abstraction, except A0, there is a language that defines a machine and vice versa. The language is often called the instruction set.

The next level of abstraction is the conventional machine language and it is more people friendly, more efficient for programming, and less subject to programming errors. For a machine with all the abstractions of Fig. 4 present, the conventional machine language is not understood by the digital logic abstraction and interpreted by the microprogram.

IV. DATA STRUCTURES IN A COMPUTER SYSTEM

In the computer, a computer byte is composed of 8 binary bits, and the computer word is composed of a number of bytes with the word length dependent on the computer. A user should read the manufacturer's documentation to determine the word size for a particular computer system. Therefore, a character, integer, or decimal number must be represented as a bit combination. There are some generally accepted practices for doing this:

- *Character*: Almost all computers today use the ASCII standard for representing character in a byte. For example "A" is represented by 65_{10}, "a" is represented by 97_{10}, and "1" is represented by 49_{10}. Most architecture textbooks will provide a table providing the ASCII representation of all character. In the past there were vendor-specific representations such as EBCDIC by IBM.
- *Integer*: The computer and user must be able to store signed (temperature readings) and unsigned (memory addresses) integers, and be able to manipulate them and determine if an error has occurred in the manipulation process. Most computers use a twos complement representation for signed numbers and the magnitude of the number to represent unsigned numbers.
- *Decimal number*: Decimal numbers are represented using a floating point representation with the most important one being the IEEE Standard 754, which provides both a 32-bit single and a 64-bit double precision representation with 8-bit and 11-bit exponents and 23-bit and 52-bit fractions, respectively. The IEEE standard has become widely accepted, and is used in most contemporary processors and arithmetic coprocessors.

The computer is a finite state machine, meaning that it is possible to represent a range of integers and a subset of the fractions. As a result, a user may attempt to perform operations that will result in numeric values outside of those that can be represented. This must be recognized and dealt with by the computer with adequate information provided to the user. Signed integer errors are called overflow errors, floating point operations can result in overflow or underflow errors, and unsigned integer errors are called carry errors.

The computers store numbers in twos complement or floating point representation because it requires less memory space. The operations are performed using these representations because the performance will always be better.

The computer architect must determine the algorithm to be used in performing an arithmetic operation and mechanism to be used to convert from one representation to another. Besides the movement of data from one location to another, the arithmetic operations are the most commonly performed operations; as a result, these arithmetic algorithms will significantly influence the performance of the computer. The ALU and Shifter perform most of the arithmetic operations on the datapath.

V. ARCHITECTURE LIFE CYCLE

A computer architecture has a life cycle that is analogous in nature to that of software. The phases might be considered:

1. *Planning*: The planning should include a needs assessment, cost assessment, feasibility analysis, and schedule.
2. *Specification*: The specification will provide a definition of the architectural requirements. Development of a test (validation) plan should be included.
3. *Architecture design*: The design phase will include such things as instruction set design and functional organization.
4. *Architecture implementation*: This portion of the life cycle is crucial to a correct instruction set. One key issue to be resolved in implementation is whether a feature should be implemented in software or hardware.
5. *Testing*: Testing will usually involve testing of units and the entire architecture.
6. *Maintenance*: Maintenance will include correcting errors and providing enhancements.

Each phase is important and should be given its due portion of the effort as a part of a process that may be cyclic rather than sequential through the phases. The life cycle may deal with one of the five major architecture components or even one of their subcomponents. As recommended by Amdahl's Law, most complete computer system developers have strived to provide a balance in the performance of the components, but memory components have lagged behind processor in performance. The computer architects strive for balance in the throughput and processing demands of the components. There are two design components that are ever changing:

A. The performance is changing in the various technology areas of the components, and rate of technology change differs significantly from one component type to another.
B. The new applications and new input/output devices constantly change the nature of the demand on the system in regard to instruction set requirements and data needs.

VI. INTERRELATION OF ARCHITECTURE COMPONENTS

The basic function of a computer is the execution of programs. The execution of a typical program requires the following:

- The movement of the program into memory from a disk or other secondary storage device
- Instructions and data are fetched by the processor from memory as needed
- Data is stored by the processor in memory as required
- Instructions are processed as directed by the program
- Communication occurs between the processor and the I/O devices as needed

All of this activity occurs as a result of instructions and is carried out by the control unit, and the five categories of activities occur (not as five sequential actions) at various points during the process of executing the program. The instructions that cause the program execution to happen may be any combination of operating system instructions and instructions in the program being processed.

A computer system consists of processor(s), memories, and I/O modules that are interrelated in an organized fashion. The interrelationship is successful if the communication between the processor(s), memories, and I/O modules is coordinated. That is, they must speak the same language and operate at a speed that is mutually acceptable. For example, two people communicating with each other can only do so if they speak at different times and in a language both understand. This section will discuss the processor, memory and I/O device and their interrelationship, and future sections will build on the topics initiated here.

A. Processor, Memory, I/O System, and the Interrelationship

A computer system is composed of components that may be classified as processor, memory, and I/O that interact or communicate with each other. There must be a path, logically depicted in Fig. 5, between these components in order for this communication to take place. Figure 5a

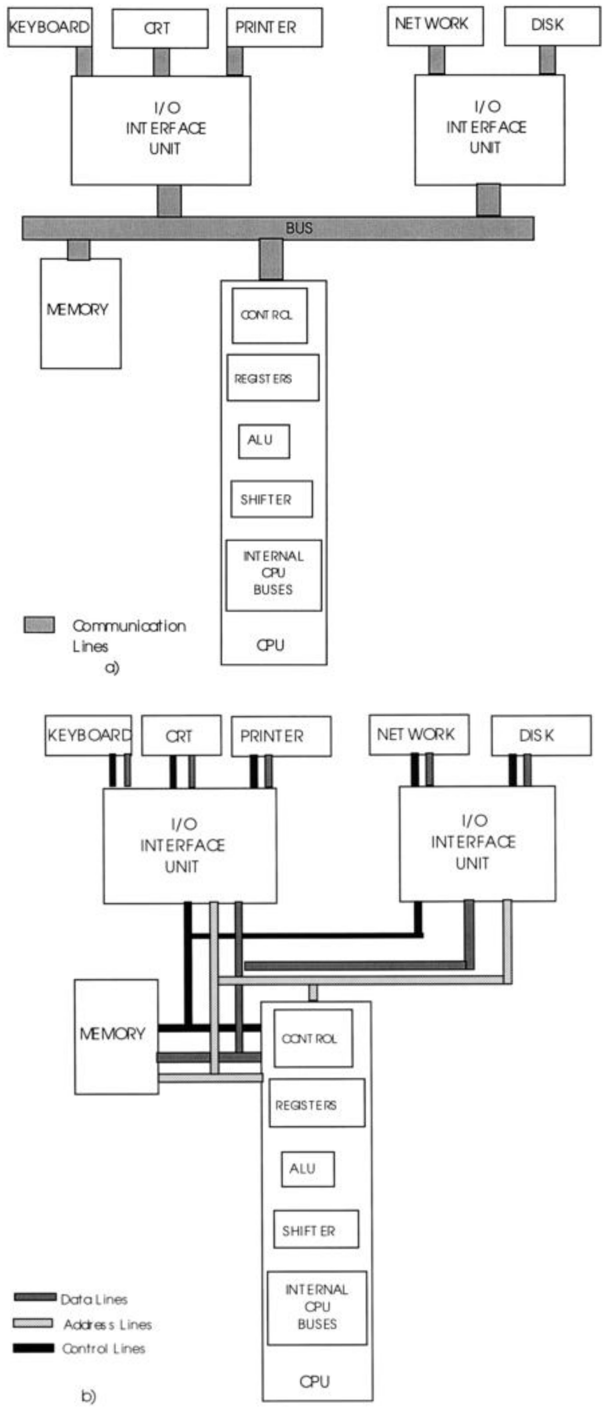


FIGURE 5 Interconnection between major computer components.

illustrates a simple single bus structure that may be used for the interaction between components, and Fig. 5b classifies the bus signals into the three possible categories of data, address, and control. The location of components relative to each other is important only in the sense that signals travel more rapidly between components that are closer to each other. Components that communicate with each other frequently should be organized so as to minimize the distance between them.

The physical connection between the components is often termed the interconnect structure. Timing considerations must be taken into account and often the interconnection will be more complex than just a set of wires and will include digital circuits used to coordinate the timing of data, address, and control signals between the components. The reason for this is because different components may operate at different speeds. The data lines are used to transfer information, instructions, and data, and are the sole reason for the existence of the communication address and control lines. It must be possible to move information in either direction between processor and memory or between I/O devices and processor or memory. The interrelationship between the components is defined as the interconnect structure as well as the signals that are transmitted between the components over the interconnection.

Figure 5 appears to allow the exchange of all types of signals in either direction. However, in practice, memory is only capable of sending data and the I/O devices do not send address signals; otherwise all other possible exchanges may occur. The following represents a summary of the activities of each of the components:

- *Processor*: The processor is the coordinator of the computer system. It processes abstraction 2 programs by fetching their instructions from memory, decoding them, and executing them. The control portion of the processor is like a director of a play and coordinates the activities of all other portions of the processor, and the other computer system components working from a script called the instructions of a program. The control unit may be strictly hardware, or a combination of hardware and software (microprogram). The processor controls all other activities if it is only processor in the computer system. If multiple processors exist, then one must be the master or arbitration occurs between the processors for control of the other components.
- *Memory*: The memory, often called the store or storage, is the location where information is stored. *Registers*, cache, RAM, ROM, disks, and tapes are different types of memory. The early computers had only one register and it was called an accumulator. However, in the context used so far in this chapter the memory component will be either ROM or RAM. The memory component will contain N words, with each word containing a fixed number of bits and located at one of the addresses $0, 1, 2, \ldots, N-1$. Memory locations could be compared to tract houses, because all of them look the same and can only be

distinguished by their addresses. To address N locations, it is necessary to have $\ln_2 N$ address lines between the processor and memory.

- *I/O modules*: The I/O modules are composed of the I/O interface unit and the peripheral(s). From the user's view, the I/O modules receive all the instructions and data from the computer user. From the processor point of view, the I/O module is analogous to the memory component in that it will be accessed at a specific address and information can be read/written to it. The I/O modules are an intermediary between the processor and I/O devices to off load some of the processor overhead. However, it is different from memory in that it may send control signals to the processor that will cause a processor *exception* to occur. Many computer manufacturers use the term exception to include interrupts, and the two terms will be synonymously used throughout this article. Examples of I/O devices are keyboards, printers, disk drives, CD-ROM drives, DVD drives, tape drives, and ZIP drives.

The scenario of the program cycle of a computer program illustrates the communication that must occur between the computer components. The sole purpose of a computer is to execute programs. In the following the size of the unit of transfer depends on the number of data lines in the communication lines. A possible set of computer-related steps for the program cycle include:

1. Create the program using an editor. This involves the following steps.
   a. Cause the editor to start executing.
      1. Issue editor execute command. The user types the command at the keyboard (from a window environment, the keyboard may be replaced with a mouse action) and it is transferred from the keyboard through the I/O interface unit and the communication lines to processor. This command will ultimately be recognized by the operating system and the operating system will cause the editor to start executing.
      2. Editor execute command is displayed on the monitor. When the command of (1), above, is received by the processor, it is conveyed by the processor through the communication lines and the I/O interface unit to the monitor where it is displayed.
      3. Editor program is loaded into memory. The operating system interprets the editor execute command and causes the processor to command the disk to load the editor program into memory. The processor carries this out by sending the command over the communication lines and through the I/O interface unit to the disk. The disk then sends the editor program to memory, one unit at a time, over the communication lines and through the I/O interface unit.
      4. Processor executes the editor initialization instructions in the editor program. This includes the tasks of moving the instructions one at a time from memory to the processor using the communication lines for execution by the processor. The instructions are executed by the processor, resulting in such things as messages transferred from the processor to the CRT via the communication lines and I/O interface unit for display on the CRT.
   b. Interact with the editor and develop the program. Among other things the editor interaction includes:
      1. Issue commands to the editor. The user issues these commands to the editor through the keyboard to the processor. The editor instructions interpret the command and if it is valid carries it out, but if it is invalid the editor instructions will cause the processor to send an appropriate message over the communication lines and through the I/O interface unit to the CRT.
      2. Retrieve existing program file. This is a user-issued command to the editor, as given in (1), but is frequently used and of special interest. The editor will prompt the user for the name of the file to be retrieved by sending a message to the CRT.
      3. Enter program instructions. The request to insert program instructions is a user-issued command to the editor, as given in (1), but is frequently used and of special interest.
      4. Save the program. The request to save a program is a user-issued command to the editor, as given in (1), but is frequently used and likewise of special interest.
2. Compile the program.
3. Execute program.

Details of an instruction cycle will be addressed in the next section.

B. Central Processing Unit

The processor has the responsibility of causing instructions to be processed. The processor processes an instruction in an instruction cycle consisting of a number of small actions. Understanding the instruction cycle is important to the understanding of the processor.

Example VIB: Considering the abstraction 2 instruction, such as the assembled version of the instruction MOVE.W FOO.L, D3 (this instruction moves a word from the memory location "L" into a register named D3.), A potential instruction cycle is as follows:

1. Fetch the instruction from memory into the instruction register.
2. Increment the *program counter* by two.
3. Decode the fetched instruction.
4. Fetch the address of FOO from memory into the processor.
5. Increment the program counter by four to point to the next instruction.
6. Fetch the content of the location FOO into the processor.
7. Place the content of FOO in D3.
8. Update the content of the CCR appropriately.

It should be noted that a program could be written to perform each of the eight steps of the instruction cycle. In actuality, the microprogram is a program that carries out the instruction cycle of a conventional machine language instruction. For each of the eight steps above, there are sets of microinstructions in the microprogram that perform the specified task.

The processor repeats the instruction cycle, with variations for different instructions, for each instruction executed. The general instruction cycle for a single instruction is illustrated in Fig. 6. Steps 1, 2, and 4 correspond to instruction fetch subcycle, step 3 corresponds to the instruction decode subcycle, and steps 5–8 correspond the execute subcycle. In the MOVE.W FOO.L, D3 instruction, after decoding the first word of the instruction it is necessary to return to the fetch portion of the instruction cycle to fetch the two words corresponding to the address of FOO, but a further decode is not necessary. The instruction, MOVE.L (6, A3, D2.W), D2, is an example of an instruction in which a return to the fetch portion of the instruction cycle must occur followed by a further decode of the instruction. The information regarding the addressing mode, (6, A3, D2.W), is stored in an extension word that must be fetched and decoded after the opcode word is fetched and decoded. All instruction cycles require an interaction between the processor and memory and possibly between the processor and other computer components. For the MOVE.W FOO.L, D3 instruction, the processor must interact with memory, memory read cycle, in steps 1, 4, and 6.

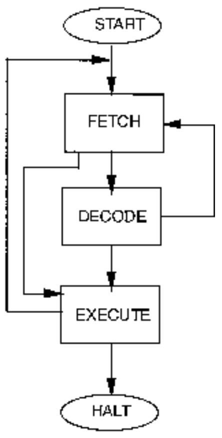


FIGURE 6 Basic instruction processing cycle.

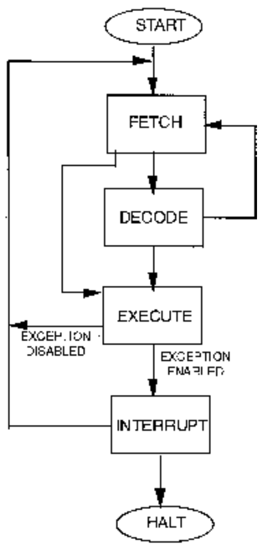


FIGURE 7 Basic instruction processing cycle with exception handling capability.

An exception subcycle occurs if an exception is pending, and this addition to the instruction cycle is included in Fig. 7. The exception is caused by an event separate from the program that is being processed and is serviced in such a way that the program processing may continue after the exception has been serviced. In the case of the instruction cycle illustrated above, none of the steps are a part of the exception subcycle, and it was assumed that no exception was pending. Exception processing will not be covered extensively here, but an exception might be considered analogous to the exception of the following example. A maskable exception is one that can be ignored for some time, but an unmaskable exception is one that must be handle immediately. An I/O device that needs service by the processor causes one type of exception. For example, when a user types information at the keyboard, the keyboard may raise an exception in the processor to alert the processor that it has received information from the user and request permission to send it. The exception is transmitted from the keyboard over the control lines to the processor. The keyboard exception is an example of

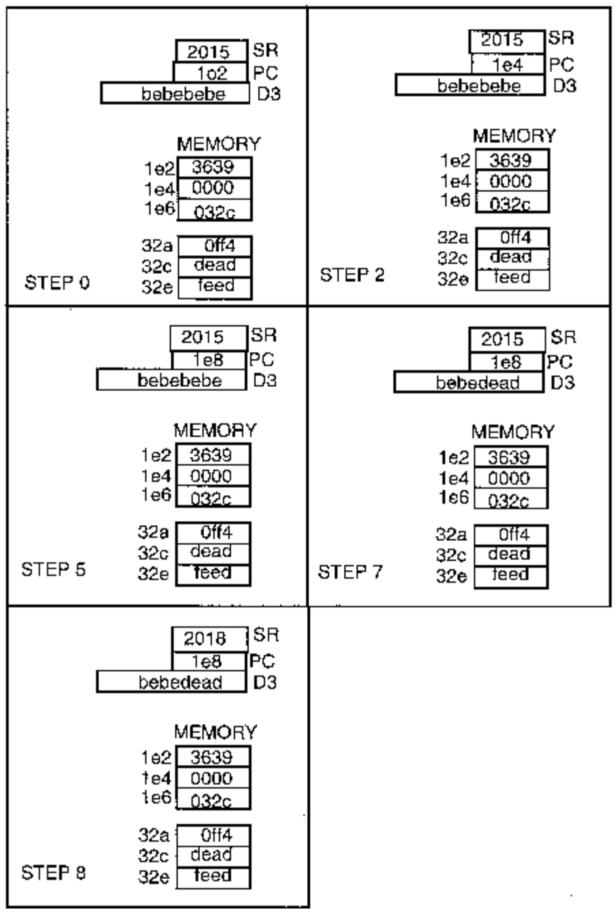


FIGURE 8 Steps in MOVE FOO.L, D3 instruction cycle in which conventional machine states are changed.

a maskable exception. A memory failure error is an example of an unmaskable exception and will be raised by the memory unit by transmitting a signal over the control lines to the processor. The signal that raises the exception is sent by the device to the processor over the communication control lines. Significant effort must go into planning the process for dealing with exceptions in order to do so in a way that will not cause performance deterioration of the processor.

Figure 8 gives the conventional machine language state changes that occur as the instruction MOVE.W FOO.L, D3 is processed. It should be noted that the conventional machine language instruction corresponding to MOVE.W FOO.L, D3 is 0x36390000032c and it is assumed that it is stored at memory location 0x1e2. At execution time the assembly language form, MOVE.W FOO.L, D3, does not exits. Only the steps given above that cause one or more conventional machine states to change are shown. The microprogram machine states will change as a result of each of the machine cycle steps above, and those steps will be carried out as a result of executing several microinstructions. All machine states are given in hexadecimal, and except for the 32-bit register D3, all of the other pictured machine states are 16-bit locations. It should be noted that only the low order word of D3 is modified (Big Endian memory configuration is assumed.) as a result of this instruction and the SR N bit is set and the Z and C bits are cleared in step 8. Likewise, it is arbitrarily assumed that FOO is at address 0x32c. The assembly language or conventional machine language programmer is only aware of machine state changes occurring at steps 0, 2, 5, 7, and 8, but microprogram abstraction machine states will change at all of the steps. For example, the first word of the conventional machine language instruction, 0x36390000032c, will be moved from memory to the processor at step 2.

The processor is a part of the hardware of a computer system. It is composed of sequential and combinational circuits. The processor may be logically subdivided into components such as the control, ALU, registers, shifter, and internal processor buses. Most every computer system that was ever developed would have all of these components present. Modern processor chips also contain additional components such as cache memory. The interconnection of processor components via the internal buses may also be viewed as a network. The control component is the part of the processor that is the director of a processor activity. As discussed above, there is an interrelationship between the major computer components such as the processor, memory, and I/O devices. Also, within the processor there is an interrelationship between its registers, ALU, etc. Each interrelationship much be synchronized.

1. ALU

The ALU is a combinational circuit, and is that part of the processor that performs arithmetic, logic, and other necessary related operations. Sometimes there is a separate component, shifter, which is used to perform the shift operations on data items. The abstraction 2 programmer usually considers the shifter activities to be a part of the ALU. The Abacus, Blaise Pascal machine, and other ancestors of the computer were really just ALUs, and for the most part the other components of modern computer systems are merely there to hold data for or transfer it to the ALU. Thus, the ALU could be considered the center of the computer system. However, it does not determine its own activities. Abstraction 1 or 2 instructions determine its activity via the control. The ALU will be actively involved in steps 2, 3, and 5 of the instruction cycle of Example VIB and may be involved in other steps.

2. Control

Control coordinates all activities. As mentioned previously, unless multiple processors exist within the computer system, the uniprocessor control directs all activities of the other elements of the processor and in general all components throughout the computer system as well as its own activity. For each instruction, control performs a series of actions. From a very high level it is possible to consider the set of actions to be the three of Figure 2. Those three steps may be further refined to the eight steps for the MOVE.W FOO.l, D3. These eight steps may be further filtered so that a set of microinstructions corresponds to each of the eight steps. Of course, these microinstructions could be broken down even further. Thus the computer activities may be studied at any level of detail. The control component will contain combinational and sequential circuits. The microprogram will be placed in the control store, which is a set of sequential circuits. However, the RISC computers do not contain a microprogram and thus control store is not present, and the control unit is strictly hardware.

3. Registers

This sequential circuit is the fastest and most expensive part of the memory hierarchy. The registers are the part of the memory hierarchy that are directly a part of the processor datapath. Because these are the fastest memory it is desirable to have all of the active data present in them. At a given time only a few are available to the programmer, because of the principle that if more were present, it would take longer to access the one desired, because the search process would take a longer period of time.

4. Exception (Interrupt) Processing

Processors vary significantly in the sophistication of their exception (interrupt) handling facilities. The exception capabilities of the Motorola 68000 family is well above the average and can be used to place all exception processing out of reach of garden variety user activities. All processors must have some degree of exception processing support, because at a minimum it must be possible to perform processor resets, which amounts to an exception.

5. Processor Organization Overview

A simplified logical view of a possible processor organization is given in Fig. 9. The registers, ALU, and shifter along with the three buses are often called the internal datapath of the processor. The datapath design should be designed to optimize the execution speed of the most frequently used abstraction 2 instructions. The datapath activity is controlled by the CONTROL component via the dashed control lines. The dashed lines represent one or more single control lines. The control component also controls its

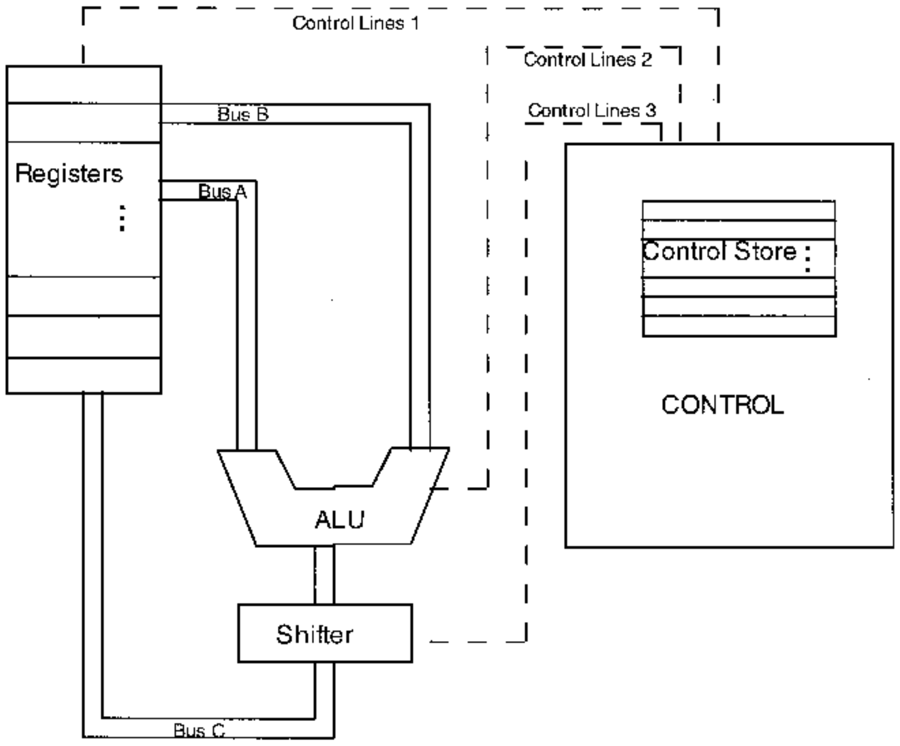


FIGURE 9 Logical view of a simple processor.

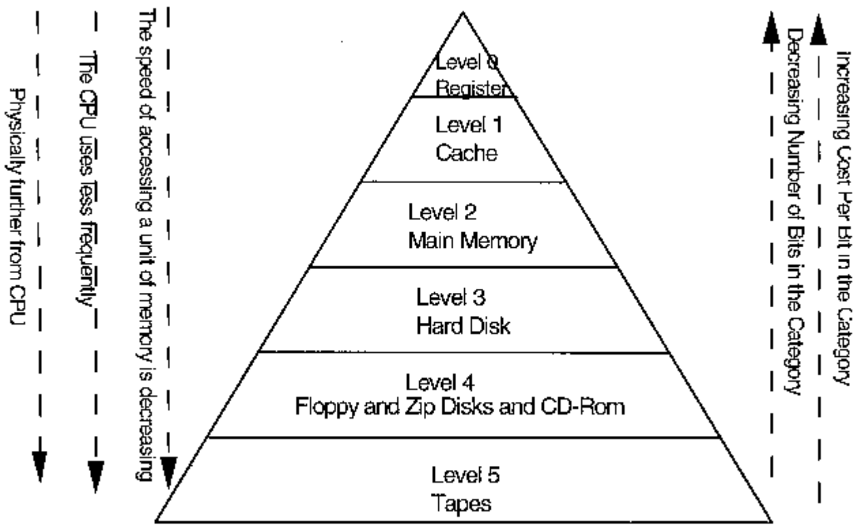

Level 0 Register
Level 1 Cache
Level 2 Main Memory
Level 3 Hard Disk
Level 4 Floppy and Zip Disks and CD-Rom
Level 5 Tapes
Physically further from CPU
The CPU uses less frequently
The speed of accessing a unit of memory is decreasing
Increasing Cost Per Bit in the Category
Decreasing Number of Bits in the Category

FIGURE 10 Hierarchal presentation of some types of memory.

own internal activity. This processor would contain more additional control lines than those shown, for example, ones that control the registers. As sure as there is water in the oceans, there is something missing, and that is the connection of the internal datapath and CONTROL signal to the main memory and I/O modules. This was done deliberately to reduce the complexity at this point. A complete dissertation of the processor can usually be found in most computer architecture textbooks. The control lines will carry out the following general tasks:

- Control lines 1: Used to select the register to be placed on Bus A and or Bus B, and/or receive data from Bus B.
- Control lines 2: Used to determine the ALU operation to be performed.
- Control lines 3: Used to determine the Shifter operation to be performed.

Control must be able to communicate commands to all of the computer components, as illustrated for the Registers, ALU, and Shifter, and the commands must be carried out in a proper sequence:

- First, the proper registers must be placed on Bus A and Bus B.
- Second, the ALU must generate the proper result.
- Third, the Shifter must produce the proper output.
- Finally, the content of Bus C must be stored back in the proper register.

For high performance, the computer architecture must be planned using the principles discussed previously and Amdahl's law.

C. Memory System Hierarchy

The memory in the memory hierarchy of a computer system is used to store information, instructions, and data that will be used by the computer system. Memory is often classified as registers, cache memory, main memory, hard disk, floppy disk, and tapes. These are pictured in a hierarchal form in Fig. 10 with locations within each type of memory randomly accessible except for tapes. Tapes are sequentially accessible, and in the long run each disk data unit is accessible in equal time, but at a given time the access time for a particular unit is dependent on the location of the disk components. The term "access" designates the memory activities that are associated with either a read or a write. Randomly accessible means that a memory location may be read or written in the same amount of time irregardless of the order of accesses of memory locations, and sequentially accessible means that the time required to access a memory location is dependent on location of the immediate prior memory access.

Example VIC: If a company stored all of their folders in a file cabinet alphabetically, then an employee would be able to find any folder in the file with roughly equal speed (randomly accessible). However, if the folders were stored flat in nonalphabetized order in a box, then the employee would have to sequentially search for the desired folder in the box. These filing systems are analogous to the randomly accessible and sequentially accessible memory storage units.

There are other forms of information storage such as CD-ROM and cassettes. The control store is a memory unit but is not considered in the memory hierarchy because it is only used to store microprogram instructions. Memory hierarchy locations are used to store abstraction

2 instructions and data. In the future it may be possible that control store will become a part of the memory hierarchy if programs are translated down to that level and the control store becomes a RAM-type memory for storage of the translated program along with the corresponding data.

In the memory hierarchy, the memory level in the higher location of the pyramid of Fig. 10 is usually physically closer to the processor, faster, more frequently accessed, smaller in size, and each bit is more expensive. Some of the characteristics of memory types are access time, bandwidth, size limit, management responsibility, and location within the computer system (within processor or on the external bus).

1. Principles of Memory Design

Size, cost, and speed are the major design parameters in the memory hierarchy. The principle behind memory hierarchy design is to keep the cost per unit of memory as close as possible to that of the least expensive memory and keep the average access time as close as possible to that of the fastest memory. To accomplish this, the design must use a minimal amount of the memory type at the top of the pyramid of Fig. 10 and attempt to keep information that will be accessed in the near future in memory at the top of the pyramid. Removable disks and tapes are of unlimited size, and that means that the user may continue to buy more of the media until his needs are met.

It would be most desirable to have every item to be used in a register. However, this is impossible, because current computers must move information into memory using level 2 instructions. In order to follow these design principles, it is necessary to study the relationship of memory references by a program and utilize those relationships that are found when possible. Studies have shown that the memory references in a small period of time tend to be clustered for both data and instructions within a few localized regions of memory, as illustrated in Fig. 11. This is known as the locality principle and predicts that there is high probability that those items referenced recently or those nearby the recently referenced memory locations will be referenced in the near future. A copy of those items that, according to the locality principle, are most likely to be used in the near future should be kept in the memory that is accessed most quickly.

Example VIC 1: Average cost and average access time.

A memory design issue is the ordering of bytes in memory that may follow either the little Endian or Big Endian definitions. Within a particular manufacturer's computers it will be either one or the other for all of their computers

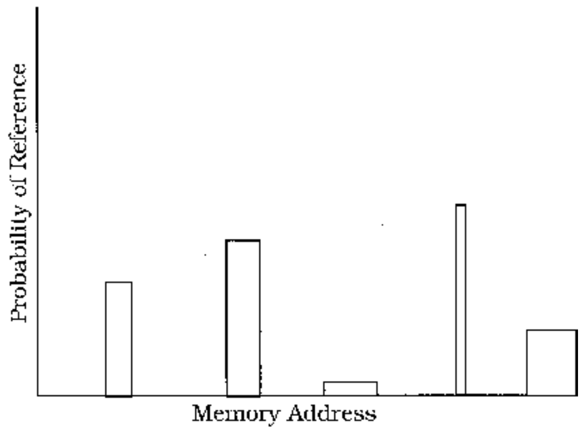


FIGURE 11 Probability of referencing memory at a snapshot of time.

within a computer family. Little Endian and big Endian are terms introduced by Cohen (1981) in an amusing article and the term "Endian" is borrowed from Jonathan Swift's *Gulliver's Travels* that poked fun at politicians who waged war over the issue of whether eggs should be cracked on the big end or little end.

D. I/O System

The I/O system is a crucial part of any computer system, but it will not be discussed in greater detail in this article.

E. Interconnection

The interconnection of computer components is critical to computer performance. The interconnection of components within a computer has been discussed, and is a key concern in the implementation of an architecture. The interconnection between computers is covered under the data and computer communications category in books and articles dealing with that topic.

VII. ENHANCING ARCHITECTURE IN THE TRADE-OFF OF PERFORMANCE AND COST

Every person who buys a computer desires one that will have maximum performance with minimum cost and meet his minimum requirements. The language of the computer is determined by the instructions that it understands. Languages of computers are quite similar in nature because computers are all built using approximately the same underlying principles. The performance of a computer system is influenced by three critical components: instruction count, clock cycle time, and clock cycles per instruction (CPI). The number of instructions required is determined by the instruction set of the computer and the efficiency of the compiler in generating the program that will be executed. The processor implementation determines the clock cycle time and the number of clock cycles per instruction. The performance of a computer can be enhanced to almost any desired level, but the performance level must be traded off with the cost. If the cost is too great, it will not be affordable and as a result will not sell. In this section, we will look at some current techniques that are used to enhance the performance of a computer.

A. Instruction Set Design

The instruction set influences the processor implementation and must be determined based on the processor efficiency. Some of the key instructions of a computer include the

1. memory reference instructions
2. arithmetic-logical instructions
3. branch instructions

Other factors in the design of an instruction set include number of instructions to be included, whether or not the instructions should have a fixed length and/or fixed format, the addressing modes available, the robustness or complexity of the instructions, and whether or not the instructions, and support data structures such as “stacks” and data types.

The guidelines—make the common case fast (make the memory access instructions fast because they occur about 50% of the time in a general purpose program) and simplicity promotes regularity (make the instructions fixed length and fixed format)—play a significant role in the definition of modern instruction sets and in particular the RISC computers. In the instruction set design, it is not always possible to have fixed length and format instructions and as a result the principle, a good design will require compromises, must be followed. The instruction set has a significant influence on the implementation, and the choice of implementation strategies will affect the clock rate and the CPI for the computer. The final principle, smaller is faster, comes into play when the number of instructions is minimized or the number of memory locations is small (in the case of registers). The number of methods and complexity of instruction access of memory (addressing modes) will play a significant role in the performance level that the computer can achieve with best performance achieved if the design principles “simplicity” and “small” are observed.

The implementation of the instruction set through the design of the datapath and control within the processor will play a major role in the performance of the computer with the technology available being an important factor. Technology will significantly influence such things as the density of transistors on a chip. The density of transistors will influence the distance between two points, and the higher density chips will provide the capacity for adding features on the processor chip as opposed to externally, thus reducing the time required to transfer information between components.

A significant influence on the instruction set design has been the past desire by computer manufacturers to provide backward compatibility (the capability of executing existing programs without modification). If this requirement were met by retaining the old instruction set in the new instruction set of the processor, it would result in greater overhead and a reduction in the potential performance of the processor. If this requirement were met by using a software interpreter, then the old programs would not achieve top performance because of the overhead of the interpreter.

B. Instruction Set Implementation Enhancements

It was mentioned above that technology advancements have played a significant role in the advancement of the performance of a processor. Implementation techniques play a major role as well with the following being some of those advancements.

1. *Pipelining*: This is an architecture implementation technique that allows multiple instructions to overlap in execution. The processor is organized as a number of stages that allow multiple instructions to be in various stages of their instruction cycle. The pipeline for instructions is analogous to an assembly line for automobiles. To implement the pipeline, additional processor resources are required. In order to achieve maximum efficiency, the instruction cycle must be divided so that approximately the same amount of work is done in each of the stages. There are a number of potential hazards that exist:
   a. *Data hazard*: This occurs if an instruction in an earlier stage of the pipeline attempts to use the content of a location that has not been written by an instruction that precedes it in the pipeline.
   b. *Branch hazard*: This occurs if instructions following the branch are brought into the pipeline when the branch will actually occur or vice versa. A method used to eliminate this problem is to provide the resources to bring both sets of possible future instructions into the pipeline. Another solution is to rearrange the order of instructions to

cause instructions preceding the branch in the program to follow the branch in the pipeline. This assumes these instructions do not influence the branch decision.

c. *Exception hazard*: This hazard could occur if an exception occurs and instructions in the pipeline are not handled properly.

All hazards can be remedied but the method used may require significant processor resources. The data and branch hazards can be remedied by using the compiler to reorder the instructions.

2. *Superpipelining*: This is a processor design that divides the pipeline up into a large number of small stages. As a result, many instructions will be at various stages in the pipeline at any one point in time. There is no standard distinguishing pipelined processors from superpipelined processor, but seven has been a commonly accepted division point. The ideal maximum speedup of the processor is proportional to the number of pipeline stages. However, the required resources for fixing the hazard problem increases more rapidly as the number of stages increases.
3. *Superscalar*: This implementation of the processor provides more than one pipeline. As a result, multiple instructions may be at the same stage in their instruction cycle. A superpipeline and a superscalar implementation may exist within the same processor.
4. *Very long instruction word* (*VLIW*): This processor implementation provides for the processor to use instructions that contain multiple operations. This means that the multiple instructions are contained in a single instruction word. It is usually the responsibility of the compiler to place compatible instructions in the same word.

All of these techniques introduce parallelism into the execution of instructions within the processor.

C. Input and Output Enhancements

As computers have evolved there has been increasing complexity placed in the function of the I/O devices. The following are some of the evolutionary steps that have occurred.

1. Initially the processor handled the peripheral devices directly.
2. The I/O device became a controller of the peripheral device, but the processor performed the programmed I/O (the processor stopped processing other instructions until the I/O activity had completed and the device had signaled its completion of the required action).
3. The I/O device is the same as (2) but the processor uses interrupts to allow itself to process instructions of another program until it is interrupted by the I/O device when the interrupt activity is completed.
4. The I/O device becomes a controller that has direct memory access capability. As a result, it can place data in memory directly.
5. In the most sophisticated case to date, the I/O device becomes a complete processor with its own instruction set designed for I/O. As a result, the processor may specify a set of I/O activities and be interrupted only after all of the requests are fulfilled.

D. Memory Enhancements

Memory speed enhancements have not kept up with processor speed improvements. As a result, other techniques have been used in order to enhance the performance of the memory system. The characteristics of a memory system that are adjusted to improve performance are included in the following table:

Location of memory in computer system
Processor
Main memory
Secondary memory
Physical characteristics
Semiconductor
Magnetic
Optical
Nonvolatile vs volatile
Nonerasable vs erasable
Speed of access
Access time
Memory cycle time
Transfer rate
Capacity
Word size
Number of words
Method of access
Random
Sequential
Direct
Associative
Organization
Location of bits in random access memory
Registers
Cache
Virtual memory
Transfer unit
Word
Block
Page

All of these characteristics of the design of a memory system will influence its performance. The location of the memory in relationship to the processor and the closer it is to the processor the faster it will be. Another key characteristic is the unit of transfer that is dependent on the size of the bus from the memory unit with increased size of the transfer unit increasing performance. Increased number of accessible units will often decrease the performance of memory, and this is remedied by increasing the word size. The first three physical characteristics have more influence on performance than do the last two. The RAID standard has been adopted by industry, and it provides a mechanism for substituting large capacity disks with multiple smaller-capacity disk drives and causes the data to be distributed in such a way as to allow the simultaneous access to data from the multiple drives. Adoption of the RAID standard provides disk access performance enhancements and provides a mechanism for allowing incremental increases in overall storage capacity of the disk system. The design of the memory system is critical to the performance of a computer system.

E. Enhancement Using Parallel Processors

Computer professionals have strived to achieve the ultimate of computer design by interconnecting many existing simple computers together. This is the basis for a computer using multiprocessors. Multiprocessors are computers that have two or more processors that have a common access to memory. Customers desire scalable multiprocessors that allow them to order the current required number of processors and then add additional processors as needed. It is also desirable to have the performance increase proportionately to the number of the processors in the system, and to have the computer system continue to function properly when one or more of the processors fail. The typical range of processors in a multiprocessor computer is between 2 and 256.

The design of the multiprocessor computer is a key factor in the performance of the resulting computer system and the following are some of the key issues that must be addressed:

- The mechanism for sharing memory between processors must be defined
- The coordination of processors must be determined
- The number of processors must be established
- The interconnection between the processors must be defined

The most common mechanism for sharing memory is through a single address space. This provides each processor with the opportunity to access any memory location through their instruction set. Shared memory multiprocessor may be symmetric multiprocessor (SMP) processors, in which the access to a word of memory takes the same time no matter which processor accesses it or which word is accessed, and the nonuniform memory access multiprocessors, which does not have the equal access characteristic. The alternative to the shared memory multiprocessor is one whose processors have private memory, and coordination is then achieved through the passing of messages. A key element for memory is coherency, which requires data validity among the different processors. Multiple processors are interconnected either by a bus or network.

The multiprocessor area of computer architecture is highly active. It is difficult to maintain awareness of current examples but the World Wide Web is an excellent source of information with links to several types of multiprocessor examples at http://www.mkp.com/books_catalog/cod2/ch9links.htm.

VIII. CONCLUSIONS

Computer architecture has been and will continue to be one of the most researched areas of computing. It is about the structure and function of computers. Although there is explosive change in computing, many of the basic fundamentals remain the same. It is the current state of technology along with the cost/performance requirements of the customers that changes how the principles are applied. The need of high performance computers has never been greater than it is now, but the difficulty of designing the ideal system has never been more difficult because of the rapid increase in speed of processors, memory, and interconnection of components. The difficulty is in designing a balanced system according to Amdahl's Law.

Professionals from every area related to computing should understand architecture principles and appropriate aspects of hardware and software. It is important for those professionals to understand the interrelationship between the assembly language, organization of a computer, and the design of a computer system. It is important to be able to sort out those concepts that are the basis for modern computers and to be able to understand the relationship between hardware and software.

SEE ALSO THE FOLLOWING ARTICLES

Compilers • Computer Networks • Microcomputer Buses and Links • Operating Systems • Process Control Systems • Software Engineering • Software Maintenance and Evolution

BIBLIOGRAPHY

Adams, D. R., and Wagner, G. E. (1986). "Computer Information Systems: An Introduction," South-Western Publishing, Cincinnati, OH.

Bell, C., and Newell, A. (1978). "Computer Structures: Readings and Examples," McGraw-Hill, New York.

Bitter, G. G. (1984). "Computers in Today's World," Wiley, New York.

Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D. (1994, June). "RAID: High Performance, Reliable Secondary Storage," ACM Computing Surveys.

Campbell-Kelly, M., and Aspray, W. (1996). "Computer: A History of the Information Machine," Basic Books, New York.

Digital Equipment Corporation. (1983). "PDP-11 Processor Handbook," DEC, Maynard, MA.

Digital Equipment Corporation. (1983). "VAX Hardware Handbook," DEC, Maynard, MA.

Gimarc, C., and Milutinovic, V. (1987). "A Survey of RISC Processors and Computers of the Mid-1980s," IEEE Computer, Long Beach, CA.

Hennessy, J. L., and Patterson, D. A. (19**XX**). "Computer Architecture: A Quantitative Approach," 2nd ed., Morgan Kaufmann, San Francisco.

Kidder, T. (1981). "Soul of a New Machine," Little, Brown, New York.

Patterson, D., and Hennessy, J. (1998), "Computer Organization and Design—The Hardware/Software Interface," 2nd ed., Morgan Kaufmann, San Francisco.

Stallings, W. (1999). "Computer Organization and Architecture," 5th ed., Macmillan, New York.

Stone, H. (1993). "High-Performance Computer Architecture," Addison-Wesley, Reading, MA.

Tanenbaum, A. S. (1995). "Distributed Operating Systems," Prentice-Hall, Englewood Cliffs, NJ.

Tanenbaum, A. S. (1999). "Structured Computer Organization," 4th ed., Prentice-Hall, Englewood Cliffs, NJ.

Computer Networks

Jordanka Ivanova
Michael Jurczyk
University of Missouri-Columbia

I. Definitions
II. Network Protocols
III. Network Classes
IV. Network Components
V. Wireless Networks
VI. Internet and Applications
VII. Security
VIII. Quality of Service in Computer Networks
IX. Trends

GLOSSARY

Computer network System consisting of an interconnected collection of computers that are able to exchange information.

Network protocol Encapsulated module of services with defined interfaces, providing communication functions to other protocols or applications.

Network architecture Structured collection of protocols and layers with specification of the services provided in each layer.

Quality of Service (QoS) Description of the quality of services provided by a computer network. QoS is characterized by parameters such as bandwidth, delay, and jitter.

Network security service Service that enforces the security policy to provide a certain level of network security.

World Wide Web (WWW) Large-scale, on-line repository of information in the form of linked documents, web pages, and web sites.

Internet The global collection of networks (internet) that uses TCP/IP protocols.

I. DEFINITIONS

For the past 10 years, computer networks have become a major part of modern communications, data processing, business, culture, and everyday life. The properties that make them so popular are access to remote data, resource sharing, and human communication. Network users can access and use remote data and machine resources as if they were local. Computer resources such as printers and database servers can be shared by many remote users, making the computing process more efficient. Resource sharing and remote data access through networking are also used to increase the reliability, accessibility, and uptime of many applications. For example, if a bank has a database server and a backup server, in case of failure of the main server the backup server can then provide service to the users, preventing loss of data accessibility.

Apart from sharing data and resources, computer networks offer a human communication medium. This is very important for large companies, which have offices around the country, or international organizations, which have offices all over the world. The efficient and convenient means of communication like e-mail, video conferencing, and the Internet as a global information environment are used to link people together. The Internet as a worldwide network brings new business opportunities, where products and services find their way to the clients much faster and with less effort. E-commerce is a growing and very successful business.

Networks are as important for personal use as for corporate use. Information access is one of the major motivations. Users can access information from millions of servers available through the Internet containing articles, publications, on-line books, manuals, magazines (e-zines), and newspapers. Video and audio, virtual reality, entertainment, and network games are other popular applications used over the Internet. On-line shopping is not just popular, but it is becoming a major competitor for conventional shopping. Personal communications is another important feature of computer networks. E-mail, on-line pagers, and Internet telephony are all network based. Newsgroups and chat rooms are interactive ways, in real time, to communicate with other people. A new approach in the educational system is distance learning. Based on on-line courses and programs, it offers course material combined with multimedia to the students. This distance learning is especially helpful for people holding full-time jobs or people with disabilities.

Computer network is a system consisting of an interconnected collection of computers that are able to exchange information. Computers connected in a network are usually called nodes. *Hosts* are nodes used by users and are usually running applications. Apart from hosts, network nodes such as routers, bridges, or gateways that execute network support functions (e.g., messages routing) might also be present. Physical links that interconnect nodes can be copper wire, fiber optics, microwave, radio, infrared, or satellite connections. There are two connectivity technologies independent of the physical link type: multiple-access and point-to-point links. A *multiple-access link* allows more than two nodes that share that link to be attached to it. In contrast, a *point-to-point link* connects two nodes only. A network that uses multi-access links and allows messages sent by a node to be received by all other nodes is called a *broadcast network*. Broadcasting is the operation of sending a message/packet to all destinations. A *point-to-point network* or *switched network* consists of many point-to-point links in an organized way forwarding data from one link to another. Usually, smaller networks tend to be broadcast networks, while larger ones are point-to-point networks.

Networks can be circuit switched or packet switched. *Circuit-switched networks* establish a connection from the sender to the receiver first. After the connection is up, the sender will send data in a large chunk over the connection. After the sending process is finished, the connection is torn down. In *packet-switched networks*, a message is split into packets. Each packet consists of a body (the actual data) and a header that includes information such as the destination address. A connection is not established, but each packet is routed through the network individually. Packet-switched networks can provide connection-oriented or connectionless services. *Connection-oriented services* establish a connection from the sender to the receiver by specifying the exact route for all the packets through the network. All packets use this route and will be received by the receiver in order. In contrast, *connectionless services* do not establish a fixed route for the packets. Packets belonging to the same connection might travel over different routes through the network and might be received out of order.

Size is an important characteristic of networks. A key factor is not the number of nodes a network connects, but the time needed for the data to propagate from one end of the network to the other. When the network is restricted in size, the upper bound of transmission time is known. In larger networks, this time is generally unknown.

Smallest networks covering usually a single room are termed *system area networks* (SAN). They are used to interconnect PCs or workstation clusters forming server systems and to connect data vaults to the system. Smaller networks, typically spanning less than 1 km, are called *local area networks* (LAN). LANs are generally used to connect nodes within a few buildings and are privately owned. They are used to connect workstations for resource sharing (printer, database server) and for information exchange among each other and often employ multi-access links to connect all of the computers. LANs can run at speeds from 10 megabits per second (Mbps) up to few hundreds Mbps and have low delay and usually low bit error rates. The next in size are *metropolitan area networks* (MAN), which usually extend up to tens of kilometers and serve nearby corporate offices or interconnect a city. The last type, *wide area networks* (WAN), has no size limit and can be worldwide. WAN spans a large geographical area, often a country or continent. Because of the great number of hosts and the distance among them, WANs are in general not broadcast-type networks, but they can provide broadcast operations. WANs use the point-to-point principle and employ a set of nodes dedicated to internetwork communication. These nodes are usually called *routers*, and the set of routers is called *subnet*. When a host wants to communicate with another host, it sends the message, often divided into several packets, to the closest router. That router stores the packets and forwards them to a router that is closer to the destination host. This repeats until the packets reach a router that is connected to the destination node. This router will then deliver the packets to the destination or to the LAN the destination is connected to.

Another common classification of networks is based on the topology of how the nodes and the links are interconnected. The topology of the network is represented by a graph in which nodes are vertices and links are edges. The network graph can be symmetric or irregular. Symmetric ones are used mainly in LANs, such as the star topology, in which all the nodes are connected to one center node, the bus topology, in which all nodes are attached to a single cable/bus, the ring network, in which nodes are connected to form a ring, or the fully connected topology, in which every node is connected to every other node. WANs are likely to have irregular topology, since they have too many links to enforce a symmetric topology.

II. NETWORK PROTOCOLS

A. Overview

One of the main objectives when designing a computer network is to keep it as simple as possible. To achieve this, different abstractions for services and objects can be used. The idea of encapsulating components of the system into an abstraction and providing interfaces that can be used by other system components hides the complexity. In networks, this strategy leads to the concept of *layering* of the network software. The general rule is to start from the underlying hardware and build a sequence of software layers on top, each one providing a higher level of service. Layering provides two features: less complexity and modularity. Instead of constructing a network with one solid mass of services, it is much easier to separate them into manageable components. Modularity gives the assurance that as long as the interface and the functions are consistent and compatible with the original specification, the layer can be modified at any time and new functions can be added. Each layer provides certain services to the layer above, implemented in terms of services offered by the layer below it. A layer on one machine communicates with exactly the same layer on another machine. Corresponding layers are usually called *peers*. Peers, together with rules and conventions that they have for communication, are called *protocol*. Each protocol defines the interface to its peers and the service interface to the other layers. Peer interface describes the specifications and format of the messages exchanged between the peers. Service interface specifies the functions provided by the layer. The set of layers and protocols is called *network architecture*. In fact, only the hardware level peers communicate with each other directly, all other layers pass messages to the next lower layer. If two applications on different computers want to communicate, the initiator sends a message containing data and some special formatting information usually called *header*. The next lower layer receives the message, adds its own header, and passes it to next lower level and so on until the message is transmitted over the link (see Fig. 1). On the receiver machine, this message will progress up the different layers until it reaches the application involved in this communication. Each layer will strip the corresponding layer header from the message, will apply its services, and will pass the message to its next higher layer. Viewing the group of protocols in a machine or a network as a stack can be helpful for visualization, but the protocols need not be in a linear configuration. More than one protocol can exist on the same level, providing similar yet different services. The *protocol graph* is used to describe the protocols and their dependencies in a network system. A chain of protocols relying on each other's services used by a specific application or a system is called *protocol stack* and is a single path in the protocol graph. The use of the term protocol is somehow ambiguous, referring sometimes to the abstract interface and sometimes to the actual implementation of the interface. To distinguish this, we will refer to the latter as *protocol specification*.

Network architectures and protocols are described in reference models. In the following subsections, a few very important reference models will be discussed: the OSI reference model, the TCP/IP reference model, and the B-ISDN ATM reference model.

B. ISO OSI Reference Model

One of the first standards for computer communications was proposed and developed by the *International Standards Organization* (ISO) in the early 1980s. This network architecture model, the *open systems interconnection* (OSI) reference model shown in Fig. 1, describes a network through seven layers. On any of these layers, one or more protocols can implement the functions specified for the layer. Some protocol specifications based on this

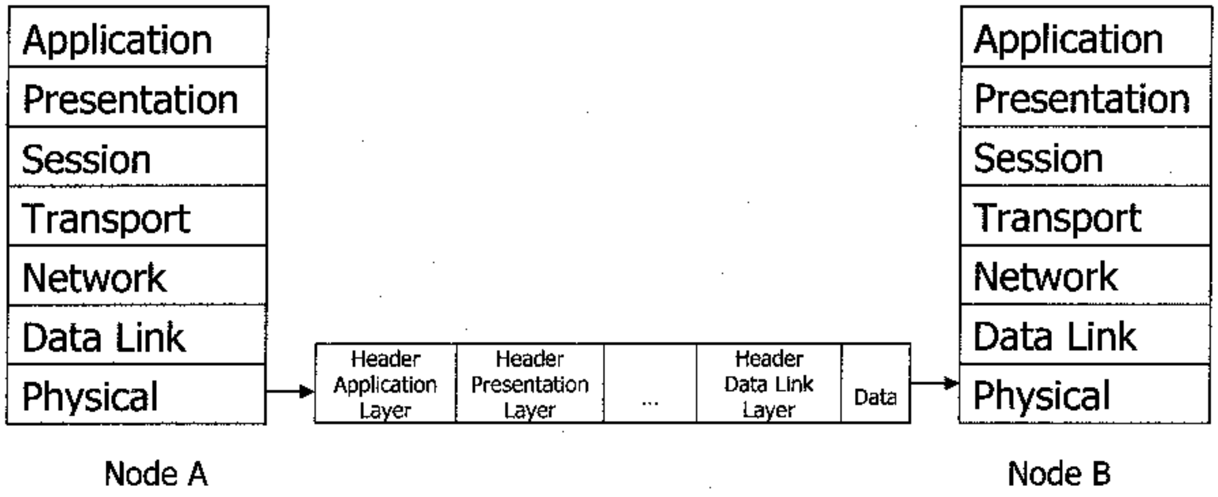


FIGURE 1 An OSI model and example of data transmission from node A to node B.

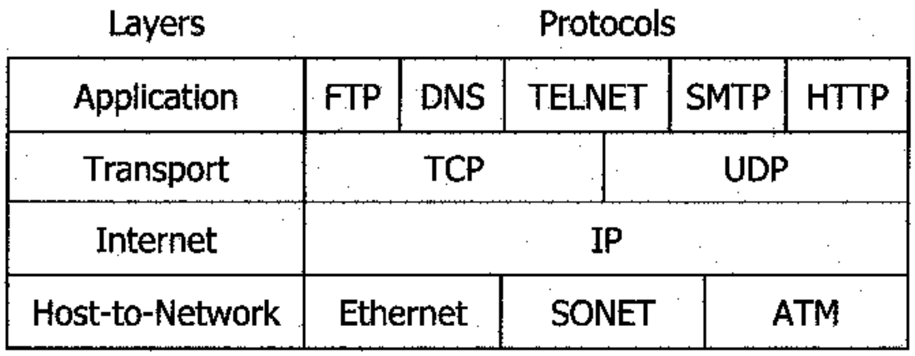
model, the "*X dot*" series X.25, X.400, etc., were specified by the *International Telecommunications Union* (ITU). Even though most of today's protocols do not follow this reference model, it captures very well the concepts of service, protocol, and interface. OSI design is general and quite complex, but it is able to present the functionality of a network in a way it can be used as a guide for designing new networks.

There are seven layers in the OSI model, starting with the *physical layer* handling the raw data transmission over a physical medium. The most common transmission media are twisted pair (copper wires), coaxial cable, and fiber optics. The *data link layer*, usually implemented in the network adaptors, is above the physical layer and is concerned with the organization of data into *frames* and the reliable transportation of these frames over a direct link. The specific problems of multi-access links such as channel allocation and collision detection are handled by the data link sub-layer called *medium access control* (MAC). Reliable frame delivery, frame ordering, and frame retransmission are provided in the layer by *sliding window protocols*. This is a set of protocols for full-duplex data frame transmission, in which the sender and the receiver both keep windows of frame acknowledgements and send frames only if a certain number of already sent frames were acknowledged by the receiver. The data link layer also includes some error detection and correction functions such as parity bit code and cyclic redundancy code (CRC).

The next higher layer is called *network layer*, and it addresses the problem of finding a route through the network from the source to the destination. This layer also addresses the problems of interconnecting different networks. Protocols in this layer are used to glue together heterogeneous networks into one scalable internetwork or internet. Mind the difference between Internet and internet, where the former is the global, widely used internetwork and the latter is the general term for a logical network consisting of a collection of physical networks. The *transport layer* takes care of the efficient and reliable delivery of data from the source to the destination node. Protocols in this layer are sometimes called end-to-end protocols. Modified versions of the sliding window protocols ensure reliable delivery, synchronization, and flow control. The fifth layer, the *session layer*, is the one that manages traffic direction and synchronization. The last two layers, the presentation and application layers, are mainly application-oriented service layers. The *presentation layer* is responsible for data representation and data coding, while the *application layer* offers a variety of services for particular applications, such as e-mail or file transfer. Security services such as encryption/decryption, authentication, and integrity are usually implemented in the last two layers.

C. TCP/IP Reference Model

While the OSI model was carefully designed, standardized, and then implemented, the *Transmission Control Protocol* (TCP)/*Internet Protocol* (IP) architecture was implemented first in the early 1980's and modeled afterwards. Well accepted in the university circles first, and later in the industry, TCP/IP, also called the Internet architecture, is nowadays one of the major architectures. However, the TCP/IP reference model is not general and consistent, but rather implementation bound, and it does not have a clear concept of services and protocols. The IP protocol serves as a joining point for many different networks, providing them with a method for communication.



Layers	Protocols				
Application	FTP	DNS	TELNET	SMTP	HTTP
Transport	TCP			UDP	
Internet	IP				
Host-to-Network	Ethernet		SONET		ATM

FIGURE 2 Protocols and layers in the TCP/IP model. Applications can either use TCP or UDP, depending on their implementation.

TCP/IP consists of four layers, as shown in Fig. 2, which can be matched to some of the OSI model layers. The *host-to-network layer* is similar to the physical and data link layers. The *internet layer* with the IP protocol corresponds to the network layer. The main concept for TCP/IP, implemented on this layer, is a connectionless packet-switched network.

Above the internet layer is the *transport layer* with two main protocols, TCP and *User Datagram Protocol* (UDP). TCP is a reliable and connection-oriented, end-to-end protocol. TCP has error detection/correction, data retransmission, and flow control mechanisms and takes care of out-of-order messages. UDP in contrast is an unreliable, connectionless protocol providing support for audio and video applications. The last layer, the *application layer*, hosts a variety of high-level protocols such as File Transfer Protocol (FTP) or Simple Mail Transfer Protocol (SMTP).

D. B-ISDN ATM Reference Model

The Asynchronous Transfer Mode (ATM) technology is the core of the Broadband Integrated Services network (B-ISDN). ATM is a connection-oriented and packet-switched technology, which emerged in the 1980s. Data in ATM is transmitted in small packets, called *cells*, which have a fixed size of 48 bytes payload and 5 bytes header. To set up a connection, a path through the network, a *virtual circuit*, is established first. This connection setup process is called *signaling*. After network resources are reserved and allocated to that virtual circuit, the source can start sending data cells. All cells transported over a specific virtual circuit take the same path/route through the network so that cells belonging to a specific connection are always received in order. Several virtual circuits can be combined into a single *virtual path*. A virtual path identifier and a virtual circuit identifier are present in each cell header to map a cell to a specific route through the network. If congestion occurs inside the network, cells are temporarily buffered in the ATM switches. If a switch buffer overflows, cells are discarded inside the switch and are lost. Cell switching is one of the biggest advantages of ATM. During the signaling process, not only appropriate routes are found but also resources are allocated in the switches. In ATM, it is therefore possible to make quality of services (QoS) reservations for a specific virtual circuit. QoS parameters, such as sustained cell rate, peak cell rate, maximum burst tolerance, cell delay, and cell delay jitter, characterize the services provided.

ATM link speed of 155 Mbps to a few gigabits per second and the capability to guarantee QoS are very important for real-time applications like audio, video, and multimedia. ATM networks are increasingly used as backbones for WANs. One example is the very high speed backbone network (vBNS), which is currently used as one of the backbones for the next-generation Internet (Internet-2). vBNS runs TCP/IP over ATM. ATM can also be used to implement high-speed LANs.

As depicted in Fig. 3, the B-ISDN ATM reference model has different layers and a different structure as compared to the OSI model. While the OSI model is two dimensional, the ATM model is three dimensional. ATM's physical layer corresponds to both the physical and data link layer of the OSI model. The physical layer deals with the physical transmission of the bit stream and therefore depends on the physical medium used. Copper cable or fiber optics can be used for ATM. Above the physical layer is the ATM layer that deals with flow control, virtual circuit management, and cell header generation. The ATM layer is functionally equivalent to the OSI data link and network layers. On top of the ATM layer sits the ATM adaptation layer (AAL) that supports the different ATM services. AAL lays somewhere between the transport and session layers in the OSI model and provides assembly and reassembly of packets that are larger than a cell. Four different services are currently defined for ATM, resulting in four different AAL classes. AAL1 supports circuit

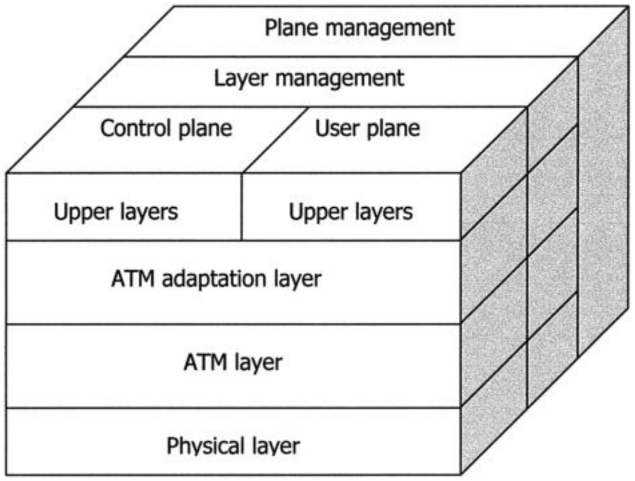


FIGURE 3 The B-ISDN ATM reference model.

emulation over ATM networks and is well suited for transporting constant bit rate (CBR) traffic. AAL2 is designed to support variable bit rate (VBR) traffic, where a timing relationship between the source and the destination is required. AAL3/4 is used to transfer data that is sensitive to loss but not to delay. Finally, AAL5 supports data traffic with no real-time constraints and is currently the most-used adaptation layer in industry.

Above the AAL layer are upper layers that are divided into a user and a control plane. The user plane is concerned with flow control, data transport, and error correction, while the control plane deals with connection management such as call admission and QoS. The layer and plane management functions in the third dimension are responsible for interlayer coordination and resource management.

To achieve high-speed data rates of up to a few gigagits per second, ATM networks use fiber optics. The physical layer standard most often used with ATM is the *synchronous optical network* (SONET), which was released in the late 1980s. SONET provides a common signaling standard with respect to wavelength, timing, framing, etc. It also offers a way to multiplex multiple digital channels together and provides support for network operation, administration, and maintenance.

III. NETWORK CLASSES

A. System Area Network (SAN)

SANs are used to interconnect PC clusters or workstation clusters forming server systems and to connect data vaults and other I/O subsystems to the overall system. SANs are also used to connect the individual processor nodes in a parallel computer (these networks are also termed *interconnection networks*). It is very common to connect SANs to LANs and WANs, and they are usually at the leading edge in terms of performance.

B. Local Area Network (LAN)

LANs have become the most popular form of computer networks; they are inexpensive and widely available. The primary topologies used in LANs are star, ring, and bus.

One bus-based standard is *Ethernet*, where computers share a single transmission medium. Ethernet segments are limited to 500 m. The speed of Ethernet starts from 10 Mbps and goes up to 1 Gbps, in the latest version called Gigabit Ethernet. If a node has something to send over the bus, it listens to the bus first. If there is no data currently on the bus, the node starts sending its data. Because there is no coordination among the nodes, two or more nodes might start sending data at the same time, resulting in a collision. To deal with collisions, *carrier sense multiple access with collision detection* (CSMA/CD) is used in Ethernet as the MAC protocol. For collision detection, every node monitors the bus while sending data. If the data on the bus is different to the data a node is currently sending, a collision is detected. After detecting a collision, the nodes involved stop sending, wait a random time of up to d seconds, and then try again. In case of a new collision, the parameter d is doubled, and the nodes try sending again. This mechanism is called *binary exponential backoff*.

An example for a ring network is the IBM *Token Ring*. In this network, computers are connected in a loop and have a medium access mechanism called *token passing*. Token is a special message that circulates in the network. Whenever a node receives the token, that node has the right to transmit data. After sending the data, it will send the token to the next neighbor node to allow that node to send data. This scheme guarantees freedom of collision and fair access to the network for all nodes. The token ring protocol can also be implemented in a bus network, resulting in a *token bus* network. In this case, a virtual ring structure is established in which each node has a fixed address and a node sends the token to the node with the next higher address.

Recently, ATM found its way into LANs. An ATM switch can be used to centrally connect up to 32 computers or other LANs, providing bit rates starting at 155 Mbps.

LANs provide a good combination of capacity, price, and speed, but they have a distance limitation of 500 m. This limitation is primarily due to the properties of the copper wire used. To extend the span of a LAN, devices such as repeaters and bridges can be used. These devices are introduced in Section IV.

C. Metropolitan Area Network (MAN)

MANs employ technologies similar to the one used in LANs. A standard called *distributed queue dual bus* (DQDB) has been defined for MANs that does not have the LAN's cable length limitations and performance problems when connecting thousands of nodes. DQDB uses two parallel, unidirectional optical buses that connect all nodes. To transmit data, a node has to know whether the destination node is to the left or to the right of it. If it is to the left, it uses bus A, while if it is to the right, it uses bus B. At the head of each bus, a steady stream of empty data frames is produced. These frames in conjunction with counters in each node are used for fair bus arbitration. DQDB networks span a distance of up to 160 km at a data rate of 45 Mbps.

D. Wide Area Network (WAN)

WAN technologies are able to provide good performance for large-size networks accommodating a huge number of nodes/connections. Usually, WANs are packet switched and use point-to-point links that interconnect the routers/switches and connect the WAN to MANs and LANs. A hierarchical addressing scheme is used that makes routing decisions easier. In most cases, *next-hop forwarding* is implemented in which routers decide to which router a packet is to be forwarded. The information about destinations and next-hop relations is kept in a routing table within each router. There are two types of routing mechanisms: static and dynamic routing. Static routing protocols use network topology information only to decide on a route, resulting in simple, low-overhead protocols. However, static protocols do not adapt to changing traffic conditions and link failures in the network. Dynamic routing protocols, on the other hand, take the current traffic conditions into account as well and are able to deal with network/link failures by routing traffic around the failed links. Most WANs use dynamic routing protocols for increased flexibility and efficiency. As mentioned before, ATM is increasingly used to implement WANs.

IV. NETWORK COMPONENTS

Components most often used in networks are repeaters, switches, routers, bridges, and gateways. The main distinction among these components is the layer it is operating on and if they change data that they relay. Networks using copper wires have a distance limitation. This limitation is primarily due to the properties of the copper wire used. A signal on a copper wire becomes weaker and weaker as it travels along the wire, which limits the distance it can travel. This is a fundamental problem with the technologies used in LANs. A common technology to overcome this signal loss is the use of repeaters. A *repeater* is an electronic device operating at the physical layer that takes the signal received on one side of the cable and transmits it amplified on the other side. To limit the propagation delay in Ethernet, a maximum of four repeaters can be used, which extends the Ethernet distance to up to 2500 m. Another device used to extend LANs is a *bridge*. A bridge operates on the data link layer and connects two LAN segments like a repeater does but it helps to isolate interference and other problems as it forwards only correct frames and ignores the corrupted ones. Bridges do not change frame information. In addition, most bridges perform frame filtering to transmit only frames that have their destination located in the LAN on the other side of the bridge. Adaptive bridges learn node locations automatically by observing source and destination addresses in the frame headers. Bridges can be used to extend LANs between two buildings or even over longer distances. Furthermore, satellite connections or fiber links can be used between two filtering bridges with buffering capabilities to further increase the LAN span.

Routers work on the network level, selecting the most convenient route based on traffic load, link speed, cost, and link failures. They route packets between potentially different networks. *Switches* also operate on the network level, transferring packets from one link to another, implementing packet-switched networks. Sometimes a device might combine router, switch, and bridge functions. A *gateway* is a device that acts as a translator between two systems that do not use the same communication protocols and/or architectures.

V. WIRELESS NETWORKS

Wireless networks have penetrated LAN and WAN systems. *Wireless LANs* provide low mobility and high-speed data transmission within a confined region such as a building/office. A base station connected to the wireline network is placed in a central spot in a building, while mobile users use a wireless transmitter/receiver with their computer/laptop (e.g., a PCMCI card) to communicate with the base station. The range of a wireless connection is around 100 m, depending on the base station location and building/office characteristics such as building material and floorplan. Recent technological developments have led to several *wireless WAN* systems aimed at personal communications services (PCS). Although the major application is voice transmission, current-generation wireless WAN systems are also capable of data communications in the range of 100 k bits per second (bps), and future-generation wireless systems are expected to provide a T3 data rate of 45 Mbps, which will enable multimedia communications including data, text, audio, image, video, animation, graphics, etc.

In general, wireless networks can be divided into two classes: (1) cellular networks and (2) ad-hoc networks. In *cellular networks*, a geographic area is divided into overlapping cells (as in cellular phone networks). A fixed base station (BS) is situated in each cell, and mobile hosts (MHs) (e.g., users with a laptop) roam these cells. Each base station is connected to a wireline network and is responsible for broadcasting data received by the wireline network to the mobile hosts currently present in the cell and for receiving data from the individual mobile hosts and forwarding the data over the wireline network to their destinations. Mobile hosts cannot directly exchange data among each other but must use the base station. Thus, to send data from MH1 to MH2, MH1 transmits the data to its corresponding base station BS1. Assuming that MH1 and MH2 are currently in different cells, BS1 will forward the data to base station BS2 of the cell MH2 resides in using the wireline network. BS2 then broadcasts the data to MH2.

In *ad-hoc networks*, there is no cellular structure and there are no fixed base stations. Instead, a mobile host communicates directly with another mobile host that is a single radio hop away. A mobile host can also function as an intermediate node relaying data between MHs that are more than a single hop apart. Ad-hoc mobile networks allow spontaneous LANs to be created anywhere and anytime, supporting nomadic collaborative computing. A route in an ad-hoc network comprises the source, the destination, and intermediate nodes. Movements by any of these nodes may affect the route. If any of the nodes involved in a specific route moves out of the range of its neighbor nodes, the route becomes invalid and the communication terminates. MHs that are able to communicate with each other (using intermediate nodes) form a group. On the one hand, independent groups that are currently not connected among each other might become connected through MH movements forming larger groups. On the other hand, a group may also break up into smaller, independent groups, depending on the node movements. The unpredictability of node movements and group memberships call for sophisticated and efficient routing protocols for ad-hoc wireless networks.

Both cellular and ad-hoc networks rely on high-speed wireless data transmission. The main problems in providing high-speed wireless connections are (1) the significantly higher bit error rate as compared to wireline networks, (2) frequency reuse in cellular wireless networks, and (3) handoff from one base station to the next in cellular networks. Most network protocols such as TCP or ATM were designed under the assumption of low bit error rates. For example, if a packet is lost under TCP, network congestion rather than poor channel conditions are assumed, and the packet will be retransmitted. If TCP would be used over a wireless channel, degraded channels would lead to a massive retransmission, resulting in very poor network performance. In cellular networks, a cell overlaps with its neighboring cell to guarantee complete network coverage over an area. Thus, if a certain frequency is used in a cell for a connection, this frequency cannot be used in any neighbor cell to avoid data collisions at the cell boundaries. This frequency restriction limits the number of different frequencies that can be used in a single cell, which limits the number of concurrent users in a cell. If a mobile host moves in a cellular network and crosses cell boundaries by leaving cell A and entering cell B, a handoff of that communication from the base station of cell A to the base station of cell B has to be performed. Thus, the connection has to be rerouted in the wireline network connecting those base stations. This rerouting has to be done in a way that the connection experiences minimized packet loss and minimized packet delay jitter during the handoff. In addition, the probability of connection termination during handoff (e.g., due to insufficient available bandwidth within the wireline network or within the new cell) should be minimal.

VI. INTERNET AND APPLICATIONS

A. Fundamentals of the Internet

The Internet was first introduced in the late 1960s. At that time, the Advanced Research Projects Agency (ARPA, later DARPA) launched an experimental network called

ARPANET. ARPANET was dedicated for university research teams working on DoD projects, allowing them to exchange data and results. This network was the first one to use store-and-forward packet switching mechanisms, which are still used in today's Internet. Starting with four nodes at 1969, ARPANET was spanning the United States, connecting more than 50 nodes just a few years later. The invention of TCP/IP (1974) and its integration into Berkeley UNIX helped ARPANET to grow to hundreds of hosts in 1983. The fact that ARPANET was not open to all academic centers led the National Science Foundation (NSF) to design a high-speed successor to the ARPANET, which was called NSFNET. In the mid 1980s, NSFNET and ARPANET were interconnected and many regional networks joined them. The name Internet was then associated with this collection of networks. Growing exponentially since 1990, the Internet now connects millions of hosts and tens of millions of users, and these numbers double every year. For a host to be "on-line" the only requirements are to run TCP/IP and to have a valid IP address. The IP address can either be permanently assigned to a node or a temporary address can be used (many users/machines are on-line only for a limited time while connected to their Internet provider so that a permanent address is not necessary). The topology of the Internet is unsymmetric, unstructured, and constantly changing. The Internet architecture is based on the TCP/IP reference model, as depicted in Fig. 4. Applications can bypass individual layers and can even access the network directly.

B. Tools and Applications

1. Domain Name System (DNS)

In the Internet, each host is assigned a unique binary IP address. In the header of each IP packet there are fields for the IP addresses of the source and the destination nodes that are used to find a route through the network from the source to the destination. For scalability reasons, the address space is divided into domains and sub-domains, resulting in an address with domain numbers separated by periods, such as 128.206.21.57. Each field in the address specifies a domain/sub-domain the node is in. This address format simplifies the routing algorithm running on each IP router. However, these addresses are usually hard to remember by users. Therefore, each node is also assigned an alphanumeric name, such as mackerel.cecs.missouri.edu that can be easier interpreted by a user. The alphanumeric names are usually used by applications to address nodes. Thus, an address translation is needed to translate the alphanumeric name into its corresponding IP address. The *Domain Name System* (DNS) is implementing this translation.

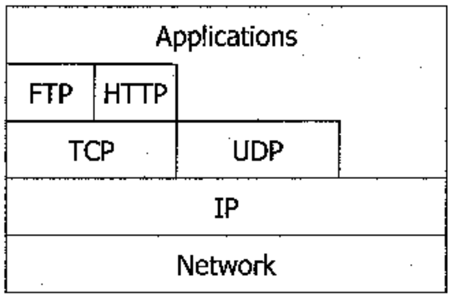


FIGURE 4 Internet architecture.

DNS is a hierarchical, domain-based naming scheme and a distributed database system that implements this naming scheme. For example, consider the node name *mackerel.cecs.missouri.edu*. The left-most segment represents the computer name (*mackerel*), the next two segments represent sub-domains: Computer Engineering and Computer Science Department (*cecs*) at the University of Missouri (*missouri*), while the right-most segment specifies the top-level domain (in our case *edu*, an educational institution). In general, the top-level domain includes names for each country (e.g., *us* for U.S.A. or *de* for Germany) or generic names such as *edu* for educational institutions, *com* for commercial, *gov* for the U.S. federal government, *int* for certain international organizations, *mil* for the U.S. armed forces, *org* for non-profit organizations, and *net* for network providers. To obtain a domain name, an organization or a user needs to register with the Internet authority. Once a domain name is given, the organization can subdivide the domain and create its own hierarchy. There are no standards for this interdomain structure, and the names of the computers belonging to this organization may not follow the same pattern. This gives the organizations freedom to change, coordinate, or create names in their own domain. The entire DNS system operates as a large distributed database. A root server occupies the top level of the naming hierarchy. The root server does not know all possible domain names, but it knows how to contact other servers to resolve an address. If a node needs an address translation, it sends a DNS request message to a local name server (the IP address of this server has to be known in advance by the node). If the server is able to resolve the name, it sends the translation back to the requesting node. Otherwise, this server contacts another name server higher up in the hierarchy to resolve the name.

2. E-mail

One of the first and very popular Internet applications was electronic mail or e-mail. It was implemented for the

ARPANET to provide the users with the ability to send each other messages over the network. This application has three components: message format standard, transfer protocol, and user interface (mail reader). The standard for message format, RFC 822, divides an e-mail message into a header and a body part. While the header contains simple information such as the names and e-mail addresses of the sender and the recipient, time of sending, and message subject, the body of the message can be more complicated. Initially, RFC 882 defined the body to contain ASCII text only. In 1993, the *Multi Purpose Internet Mail Extensions* (MIME) was introduced, which made it possible to also transmit binary files such as executables, images, audio, or video files. In MIME, the binary data is first encoded into plain ASCII text and then sent over the network. The protocol for transferring e-mail messages over the Internet is called *Simple Mail Transfer Protocol* (SMTP). SMTP requires a mailer daemon to run on the hosts. To send a message, the mailer daemon on the sending machine connects to the destination machine's daemon using SMTP over TCP and transmits the message, as well as receives messages if there are any. Received messages are then moved to the user's inbox. SMTP as the name suggests is a simple, ASCII-based protocol that uses commands of the type HELLO, MAIL, etc. This only works if both the sender and the receiver nodes are directly connected to the Internet. In the case that a node has no direct access to the Internet, it can still send and receive e-mail by using an e-mail server that is connected to the network. E-mail is received by the server and stored in a remote mailbox on the server. The destination node then connects to the server and fetches the e-mail from that server. Two simple protocols can be used for e-mail fetching: POP3 and IMAP. The *Post Office Protocol* (POP3) allows a user to connect to the mail server to download received messages. Once a message is downloaded, it is deleted from the server. The *Interactive Mail Access Protocol* (IMAP) transforms the e-mail server into an e-mail repository. A user can access the server and can read the messages without downloading them to the local machine. This protocol allows users to access their e-mail from different machines.

The last component necessary for e-mail, the mail reader or client program, organizes e-mail messages into folders and does the MIME encoding/decoding. Some popular mail readers are Qualcomm's Eudora, Netscape Browser, Microsoft Outlook, etc.

3. File Transfer Protocol (FTP) and Telnet

File Transfer Protocol is one of the oldest Internet applications and is still widely used. First implemented for the ARPANET, the FTP was designed to transfer/copy files from one host to the other over the Internet. FTP first establishes a TCP connection between the two hosts and requests an authorization from the user. After the user supplies a valid user name and password, an FTP control session starts. Simple commands such as append, open, send, and rename are used to transfer files. The user can either supply these commands directly or a file transfer application program can be used. Data transfer can be done in two modes, binary and ASCII mode, allowing the transfer of text files as well as binary files such as executables, images, or video/audio files. One or more FTP connections can exist concurrently between any pair of computers.

The telnet application is used to log-on to a remote computer. It enables a user to access and work on a remote node as if the user were directly connected to that node.

4. World Wide Web (WWW)

Applications like e-mail, FTP, and Telnet are popular and brought many people to use the Internet, but the real Internet boom started with the introduction of the World Wide Web (WWW). It started in 1989 with a proposal from the physicist Tim-Berners Lee from the European Center for Nuclear Research, CERN. CERN had several large international teams and many research projects with complex experiments. Since team members were distributed across several countries, it was very difficult for them to constantly exchange reports, drawings, plans, and other documents. The Web was intended to satisfy their needs by creating a web of linked documents easy to view and modify. Soon after 1993, when the first graphical interface, Mosaic, was released, it became very popular—so popular that it brought \$1.5 billion in stock for its successor Netscape. A year later, MIT and CERN created a WWW Consortium, an organization to further develop the Web, standardizing protocols and providing control over the Web. The main reason for the Web's popularity is the easy-to-use graphical interface and the enormous amount of information available to anyone. Not only is information available, but many companies offer services over the Internet such as on-line shopping, banking, education, and entertainment. The new possibilities for business are a very important factor; it is estimated that by 2003 the business-to-business commerce over the Internet will reach \$1.3 trillion. The Web is a set of servers, clients, and linked documents, and it uses the *Hypertext Transfer Protocol* (HTTP). Linked documents can contain text, image, data, audio, video, and, of course, links to other documents or parts of documents. A single document is usually called a page, and a set of linked pages is called a site. Web servers have a process listening on TCP port 80 for clients. Clients (Web browsers) request linked documents from a server and view them on the client's computer. The most common operations are GET and POST, to fetch a Web site and to append a web page, respectively. HTTP, like SMTP, is a text-oriented protocol with human understandable commands. An HTTP message has a header and a body. The message body is the data requested by the client or is empty in request messages. A message header contains information similar to the one in e-mail headers, such as the web page address. Instead of using the IP address of the web page, a *uniform resource locator* (URL) is used. The URL is a compact representation of the location and access method for a resource such as a web page. The URL provides a pointer to any object (files, file directories, documents, images, video clips, etc.) that is accessible on any machine connected to the Internet. Each URL consists of two parts: access method and location. For example, in the URL *http://www.cecs.missouri.edu, http* indicates that the HTTP protocol is to be used to access the web page located on the machine *www.cecs.missouri.edu*. Other access methods are ftp, mailto, news, telnet, etc.

The first version of HTTP1.0 establishes one connection per client and allows only one request per connection. This means that if a client wants to retrieve more than one web page it needs to set up multiple connections to the server. This is improved in the latest HTTP version, which allows persistent connections. Caching web pages is done either on the client side or on proxy servers to reduce Internet traffic and web page access time. HTTP supports caching by providing an expiration field in the web page header. A client can use the cached copy of the page until the date specified in this field expires, in which case the client has to reacquire the page from the same node.

5. Multimedia Applications

The success of the World Wide Web and the HTTP protocol led to the introduction of many new applications, such as multimedia. Multimedia applications combine text, image, video, and audio. While the transmission of text and images over the Internet is non-critical, video and audio transmissions are more problematic because of the relatively high transmission bandwidth requirements, especially for video.

Two general types of audio/video applications are streaming and conferencing. Streaming applications typically have servers delivering audio and/or video streams to the client. Examples are video broadcast, radio broadcast, and video on demand. It is common for these applications to broadcast the streams so that multiple clients are able to receive them. The second type, conferencing applications (e.g., video conferencing), is more interactive. Although different in nature, these applications need high bandwidth, low delay, and low delay *jitter* (variance of the delay).

Another important characteristic of multimedia applications is the need for multicast. During a video broadcast, a video stream has to be delivered to multiple destinations. The same is true for video conferencing applications. During a multicast, a sender could send the multicast stream multiple times to each individual destination. This, however, would result in a high traffic load in the network. Thus, the network itself should be able to distribute/copy packets of a single video stream. In IP networks such as the Internet, *IP Multicast* was therefore introduced and relies on special multicast routers. A multicast router is responsible for copying incoming packets belonging to a multicast stream to specific router outputs to generate the multicast distribution tree. An announce/listen mechanism was adopted in which multicast senders periodically send session announce messages. Nodes that want to join a certain multicast group will listen to these messages and will send a message announcing their join request. After joining the session, each node periodically sends announce messages. The multicast routers will listen to the announce messages and will generate the multicast tree according to the current session membership. If a node leaves the session or a temporary network failure occurs, the nearest router implementing the session will stop receiving the node's announce messages and will stop sending any multicast packets to that node. Thus, changes in session membership are handled locally in the network, resulting in scalability of the IP multicast.

Another important mechanism for multimedia communication is the handling of temporary network congestion during a multimedia transmission. The *real-time transport protocol* (RTP) in conjunction with the *real-time control protocol* (RTCP) is most often used today to deliver streaming video over the Internet that can be used to react on network congestion. RTP is not a protocol layer, but rather a tree-based hierarchy of interdependent protocols supporting different video and audio coding schemes. Among the services provided by RTP are delivery monitoring, time stamping and sequence numbering of packets for packet loss detection and time reconstruction and synchronization of multimedia data streams, payload and coding scheme identification. RTP/RTCP runs over UDP/IP and uses IP multicast. RTCP is responsible for generating periodic sender and receiver reports that can be used by the multimedia application to detect temporary network congestion. A source could, for example, adjust the video coder rate to the current network state (e.g., decrease the rate during network congestion), resulting in a varying quality of the received video that depends on the current network conditions. The *real-time streaming protocol* (RTSP), a client-server multimedia presentation protocol used with RTP/RTCP, provides a "VCR-style" remote control functionality to audio/video streams. RTSP provides functions such as pause, fast forward, rewind, absolute positioning in a stream, etc.

VII. SECURITY

Computer networks have a very important feature—sharing. Sharing resources and data as well as extensive access to the Internet imply the need of some security mechanisms. While very intuitive, computer security is a quite ambiguous term. It is well known that a system (computer network) is as secure as its weakest element. However, defining security in general is quite hard. The intrinsic problem is that in order to define what a secure system is, the system's assets have to be evaluated. Since a computer network is a dynamic structure, the security should be a dynamic rather than a static process. The description of the levels and types of security services a network needs is called *security policy*. Some basic services to be provided in a network are authentication, authorization, integrity, and confidentiality. In the real world, we have established mechanisms to implement and enforce similar services as well. For example, passport and/or signature are enough to authenticate a person. In computer networks, analogous mechanisms are deployed. Usually, user name and password are sufficient to verify and accept a valid user of the network, relying on the authentication protocol. Preventing security holes and attacks is necessary, but sometimes detection and recovery from a system failure is equally or more important.

A. Security Services

Security services are intended to protect a system from security attacks, to prevent attacks, or both by utilizing different security mechanisms. *User authentication* is the process of verifying the identity of a user. In the case of a user-to-user communication, both users have to be checked. Traditionally, in the client–server domain, the authentication is focused on the client side, since the system should be protected from users and not vice versa. However, for some applications such as e-commerce, server authentication is equally important to ensure that it is the correct server a customer is communicating with. *Data authentication* describes the verification of a particular data or message origin.

Authorization refers to the restriction of access to data and/or nodes. A user can be accepted into the network/node through authentication, but he/she might not have access to all of the files. Restriction lists or access lists and membership control are generally provided by the operating system. Another important service is *integrity*. It protects transmitted data from changes, duplication, or destruction. Modifications due to an error or intruder can usually be detected and fixed by the network protocol. If the data is sensitive, then integrity is combined with confidentiality service. *Confidentiality* is a service that protects all user data transmitted over a network. Even if data is intercepted by a third party, that third party will be unable to read the data. The *non-repudiation* service prevents a sender or receiver from denying a transmitted message (e.g., for on-line purchase proof).

Security attack is defined as any action, intended or not, that compromises the security of the information and/or system. Attacks can generally be passive or active. Passive attacks can be the copying of information or a traffic analysis. Active attacks involve some modification of the original data or fabrication of new data, such as replay or interruption of data. Security mechanisms are designed to prevent, protect, and recover from security attacks. Since no technique is able to provide full protection, the designers and/or system administrators of a network are responsible for choosing and implementing different security mechanisms.

B. Security Building Blocks

To incorporate and enforce the security policy, appropriate mechanisms are needed. Some very fundamental mechanisms are encryption/decryption, security management tools, firewalls, and detection and recovery tools. The choice of a specific mechanism depends on the level of security needed.

1. Cryptography

Cryptography is one of the oldest and most powerful security-providing technologies. The word cryptography, science of information security, comes from the Greek word *kryptos*, meaning hidden. It is mainly associated with *encryption*, the process of scrambling data with a secret parameter called an encryption key into *ciphertext*. The opposite process, *decryption*, takes the ciphertext and converts it back to plain text with the help of a decryption key. The mechanism of breaking ciphers without having the decryption key is called *cryptanalysis*. This is usually done by an *intruder* who has access to the data transmission channel.

There are few different types of encryption methods: substitution, transposition, and one-time pad cipher. One of the oldest ciphers is *substitution cipher*, known as Caesar cipher, which substitutes every symbol with another from its group. For example A with D, B with G, and so on. This is very easy to analyze and break with common letter statistics. *Transposition cipher* preserves the symbol meanings but reorders them in a new way depending on the key word. *One-time pad* is a theoretically unbreakable cipher. It combines (e.g., using *exclusive or*) two strings, the plaintext and a secret key. The result is a cipher, which holds no information about the plaintext, and every plaintext is an equally probable candidate for a specific ciphertext. One general rule is that the more secure the method is, the more redundancy information is required. However, the more data that is available for a cryptanalyst, the higher the chances are to break the code. Modern cryptography used in computer systems applies the basic techniques described here with a tendency toward very complex algorithms and short secret keys. There are two main classes of encryption algorithms: symmetric and asymmetric.

Symmetric algorithms use one key to encrypt and decrypt the data. Therefore, the key should be distributed between the communicating parties in a safe way. An example is the *data encryption standard* (DES) algorithm. This cipher developed by IBM and standardized by the National Security Agency (NSA) was initially used with 56-bit keys. Unfortunately, this key size results in around 7.2×10^{16} different key combinations only, and it is possible to break a DES cipher using exhaustive search within a few hours using powerful PCs. To increase the security level of DES, a 128-bit key is used now. Using this key size, it requires around 100 million years to break a DES cipher with exhaustive search.

The problem of symmetric algorithms is the secure distribution of the secret keys among nodes. This distribution problem is solved by *asymmetric algorithms*. The idea is to use two different keys, one encryption (public) key, which is available to everyone, and one decryption (private) key, which is known only by the user owning the key. This scheme is also called *public key cryptography*. One efficient method developed in 1978 by a group at MIT is the *RSA algorithm* named after the authors Rivest, Shamir, and Adleman. The RSA algorithm uses the product of very large prime numbers to generate the public and private keys. In order to decrypt a cipher, an intruder knows the public key but has to calculate the private key by factorizing the public key. However, factorization of very large numbers is not trivial (for example, for a 200-digit public key, a billion years of computational time is required to calculate the private key). This makes the RSA algorithm very secure.

2. Authentication Protocols and Digital Signatures

Authentication protocols can be based on shared secret key, public key, key distribution center, or the *Kerberos protocol*. The protocol based on *shared secret key* requires users A and B to share a secret key in order to use the protocol. The protocol consists of five message exchanges. A first sends a communication initiation message to B. B does not know whether this message is from A or from an intruder, so B sends a very large random number to A. To prove its identity, A then will encrypt the number with the shared secret key and return it to B. B will decrypt the message to obtain the original number back. Because only A and B know the secret key, B now knows that the message is coming from A. Next, A sends a challenge (large random number) to B, B encrypts the number with the secret key and sends it back to A, and A decrypts the number to find out whether the message actually came from B. After this, the real communication can start. A problem with the secret key authentication is the secure distribution of the secret key.

The *public key authentication protocol* uses two keys per node, a public key for encryption and a private key for decryption. Everybody has access to the public key of a node, while the private key is secret. During authentication, random numbers are generated and exchanged, similar to the shared secret key protocol. The only difference is that the public key of the receiving node is used by the sending node to encrypt the random number, while the secret key of the receiving node is used to decrypt the received number. A disadvantage of this protocol is the non-trivial distribution of the public keys.

Another authentication method uses trusted *key distribution centers* (KDC). Each user has only one key that is shared with the distribution center. Whenever A wants to communicate with B, it generates a session key, encrypts the key with its own secret key, and sends it to the distribution center. The center knows A's secret key and is able to decrypt the session key. It then encrypts the session key with B's secret key and sends it to B, which is able to decrypt the session key again. The session key is then used for secure communication between A and B. To avoid replay attacks where intruders copy messages and resend them at a later time, time stamps or unique numbers are included in the messages to detect the message resending.

The *Kerberos authentication protocol* consists of a set of two additional servers, the *authentication server* (AS) and the *ticket-granting server* (TGS). The AS is similar to a key distribution center in that it shares a secret key with each user. To start a communication between users A and B, A contacts the AS. The server will send back a session key KS and a TGS ticket to A, both encrypted with A's secret key. The TGS ticket can later be used as proof that the sender is really A whenever A requests another ticket from the TGS server. A sends the request to communicate with B to the TGS. This request contains KS and the TGS ticket that A received from the AS. This ticket was encrypted by the AS using a secret TGS key. By decrypting the ticket, the TGS is therefore able to validate that it is communicating with user A. The TGS then creates a session key KAB for the A/B communication and sends two versions back to A: one version encrypted with KS and one version encrypted with B's secret key. A decrypts the first version by using its session key to obtain KAB. It then sends the second version to B, which is also able to decrypt KAB. Now A and B both have the same secret session key KAB to start a secure communication.

Authentication protocols authenticate users only. In many applications, such as financial transactions or e-commerce, messages themselves have to be authenticated as well. Digital signatures were therefore introduced. *Digital signatures* are used to verify the identity of the sender, to protect against repudiation of the message by the sender later on, and to detect if a receiver has concocted a message himself. In general, any public key authentication algorithm can be used to produce digital signatures. For example, if user A wants to send a message with a signature to user B, A first generates the signature (by applying a cryptographic function such as a hash function to the message) and encrypts it with his private key and then with B's public key as shown in Fig. 5. After receiving the message and the signature, user B will decrypt the signature first with his private key and then with A's public key. After decrypting the actual message, B can generate a message signature and can compare it with the decrypted signature to verify the message and its sender. The *Digital Signature Standard* (DSS) introduced in 1991 uses a similar mechanism.

3. Firewalls

A *firewall* is a device barrier between a secure intranet and the outside world (e.g., the Internet). Firewalls are typically implemented as "screening routers." A *screening router* is a router or gateway that examines the incoming/outgoing traffic and selectively routes packets based on various criteria. A firewall may range from impermeable (allowing little or no traffic in or out) to porous (allowing most or all traffic in or out). For example, a typical screening router may block inbound traffic traveling on any TCP/IP port except port 80 (generally used for WWW services). An "absolute" protection against flooding the intranet of packets from an external host, called denial-of-service attack, can be achieved by forbidding any inbound traffic but allowing outbound traffic. Given that some other security measures exist in the network and that users will want to use the network to share and exchange data, this measure is extreme in most cases. Some firewalls have the ability to provide dynamic port selection. This is helpful in cases when the ports are specified during the transmission. A second common type of firewall is a proxy-based firewall. *Proxy*, in general, is a process/computer situated between a client and a server. When a client sends a request to a server, or vice versa, the request is actually sent to the proxy, and the proxy passes the request to the server. These proxies can help to enforce security policies by examining the packets and dynamically deciding which packets to forward and which packets to drop. Proxies have to understand the application layer protocol used (e.g., HTTP, Telnet, FTP). Firewalls add a layer of protection to the whole network connected to it. Depending upon the setup of the firewall, it can be used for prevention (stop the traffic in particular direction and port), protection, and recovery. Protection can be implemented by analyzing the traffic streams. If the traffic behavior deviates from the normal/anticipated behavior, the traffic source can be localized, and access for this host can be prevented. Recording of traffic information and data transmitted is possible at a firewall and can be later used for recovery. Although firewalls are useful for protection from external users, they cannot protect the system from malicious internal users. Another vulnerability of firewalls is the use of mobile codes or tunneling techniques. While the inspected packets look innocent to the firewall, they might carry programs threatening the security of the system. Thus, firewalls should always be used together with security support services available inside the intranet.

VIII. QUALITY OF SERVICE IN COMPUTER NETWORKS

Many applications such as multimedia need some network guarantees to deliver a certain level of quality independent of the current or future network state. For example, to deliver a high-quality video, the network should provide a video connection with a guaranteed bandwidth of, e.g., 4 Mbps. Thus, a network should provide a certain level of *quality of service* (QoS). QoS is characterized by QoS parameters such as bandwidth, delay, and delay jitter. The user/application negotiates a QoS contract with the network by specifying QoS parameters that the network will guarantee during the lifetime of a connection.

In ATM networks, QoS guarantees are explicitly included through resource reservation, while IP networks are best-effort networks without any QoS guarantees. To enable QoS in IP networks such as the Internet, QoS mechanisms were introduced in IP. The two most important ones are *Integrated Services* (IntServ) and *Differentiated Services* (DiffServ). IntServ provides two different classes of services over the Internet: (1) the guaranteed service and (2) the controlled-load service. The *guaranteed service* class guarantees bandwidth and delay requirements for a connection. This service can be used for real-time audio/video connections with hard delay requirements. The

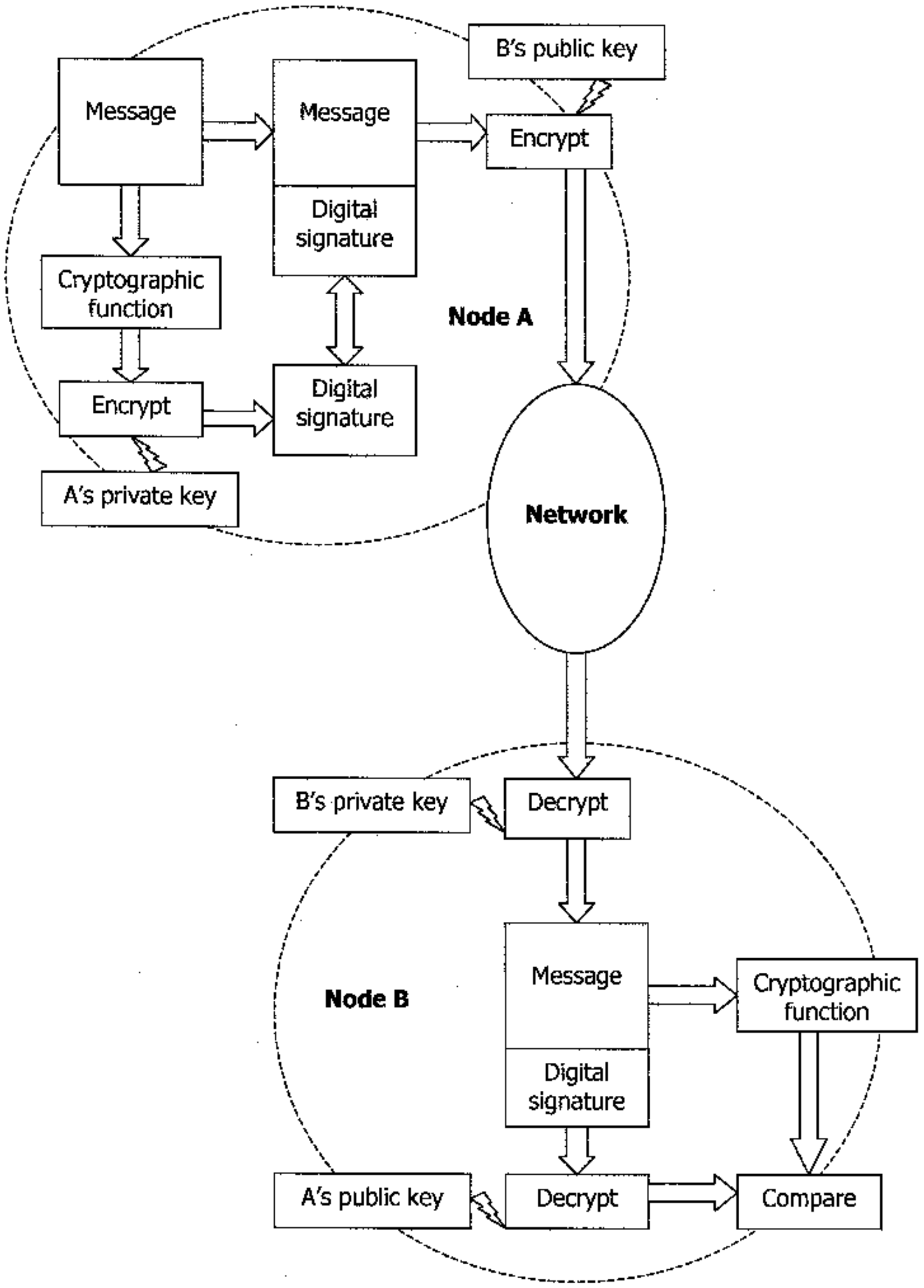


FIGURE 5 Example of public key encryption and digital signature. A sends B an encrypted message signed with A's digital signature. B decrypts the message and the signature and checks if the signature corresponds to the right message and right person.

controlled-load service class does not guarantee anything but tries to minimize packet loss and delay of a connection. This service can be used for adaptive audio/video applications.

IntServ relies on resource reservation and uses the *resource reservation protocol* (RSVP). RSVP is a signaling protocol that supports IP Multicast and permits receiver-driven resource reservation of an IP packet flow without establishing an explicit connection. To accomplish this, special RSVP routers are needed. The main problem with RSVP is that its applicability and scalability over large networks such as the Internet are limited. Because RSVP implements per-flow resource reservation, it requires very large state tables to be maintained in the core network routers, which limits its scalability.

The current approach for supporting IP QoS is *differentiated services*. Instead of reserving resources for each individual flow, DiffServ classifies these flows at the network edge and applies a per-class service in the network. Each service class is associated with a per-hop behavior (PHB) in the network. Networks supporting DiffServ therefore need two kinds of routers: boundary routers at the edge of the network that classify and shape/police incoming traffic and interior routers that apply PHBs to the different classes. Currently, three classes are supported: (1) best effort (regular IP traffic), (2) *expedited forwarding* (EF), and (3) *assured forwarding* (AF). The EF service supports low loss, low delay, and low delay jitter connections with a guaranteed peak bandwidth. EF emulates a point-to-point virtual leased line. The AF service defines four relative classes of service with each service supporting three levels of packet drop precedence. If a router encounters congestion, packets with higher drop precedence will be dropped ahead of those with a lower precedence. No specific bandwidth or delay constraints are defined for the different AF classes. EF and AF services are implemented through internal router queue management and scheduling such as random early detection and weighted fair queuing.

IX. TRENDS

The backbones of today's networks are telephone lines, coaxial cables, and optical fibers. This is going to change with the growing number of cell phones (predicted to outnumber PCs by 2005) that will result in more people being connected to the Internet via wireless than wired connections by 2008. The cell phone manufacturers are adopting new protocols for wireless communications. The current lack of interoperability (European cell phones do not work in the United States and vice versa) and the low bandwidth of current wireless connections are going to be changed by the new cell phone standard 3G (third generation). By 2004, 3G promises to bring interoperability, increase the bandwidth, and drop the average cost of making a cell phone call. The bandwidth predictions are for 144-Kbps access for automobile passengers and for up to 2-Mbps connectivity for stationary users for as low as 2 cents per minute. This will allow home users to receive high-quality movies and business travelers on the road to receive their multimedia e-mail. The future ubiquitous Internet (UbiNet) may evolve into a network of networks, all running IP over a combination or wired and wireless networks, and might be a reality by as early as 2008.

SEE ALSO THE FOLLOWING ARTICLES

ARTIFICIAL NEURAL NETWORKS • COMPUTER ARCHITECTURE • DATABASES • SOFTWARE ENGINEERING • TELECOMMUNICATION SWITCHING AND ROUTING • WIRELESS COMMUNICATION • WWW (WORLD-WIDE WEB)

BIBLIOGRAPHY

Comer, D. (1999). "Computer Networks and Internets," 2nd ed., Prentice-Hall PTR, Upper Saddle River, NJ.

Foster, I., and Kesselman, C., eds. (1999). "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kauffman, San Francisco, CA.

Händel, R., Huber, M., and Schröder, S. (1995). "ATM Networks: Concepts, Protocols, Applications," Addison-Wesley, Berkeley, CA.

Jurczyk, M., Siegel, H. J., and Stunkel, C. (1999). Interconnection Networks for Parallel Computers. In "Wiley Encyclopedia of Electrical and Electronics Engineering" (J. G. Webster, ed.), Vol. 10, pp. 555–564, Wiley, New York.

Kuo, F., Effelsberg, W., and Garcia-Luna-Aceves, J., eds. (1998). "Multimedia Communications," Prentice-Hall PTR, Upper Saddle River, NJ.

Lewis, T. (1999). "UbiNet: The ubiquitous Internet will be wireless," *IEEE Computer* **3210,** 127–128.

Peterson, L., and Davie, B. (2000). "Computer Networks: A System Approach," 2nd ed., Morgan Kauffman, San Francisco, CA.

Prycker, M. (1995). "Asynchronous Transfer Mode, Solution for Broadband ISDN," 3rd ed., Prentice-Hall PTR, Upper Saddle River, NJ.

Stallings, W. (1999). "Cryptography and Network Security: Principles and Practice," 2nd ed., Prentice-Hall PTR, Upper Saddle River, NJ.

Tanenbaum, A. (1996). "Computer Networks," 3rd ed., Prentice-Hall PTR, Upper Saddle River, NJ.

Toh, C.-K. (1997). "Wireless ATM and Ad-hoc Networks," Kluwer Academic, Boston, MA.

Microcomputer Design

Samuel C. Lee
University of Oklahoma

I. Introduction
II. From Computer to Microcomputer
III. Eight-Bit, 16-Bit, 32-Bit, and 64-Bit Microprocessors and Microcomputers
IV. Microcomputer-Based Systems Design

GLOSSARY

Central processing unit (CPU) Unit of a computer that includes the circuits controlling the interpretation and execution of instructions. The CPU of a computer system contains main storage, arithmetic unit, control registers, and scratch-pad memory.

Input/output (I/O) ports A typical input/output port consists of an 8-bit latch with tristate output buffers, along with control and device-selection logic.

Memory One of the three basic components of a computer. Memory stores information in binary form. Various types of memory are disk, drum, semiconductor, magnetic core, charge-coupled devices, bubble domain, etc.

Microinstructions/microprograms The execution of an instruction involves a sequence of information transfers from one register in the processor to another. A machine or computer instruction is made up of a number of these transfers, which can be likened to a program, and hence, the term for individual steps is microinstruction and the sequence of steps a microprogram.

Microprocessor Semiconductor central processing unit, consisting of the arithmetic logic unit and the control logic unit. The elements of the microprocessor are frequently contained on a single chip.

Microprocessing unit (MFIU) Main constituent of the hardware of the microcomputer, consisting of the microprocessor, the main memory (composed of read/write and read-only memory), the input/output interface devices, and the clock circuit.

MICROCOMPUTER DESIGN involves microcomputer hardware design and microcomputer software design. The former includes microcomputer memory design, microcomputer input/output design, and timing of microcomputer components. The latter includes the development and operation of microcomputer programs of machine, assembly, and high-level programming languages.

I. INTRODUCTION

The development of the microcomputer during the 1970s brought about a revolution in engineering design. The industrial revolution at the turn of the 19th century heralded the development of machines that could replace physical drudgery by mechanical means. Apart from a few exceptions, however, these machines required manual supervision, because the problem of controlling this mechanical power was not at all straightforward. Many types of automatic control system have appeared during the 20th century, based upon electric, mechanical, hydraulic, and fluidic principles. In each case the design techniques have been similar because each component of the system usually contributes a single well-defined function to the system's behavior. The microcomputer represents a fundamentally different approach to the design of a system. Its physical form is quite simple and reliable, consisting of a few general-purpose elements that can be programmed to make the system function as required. It is the controlling program that must be designed to give the system the required behavior, and that will contain "components" and "subassemblies" just like any other kind of engineering. The program, or *software*, is just as much a part of the engineered system as the physical *hardware*, but it is much less susceptible to failure, provided that it is designed properly. This new method uses a microprocessor containing a microprogrammed read-only memory (ROM). The microprocessor replaces hard-wired logic by storing program sequences in the ROM, rather than implementing these sequences with gates, flip-flops, counters, and so on. It can be said that 8–16 bits of ROM is the logical equivalent of a single gate. Assuming that on the average an integrated circuit (IC) contains 10 gates, the number of ICs that are replaced by a single ROM is as follows:

ROM memory sized bits	Gates replaced	ICs replaced
2048	128–256	13–25
4096	256–512	25–50
8192	512–1024	50–100
16,384	1024–2048	100–200
32,768	2048–4096	200–400

This method has the following advantages over the conventional hardware logic:

1. Manufacturing costs of products can be significantly reduced.
2. Products can get to the market faster, providing a company with the opportunity to increase product sales and market share.
3. Product capability is enhanced, allowing manufacturers to provide customers with better products, which can frequently command a higher price in the marketplace.
4. Development costs and time are reduced.
5. Product reliability is increased, which leads to a corresponding reduction in both service and warranty costs.

Before introducing this new method, let us compare three information-handling systems: the conventional computer system, the person–calculator system. and the microprogrammed computer system, by which the general structure and the operation of a microcomputer is described.

II. FROM COMPUTER TO MICROCOMPUTER

A typical digital computer consists of the following three parts:

1. A central processor unit (CPU)
2. A memory
3. Input/output (I/O) ports

The CPU contains two major parts: an arithmetic/logic unit (which is often simply referred to as the ALU) and a control unit. A block diagram showing these basic elements and the organization of a digital computer is shown in Fig. 1. The memory is composed of storage space for a large number of "words," with each storage space identified by a unique "address." The word stored at a given address might be either computational data or a machine directive (such as add, or read from memory). Two temporary store registers, each capable of containing one word, complete our memory. These registers are designated as memory address register (MAR) and memory data register (MDR). The MAR contains the binary representation of the address where information is to be read from memory or written (stored) into memory, and the MDR contains the data being exchanged with memory.

Based on its usage. a memory may be referred to as a *program memory* or a *data memory*. The program memory

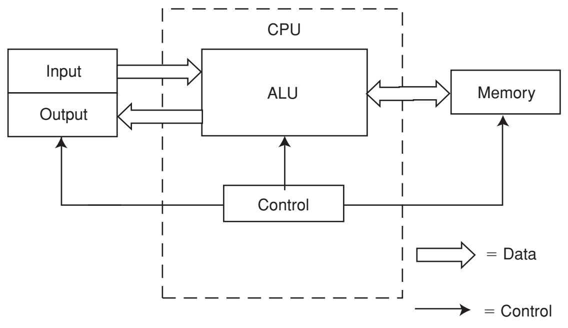


FIGURE 1 Basic elements of a digital computer.

serves primarily as a place to *store instructions*, the coded pieces of data that direct the activities of the CPU. A group of logically related instructions stored in memory is referred to as a *program*. The CPU "reads" each instruction from memory in a logically determinate sequence and uses it to initiate processing actions. If the program structure is coherent and logical, processing produces intelligible and useful results. The data memory is used to store the data to be manipulated. The CPU can access any data stored in memory, but often the memory is not large enough to store the entire data bank required for a particular application. The program can be resolved by providing the computer with one or more *input ports*. The CPU can address these ports and input the data contained there. The addition of input ports enables the computer to receive the information from external equipment (such as a paper tape reader) at high rates of speed and in large volumes.

Almost any computer requires one or more *output ports* that permit the CPU to communicate the result of its processing to the outside word. The output may go to a display, for use by a human operator; to a peripheral device that produces "hard copy," such as a line printer; or to a peripheral storage device, such as a magnetic tape unit; or the output may constitute process control signals that direct the operations of another system, such as an automated assembly line. Like input ports, output ports are addressable. The input and output ports together permit the processor to interact with the outside world. Data exchanges take place with external or peripheral devices via an I/O register.

The CPU unifies the system. It controls the functions performed by the other components. The CPU must be able to fetch instructions from memory, decode their binary contents, and execute them. It must also be able to reference memory and I/O ports as necessary in the execution of instructions. In addition, the CPU should be able to recognize and respond to certain external control signals, such as INTERRUPT and STOP requests. The functional units within a CPU that enable it to perform these functions are described below.

A. Registers

Registers are temporary storage units within the CPU. Some registers, such as the program counter and instruction register, have dedicated uses. Other registers, such as the accumulator, are used for more general purposes.

B. Accumulator

The accumulator usually stores one of the operands to be manipulated by the ALU. A typical instruction might direct the ALU to add (or to perform similar logical operations upon, e.g., OR) the contents of some other register to the contents of the accumulator and store the result in the accumulator. In general, the accumulator is both a source (operand) and destination (result) register. Often a CPU will include a number of additional general-purpose registers that can be used to store operands or intermediate "scratch-pad" data.

C. Program Counter (Jumps, Subroutines, and the Stack)

The instructions that make up a program are stored in the system's memory. The central processor examines the contents of the memory to determine what action is appropriate. This means that the processor must know which location contains the next instruction. Each of the locations in memory is numbered, to distinguish it from all other locations in memory. The number that identifies a memory location is called its *address*. The processor maintains a counter that contains the address of the next program instruction. This register is called the *program counter*. The processor updates the program counter by adding "1" to the counter each time it fetches an instruction, so that the program counter is always current. The programmer therefore stores his instructions in numerically adjacent addresses, so that the lower addresses contain the first instructions to be executed and the higher addresses contain later instructions. The only time the programmer may violate this sequential rule is when the last instruction in one block of memory is a *jump* instruction to another block of memory. A jump instruction contains the address of the instruction that is to follow it. The next instruction may be stored in any memory location, as long as the programmed jump specifies the correct address. During the execution of a jump instruction, the processor replaces the contents of its program counter with the address embodied in the jump. Thus the logical continuity of the program is maintained.

A special kind of program jump occurs when the stored program accesses or "branches" to a subroutine. In this kind of jump, the processor is logically required to "remember" the contents of the program counter at the time that the jump occurs. This enables the processor to resume execution of the main program when it is finished with the last instruction of the subroutine. A *subroutine* is a program within a program. Usually it is a general-purpose set of instructions that must be executed repeatedly in the course of a main program. Routines that calculate the square, the sine, or the logarithm of a program variable are good examples of the functions often written as subroutines. Other examples might be programs designed for inputting or outputting data to a particular peripheral device.

The processor has a special way of handling subroutines, in order to ensure an orderly return to the main program. When the processor receives a jump to subroutine instruction, it increments the program counter and stores the counter's contents in a register memory area known as the *stack*. The stack thus saves the address of the instruction to be executed after the subroutine is completed. Then the processor stores the address specified in the subroutine jump in its program counter. The next instruction fetched will therefore be the first step of the subroutine. The last instruction in any subroutine is a branch back. Such an instruction need specify no address. When the processor fetches a branch-back instruction, it simply replaces the current contents of the program counter with the address on the top of the stack. This causes the processor to resume execution of the program at the point immediately following the original branch.

Subroutines are often *nested;* that is, one subroutine will sometimes call a second subroutine. The second may call a third, and so on. This is perfectly acceptable, as long as the processor has enough capacity to store the necessary return addresses, and the logical provision for doing so. In other words, the maximum depth of nesting is determined by the depth of the stack itself. If the stack has space for storing three return addresses, then three levels of subroutines may be accommodated. Processors have different ways of maintaining stacks. Most have facilities for the storage of return addresses built into the processor itself. The integral stack is usually more efficient, since fewer steps are involved in the execution of a call or a return.

D. Instruction Register and Decoder

Every computer has a *word length* that is characteristic of that machine. A computer's word length is usually determined by the size of its internal storage elements and interconnecting paths (referred to as *buses*), for example, a computer whose registers and buses can store and transfer 8 to 32 bits. The characteristic 8-bit field is referred to as a *byte*. Each operation that the processor can perform is identified by a unique binary number known as an *instruction code* or *operation code* (OP code). An 8-bit word used as an instruction code can distinguish among 256 alternative actions, more than adequate for most processors.

The processor fetches an instruction in two distinct operations. In the first, it transmits the address in its program counter to the memory. In the second, the memory returns the addressed byte to the processor. The CPU stores this instruction byte in a register known as the *instruction register* and uses it to direct activities during the remainder of the instruction execution. The mechanism by which the processor translates an instruction code into specific processing actions requires more elaboration than we can afford here. The concept, however, will be intuitively clear to any experienced logic designer. The 8 bits stored in the instruction register can be decoded and used to selectively activate one of a number of output lines, in this case up to 256 lines. Each line represents a set of activities associated with execution of a particular instruction code. The enabled line can be combined coincidentally with selected timing pulses, to develop electrical signals that can then be used to initiate specific actions. This translation of code into action is performed by the *instruction decoder* and by the associated control circuitry.

An 8-bit field is more than sufficient, in most cases, to specify a particular processing action. There are times, however, when execution of the instruction code requires more information than 8 bits can convey. One example of this is when the instruction references a memory location. The basic instruction code identifies the operation to be performed, but it cannot also specify the object address. In a case such as this, a two-word instruction must be used. Successive instruction bytes are stored in sequentially adjacent memory locations, and the processor performs two fetches in succession to obtain the full instruction. The first byte retrieved from memory is placed in the processor's instruction register, and the second byte is placed in temporary storage, as appropriate. When the entire instruction is fetched, the processor can proceed to the execution phase.

E. Address Register(s)

A CPU may use a register or register pair to temporarily store the address of a memory location that is to be accessed for data. If the address register is *programmable* (i.e., if there are instructions that allow the programmer to alter the contents of the register), the program can "build" an address in the address register prior to executing a *memory reference* instruction (i.e., an instruction that reads data from memory, writes data to memory, or operates on data stored in memory).

F. Arithmetic/Logic Unit (ALU)

By way of analogy, the ALU may be thought of as a super adding machine with its keys commanded automatically by the control signals developed in the instruction decoder and the control circuitry. This is essentially how the first stored-program digital computer was conceived. The ALU naturally bears little resemblance to a desktop adder. The major difference is that the ALU calculates by creating an electrical analogy rather than by mechanical analogy. Another important difference is that the ALU uses binary techniques, rather than decimal methods, for representing and manipulating numbers. In principle, however, it is convenient to think of the ALU as an electronically controlled calculator. The ALU must contain an adder that is capable of combining the contents of two registers in accordance with the logic of binary arithmetic. This provision permits the processor to perform arithmetic manipulations on the data it obtains from memory and from its other inputs. Using only the basic adder, a capable programmer can write routines that will subtract, multiply, and divide, giving the machine complete arithmetic capabilities. In practice, however, most ALUs provide other built-in functions, including hardware subtraction, Boolean logic operations, and shift capabilities.

The ALU contains *flag bits*, which register certain conditions that arise in the course of arithmetic manipulations. Flags typically include *carry* and *zero*. It is possible to program jumps that are conditionally dependent on the status of one or more flags. For example, the program may be designed to jump to a special routine, if the carry bit is set following an addition instruction. The presence of a carry generally indicates an overflow in the accumulator and sometimes calls for special processing actions.

G. Control Circuitry

The control circuitry is the primary functional unit within a CPU. Using clock inputs, the control circuitry maintains the proper sequence of events required for any processing task. After an instruction is fetched and decoded, the control circuitry issues the appropriate signals (to units both internal and external to the CPU) for initiating the proper processing action. Often the control circuitry will be capable of responding to external signals, such as an interrupt request. *An interrupt* request will cause the control circuitry to temporarily interrupt main program execution, jump to a special routine to service the interrupting device, then automatically return to the main program. A diagram of a simplified CPU plus a memory is depicted in Fig. 2. A sound understanding of these basic operations is a necessary prerequisite to examining the specific operations of a particular computer.

H. Timing

The activities of the central processor are cyclical. The processor fetches an instruction, performs the operations required, fetches the next instruction, and so on. Such an orderly sequence of events requires timing, and the CPU therefore contains a free-running oscillator clock, which

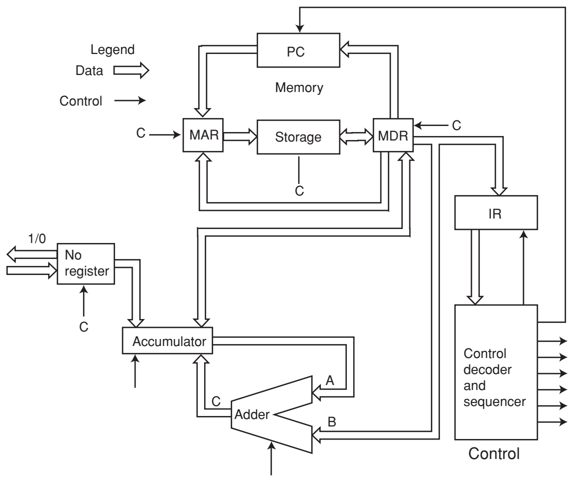


FIGURE 2 Simplified CPU plus memory.

furnishes the reference for all processor actions. The combined fetch and execution of a single instruction is referred to as a *machine cycle*. The portion of a cycle identified with a clearly defined activity is called a *phase*. And the interval between pulses of the timing oscillator is referred to as a *clock period*. As a general rule, one or more clock periods are necessary to the completion of a phase, and there are several phases in a cycle.

I. Instruction Fetch

The first phase(s) of any machine cycle will be dedicated to fetching the next instruction. The CPU issues a read operation code and the contents of the program counter are sent to program memory, which responds by returning the next instruction word. The first word of the instruction is placed in the instruction register. If the instruction consists of more than one word, additional cycles are required to fetch each word of the instruction. When the entire instruction is present in the CPU, the program counter is incremented (in preparation for the next instruction fetch) and the instruction is decoded. The operation specified in the instruction will be executed in the remaining phases of the machine cycle. The instruction may call for a memory read or write, an input or output, and/or an internal CPU operation, such as a register-to-register transfer or an add-registers operation.

J. Memory Read

The instruction fetched may then call for data to be read from data memory into the CPU. The CPU issues a read operation code and sends the proper memory address; memory responds by returning the requested word. The data received are placed in the accumulator (not the instruction register).

K. Memory Write

A program memory write operation is similar to a read except for the direction of data flow. The CPU issues a write operation code, sends the proper memory address, then sends the data word to be written into the addressed memory location.

L. Input/Output

Input and output operations are similar to memory read and write operations, with the exception that a peripheral I/O port is addressed instead of a memory location. The CPU issues the appropriate input or output command, sends the proper device address, and either receives the data being input or sends the data to be output. Data can be input/output in either parallel or serial form. All data within a digital computer are represented in binary-coded form. A binary data word consists of a group of bits: Each bit is either a 1 or a 0. *Parallel I/O* consists of transferring all bits in the word at the same time, one bit per line. *Serial I/O* consists of transferring one bit at a time on a single line. Naturally, serial I/O is much slower, but it requires considerably less hardware than does parallel I/O.

M. Interrupts

Interrupt provisions are included on many central processors as a means of improving the processor's efficiency. Consider the case of a computer that is processing a large volume of data, portions of which are to be output to a printer. The CPU can output a byte of data within a single machine cycle, but it may take the printer the equivalent of many machine cycles to actually print the character specified by the data byte. The CPU will have to remain idle waiting until the printer can accept the next data byte. If an interrupt capability is implemented on the computer, the CPU can output a data byte, then return to data processing. When the printer is ready to accept the next data byte, it can request an interrupt. When the CPU acknowledges the interrupt, it suspends main-program execution and automatically branches to a routine that will output the next data byte. After the byte is output, the CPU continues with main program execution. Note that this is, in principle, quite similar to a subroutine call, except that the jump is initiated externally rather than by the program. More complex interrupt structures are possible, in which several interrupting devices share the same processor. Interruptive processing is an important feature that enables maximum utilization of a processor's capacity.

N. The Person–Calculator System

Having introduced the structure and operations of a digital computer, we now turn to another information-processing system, the person–calculator system. This system has all the "functional parts" that a digital computer has. The person's fingers represent the input; his or her eyes, coupled with the calculator's output, represent the output; the brain is the control element, while the calculator electronics function as the ALU; and the brain also serves as the memory.

Let us examine the sequence of events that occur when our person-calculator solves the problem $2 + 3 = ?$:

1. Brain accesses first number to be added, a 2.
2. Brain orders hand to depress 2 key.
3. Brain identifies addition operation.
4. Brain orders hand to depress + key.
5. Brain accesses second number to be added, a 3.
6. Brain determines that all necessary information has been provided and signals the ALU to complete computation by ordering hand to depress = key.
7. ALU (calculator) makes computation.
8. ALU displays result on readout.
9. Eyes signal brain; brain recognizes this number as the result of the specified calculation.
10. Brain stores result, 5, in a location, which it appropriately identifies to itself to facilitate later recall.

Now let us see how a digital computer solves this problem without having any human intervention. The task involved in solving this problem may be described as follows: "Read in a number from the I/O. Store it in memory location 10. Read in another number from the I/O. Store it in memory location 11. Add the two numbers together. Store the result in the memory location 12, print out the output, and halt."

A "program" has been written to execute this task and is stored in consecutive memory locations beginning at 0. The program, written in an artificial symbolic "language," is shown in Table I. Suppose that our machine is an 8-bit word machine. Each instruction is described by one word. Half of it (4 bits) is used for describing the operation code (OP code) and the other half (4 bits) for describing the address (ADDR). Address would be longer in actual design; so would the OP code. The preceding sample problem, for example, involves five distinct operations: CLEAR, INPUT, STORE, ADD, and HALT. They

TABLE I Program

Memory location	Instruction (contents)
0	Input to accumulator.
1	Store accumulator at location 10.
2	Input to accumulator.
3	Store accumulator at location 11.
4	Add accumulator at location 10.
5	Store accumulator at location 12.
6	Display accumulator to I/O.
7	Halt.

are usually abbreviated by three-letter words and coded in binary codes. For example:

Operation	OP code	Meaning
INP	0100	Input a number from 1/0 device I (0001) into the accumulator.
STR	0110	Store the number in the accumulator to a memory location.
ADD	0101	Add the number in a memory location to the number in the accumulator.
DSP	0111	Display the number in the accumulator on 1/0 device 1.
HLT	0001	Stop operation.

With these OP codes, one can rewrite the program in Table I in a binary coded form as shown in Table II. The operation is complete! No human intervention is required—all operations are automatic.

All computers (processors, CPUs, etc.) operate in a similar manner, regardless of their size or intended purpose. It must be emphasized that many variations are possible within this basic architectural framework. More common variations include highly sophisticated I/O structures (some of which have direct and/or autonomous communication with memory), multiple accumulators for programming flexibility, index registers that allow a memory address to be modified by a computed value, multilevel interrupt capability, and on and on.

TABLE II Program in Table I Described in Binary Code

	Instruction(contents)	
Memory location	OP code (OPR)	ADDR (OPA)
0000	0100	0001
0001	0110	1010
0010	0100	0001
0011	0110	1011
0100	0101	1010
0101	0110	1100
0110	0111	0001
0111	0001	—

One of the most exciting architectural concepts in the early 1970s was that of microprogrammed control. A microprogrammed computer differs from the classical machine in the control unit implementation. The classical machine has for its control unit an assemblage of logic elements (gates, counters, flip-flops, etc.) interconnected to realize certain combinatorial and sequential Boolean equations. On the other hand, a microprogrammed machine utilizes the concept of a "computer within a computer." That is, the control unit has all the functional elements that comprise a classical computer, including ROM. This microprogrammed control unit is called a *microprocessor*. The purpose of this "inner computer," which is not apparent to the user, is to execute the user's program instructions by executing a series of its own microinstructions, thereby controlling data transfers and all functions from computed results. Herein lies the key distinction: the control signals, and hence the very "personality" of the computer, are controlled by computed results. The implication is immediately obvious—by simply changing the stored microprogram that generates the control signals, we may alter the entire completion of the computer. By altering a few words stored in the ROM, we can cause our computer to behave in an entirely new fashion—to execute a completely different set of instructions, to emulate other computers, to tailor itself to a specified application. It is this capability for "custom tailoring" that allows such a machine to be optimized for a given us. By so extracting the utmost measure of efficiency, a microprogram-controlled machine is more reliable, less costly, and easier to adapt to any given situation. Any computer that uses a microprocessor as its main processor is considered a *microcomputer*. A microcomputer is generally used for a dedicated task as part of a system.

III. EIGHT-BIT, 16-BIT, 32-BIT, AND 64-BIT MICROPROCESSORS AND MICROCOMPUTERS

Since their inception, digital computers have continuously become more efficient, expanding into new applications with each major technological improvement. A time chart of computer technology is shown in Fig. 3. The advent of minicomputers enabled the inclusion of digital computers as a permanent part of various process control systems. Unfortunately, the size and cost of minicomputers in dedicated applications has limited their use.

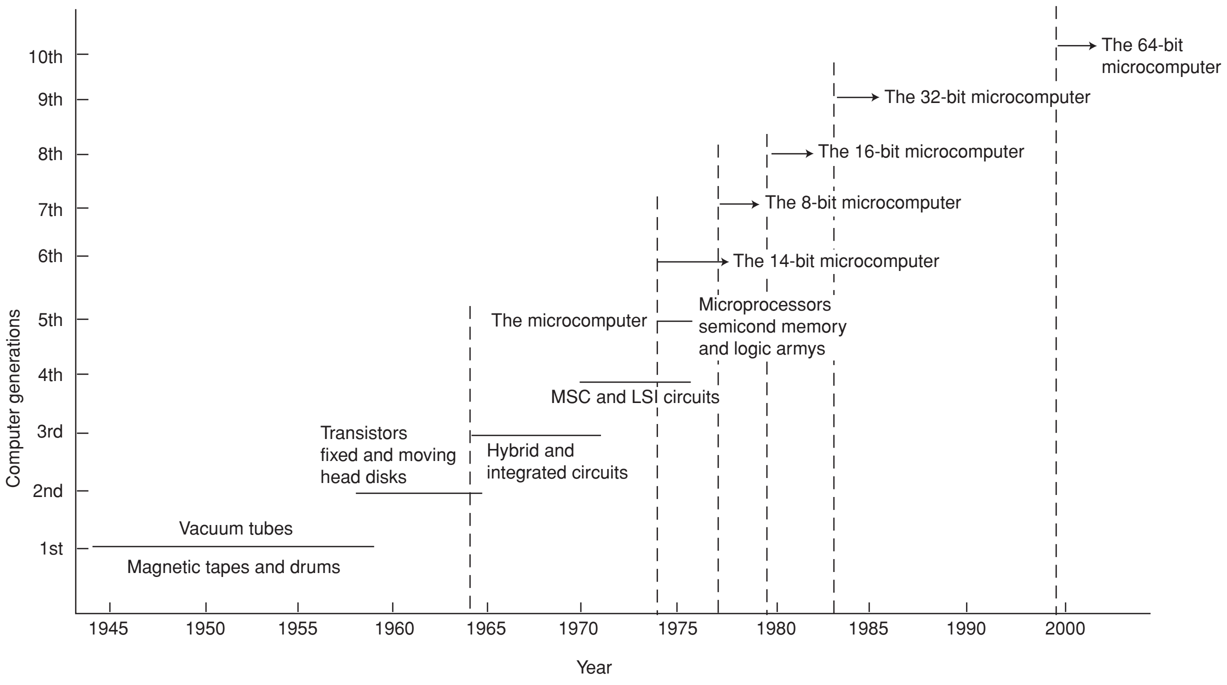


FIGURE 3 Time chart of computer technology.

The microprocessor age began when integrated circuit technology had advanced enough to put all of the necessary functions of a CPU into a single IC device. Since that time, the advancement of microprocessors has paralleled the advancement in microelectronic technology and the ability to include more and more logic on a single IC surface. Although the cost of a microprocessor increases with its complexity, it is much lower than the cost of the equivalent logic scattered over several less capable ICs. In addition to reducing the number of ICs needed to perform a given function, the total number of pins is reduced, and thus the assembly costs are decreased.

As microelectronics fabrication techniques have improved from the LSI level to the VLSI level, microprocessors have advanced from having four data lines and 12 address lines, four of which were also the four data lines, to 16 data lines and 20–24 independent address lines. The former were good only for constructing calculators and simple controllers; the latter can be used as CPUs in sophisticated general-purpose computers that rival the medium to large computers. The 16-bit single-chip microprocessors accommodate up to 24 megabytes of memory and include both multiprogramming and multiprocessing features. The latest 32-bit single-chip microprocessors, which can accommodate up to 4 gigabytes of memory, are designed for applications in graphics, workstations, parallel processing, multiuser systems, and real-time systems. To accompany these advanced processors, the IC manufacturers are also improving the variety and complexity of supporting interface, bus logic, and memory devices.

In the early 1970s, Intel succeeded in its manufacturing of 4-bit single-chip microprocessors 4004 and 4040 and their associated 4000 series chips. From 1974 to 1977, this family has expanded from the 4-bit microprocessor to the 8-bit 8008, 8080, and 8085 and their derivatives and supporting devices. During the same period, many other 8-bit microprocessors with different architectures and design, and using different semiconductor technology—for example, Motorola's MC6800, Zilog's Z-80, Mostek's 6502, and National Semiconductor's IMP8—were also available to the user. Beginning in 1979, 16-bit microprocessor series, such as 8086/8088, MC68000, LSI- II, and T19900, became available. These machines have more circuitry, higher speed, and more instructions and supporting system software than their predecessors, and thus provide more flexible computation power, power that is particularly useful when designing complex microcomputer systems. The latest 32-bit microprocessors from AT&T (WE32100 family), Intel (80X86 family), Motorola (68KC family), National (NS32332 family), Zilog (Z80000 family), and Advanced Micro Devices (AM29325 family) provide minicomputer performance in a desktop package.

The microcomputers (microcontrollers) are designed to be used in many dedicated systems, such as data- acquisition systems, process-control systems, data-processing systems, and remote-terminal control systems. There is a mushrooming variety of microprocessor applications. A few of them are special (high-level) language computers, microprocessor-based instruments, intelligent CRT (cathode-ray tube) terminals, point-of-sale terminals, computer peripheral controllers, on-board automobile control, control of automation and continuous processes, and home entertainment and games.

Generally, microprocessors are evolving into two directions, performance and integration. The performance direction emphasizes increased and faster processing power with the ability to store more data. Computers use these microprocessors as CPUs. The last 64-bit Intel Pentium III is an example of this type and the general public is more aware of it because of the popularity of personal computers. The integration direction emphasizes reduced chip count. The latest 32-bit Intel 80C251 and Motorola MC68CH11 microcontrollers are examples of this type. It is estimated that there are 10 times as many microcontrollers as general-purpose microprocessors used in consumer products. This is because microcontrollers are used in many machines, instruments, and control applications.

IV. MICROCOMPUTER-BASED SYSTEMS DESIGN

A. Design Requirements

One problem with the specification of design requirements is that everyday natural language is not always precise or unambiguous enough for the purpose. The result of discussions between the engineers and the customers for whom microcomputer-based systems are developed should be a document that sets out the agreed requirements for the system. The following six requirements for such a document are recommended.

1. It should specify only external system behavior. There is no reason why the requirements should specify internal behavior, and not doing so allows the designers greater freedom in performing their job.
2. It should specify constraints on the implementation. The constraints may be physical, such as size or weight, or otherwise, such as cost or country of origin of components.
3. It should be easy to change. A careful document-control system is needed to ensure that everyone involved with a project knows the current version of the requirements.
4. It should serve as a reference tool for those involved in system maintenance. Understanding exactly what a system has to do can be important In trying to find out what is wrong when it malfunctions!
5. It should record forethought about the life cycle of the system. This includes the way in which it is to be designed, implemented, commissioned, manufactured, tested, and maintained.
6. It should characterize acceptable responses to undesired events. When things go wrong, it should behave in a predictable (and, one hopes; responsible) way.

B. Design Philosophy

The approach normally taken to the problem of designing microcomputer-based systems is the so-called top-down approach. Here the designer breaks down the problem into smaller and smaller sections, and once it has been analyzed in enough detail a solution can be proposed to each part of the problem. The design process proceeds from the overall general view of the problem into greater and greater detail until the design has been fully described. The specification of the subsystem needed to handle each part of the problem must be carefully written and the interaction between the subsystems carefully defined. Once all the sections have been specified, the designer can decide which of the sections are to be constructed in hardware and which are to be based on software methods.

C. Hardware Design

Once a system has been specified by its designers, a decision must be made concerning the methods of construction: In some cases the same function could be performed using hardware or software. The microcomputer appears in an enormous range of forms, from single integrated circuits to boxed units with very considerable computing power. The designer of a system must also determine whether the project should use an "off the shelf" computer or, if some electronic construction is to be undertaken, the necessary approach. A number of factors must be weighed when deciding how a system is to be built. Designing a computer is not a task to be undertaken lightly, and for this reason it is almost always better to make use of "standard" equipment wherever possible. For many applications where only one or a few of a design are going to be produced, the best solution will be to make use of a boxed system from a microcomputer manufacturer. A standard chassis and other subassemblies are available from a number of manufacturers for each of the widely used standard sizes of printed circuit board, and this allows a system to be built up with a suitable mixture of processor, memory, and input/output for most applications.

In very-large-scale production, another possibility presents itself: the design of special integrated circuits. It is quite possible to incorporate special-purpose analog and/or digital interface circuitry on the same chip as the microprocessor, ROM, and RAM. This provides a low-cost solution for applications such as controlling the frequency of a television set or sensing the temperature within a microwave oven. In this way the unit cost can be minimized, but the high design cost means that a very large production run is needed to make this approach cost effective.

The hardware components must interact to form a composite system, and the exact nature of all these interactions must be set out in the specification. The precise relationship between the inputs and outputs of each function must be specified. Once this has been done, hardware must be bought or built to produce modules that carry out these functions. This can be done at the level of integrated circuit components, printed circuit boards, or boxed systems, depending upon the application.

D. Software Design

Because the software is designed according to the same top-down principles that are used with the hardware, it is not surprising that the same need to specify functions and document their interactions applies to the software. The software that results from a top-down approach to design will consist of a main program that consists largely of calls to subroutines, which will in turn reference other subroutines, and so on, down to the lowest level of subroutines and the instructions or statements that make them up. Thus the subroutines are the functions that comprise the overall software, and it is these subroutines that must first be specified in terms of what they do and how they interact with the rest of the system.

Each subroutine must be called with the parameters being presented in a defined way, and the manner in which any parameters are returned to the calling program must also be defined. The subroutine will carry out a set of operations on the input parameters to produce the output parameters. and these operations must be closely specified.

The specification for the subroutine should contain:

1. The name of the subroutine.
2. The number of parameters passed to the subroutine, the number of parameters returned, and the manner in which they are passed between the calling program and the subroutine.
3. The specification of the subroutine, in terms of the operations that it carries out on the input parameters and how the output parameters are generated. If the subroutine handles any input/output functions, then the way in which these relate to the parameters must be specified.
4. The time taken to carry out its function, and any restrictions on values of the input parameters. Although average timings are useful, worst-case timings are needed because these are used to ensure that there will always be enough time for all the subroutines to run in the application envisaged.
5. Any unusual action that the subroutine may take in the case of an “exception” condition, such as an attempt to divide by zero, or as the result of being presented with “illegal” data.
6. The amount of memory used by the subroutine, for code storage, data storage, and stack usage. Again, worst-case figures must be specified for the stack usage if this is variable.
7. The names of any other subroutines that are called in turn by this subroutine, so that any interaction between them can be checked.

If subroutines are specified in this way, enough information should be available to allow the software designer to check that there will be enough memory for the subroutines to operate together and there will be enough time for them all to perform their tasks. The interactions between the subroutines should only ever take place via the parameters. If global variables are used that are shared between two or more subroutines, then this fact should also be recorded in the specification of the subroutines. Where possible, the avoidance of global variables makes for simpler interactions between subroutines and for easier checking of overall operation.

E. Hardware/Software Trade-Offs

There are two main aspects to the design of a microcomputer-based system: the hardware and the software. It is only after one gets thoroughly into the use and application of microcomputers that one discovers the real underlying appeal: Hardware and software are essentially interchangeable. When a system is being designed, decisions often have to be made as to whether a function can best be performed using hardware or software. This fact can be used to bias the economics in favor of whatever objective you have established.

Software is generally more expensive to implement than hardware, but it is less expensive to duplicate. Because of the inherently sequential nature of computer programs, they are at least an order of magnitude more difficult to design, test, and debug than conventional electronics. However, once software has been correctly implemented, the cost of replicating it is very low in comparison with the equivalent amount of electronics.

As a general rule, if one is trying to minimize development costs, then a generalized microcomputer with simple software is often the best choice. When production volumes warrant, however, one must look carefully at the hardware/software trade-off to minimize unit replication costs. This usually means more software and less hardware.

As an example, consider the problem of counting external events that appear to the microcomputer as a series of pulses from some kind of transducer. If these pulses occur relatively infrequently, perhaps because of the movement of a mechanical component in the system, they can be counted in software by means of a suitable subroutine, possibly using an interrupt. Each time the count is incremented, a fixed amount of processor time is used, so that the faster the rate of arrival of pulses, the greater the proportion of the processor time that is taken up by the relatively simple operation of counting pulses. Thus, another method must be used to count pulses if the pulse frequency is expected to be high, and this will mean the inclusion of extra electronic hardware to perform the counting operation. The designer is faced with a choice between increasing the complexity of the hardware to count the pulses and increasing the program complexity—and possibly increasing the problems in debugging the software. Any function performed by software could in principle be carried out using hardware, but it is not always possible to replace hardware with software.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ARCHITECTURE • HUMAN-COMPUTER INTERACTION • INTEGRATED CIRCUIT MANUFACTURE • MAGNETIC RECORDING • MICROCOMPUTER BUSES AND LINKS • MINICOMPUTERS • SOFTWARE ENGINEERING

BIBLIOGRAPHY

Ayala, K. (2000). "The 80251 Microcontroller," Prentice-Hall, Upper Saddle River, NJ.

Cannon, D. L. (1982). "Fundamentals of Microcomputer Design," Texas Instruments Learning Center.

Lee, S. C. (1977). "Microcomputer Design and Applications," Academic Press, New York.

Rafiguzzaman, M. (1982). "Microcomputer Theory and Applications," Wiley, New York.

Spasov, P. (1999). "Microcontroller Technology: The 68HC11," 3rd ed. Prentice-Hall, Upper Saddle River, NJ.

Tocci, R. J., and Laskowski, L. P. (1982). "Microprocessors and Microcomputers," 2nd ed. Prentice-Hall, Englewood Cliffs, NJ.

Wise, K. D., Chen, K., and Yokely, R. E. (1980). "Microcomputers: A Technology Forecast and Assessment to the Year 2000," Wiley, New York.

Microcomputer Buses and Links

J. D. Nicoud
Ecole Polytechnique Fédérale de Lausanne

I. Single Master Systems
II. Multimaster Operations
III. Standard Buses
IV. Future Trends

GLOSSARY

Decoder Circuit capable of deciding from a set of signals (for example, memory address lines) whether a device should participate in an operation (for example, a memory transfer).

Hot plug Capability for a bus to accept I/O devices to be inserted or removed during operations and be recognized as such by the software. Common with recent serial buses, it is expensive and seldom supplied with boards.

Input/output device Any piece of equipment connected to a processor and used for introducing information (keyboard, disk) or extracting information (printer, robot). Often referred to as an I/O device.

Link Data transmission channel, usually implemented as a set of data, control lines, and ground lines.

Memory Electronic device capable of storing information. A set of 10 address lines can access $2^{10} = 1024 = 1$ kilo words of data; each word is usually 8, 16, or 32 bits wide.

Plug and play Capability for the processor to recognize the connected I/O devices and install the corresponding software drivers at power-up.

Processor Electronic device that processes binary information (add, compare, move, etc.) according to instructions located in a memory. The information is then transferred from and to I/O devices.

Protocol Set of hardware or software rules used when transferring information in order to guarantee a correct handshake and reliable transfers.

MICROCOMPUTERS communicate over telephone lines and local networks. They transfer information with peripherals over more dedicated lines. Inside the microcomputer box, a set of lines called a bus allows for fast transfer between the microprocessor, the memory, and the input/output modules.

Standard buses and links have been defined in recent years for major groups of applications. Their complete understanding is a specialist's work. This paper emphasizes the principles of addressing and data transfer over parallel and serial buses. The trend is for smart I/O devices connected on a serial bus such as USB.

I. SINGLE MASTER SYSTEMS

A. Introduction

In any general-purpose computer, workstation, or dedicated controller based on a microprocessor, data transfers

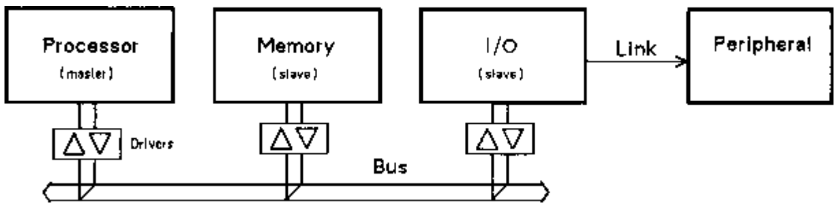


FIGURE 1 Typical computer system.

are continuously being performed between the processor, the memory, and the input/output (I/O) devices. Frequent transfers imply a high bandwidth, economically feasible only for short distances. For distances greater than a few meters, the cost of the electrical or optical lines forces the serialization of information.

A typical computer system consists of the processor (master) and several memory and I/O devices (slaves) interconnected by a set of data and control lines named buses (Fig. 1). These devices are generally clearly recognizable when they are connected by a backplane bus (Fig. 12). They are frequently mixed on a single board computer. The bus allows bidirectional transfers between a possibly variable set of devices. The links toward the peripherals have a simpler structure since they are point to point. Connecting several devices on a bus, or transferring data over long distances, implies solving many electrical problems correctly and taking care of the propagation time inside devices and over the transmission lines.

B. Simple Point-to-Point Transfers

All microprocessor systems have a strong hierarchical dependence. The processor (master) reads and writes the information with each device, which cannot communicate directly. A simple write cycle (Fig. 2a) needs a set of data lines Di for the information (for example, 8, 16, or 32 bits) and one line for the synchronization (write strobe WRS). The minimum setup time t_s and hold time t_h of data with respect to the WRS, as well as the write time t_w, depend on the technology and on the static or dynamic (edge-triggered) nature of the storage elements.

For a read cycle, in response to the request (read strobe RDS), the new data is available with an access time t_a (Fig. 2b). Usually, the communication bus is in a neutral inactive state as long as the selection has not been performed and one can define an activation time t_{on} and a disable time t_{off}.

Processors perform both read and write cycles. The data lines Di can be multiplexed if the outputs are three-stated when not selected. Two separate WRS and RDS lines, never activated simultaneously, can control the transfer (Fig. 2c). This solution is preferred by several manufacturers and used, for example, on the early IBM-PC bus. Another solution is to have a single strobe line ST and a direction line $\overline{\text{RD}}$/WR (Fig. 2d). When the $\overline{\text{RD}}$/WR is active, a pulse on the ST line triggers a write transfer from the master toward the slave; otherwise, it is a read cycle.

All of these cycles are known as synchronous, since the transfer cycle is performed in a given lapse of time dependent on the strobe duration. If the slave is too slow, data may be lost; one needs a feedback signal. A first solution is to activate the signal only when the cycle has to be slowed down (WAIT signal, Fig. 3a). A better solution, used on all 32-bit microprocessors, is to wait for a positive acknowledge, AK, in response to the strobe ST (Fig. 3b). If the slave is not able to provide the information, a negative acknowledge, NK (frequently named bus error), must be generated, either by the slave or by a dedicated circuit. Signals in Figs. 3 and 4 are active high. They are frequently inverted (active low) for electrical reasons.

C. Selection

In order to select the devices on a bus, as well as the memory and I/O registers inside the devices, processors use a set of address lines Ai, and a selection cycle that activates the three-state outputs of only one device on the bus. The selection cycle can be implemented as a dedicated write cycle, with an address strobe, AS, and an address acknowledge, AK, signal. The data strobe, DS, and data acknowledge, DK, lines synchronize the read or write data transfer that follows (Fig. 4a). The selection cycle is frequently identified with the transfer cycle and uses the same signals, as shown in Fig. 3. The transfer is delayed by the time the decoder needs to activate the selected device (Fig. 4b). Data acknowledge frequently anticipates data valid, assuming how much time the processor will take to really need the data.

Address and data can be multiplexed. This is slightly slower and costs address latches, but saves a lot of lines, especially with 32-bit processors. The address strobe is frequently named address latch enable, ALE, and directly controls the address latch (Fig. 5), which is either on the master board (demultiplexed bus) or on each slave board (multiplexed bus). The data transfer is continued as before using, for example, separate RDS and WRS signals.

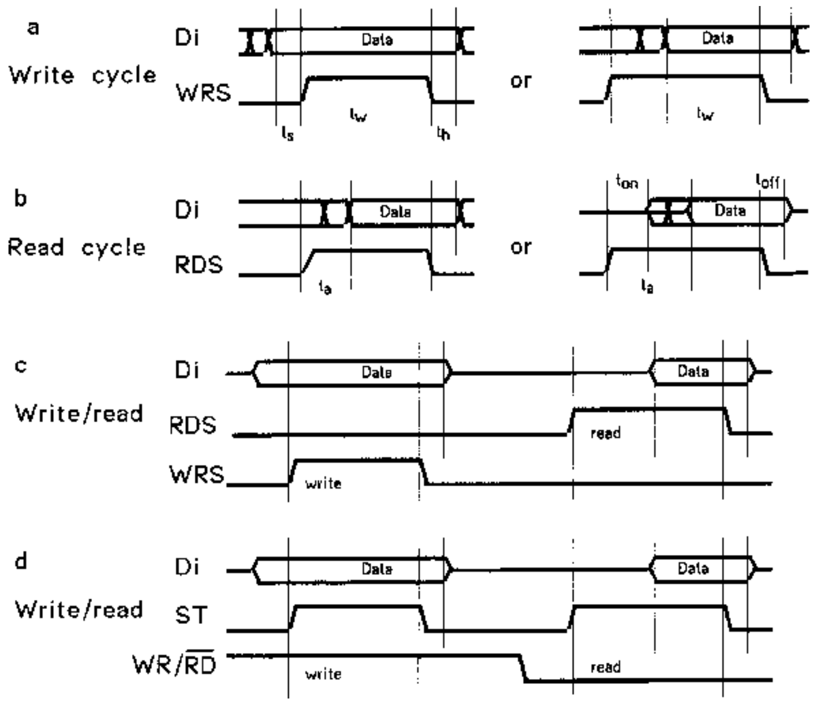


FIGURE 2 Simple cyles.

Geographic addressing is very different. Its purpose is to recognize which board is in a given slot of the bus in order to establish the memory map of the system and then correctly address the available resources. Plug and play (PnP) has been implemented on the old modified ISA bus by a tricky sequence supported by Windows 95. EISA and PCI support geographic addressing and make PnP software simpler.

These transfer protocols are known as asynchronous or handshaken, that is, bus signals are activated as soon as the function they express is valid. As a result, a mixture of fast and slow devices, using both old and new technologies, can share a bus; bus speed adapts automatically to the requirements of the devices.

Synchronous buses have a central clock oscillator that drives a bus signal line to distribute timing information throughout the system. Pure synchronous systems provide one clock period for every operation. Semisynchronous or clocked operations afford several clocks for an operation. Figure 6 shows a read operation using the clocked protocol used on the old Macintosh II Nubus and more recent PCI bus. The rising edge of the clock is the time when bus signals make their changes. Signals are assumed to be valid, that is, to have successfully propagated throughout the system, just before the next rising clock edge.

In a read cycle (Fig. 6a), a signal (Start) marks the presence of the address and control information on the multiplexed bus lines. When the slave recognizes its address and finds the requested data, it puts the data and status on the bus and marks their presence with an acknowledge (Ack) signal. A write operation is similar (Fig. 6b), except that data are supplied by the master at the next clock after the address and remain on the bus

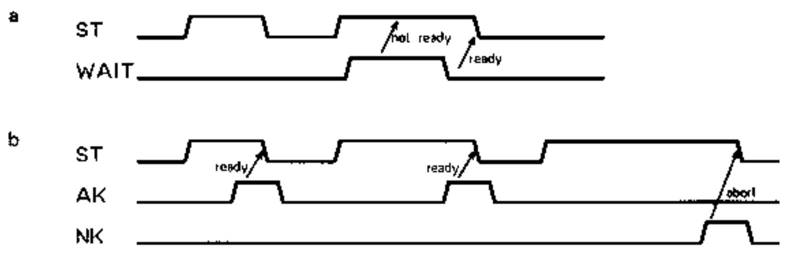


FIGURE 3 Asynchronous (handshaken) protocols.

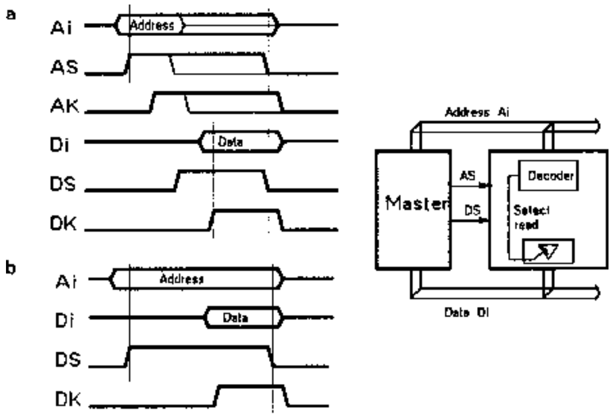


FIGURE 4 Addressed transfers.

until the acknowledge and status signals are sent by the slave.

The clock frequency and bus length must be chosen so as to allow enough time for signals to flow from any starting point to every other point well before the end of the clock cycle, allowing for differences in clock arrival time as well. Thus, shorter buses can be designed to run faster. One can notice that the bandwidth of the clock is twice that of one of the signals. In order to accelerate data transfers, both edges of the clock signal can be used. Most recent buses use this scheme, initially proposed by Rambus for its memory concept, which accepts only burst transfers on an 8-bit bus at 500 MHz.

D. Special Cycles

Frequently, consecutive memory locations must be transferred over the bus. Moving the address once and incrementing a counter on the memory are more efficient than sending the address each time, especially if the bus is multiplexed. Block transfers are supported by the most complex buses and are available with limitations on the most recent 32-bit microprocessors.

Reading from many devices simultaneously is called broadcall. This costs additional bus lines and transfer time and is not frequently implemented.

Combined cycles are possible. Read–modify–write cycles are interesting in multiprocessor systems. Read-after-write cycles are never implemented because one has confidence in what has been written.

Finally, split cycles are a way of having only write cycles in the system. In order to read some information, one writes a read request and waits. The slave interrupts the master to get its information read, and the process may continue. This is useful when the slave, for example, a disk interface, transfers blocks of data and is slow: It would be inefficient to wait for an acknowledge.

E. Slave Operation

The only initiative left to a slave is to interrupt the master and wait to be serviced. This is done by a dedicated line called IntReq; the wired-OR function is required to drive this line (Fig. 7). In response to the interrupt, the processor starts an identification procedure that can be software driven (polling), as with simple microprocessor systems, or can use some additional hardware.

Hardware identification of the source of interrupt can be accomplished by

1. The broadcall cycle: Requesting devices activate the data bus line(s) to which they have been attributed.
2. Proximity addressing: A daisy chain (Fig. 9) allows the requesting device that is the closest to the commander to put an identification word called a vector on the lines. This vector can be the address of the interrupting device or the address (direct or preferably indirect) of the routine that must be executed when servicing the interrupt.

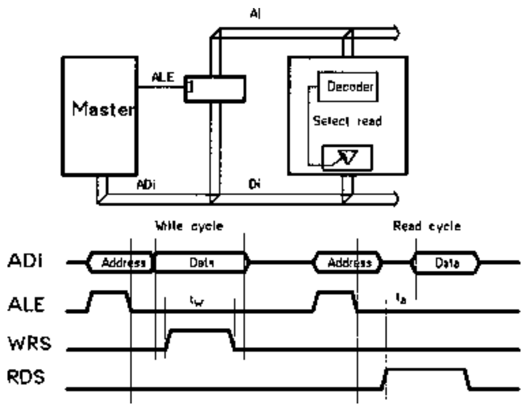


FIGURE 5 Multiplexed processor.

II. MULTIMASTER OPERATIONS

If several processors must access common resources, such as memory or I/O over the bus, they start by arbitrating in the case of a simultaneous request, since only one can use the resource at a time. A dedicated circuit, the arbiter, must be implemented and can be centralized or distributed.

A. Central and Distributed Arbiters

The simplest and most reliable way of designing an arbiter is to build a sequential clocked system that scans all the

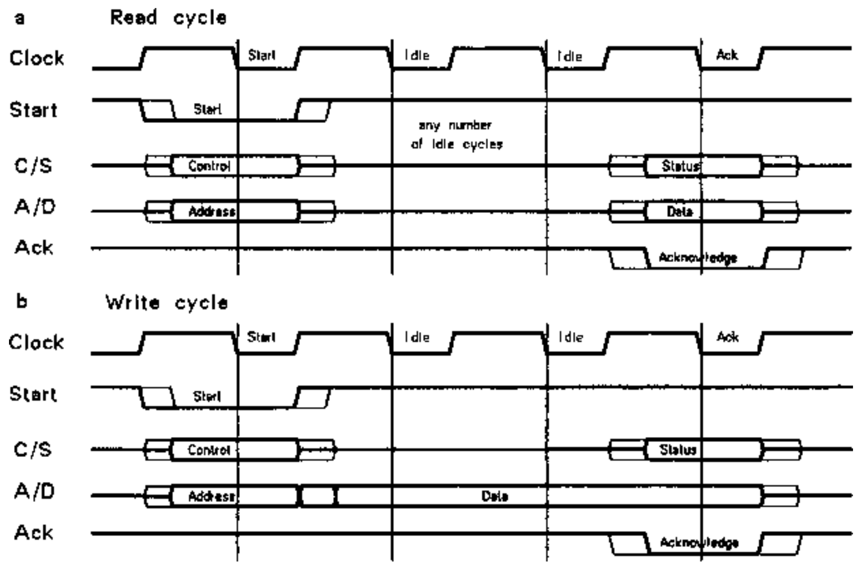


FIGURE 6 Clocked data transfer protocol.

masters in turn and gives the bus to the first one found having a request. This, however, is slow and therefore asynchronous arbitrations are attractive.

If there are only two contenders, the scheme of Fig. 8 gives priority to the first one. If they are absolutely simultaneous, the circuit may hesitate for some time (metastable state) before giving the priority to one or the other.

The arbitration process may be distributed along the bus; this provides a greater flexibility of design and usually a lower cost, but it is slower. The daisy chain (Fig. 9) is frequently used; it is a selection scheme that selects the first requesting device in geographical order depending on the wiring.

The would-be masters set their requests on the bus, using a wired-OR scheme. The active master, when finished, issues a grant pulse that is gated by each master on the bus. It is clear from the schematic in Fig. 9 that the first would-be master on the bus will get the pulse and will cut the chain. Some synchronization is required for correct operation. With most computer systems, there is often a permanent master at the beginning of the chain, and the

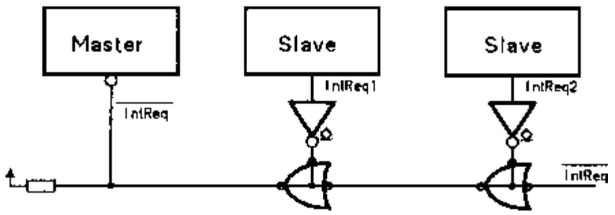


FIGURE 7 Interrupt bus.

control of the bus is passed back to it by each master when it is finished.

A more symmetrical scheme is the self-selection arbiter used in all modern buses (Futurebus P896, EISA, PCI). It makes use of a set of wired-OR priority lines and compares the values on the lines with the value that settles on the bus (Fig. 10). A careful design must take care of propagation delays.

The comparator is simpler if the priority vectors are not encoded, that is, if one line is assigned to each master, as on the SCSI bus. A common request line is activated each time an arbitration phase starts. As soon as that request has been recognized, no new priority vector is allowed.

A delay or a clock period signals when the comparison phase is over in each module; the winning master assigns a "bus busy" line during the data transfer.

B. Bus Transactions

An access from a processor to a memory via the system bus requires three different operations, performed in three consecutive transfer cycles: the bus allocation (or arbitration), the selection of the register (addressing), and finally one or more data transfers. These operations can be performed in sequence or overlapped, with different tradeoffs between speed and bus size (Fig. 11). For a given bus width, one can either allocate different paths to the different operations in order to allow concurrency or keep the maximum width with fully sequential operations; the current trend is toward maximum concurrency, which in multiprocessor systems is achieved by pipelining operations from different processors.

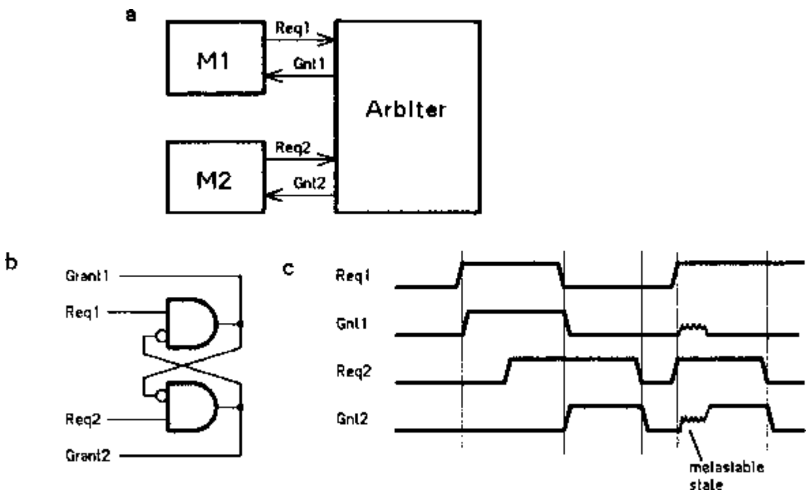


FIGURE 8 Central arbiter for two masters.

C. Multiprocessor Hierarchy

One can consider three general classes of multiprocessor systems:

1. A centralized multiprocessor, in which one processor runs the kernel and treats the other as slaves. This processor has access to the local memory spaces of the slaves.
2. A federalist multiprocessor, in which all processors own a copy of the kernel and communicate over a shared memory, using a shared memory space, but also have a private memory space.
3. An autonomous multiprocessor, in which the processors do not share a common address space but communicate by messages, as in a network.

Modern multiprocessor buses support the federalist type. They have the following:

1. A decentralized arbitration with a fairness strategy, guaranteeing that every module will receive access to the bus within a limited time.
2. A decentralized interrupt that allows every module to send an interrupt to all other modules or to a group of destinations.
3. A locking mechanism for semaphore operations.
4. An initialization line that allows a defined start-up of the system (reset-not-complete).
5. A broadcast mechanism that allows implementation of a replicated global memory and considerably speeds up the access to share data.

PCI (peripheral component interconnect) is a good example of such a bus. It is not as sophisticated as the Futurebus P896, but its adequate set of features has spread its usage on the PCs and workstations of the late 1990s. Transfer speed is up to 120 MBytes/s on a 32-bit wide

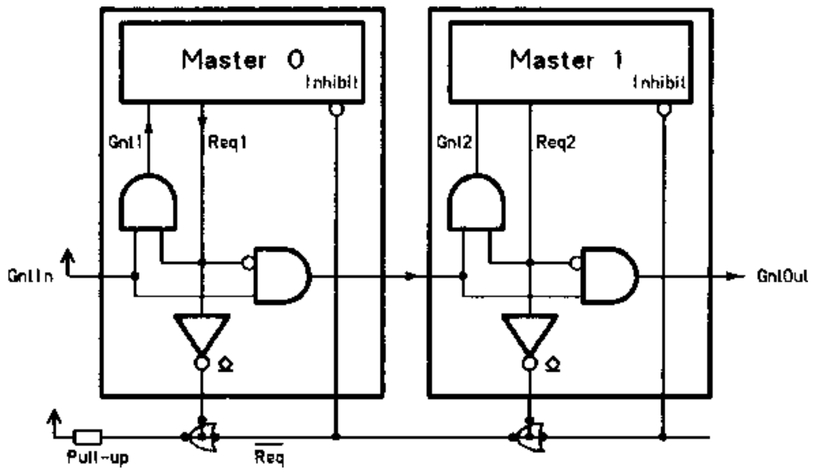


FIGURE 9 Daisy chain arbiter.

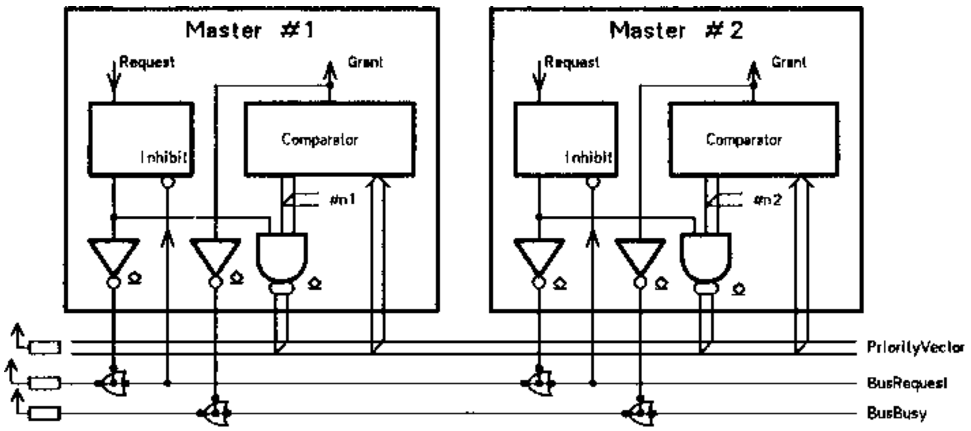


FIGURE 10 Self-selection arbitration scheme.

bus with a bus clock of 33 MHz. The 64-bit wide implementation double that speed. Some of the PCI features, like handshake, arbitration, and cache coherence, have the drawback to slow down the transfer cycle. Hence there is the need for a more direct link between the processor and the screen memory. The AGP (accelerated graphic port), the successor to VESA local bus, is a solution that may also last for several years only. The need for as high as possible bandwidth between the processor, the main memory, and the graphic screen memory and logic, may be better served in the future by a Rambus-like scheme.

D. Electromechanical Interface

Just putting the information on the bus is not so simple. Address and data drivers, which isolate and amplify the signals, are required. Driving a highspeed, well-adapted bus requires a lot of power because of the low impedance (below 100 Ω) of the transmission lines. TTL circuits have been replaced by special low-voltage swing drivers in order to reduce power dissipation. Trapezoidal waveforms reduce crosstalk between adjacent lines, which are implemented as twisted differential pairs and shielded when long serial lines are involved. Speeds of up to 1 Gigabit/s are then possible.

The bus usually looks like a printed circuit board covered with connectors. Most of the lines are parallel and run from one connector to the other. Adapting resistors exist at the line extremities for a correct line termination. The processor, memory, and I/O board plug into these connectors and are maintained by a chassis. Indirect connectors are preferred for industrial applications, but PCs still use direct connectors, which allows for a reduced board height for the same complexity. Indirect connectors have up to 5 rows of pins, suitable for wide buses (many data/address lines) and a dense array of ground lines in between. The card size depends on the bus objective; 32-bit buses need a lot of interface circuitry, and large boards are more efficient. Personal computer buses have narrow boards that are difficult to lay out (Fig. 12).

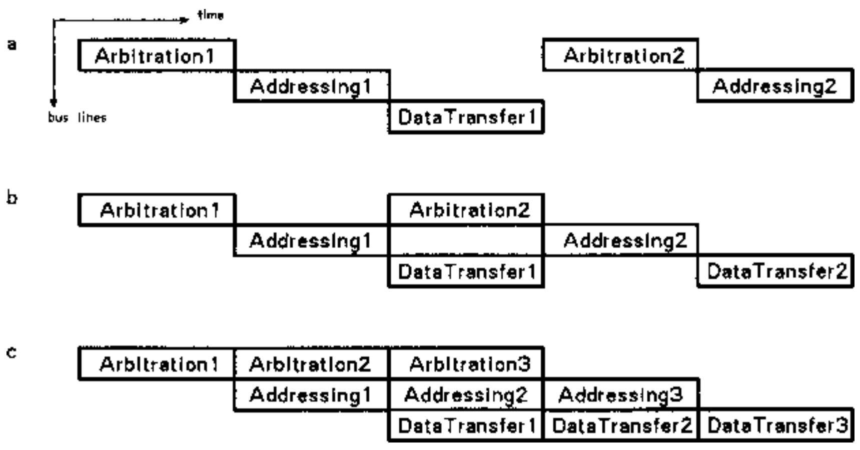


FIGURE 11 Concurrency on the infromation paths.

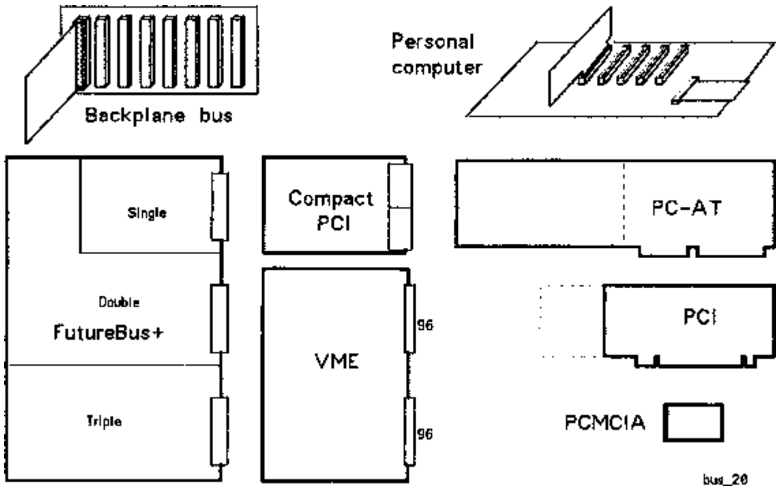


FIGURE 12 Typical board size for standard buses.

E. Software Aspects

A bused system can be configured in many different ways. For the old solution, the user had to painfully move jumpers on each card and set long lists of parameters in an initialization procedure. Now, the system should initialize itself; this implies some form of geographic addressing in order to recognize which card is in which slot, plus a parameter ROM and a programmable address decoder on each board.

In a multiprocessor system, the reconfiguration in the case of failure of a processor is a difficult problem. Communications must occur between the masters at a different level in order to distribute the load and check for the correct operation. This may be solved with a special control and status register space (CSR) or with a message-passing mechanism over the parallel bus or over a dedicated serial bus. Futurebus+ is the only bus to support the complete set of these sophisticated features.

III. STANDARD BUSES

A. Classification

It is not possible to explore in a few pages all the solutions used in the many existing buses and links that more or less follow the accepted standards. In Fig. 13, going from the simple point-to-point transfer (Centronics) to the federalist multiprocessor, many features have to be added; full addressing, multiprocessor features, plug and play (which implies geographic addressing), message passing (that is, no dependency or interrupt lines), and hot plugs are the major complexity steps. In Fig. 13, the more complex and more recently developed buses are at the bottom of the figure. Performance and complexity are required and better handled by the latest technology.

Such an evolution may not continue at the same speed: Electrical limits are reached and future hardware functions will not improve a lot. Using gate array, special VLSI circuits, and new connectors to make the bus interface more compact is the only way to lower the price and improve the bus performance. Optical transmission may add new possibilities, but it is not suitable for buses and does not yet seem ready to leave the laboratory. The evolution is not very fast, because of the need to set up multivendor standards, develop special VLSI circuits, and get some return on investment before the next generation.

Standardization means reducing options and developing wide acceptance. The dilemma of standardization is that a general-purpose solution is normally technically inferior to a specialized solution, so there is always a reason why one should not follow the standard and instead make one's own bus. The standardization of several designs is only justified if they differ from one another sufficiently.

The instrumentation bus GPIP, or IEEE488 or IEC621, was initially developed by Hewlett-Packard. A commander assigns one talker and one or several listeners; the broadcast protocol allows, for example, the sending of information from data acquisition equipment to both a computer and a recorder. Hundreds of instruments exist for the 8-bit bus, transferring data as fast as 1 Mbyte/s.

The SCSI (small computer system interface) bus was designed to link one or several disks to a microcomputer system. Other devices, such as printers or scanners, can be connected as well. Integrated circuits exist to make the microprocessor interface very easy. A single 8-bit

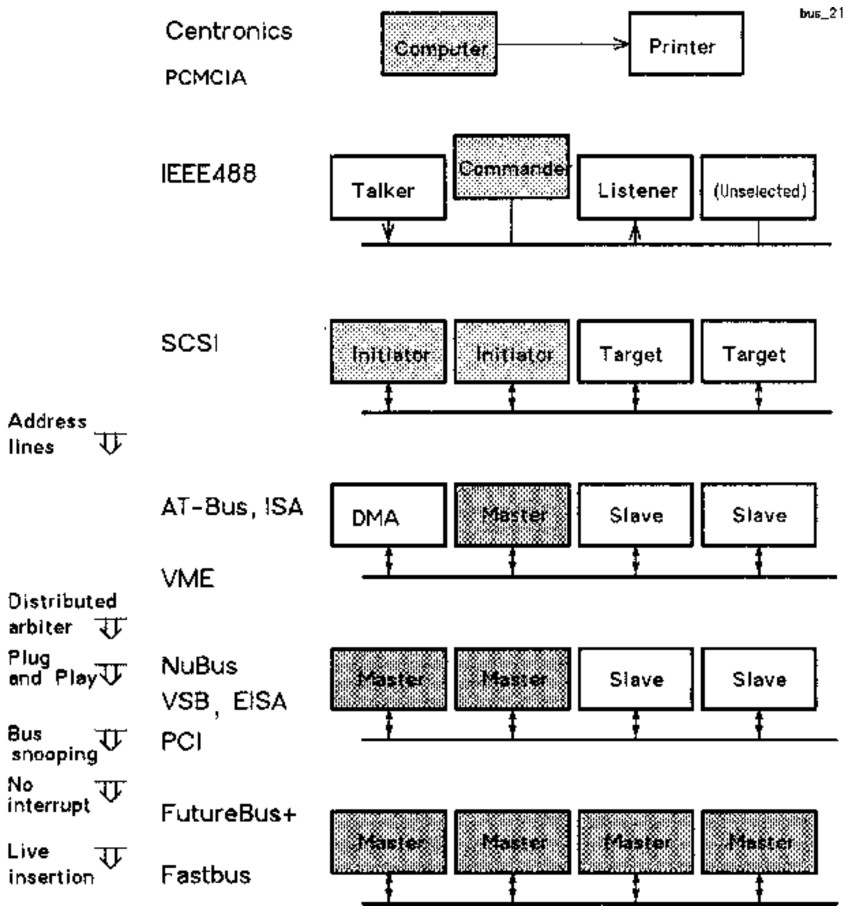


FIGURE 13 Bus classification.

information path is used for arbitration, addressing, and data transfer.

B. Parallel Links and Buses

A parallel link connects the master processor to an I/O or memory slave via a set of parallel lines. Centronics has been used for years for the connection of printers and scanners, together with the serial RS232 explained in the next section. Both should disappear soon. PCMCIA has been defined for portable PCs and uses a 64-pin connector to connect a memory card or a modem, but also accepts disks and fast video devices. New standards will be defined for the coming smaller disks and flash memories, to be used mostly for picture cameras and wearable computers.

A bus allows connection of several units in a flexible way. At every instant, a master communicates with usually one slave at a time. A supervisor is required in multimaster systems to avoid conflicts, also possible at a higher level than bus arbitration.

Microprocessor buses have, as previously explained, a set of lines for address, data, and control. Many microprocessor buses have been defined and standardized for 8-, 16-, 32-, and now 64-bit processors. New features have been added to support multiprocessor, PnP, and hot plug. Bus speed cannot progress as fast as processor speed, and universal buses are no longer used for main memory, but only for I/O and satellite processors.

The IBM-PC bus was not designed as a bus, but as a means of connecting cards in the PC. The bus has been extended to 16 bits on the PC-AT and has been frequently used in industrial applications. It is a monoprocessor bus with few dedicated interrupt lines and two DMA channels available, now standardized under the name ISA (Industry Standard Architecture). The 32-bit bus extension EISA accepts ISA boards.

The VMEbus (IEC821, IEEE1014) is broadly used for industrial applications in its 32-bit extension. The standard includes an auxiliary parallel bus named VSB, the possibility to use 32-bit processors, and interprocessor communication primitives, and it reserves two pins for a serial bus named VMSbus (IEC823, IEEE1096). Three compliance levels are defined, with different widths for the address and data fields. The extended level uses 32-bit data and address, but requires a second connector.

C. Serial Links and Buses

Data can be transferred one bit at a time, and adequate protocols permits to distinguish control, address, and data information. It is easy to convert data from parallel to serial with a shift register. The clock information may be transmitted together with the data on two wires or optical fibers, or on a single one if an adequate modulation is used. Figure 14 surveys several transfer schemes:

(a) Data and clock, directly compatible with shift registers
(b) Data and sync, as used on Firewire
(c) Data and clock, as used on 12C/SMBus
(d) Serial data, as used on USB (NRZ "non return do zero" encoding)
(e) Serial data, as used on RS232 (start and stop bits)

Serial links and buses can also be classified according to their hardware and software functionality. RS232 was a mess of variants until all machines became PC compatible. RS232 is indeed an improper designation, but it does not matter when there is no confusion. After too many years of service, it will disappear soon.

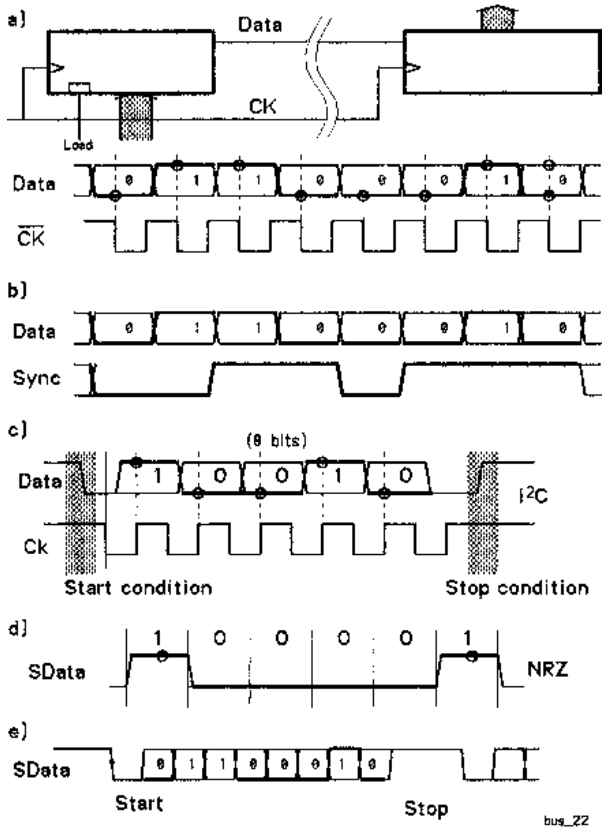


FIGURE 14 Serial transfer schemes.

EISA, defined as an extension of the ISA bus of the second generation PCs, is replaced by PCI, which offers more performance and functionality. Bus snooping is required for maintaining coherence among the many caches on a multiprocessor system. The next step in a multiprocessor system is to suppress the interrupts and use an adequate message passing system. Live insertion (hot plugs) requires special connectors. These features are available on the highest performance Futurebus+ and Fastbus, which are used mostly in industry and nuclear research.

Its successor is USB (universal serial bus), which provides speeds in the range of 1 to 100 megabit/s. All early 2000 PCs and Macs are equipped with 1 Megabit/s (for mice and keyboards) and 12 Megabit/s (for audio and slow video). The faster USB version, at 100 Mb/s, competes with the Firewire 1394 standard, proposed and used for higher speed up to 400 Megabit/s. Firewire is a multimaster bus on special six-wire cables: two pairs for data and clock (Fig. 13b) and one pair for power. Firewire is well suited for interconnecting disks, video cameras, and set-top boxes, but the interface components are more complex than those of the USB, and the market is smaller; the resulting higher price may limit the solution to professional equipment.

USB is not a bus, but a set of addressable PnP links with a single master. It uses four-line cables, two lines for the NRZ data on a differential pair (Fig. 13d), and two lines for the power supply. Providing the power to the peripherals is a great advance, but implies some power management scheme. Current is limited to 100 mA when the device is connected and the PnP sequence is executed. More power can be attributed by the PC if the device needs it, and if no other higher priority device is already using the available power. Hence, if the PC provides 2 amps, about 10 USB peripherals can be connected, some of them using up to 500 mA of currant, but not all the high-power devices will be allowed to work together. Cables are short and active hubs, including a processor, are used to connect additional peripherals; they recognize whether a plug is connected or not, and they include power management circuits and forward the data to the selected peripheral, according to an address defined at configuration.

The USB protocol transfers a set of packets (a frame) of up to 1024 bytes of data every millisecond. Some packets, said to be isochronous, have a fixed length and are guaranteed to be repeated for every frame. For instance, a voice channel will use 64 bits. If there is a transmission error, there is no repetition. Bulk transfers uses the remaining bandwidth. If there is a transmission error, the erroneous packet will be repeated. This is required for data transfer, e.g., from a disk.

A much simpler bus also exists inside PCs. The SM-bus interconnects temperature sensors and switches with the processor, over two lines (Fig. 13c) in order to save money since speed is not a requirement. SM-Bus is indeed almost identical to the I^2C bus proposed by Philips to interconnect the controllers of audio and video consumer units. A good set of interface circuits is available (calendar clocks, nonvolatile memories, analog/digital converters, sensors). Transfer speed is up to 100 kilobit/s and 127 units can be addressed. Multimaster configurations are possible, and timing requirements are easy to match.

Other buses are used on industrial machine tools and inside cars, compatible with longer lines, that is, they use coaxial cables, twisted pairs, and, for the longer distances, telephone lines. Can-Bus, Profi-Bus, and Lon are solutions that will continue to compete. The 220- or 110-V lines can be modulated at 1 kHz and used for home appliance control, with extremely low transfer speed (10 bits/s).

D. Comparison

Several authors have succeeded in comparing buses by listing the main features on a single page. Such charts are a help when eliminating buses. Selecting one is difficult and should depend on the following:

1. Technical criteria: Is the design adequate for the application (bus size, speed, features)? Will it not be obsolete soon?
2. Usage criteria: How difficult is it to configure? Do adequate software primitives, operating systems, and drivers exist for each board?
3. Commercial criteria: Are the companies and distributors reliable enough? What are the price and delivery delays?
4. Production criteria (if own production): Are the components available? Is it worth designing and producing?

Selecting a bus forces the selection of a processor, an operating system, and a group of vendors. It is greatly influenced by nontechnical aspects, such as a friend's system, company policy, and vendors' proximity.

IV. FUTURE TRENDS

Parallel buses such as Futurebus+ have a small market compared to the PC board market. Interconnection of PCs with fast links is the modern trend to get more power and more reliability (uninterruptible systems). Inside the PC, the need for a bus for interconnecting the essential peripherals has disappeared since the serial, parallel, and disk ports are available on the processor itself. With USB and Firewire, the only other need for an extension bus is for memory access and ultrafast transactions with the graphic board. New solutions will have to appear over the next 10 years. They will be based on very fast synchronous block transfers, large caches being available on every device. The width of the bus will determine the cost of the solution; 128-bit buses will be proposed, but the huge market will be for bit and byte serial buses.

VLSI and optical transmission will certainly provide novel approaches. Once a solution has been proposed by a large manufacturer, more than 5 years will elapse before an adequate commercial base has been built.

SEE ALSO THE FOLLOWING ARTICLES

COMPUTER ARCHITECTURE • MICROCOMPUTER DESIGN

BIBLIOGRAPHY

Del Corso, D., Kirrmann, H., and Nicoud, J. D. (1986). "Microcomputer Buses and Links," Academic Press, London.

Finkelstein, E., and Weiss, S. (1999). Microprocessor system buses: A case study. *Systems Architecture* **45,** 1151–1168.

Folts, H. C. (1982). "McGraw-Hill Compilation of Data Standards," McGraw-Hill, New York.

Gustavson, D. B. (1984). Computer buses—a tutorial. *IEEE MICRO* **4**(4), 7–22.

Stallings, W. (1996). "Computer Organization and Architecture," Prentice Hall.

Parallel Computing

David W. Walker
Cardiff University

I. Scientific Motivation
II. Technical Motivation
III. Nature's Parallel Computers
IV. Domain Decomposition
V. Parallel Computers
VI. Parallel Programming Paradigms
VII. Future Prospects

GLOSSARY

Amdahl's law The maximum concurrent speedup for a concurrent algorithm is limited by its sequential components. Thus, if the sequential component takes a fraction f_{seq} of the total run time on one node of a concurrent processor, the maximum possible speedup is $1/f_{\text{seq}}$.

Architecture The design of the hardware components of the computer system and the ways in which these components interact to produce the complete machine. The architecture of a concurrent computer includes both details of the nodes and the topology with which they are interconnected.

Artificial intelligence A class of problems such as pattern recognition, decision making, and learning in which humans are clearly very proficient compared with current computers. The definition is imprecise and time dependent; some include computer algebra, computer vision, and game playing in this class.

Asynchronous (MIMD) An architecture where each node of the parallel computer operates independently with different instruction streams (also "loosely synchronous").

Cluster computing A type of parallel computing in which a collection of interconnected standalone computers (typically PCs or workstations) work together as a single, integrated, computing resource.

Concurrent computer A collection of individual von Neumann, or sequential, nodes joined together to work on the same problem. We use "concurrent" and "parallel" as synonymous terms in this article. Sometimes concurrent is used to denote asynchronous and parallel to denote synchronous or lock-step modes. However, this distinction is not generally accepted.

Distributed memory The memory of a concurrent processor, split into segments, each of which may be directly accessed by only one node.

Grid computing Computing which uses distributed hardware and software infrastructure to provide reliable, pervasive, and inexpensive access to computational resources irrespective of their physical location or access point. This infrastructure is collectively referred to as "The Grid."

Megaflop (Mflop) A megaflop is a rate of computing of 10^6 floating-point operations per second, and is commonly used in measuring the performance of a computer. This performance measure depends intricately on the application. A gigaflop (Gflop) is a unit of 1000 times a Mflop, and a teraflop (Tflop) is a unit of 1 million times a Mflop.

Microprocessor A single chip with an area of approximately 4 cm^2 containing the main functions of a computer, including its CPU. In 2001 a microprocessor can have up to 2×10^8 transistors.

Sequential computer Synonymous with a von Neumann computer; a "conventional" computer in which only one node works on a given problem at a time.

Shared memory A memory that is directly accessed by more than one node of a concurrent processor. This is an important architectural feature of many supercomputer designs.

Supercomputer The most powerful available computers; the typical cost of a single supercomputer is over $20M.

Symmetric multiprocessor A parallel computer with multiple, similar, interconnected processors controlled by a single operating system, and with each processor having equal access to all I/O devices.

Synchronous (SIMD) An architecture where each node of the parallel computer performs identical instructions in lock step.

ADVANCES IN TECHNOLOGY have led to dramatic decreases in size and cost of basic computer components. These are the building blocks of very powerful parallel machines designed so that very many (10^3–10^6) individual computers can work together to solve a given problem. This approach has striking analogies with nature's and society's approach to problem solving. Use of parallel supercomputers is revolutionizing many fields of science and, indeed, our way of life.

I. SCIENTIFIC MOTIVATION

Computers have grown in performance, capability, and usefulness since their origins as mechanical machines in World War II. This growth is illustrated in Fig. 1. The early Los Alamos calculations already used parallelism with several human operators, each with a separate mechanical machine, working in a pipeline on nuclear simulations. In this article, we will describe a more sophisticated and entirely electronic form of parallelism with many individual computers linked together to simulate a single problem. We will not distinguish between parallel and concurrent computers, although some experts choose to do so. We will use the term sequential to describe a single individual computer. A parallel computer consists of many sequential processors or nodes. Until the mid-1980s, the evolution of computers was dominated by the development of sequential computers of ever-increasing power. An early machine in Fig. 1 capable of little more than one floating-point (scientific) calculation each second (one "flop" in the jargon) has become a household microprocessor which, in 2001, can be a factor of 5×10^8 times more powerful (the 1.5-GHz Intel Pentium IV processor).

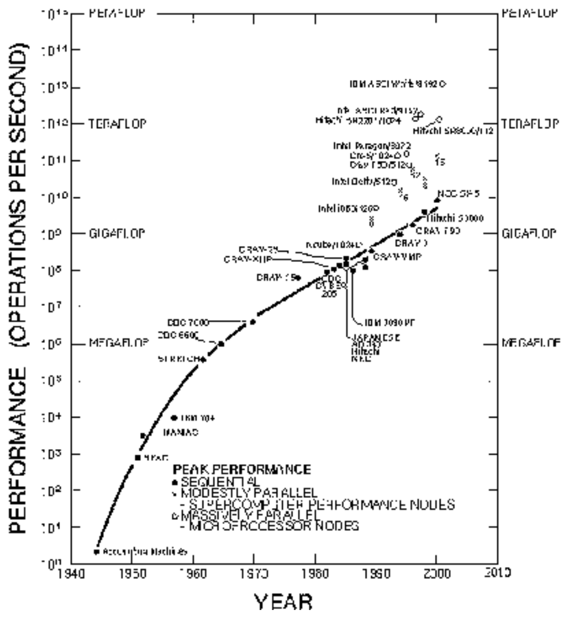


FIGURE 1 The performance of sequential and parallel computers between the years 1940 and 2000. The small crosses (×) give the performance of multiprocessor versions of the single-processor machines below them marked by a filled circle (●). The number next to a cross gives the number of processors. For massively parallel machines (○) the number of processors is given after the forward slash (/) following the name.

Before continuing this story, let us note that parallel computing has been driven by a happy confluence of two forces. Not only is it technically possible to build more-powerful computers, but also there are many computational fields that are hungry for huge increases in computer performance to be able to complete realistic calculations. Current and potential uses of supercomputers are illustrated in Table I. The supercomputer uses outlined in Table I are both academically and usually commercially interesting. Other major fields of interest to industry and the military are shown in Table II.

II. TECHNICAL MOTIVATION

The driving force for the increase in computer performance is the improvement in design and fabrication of

TABLE I Current and Potential Academic and Commercial Uses of Supercomputers

Major field	Application
Aerodynamics	Design of new aircraft
Astrophysics	Stellar structure, evolution of galaxies, stellar and black hole dynamics
Biology	Mapping genome, protein folding, rational drug design
Computer science	Simulation of chips and circuits
Chemistry	Prediction of reaction and scattering rates, simulation of reactive flows
Engineering	Structural analysis, combustion simulations
Geology and earth science	Seismic exploration, modeling earth's convection
High energy physics	QCD computations
Material science	Simulation of new materials
Meterology	Accurate weather prediction, severe storm forecasting
Nuclear physics	Weapon simulation
Plamsa physics	Fusion reactor simulation
Business	Portfolio management, derivative pricing, transaction analysis
Motion picture production	Animation, special effects and graphics

computer chips. The fabrication improvement leads to decreasing feature size of the basic components on a chip. Currently, this feature size is between 100–200 nm, and we can expect a further decrease in this linear dimension to about 70 nm before one enters new regimes. At smaller feature sizes, issues such as the limited number of electrons (and corresponding large fluctuations) involved in transitions could become important by leading to nonreproducible behavior.

This is illustrated in Table III, which shows that the state-of-the-art microprocessor chip in 1980, the Intel 8086 with 50,000 transistors, increased in complexity by a factor of 10 in eight years to give the Intel 80386. Since then chips have increased in complexity by a further factor of 100, as in the Intel Pentium IV. Further dramatic increases in complexity are unlikely using conventional chip technologies. The rapid increase in complexity seen to date can be quantified in terms of the fraction by which

TABLE II Supercomputer Uses of Interest to Industry and the Military

Robotics	Develop a machine with human capabilities
Space	Real-time control of sensors, processing of satellite data
Signal processing	Analysis of data and feature extraction
Defense	Tactical and strategic control of nation's defense, aiming and avoiding missiles, cracking codes, analysis of satellite surveillance data

TABLE III Change in number of Transistors in State-of-the-Art Microprocessor Chips 1980–2005

	1980	1988	1989	2001	2005
Transistors per complex chip	5×10^4	5×10^5	10^6	5×10^7	2×10^8
Examples Intel	8086	80,386	i860	P IV	
Motorola	68,000	68,030	88,000	G4	

technology improvements decrease the feature size, f. This leads to two sources of performance increase. First, the area of the system decreases by a factor of f^2. A factor of 10 decrease in f results in a 100-fold increase in chip density. Second, the decreased feature size also decreases switching times inside the chip and hence increases the clock speed approximately linearly. Thus, we are led to a cube, f^3, law by which the technology improvement translates into performance gains.

Over the past two decades, these very large scale integration (VLSI) improvements fueled the growth of the personal computer and workstation industry. Mainframe functionality has moved into microprocessor systems. This revolution has continued with the integration of computers into the home and the very fabric of society. This trend has also driven parallel computing. The chip designer has used the technical progress of the early 1990s to build a full-function computer on a chip. Subsequent progress has resulted not only in refinements to single-chip sequential computers—a recent trend has been to use the increased transistor budget for a chip to replicate a sophisticated computer design within a single chip. Thus, IBM's proposed Blue Gene computer will feature chips containing an array of processor–memory pairs—each essentially a separate computer. These chips will then be wired into a 3D mesh to create a massively parallel computing system. This contrasts quite strikingly with the 64 Ncube-1 nodes shown in Fig. 2. This is part of one of the earliest commercial parallel computers from circa 1987. We see replication of an individual computer consisting of seven chips—one integrated microprocessor (enhanced with communication channels to support parallelism) and six memory chips. This large printed-circuit board (about 2 ft^2) may soon be shrunk onto a single parallel computer chip.

Parallel computers have revolutionized supercomputing and dominate the area to the extent that all current supercomputers are parallel computers. The list of the world's 500 most powerful computers (as of November 2000) shows that the 5 fastest computers have over 5000 processors each. We have to go to position 24 in the list before finding a supercomputer with fewer than 100 processors. Supercomputers are currently exemplified by machines such as IBM's ASCI White system, which has 8192

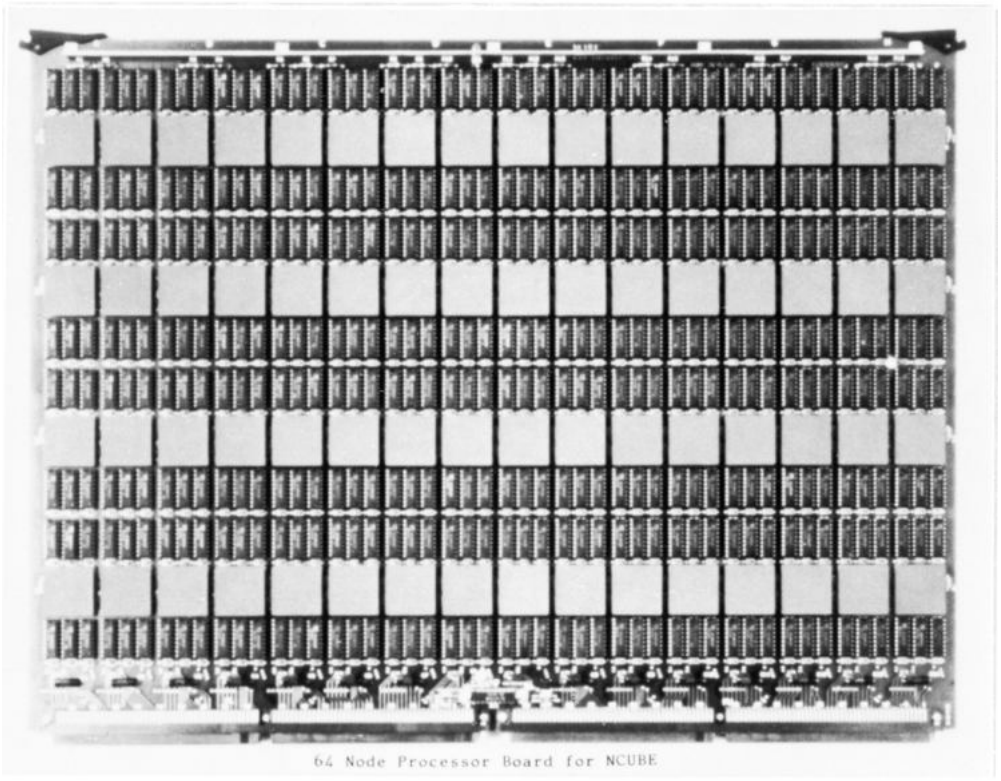


FIGURE 2 A single 64-node board of the Ncube-1 parallel supercomputer. Sixteen of these boards make up a complete system.

processors, over 6 Tbytes of memory, and a peak performance of over 12 Tflops.

The parallel computing economics of the last decade has converted increases in chip density to increased performance for machines of approximately fixed physical size and cost. This has led to a factor of 1000 or more increase in power in that period. Over the next decade, further increases in performance will come from exploiting very high levels of parallelism, with parallel systems containing 10^5–10^6 processors. The computer science challenge is to develop methods and software appropriate for such powerful machines; the computational science challenge is to use these computers to advance human understanding.

III. NATURE'S PARALLEL COMPUTERS

We can learn quite a bit about the use and design of parallel computers by studying parallelism in nature and society. In fact, one can view society or culture as a set of rules and conventions to allow people to work together, that is, in parallel, effectively and harmoniously.

A simple illustration is the way we tackle a large project, say, the construction of the space shuttle. It would be attractive to solve this sequentially by hiring a single superman to complete the project. This is prohibited by current physical phenomenology; instead one puts together a team, maybe in this case involving 100,000 "ordinary" people. These people work in parallel to complete the shuttle. A parallel computer is quite similar: we might use 10^5 digital computers working together to simulate airflow over a new shuttle design. The key in NASA's shuttle project is the management structure. This becomes, in the analogy, the issue of computer hardware and software architecture—a key research area in computer science.

We can illustrate these ideas by the building of a large wall, perhaps that built by Hadrian to protect the Roman empire in Britain. This is depicted in Fig. 3a, which shows $N = 8$ masons used to build this wall. We assume each mason works at the same rate, an idealization for people, but valid for the computer analogy. We build the wall quickest by dividing the wall into eight pieces by vertical sections of length l. This is called *domain decomposition* and is the key idea in parallel computing. At the heart of a computation is an algorithm or (mathematical) method applied to a dataset. The dataset could be the earth's atmosphere and the algorithms the physics governing its evolution: alternatively, the dataset could be the database records in a bank and the algorithm the update and querying of this database. These are illustrated in Fig. 4. Parallelism is achieved by

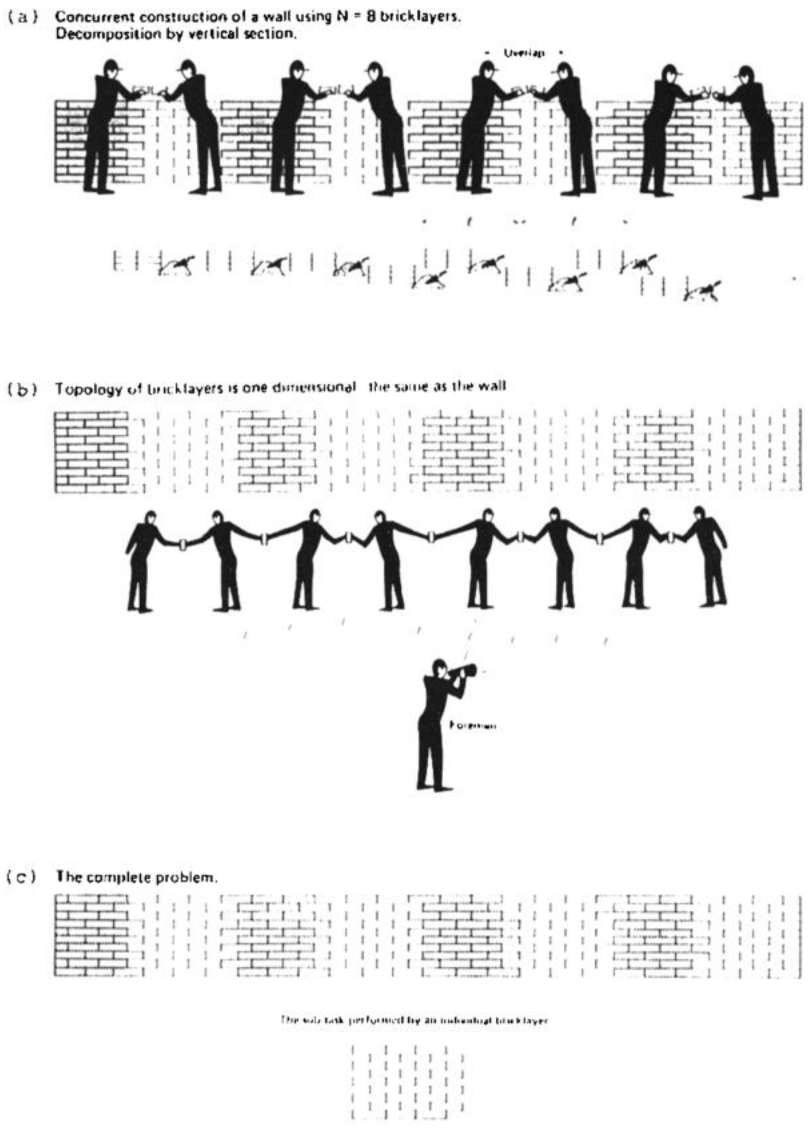

(a) Concurrent construction of a wall using N = 8 bricklayers. Decomposition by vertical section.
Overlap
(b) Topology of bricklayers is one dimensional the same as the wall
Foreman
(c) The complete problem.
The sub task performed by an individual bricklayer

FIGURE 3 The bricklaying problem as an analogy to parallel computing: (a) Decomposition, (b) topology, (c) comparison of the complete problem and the task performed by a given mason.

dividing a dataset into parts, one part for each computer (person), and acting in parallel on each part. Now this is not trivial, as seen in Fig. 3a: each mason can usually lay bricks quite independently, but there is a complication at the boundaries where the individual parts join. Here, coordination is needed between the masons. We can analyze this quantitatively in terms of the speedup S produced by the N masons:

$$S(N) = \frac{\text{Time to solve problem with 1 mason (computer)}}{\text{Time to solve with } N \text{ masons}}. \tag{1}$$

We often use the efficiency ε defined by

$$S(N) = \varepsilon N. \tag{2}$$

In this case, we can measure the overheads by an overlap extent l_{overlap} (approximately 1 m) over which horizontal distance adjacent masons must coordinate their work at the join between the different regions assigned to them. We can estimate the efficiency as

$$1/\varepsilon - 1 \sim \text{const} \cdot l_{\text{overlap}}/l, \tag{3a}$$

or, if L is the total length of the wall,

$$1/\varepsilon - 1 \sim \text{const} \cdot (l_{\text{overlap}}/L)N. \tag{3b}$$

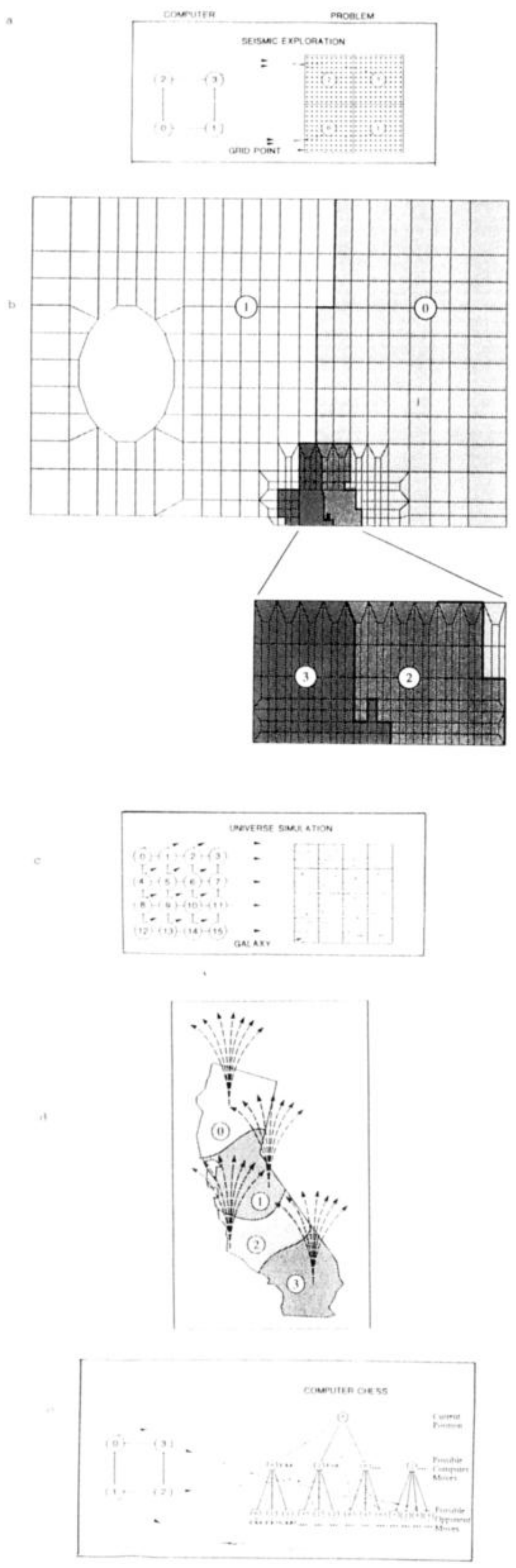


FIGURE 4 Five decompositions discussed in the text.

We wish to keep $S(N) \sim N$, so efficiency $\varepsilon \sim 1$: in other words, the right-hand sides of Eqs. (3a) and (3b) are small. In Fig. 5, we plot the speedup $S(N)$ for some reasonable choices, $\text{const} \cdot l_{\text{overlap}} = 10$ m and $L = 10^5$ m. We see that the efficiency is near one, $S(N) \sim N$, if N is less than about 1000, but as the number of masons increases past this value, the speedup levels off at 10^4. This is well known in society as “too many cooks spoil the broth” and is dignified as “Amdahl’s law” in the computer science literature. In Fig. 6, we show the measured speedups for a sophisticated astrophysical particle-dynamics simulation. Similar effects are observed with N_p (number of particles) playing the role that L did in Fig. 5. We see that large problems can use massively parallel computers (large labor forces): that is, superproblems are for supercomputers, small problems are suitable for sequential computers (small labor forces). The main criterion for the applicability of an application to parallel processing is not the availability of parallelism (which is almost always present), but whether the problem is large enough.

It is helpful to use our analogy to society to contrast parallel and distributed computing. Society used large teams to build large walls in Scotland, China, and Berlin; however, we also use many small teams or individuals to build the many small walls needed. This uses similar parallelism (individuals are laying bricks at the same time), but much less coordination: each individual wall can be constructed independently. Probably, many more bricks are laid in this independent fashion than in superprojects. Similarly, supercomputing and parallel processing is, and probably will remain, a small fraction of the computer industry. The network of individual computers, each solving small problems, sums to perhaps two orders of magnitude more computing resource than that represented by supercomputers. However, parallel supercomputers are important due to the possibility that solution of major problems, such as those noted in Section I, will lead to qualitative new advances in knowledge.

We now complete some details of the analogy developed earlier in this section. In Fig. 3b, we idealize the organization of the masons. The one-dimensional wall is built by masons who are themselves arranged in a line. In a computer science language, we say that the parallel computer should have a topology similar to that of the problem. Masons, like many computer designs, can be arranged in a flexible way to match the problem in hand. Appropriate topologies are a major issue in computer science research. Finally, in Fig. 3c, we show that a mason working as part of a big project uses similar basic techniques and, indeed, “solves a similar problem” to that involved as an individual building a complete (small) wall. In scientific computing terminology, we find that the basic numerical algorithm is unchanged, while boundary values and geometry are

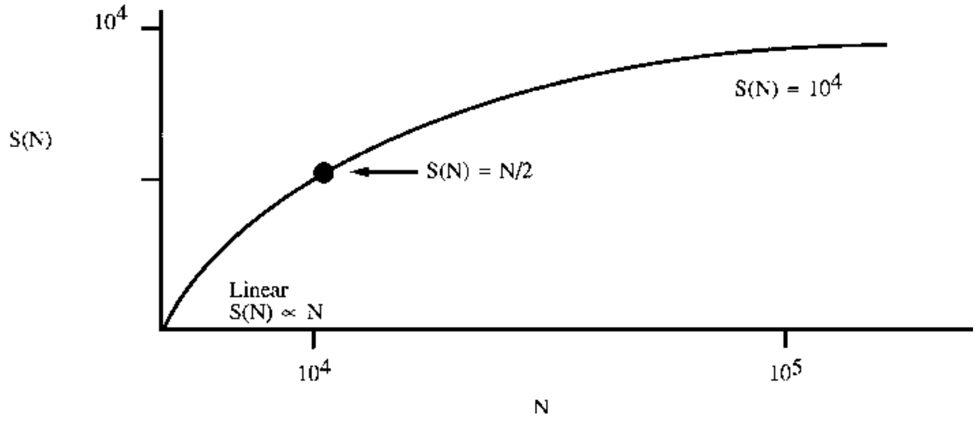


FIGURE 5 Speedup of the bricklaying problem as a function of *N*, the number of masons employed.

altered between sequential and parallel computing. This implies that, for the parallel computer, one can use similar software and algorithms as for the sequential case. Indeed, one can develop software that will run on either parallel or sequential computers; often over 90% of the code is identical for these two cases.

One can look at society at many different scales: one finds parallel computers from neurons in the brain, to bees in a swarm, to people in society. In every case, the underlying methodology is that of asynchronous computing elements (neurons, bees, people) communicating by messages. This so-called message-passing MIMD computer

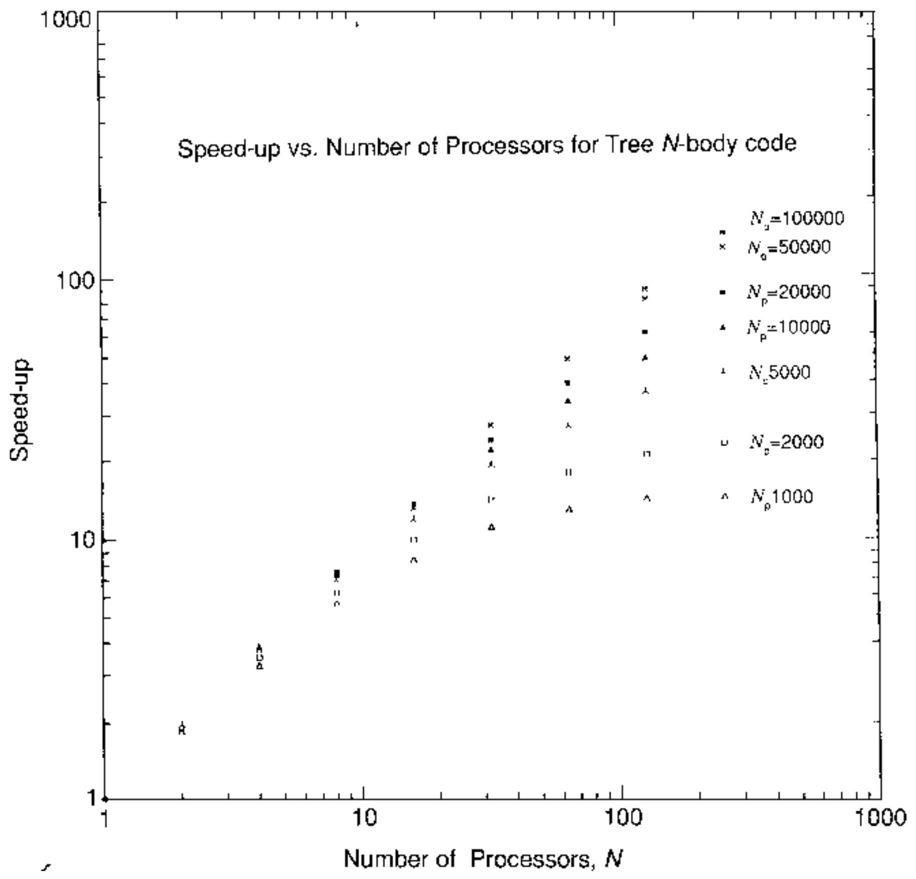


FIGURE 6 Measured speedups of an astrophysical simulation with N_p particles on an Ncube-1 with up to 256 processors.

structure is, as we shall see in Section V, only one of the possibilities for digital machines.

IV. DOMAIN DECOMPOSITION

We have already illustrated, in detail with Fig. 3 and the discussion of Section III, the use of domain decomposition in a particular example. Although not the only possible realization of parallel computing, all successful applications utilizing more than a few nodes have employed domain decomposition. This is often called data parallelism, which better expresses the key idea: parallelism is obtained by acting simultaneously on different parts of the data. Several examples are shown in Fig. 4. Figure 4a shows a regular set of mesh points. This might come from, say, a finite-difference approach to the simulation of a seismic disturbance in the Los Angeles basin. Different sets or mesh points, that is, different regions of Los Angeles, are assigned to separate nodes of the parallel processor. Figure 4b shows a finite-element mesh from the simulation of the structural dynamics of a plate. This is a similar decomposition, but no longer does each processor simulate equal areas; rather, the nodes are concentrated where the action is. Figure 4c shows galaxies scattered throughout the universe; these are evolved in time with separate parts of the universe assigned to different nodes. Galaxies move from node to node as they cross region boundaries. In Fig. 4d, the data are missiles launched by a hypothetical enemy. They are tracked in parallel by assigning different missiles to different processors. Finally, Fig. 4e shows a very different example where the data are the complex pruned tree generated by a computer chess program. This time the data are generated by the computer itself, but again we divide the data up and process separate branches of the tree on different nodes.

Now we return to the example of Fig. 4a and analyze it in more detail. We take, as shown in Fig. 7, a problem with 256 grid points arranged as a 16 × 16 mesh. We decompose it onto 16 processors. We replace the wave equation of a seismic simulation by the similar iterative solution to Laplace's equation where potential $\phi(i, j)$ is to be determined at grid points (i, j). Iteration is not the best way to tackle this particular problem. However, more sophisticated iterative techniques are probably the best approach to large three-dimensional finite-difference or finite-element calculations. Thus, although the simple example in Fig. 7 is not "real" or interesting itself, it does illustrate important issues. The computational solution consists of a simple algorithm

$$\phi(i, j) = \tfrac{1}{4}[\phi(i+1, j) + \phi(i-1, j) + \phi(i, j+1) + \phi(i, j-1)] \qquad (4)$$

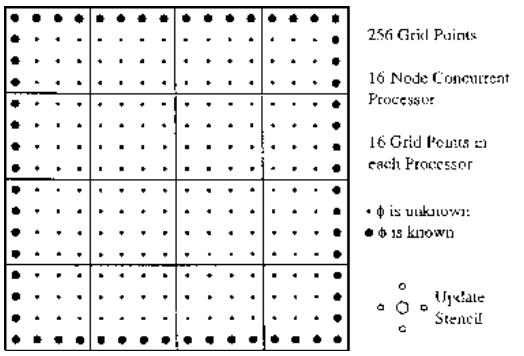

256 Grid Points
16 Node Concurrent Processor
16 Grid Points in each Processor
• ϕ is unknown
● ϕ is known
Update Stencil

FIGURE 7 A finite-difference grid decomposed among the nodes of a concurrent processor. Each processor is responsible for a 4 × 4 subgrid. Laplace's equation is to be solved for ϕ using a simple relaxation technique, Eq. (4). The five-point update stencil is also shown.

applied to each point of the dataset. As stated earlier, we get concurrency by applying Eq. (4) simultaneously in different parts of the underlying data domain. Thus, we leave the algorithm unchanged and executed sequentially within each node, and divide up the underlying dataset to get parallelism. One attractive feature of this method of domain decomposition is that it can be extended to very large machines. A 500 × 500 × 500 mesh with 1.25×10^8 points is not an atypical problem: nowadays, we divide this domain into up to thousands of parts. Clearly, such a problem can be divided into many more parts and can use future machines with very many nodes.

Returning to the "toy" example in Fig. 7, we associate a 4 × 4 subdomain with each processor. Let us examine what any one node is doing: this is illustrated in Fig. 8. We see that, in this case, an individual processor is solving the "same" problem (i.e., Laplace's equation with an

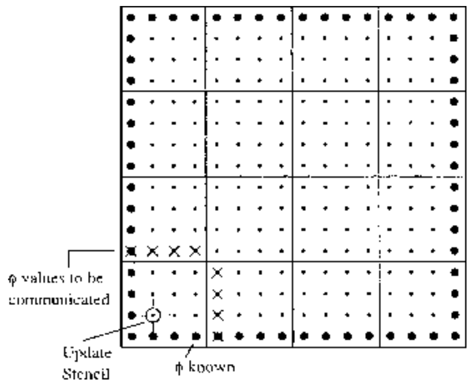


FIGURE 8 The role of processor 0 (in the bottom left corner). The update stencil, Eq. (4), is applied to each point at which ϕ is unknown. To update the grid points on the upper and right-hand boundaries, processor 0 must communicate with the processors above and to the right of it.

iterative algorithm) as a sequential computer. We have already seen two important differences in the bricklaying analogy of Section III and Fig. 3c.

1. The concurrent algorithm involves different geometry; the code should not address the full domain, but rather a subset of it.
2. The boundary conditions are changed. Referring to Fig. 8, one finds conventional boundary conditions: "ϕ is known" on the left and bottom edges of the square subdomain. However, on the top and right edges, one finds the following unusual constraint: "Please communicate with your neighboring nodes to update points on the edge."

Figure 7 and the novel boundary condition cited above make it clear that communication between nodes of the parallel computer is associated with the edge of the region stored in each node. We can quantify the effect of this on performance by introducing two parameters t_{calc} and t_{comm} to describe the hardware (see Fig. 9). Here we are specializing to the MIMD message-passing architecture described in Sections III and V.

- t_{calc}: The typical time required to perform a generic calculation. For scientific problems, this can be taken as a floating-point calculation $a = b * c$ or $a = b + c$.
- t_{comm}: The typical time taken to communicate a single word between two nodes connected in the hardware topology.

The times t_{calc} and t_{comm} are not precisely defined and depend on many parameters, such as length of message for t_{comm} and effectiveness of use of memory, cache, or registers for t_{calc}. The overhead due to communication depends on the ratio τ given by

$$\tau = t_{\text{comm}}/t_{\text{calc}} = 10\text{--}30, \tag{5}$$

where we have quoted typical values for current parallel computers. For problems similar to those in Fig. 9, we have

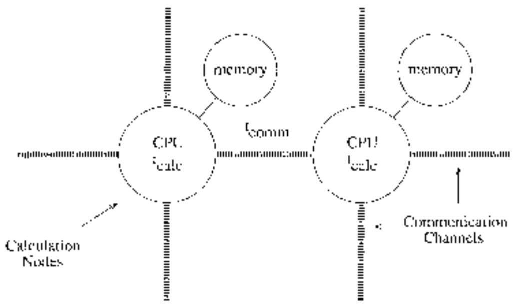


FIGURE 9 Nodes of a message passing parallel computer.

measured the performance of parallel computers. We can express the observed speedup S, defined in Eq. (1), as

$$S = \frac{N}{1 + f_C}, \tag{6}$$

where the problem runs S times faster than a single node on a parallel computer with N nodes. f_C is the fractional concurrent overhead, which, in this problem class, is due to communication. As the latter is an edge effect, one finds that

$$f_C = \frac{1}{\varepsilon} - 1 = \frac{0.5}{n^{1/2}}\left(\frac{t_{\text{comm}}}{t_{\text{calc}}}\right) = \frac{0.5\tau}{n^{1/2}}, \tag{7}$$

where one stores n grid points in each node: $n = 16$ in the example of Figs. 7 and 8; $4n^{1/2}$ is the ratio of edge to area in two dimensions. We see that f_C will be less than about 0.1 in this example for $\tau = 20$: in other words, the efficiency will be more than about 90% of optimal, as long as one stores at least 10,000 grid points in each node. This is an example of how one can quantify the importance of the problem being large, an issue central to Section II. On a machine with N nodes, the parallel computer performs well on two-dimensional problems with at least 10,000 grid points on each node, that is, on a problem with a total of at least 10,000N points.

We might have thought that one obtained small f_C only for local (nearest neighbor) problems such as that shown in Fig. 7. This is not true, as shown in Fig. 10, which demonstrates that f_C decreases as one increases the "range" of the algorithm. Indeed, long-range force problems are some of the lowest overhead ones for parallel machines. What counts is not the amount of communication (minimized by a local algorithm), but the ratio of communication to calculation.

In many problems, there are other degradations in the performance of current computers and, besides communication, load imbalance is often a significant issue. One needs to parcel out work to the nodes so that each has approximately the same amount of computation. This point is clear in the analogies with society. Load balance was trivially achieved in Fig. 7 by ensuring that each node processes an equal number of grid points. One can show that load balancing can be viewed as an optimization problem and that one can apply a variety of techniques, such as simulated annealing and neural networks. These methods are quite interesting as they involve a deep analogy between general problems and a physical system.

Load balancing is illustrated in Fig. 4b and better in Fig. 11, which illustrates the simulation of Mach 3 flow over a step on a 32-node parallel processor. The shock wave and structure near the step are modeled with an adaptive fine mesh shown in the top picture of the figure. We

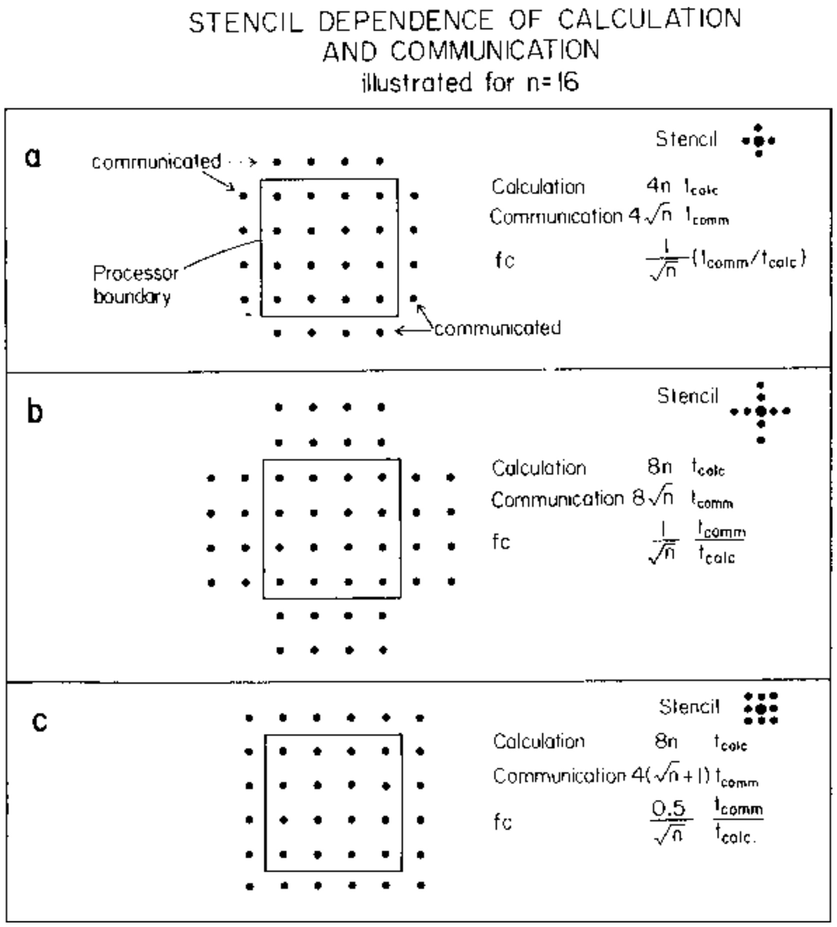

STENCIL DEPENDENCE OF CALCULATION AND COMMUNICATION
illustrated for n=16
a
communicated
Processor boundary
communicated
Stencil
Calculation 4n t_{calc}
Communication $4\sqrt{n}$ t_{comm}
fc $\frac{1}{\sqrt{n}}(t_{comm}/t_{calc})$
b
Stencil
Calculation 8n t_{calc}
Communication $8\sqrt{n}$ t_{comm}
fc $\frac{1}{\sqrt{n}}\frac{t_{comm}}{t_{calc}}$
c
Stencil
Calculation 8n t_{calc}
Communication $4(\sqrt{n}+1)$ t_{comm}
fc $\frac{0.5}{\sqrt{n}}\frac{t_{comm}}{t_{calc}}$

FIGURE 10 The effect of differing stencils on the communication overhead f_C. For the stencils in panels (a)–(c), f_C is proportional to $1/n^{1/2}$, where n is the number of grid points per processor. As the stencil size increases (d), f_C decreases until, when the stencil covers the full domain of the problem (e), f_C is proportional to $1/n$. This corresponds to a long-range force problem.

optimize the parallel computation by the irregular decomposition shown by the thick lines in this picture. This ensures approximately equal numbers of mesh points (elements) in each node of the parallel computer. Also note that the algorithm requires communication when physically adjacent mesh points are placed in different nodes. The decomposition of Fig. 11 minimizes this communication by assigning contiguous and roughly square physical regions to each node.

V. PARALLEL COMPUTERS

Traditionally, parallel computers are classified according to Flynn's taxonomy, which is based on whether:

- Each processor has a distinct instruction stream controling its execution, or each processor has the same instruction stream. If there is only one instruction stream, then each processor executes the same code. If each processor has its own instruction stream, then each can execute a different code.
- Each process receives data from a distinct data stream, or each processor receives the same data stream.

Each processor applies the instructions in its instruction stream to the data in its data stream. This leads to four possible types of parallel computer.

1. Single instruction stream, single data (SISD) stream. This corresponds to a sequential computer.
2. Single instruction stream, multiple data stream (SIMD). Each processor operates in lock step, with an identical instruction stream but different data streams.
3. Multiple instruction stream, single data stream (MISD). Each processor applies a different instruction stream (i.e., a different algorithm) to the same data stream.

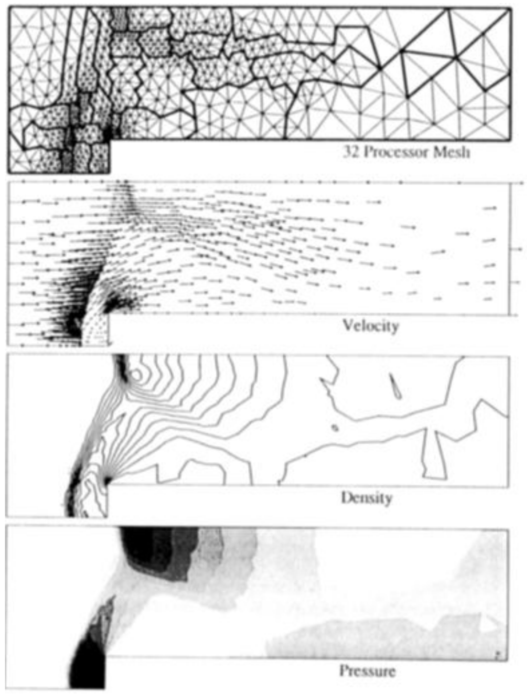


FIGURE 11 Simulation of a Mach 3 flow over a step showing fluid velocity, density, and pressure. This was done for a 32-processor parallel computer with dynamical load balancing shown by thick solid lines in the top diagram.

4. Multiple instruction stream, multiple data stream (MIMD). This is the most general case in which each processor has its own instruction stream applied to its own data stream.

Most high-performance parallel computers are now of the MIMD variety. Up to the early 1990s SIMD machines, such as the CM-2 manufactured by Thinking Machines Corporation, were used for scientific supercomputing but their use has now declined as they were found to be suitable only for *synchronous* problems. Roughly, synchronous means that the algorithm corresponds to the same numerical operation on every data element; this is typically true on geometrically regular scientific simulations.

MIMD parallel computers are usually divided into *shared* and *distributed* memory types. In the shared memory case, multiple processors are connected to memory by a switch so that any processor can access any memory location, and all processors have access to the same global name space. Thus, when each processor refers to the variable x they are all refering to the same location in the shared memory. The shared memory approach to parallelism is attractive because in porting an application from a sequential to a shared memory parallel computer usually only a few changes are required in the sequential source code, and these can often be performed with the help of a parallelizing compiler. The main drawback of the shared memory approach is that it is difficult to connect efficiently more than a few tens of processors to a single shared memory. Thus, shared memory parallel computers tend to have a have a relatively small number of powerful processors.

In the distributed memory case, each processor has its own local memory. This collection of independent computers communicates via messages, and is the clearest digital analog of nature's parallel computers discussed in Section III. Each processor–memory pairs forms the basis of a *node* of the parallel computer. Much computer science research has been done into the appropriate way to interconnect the nodes. Examples include the following:

1. A binary hypercube, where 2^p nodes are logically placed at the corners of a p-dimensional cube and connected along the edges. This was the topology used in the Caltech Mark I, II, and III parallel computers in the mid to late 1980s. The Ncube series of parallel computers, first introduced in 1986, also used a hypercube topology. A disadvantage of the hypercube topology is that it cannot be expanded in small increments, and the number of connecting wires increases rapidly with the hypercube dimension p. This topology is no longer widely used for parallel supercomputers.
2. A 2D or 3D mesh, where nodes are connected to their neighbors. For example, the Cray T3E-1200 consists of 1200 processors connected as a 3D bidirectional torus. Regular meshes form the basis of the interconnect topology of many current parallel supercomputers, as they can be expanded incrementally, and the number of connections per node is independent of the total number of nodes.
3. A switch to connect the nodes. The Butterfly series of shared memory parallel computers manufactured by BBN made use of a type of switch shown in Fig. 12 which connected each processor to every memory unit. Switch-type connections are currently used in symmetric multiprocessors (SMPs). An SMP is a parallel computer in which each processor has equal access to all I/O devices (including memory). SMPs form the basis of IBM's ASCI White computer, which is currently the fastest in the world. The ASCI White machine is made up of 512 SMP nodes, each with 16 processors, for a system total of 8192 processors.

The 1990s saw an increased use of commercial off-the-shelf (COTS) components for constructing low-cost parallel computers. This trend began with the use of networks of workstations (NOWs) for parallel computing.

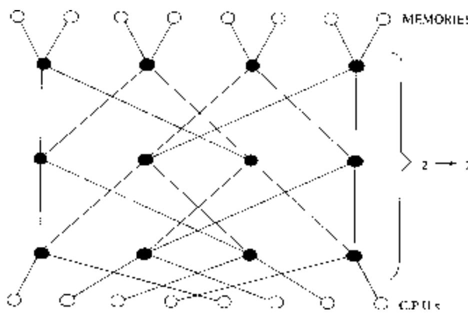


FIGURE 12 A shared-memory architecture illustrated for a shuffle-exchange switching network.

Here the typical approach is to use workstations that are connected via Ethernet on a company or departmental local area network (LAN) to perform parallel computations. NOWs provide a cheap source of computational power, using workstations and PCs that would otherwise be idle. This approach has branched out in two important directions. If a similar idea is applied on a wide-area network (WAN), then we are led into the realm of distributed computing. Because WANs have quite severe bandwidth and latency limitations this distributed approach can be applied only to applications with relatively little exchange of data between processors, i.e., they must be loosely coupled. *Grid computing* will make distributed computational resources available from any suitable access point regardless of their physical location. The "resources" include compute servers, information repositories, sensors, instruments, and people. The second direction has been the development of *cluster computing* in which a dedicated network, such as Myrinet, is used to connect the COTS components, which are usually rack-mounted. Beowulf clusters exemplify this approach, using only commercially available hardware and software and running a free operating system such as Linux or FreeBSD. These cluster computers are now capable of providing supercomputer performance at a fraction of the cost of a traditional commercial supercomputer.

VI. PARALLEL PROGRAMMING PARADIGMS

The processors of a machine executing a parallel application need to be able to share data. We saw this in our discussion in Section IV of the parallel solution of Laplace's equation—processors need to share data associated with the boundaries of their subdomains. There are two mains ways for processors to share data: through message passing or a global name space, and we now give an overview of these approaches.

A. Message Passing

In this approach, processors cooperate to perform a task by independently computing with their own local data, and communicating data between processors by explicitly exchanging messages. The message passing approach is particularly well suited to computers with physically distributed memory since there is a good match between the distributed memory model and the distributed hardware. However, message passing can be used on shared memory and sequential computers, and, indeed, can be used as a basis for the development of portable and efficient programs on all these architectures.

There are two main ways of writing a parallel message-passing program. The first is through a coordination language such as Fortran M or Occam. These are specialized languages for specifying concurrency, communication, and synchronization. The second way of performing message passing is with calls to a message passing library from within a sequential program. This has proved to be a very popular way of writing concurrent applications since it is expressive, closely models the functionality of the parallel hardware, and permits explicit management of the memory hierarchy. A number of different message passing libraries have been developed over the past decade, but the most widely used are PVM and MPI. The MPI message passing library is available on all parallel computers currently marketed.

B. Global Name Space

An alternative approach to data sharing is based on the shared memory model, in which all processes have access to the same global address space. As might be expected, this approach works well on shared memory architectures, but may also be supported in software on distributed memory computers. An example is the High Performance Fortran language, which provides data parallel language extensions to Fortran. OpenMP is a more recent development that supports shared memory parallel programming in Fortran and C/C++ through a set of compiler directives, library routines, and environment variables.

VII. FUTURE PROSPECTS

Parallel computing has advanced rapidly with the continuing stimulation of the rapid improvement of VLSI technology. However, there is limited scope for further improvements in microprocessor performance using conventional technologies. We expect future dramatic improvements in parallel performance in the short term to come from the use of very high levels of parallelism, involving up to 1 million processors. On a 5- to 15-year time scale, the use of superconducting processors with clock speeds in excess of 100 GHz and pulsed light microprocessors may lead to a new basis for supercomputer performance in the petaflop/sec (10^{15} floating-point operations per second) range. On a longer time scale, innovative approaches such as quantum computing and macromolecular computing may completely revolutionize the entire field of high-performance parallel computing, transforming all fields of scientific research and having a profound impact on society as a whole.

SEE ALSO THE FOLLOWING ARTICLES

ARTIFICIAL INTELLIGENCE • CLUSTER COMPUTING • COMPUTER ARCHITECTURE • SUPERCOMPUTERS

BIBLIOGRAPHY

Foster, I. T. (1995). "Designing and Building Parallel Programs," Addison-Wesley, Reading, MA.

Foster, I. T., and Kesselman, C. (eds.). (1998). "The Grid: Blueprint for a New computing Infrastructure," Morgan Kaufmann, San Francisco.

Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. (1988). "Solving Problems on Concurrent Processors," Vol. 1, Prentice-Hall, Englewood Cliffs, NJ.

Fox, G. C., Williams, R. D., and Messina, P. C. (1994). "Parallel Computing Works!" Morgan Kaufmann, San Francisco.

Pacheco, P. S. (1997). "Parallel Programming with MPI," Morgan Kaufmann, San Francisco.

Quinn, M. J. (1994). "Parallel Computing: Theory and Practice," McGraw-Hill, New York.

Zomaya, A. Y. H. (ed). (1996). "Parallel and Distributed Computing Handbook," McGraw-Hill, New York.

Minicomputers

Joe Grimes
California Polytechnic State University

I. History
II. Architecture, Organization, and Implementation
III. Applications
IV. Future Directions

GLOSSARY

Accumulator Usually a part of the arithmetic-logic unit of a computer that is used for intermediate storage.

Address Name identifying a location where data or instructions may be stored.

Assembler Language translator program that converts assembly language instructions into conventional machine language.

Cache Memory that is accessed more rapidly than main memory but less rapidly than a register. Today often found in processor chips with several levels of cache present in the computer.

Compiler Language translator program that converts application program language instruction, such as Pascal or FORTRAN, into a lower level language.

Computer architecture The study of the computer as viewed by the programmer.

Debugger Tool used by a programmer to locate and correct an error in a computer program.

Interrupt Special control signal(s) that diverts the flow of instruction execution to the instruction in the location associated with the particular interrupt.

Memory consistency The same data stored in different modules is correctly stored.

Memory cycle Operations required to store or retrieve information from memory.

Microprogram Program that interprets conventional machine language instructions that are interpreted directly by the hardware.

Operating system Set of programs that monitor and operate computer hardware and simplify such tasks as input/output (I/O), editing, and program translation. Also serves as an interface with the user.

Program counter Register in which the location of the current instruction is stored.

Virtual memory Approach to extending memory by using main memory and secondary memory to store a program.

THE ONCE commonplace term *minicomputer* is seldom used in current discussions about computers. The name "minicomputer" is believed by some to have come from the miniskirt that was popular when the first minicomputer was initially developed and introduced between 1959 and 1965, but the definition of minicomputer has continued to evolve as the technology has advanced. Prior to 1976, the minicomputer was defined as a small computer manufactured by a specialty company and was often sold for incorporation in larger products by other equipment manufacturers (OEMs). The classification of a computer as a mainframe, mini, or personal (microcomputer) has been based on both market and architectural differences, although today the architectural differences are few. At one time an 8-bit word-length computer was classified as a *microcomputer* (*personal computer*), a 16-bit word-length computer was classified as a minicomputer, and a 32-bit word-length computer was classified as a *mainframe*. Since then, the distinction between mainframes, minicomputers, and microcomputers has become quite nebulous and some might say nonexistent. A broad classification for the late 1980s era minicomputer was that it was a 32-bit word-length computer with *physical memory* size in the range of 4 Mbytes and 64 Mbytes with some *cache memory* and with a range in price from less than $40,000 to as much as $500,000. During the 1990 era, minicomputers incorporated some sort of parallel structure in their organization with a high-speed bus and a ever-decreasing memory cycle. One minicomputer definition that stood the test of time until about 1990 was that a minicomputer does not require a raised floor. During the late 1980s timeframe, the casual observer could have been overwhelmed by the terms minicomputer, *super minicomputer* (advanced computer that was classified as a minicomputer in the future), and *mini supercomputer* (computer with supercomputer capabilities, but near the minicomputer price range). Some important characteristics that were used to differentiate the types of early computers are instruction set repertoire, complexity of *I/O* modules, speed, number of CPU registers, word length, virtual address space, and cost.

By the year 2000, a minicomputer was typically called a server and was a class of computers in the range between the personal and a mainframe, but with a capacity far exceeding the most advanced mainframe available when the first minicomputer was introduced with a cost close to the range of the earlier era minicomputers. These minicomputers have tremendous capabilities for handling the high-speed network interfaces that are currently available.

Because the history of computing was never clearly recorded nor the definitions clearly established, many of the statements about the history and definition of minicomputers will have variations between articles on the subject matter.

I. HISTORY

There are debates as to which computer was the first minicomputer, but many credit Digital Equipment Corporation (DEC), for many years the largest company in the minicomputer field, with the introduction of the first commercial minicomputer in 1959–1960, the PDP-1, a *second-generation computer* with a word length of 18 bits, memory of 4K words, and 28 instructions. It cost $120,000, a very attractive price because at the time most other computers sold for more than $1 million. The third-generation, 12-bit, $16,000 PDP-8 was an immediate hit and started DEC toward its fortune. Their success continued with the 16-bit PDP-11 and 32-bit *VAX* families. Some attributed part of DEC's success to the fact that DEC donated PDP-11s to almost all universities that had a computer science degree program. This was a reasonable conclusion because a significant amount of the programming of the PDP-11 was done in assembly language (used very little today), which was computer dependent, and graduates of these universities often chose the computer whose assembly language was familiar to them. Early minicomputers were short on the amount of memory that programs could use because they were 16-bit computers capable of addressing only 2^{16} memory locations, and this led to the 32-bit addressable computer such as the DEC VAX. The 32-bit and larger minicomputers continued the trend of the 16-bit minicomputer, supporting more simultaneous users, more *peripherals*, and larger memories than previous ones. Many of the current minicomputers are less general purpose and designed for application-specific use. The evolution of minicomputers has seen exponential growth in their features and capabilities.

Many of the original minicomputers were only super components with a market to OEMs for inclusion in larger products. Minor efforts were put forth in the development of *software*, with enhancements limited to hardware. Prior to 1975, only a mainframe using innovative architectural techniques could achieve a high performance level. By 1975, sophisticated application software and *operating systems* had been developed to meet demands of universities and other research institutions and data-processing users who found minicomputers more cost effective and beneficial. By 1980 such performance could be achieved using a comparatively simple minicomputer architecture coupled with faster electronic components that were available and assisted by *microprocessor*-controlled input/output devices. These computers were often called super minicomputers, but were later classified by most as minicomputers. The computing capacities of most of these super minicomputers were in excess of those of mainframes designed and sold before 1980.

Most recent minicomputer enhancements are attributable to technology improvements, breakthroughs in compiler design, and significant improvements to the processor architecture. The minicomputer of 2000 evolved with technological advances such as evolution from the Complicated Instruction Set Computer (CISC) to the Reduced Instruction Set Computer (RISC), pipelining, superpipelining, superscalar, very long instruction word (VLIW), branch prediction, out-of-order instruction execution, multilevel and nonblocking caches, new buses, nonuniform-memory access, shared memory multiprocessors, and the ability to include multiple processors in a single server. In some cases, these minicomputers, such as models of the IBM AS/400, have been designed and implemented to perform specific tasks such as Web serving or to interface between a mainframe and wide area networks.

II. ARCHITECTURE, ORGANIZATION, AND IMPLEMENTATION

Most minicomputers have the same basic structure as any other general-purpose *digital computer*, consisting of datapath, control (datapath and control are usually called the processor), internal storage, input, and output units. Although minicomputers share a basic structure with other computers, they differ in the way these components are implemented and interfaced, but today often use the same processor as personal or mainframe computers. The approach that minicomputer architects have used to support high-level languages evolved from the simple stack architecture of the 1960s, to the replacement of software by hardware (high-level language hardware) or the powerful architectures with a large number of addressing modes, multiple data type, and highly *orthogonal architecture* of the 1970s, and to sophisticated *compilers* of the 1980s that renewed emphasis on machine performance with the simpler load/store (RISC) style of machine. The competitive nature of the current marketplace has caused manufacturers to pay close attention to the cost/performance ratios and use standard benchmark performance measures, such as the SPEC suites that allow users to better compare the computers of different manufacturers.

A. Design Advances

Because of technological advances, processor features have advanced significantly. Two significant changes have made it possible for new architectures to be commercially successful. The near extinction of assembly language coding and the rise of the Unix/Linux, vendor-independent operating system has reduced the risk of bringing out new architectures and caused a renaissance in computer design, and it appears that this will continue for some time to come. The popularity of the World Wide Web has created new network interface requirements for all modern computers, and minicomputer designers have risen to meet the challenge.

B. Machine Levels

As with any other computer, hardware and software levels are logically equivalent in a minicomputer. Most minicomputers have the *digital logic*, *microprogramming* (not present on a true RISC computer with conventional machine language instructions directly executed by the hardware), conventional machine, operating system, *assembly language*, and problem-oriented language (high-level language such as JAVA or C++) levels.

C. Multiple Processors in Minicomputers

Most modern minicomputers have multiple processors with a single address space that, as such, are termed multiprocessors. The processor count present in a purchased computer is based on the user needs, usually with the capability to add additional processors as use requirements increase. These processors are typically the same ones that are used in personal and mainframe computers. It is generally agreed that the multiprocessor composed of several uniprocessors provides a better cost/performance ratio than building a comparable uniprocessor using enhanced technology.

The multiple processors are interconnected using a complex network and are classified as symmetric multiprocessors (SMP) [often termed uniform memory access (UMA)] or nonuniform memory access (NUMA). The SMP multiprocessor, with support of the operating system, is one in which each processor has equal access to all memory modules and all I/O devices. The NUMA has the advantage of being able to scale to a larger number of processors. One of the greatest challenges in the design of a multiprocessor is *memory consistency* of the data.

D. Variations among Minicomputers

Minicomputers can be compared by support and internal features. It is most appropriate for a programmer to study a computer's internal features at the level at which it is being programmed, but most early computers were compared at the conventional machine level. The internal features include richness of the *instruction set*, *clock speed*, number of instructions executed per second, I/O structure, number of registers, word size, memory management approach, *addressing modes*, and *interrupt* techniques. Although these are important characteristics, the modern minicomputer is usually programmed in a high-level language and the purchase is decided on a cost/performance ratio and according to the number of processors that can be added in the future. Support characteristics of minicomputers include *maintenance*, operating system, available application software, and customer relations. If the preceding variety is coupled with stability of the manufacturing company, then the number of variations for consideration is limitless. There is actually a spectrum of computer types, and the differences among computers within each of the three major classes (mainframes, minicomputers, and personal computers) are as great as the difference between neighboring classes.

E. Internal Features of Processors

The internal features of processors are less important today in the minicomputer procurement process and are often transparent to the user because of the rich set of support tools available. However, the features are often critical and do play a significant role in the performance. They include (1) richness of the instruction set; (2) I/O structure and interrupt techniques; (3) internal communication structure; (4) number of registers and general-purpose registers available; (5) *memory management* approach; and (6) addressing modes available.

F. Support of Minicomputers

Often the prime cost consideration in buying any computer is its purchase price, which is really not its total cost. As with any other computer, the support of a minicomputer should be a key evaluation factor. Maintenance, available operating system(s), application software, and customer relations are highly variable among vendors. The minimum requirements of the operating system are that it contains the tools (such as assemblers, compilers, debugger, and editor) necessary to perform the application requirements. Other considerations include available application software (such as e-mail, scheduling, payroll, accounts payable, and accounts receivable) and software tools (such as code generators and statistics packages).

As memory and mass storage increase, there is a coinciding increase in the amount of support software that can be made available to the user. Current minicomputers have vast amounts of storage and are general-purpose computers.

III. APPLICATIONS

Minicomputers, like other computers, are used because they perform certain tasks better and more efficiently than do humans. Today, minicomputers often are considered as one of the tiers in a three-tier approach to computing. The mainframe is considered in this framework to be a central computer, the minicomputer is a regional computer, and the personal local computer is typically used by an individual. Minicomputer applications include the following.

Special-purpose computing. In laboratory experiments and scientific applications, minicomputers are use to control experiments and process information generated by the experiment. Minicomputers are powerful tools for automating manufacturing and for testing products. They also can be used for controlling machines such as large planes and ships.

Front-end processors for mainframes. Mainframes are complex, and as a result, minicomputers can be used for mainframe I/O interface.

Network processors and managers. As communication networks increase in numbers and sophistication, minicomputers are being used increasingly as a network processor and manager. This is one of the most significant uses of minicomputers today, with applications including World Wide Web server, e-mail server, news server, and scheduling server. Often the minicomputer is the server in a client/server environment. For example, with e-mail, the server has responsibility for receiving, sending, and storing e-mail messages. Typically the client on a personal computer is a software application that provides a user with a graphical user interface on a personal computer to be used to interact with the server and manipulate the user's e-mail messages. The client/server environment provides a reduction in the load on the server.

These are but a few of the applications of minicomputers. With continued minicomputer enhancements there will be an increased use of minicomputers for new applications.

IV. FUTURE DIRECTIONS

Exceptional developments in computing have been made in the areas of hardware, *optimizing compilers*, and operatings system support. It is anticipated that the cost/performance ratio for minicomputers will continue to improve substantially with additional hardware breakthroughs. Future hardware developments will implement conventional processor and memory architectures more compactly in silicon or other materials. Multiprocessors have established themselves, and significant growth will occur in the way multiple processors are used in a computer. The performance of the individual processors is expected to rapidly increase with the clock speed of announced processors exceeding 1 GHz clock speed.

Minicomputers, and computers in general, are already the most complicated devices ever developed. Revolutionary architectures will raise this complexity by many orders of magnitude. Revolutionary architectures will break from the traditional *von Neumann model* of computer architecture because of the requirements for layers of parallelism and emphasis on inference.

The number of minicomputer manufacturers has decreased, and it is anticipated that this consolidation trend will continue into the future.

SEE ALSO THE FOLLOWING ARTICLES

Computer Architecture • Microcomputer Design • Operating Systems

BIBLIOGRAPHY

Adams, D. R., and Wagner, G. E. (1986). “Computer Information Systems: An Introduction,” South-Western Publishing, Cincinnati.

Bitter, G. G. (1984). “Computers in Today’s World,” Wiley, New York.

Digital Equipment Corporation (1983). “PDP-11 Processor Handbook,” DEC, Maynard, MA.

Digital Equipment Corporation (1983). “VAX Hardware Handbook,” DEC, Maynard, MA.

Gimarc, C., and Milutinovic, V. (1987). “A Survey of RISC Processors and Computers of the Mid-1980s,” Computer.

Kidder, T. (1981). “Soul of a New Machine,” Little, Brown, and Co., New York.

Lenoski, D. (1988). A highly integrated, fault-tolerant minicomputer: The nonstop CLX. *In* “Proceedings, COMPCON, spring 1988.”

Patterson, D., and Hennessy, J. (1998). “Computer Organization and Design—The Hardware/Software Interface,” 2nd ed., Morgan Kaufmann, San Francisco.

Stallings, W. (1999). “Computer Organization and Architecture,” 5th ed., Macmillan, New York.

Tanenbaum, A. S. (1995). “Distributed Operating Systems,” Prentice-Hall, Englewood Cliffs, NJ.

Tanenbaum, A. S. (1999). “Structured Computer Organization,” 4th ed., Prentice-Hall, Englewood Cliffs, NJ.

Supercomputers

Aad J. van der Steen
Utrecht University

I. The Use of Supercomputers
II. The Evolution of Supercomputers
III. New Developments
IV. Software for Supercomputers

GLOSSARY

Cache memory Small fast memory located near a CPU. Used to mitigate the difference in speed between a CPU and main memory.

ccNUMA system Cache coherent nonuniform memory access system. A computer system in which not all memory locations can be reached in the same time span.

Chaining Direct transmission of a functional unit's result to another functional unit before storing it a register. In this way the pipelining in both functional units is combined.

Clock cycle Basic unit of time in a computer systems. All processes take place in a whole number of clock cycles. A clock cycle is of the order of a few nanoseconds (10^{-9} sec). Related is the clock frequency, the number of clock cycles per second, usually expressed as MHz.

Computer architecture High-level description of the components and their interconnections in a computer as experienced by the user.

Directive Information given to a compiler to induce a certain behavior. Directives take the form of special comment lines in a program, for instance, to force parallelization of a program fragment.

Distributed-memory system Computer in which the memory is divided over the processing nodes. Each node can address only its own memory and is not aware of the nonlocal memories.

Flop/sec Floating-point operations per second. The speed of a supercomputer is often expressed as megaflop/sec (Mflop/sec) or gigaflop/sec (Gflop/sec).

Functional unit Part of a CPU that performs a definite function such as floating-point addition or calculation of memory addresses.

Interconnection network Network that interconnects memory modules and/or processing nodes in a parallel computer.

Latency Time from the initiation of an operation to the actual start of producing the result.

Parallel process Part of a program that may be executed independently. A process possesses its own program counter and address space.

Pipelining Organization of a functional unit in stages such that it can accept a new set of operands every clock cycle similar to an assembly line in a factory.

RISC processor Reduced instruction set computer processor. A processor with an instruction set that is small compared to that of the earlier complex instruction set processors. Presently, apart from the Intel IA-32-like processors, virtually only RISC processors are built.

Shared-memory system Computer in which all of the memory is adressable by all processing nodes in the system.

Thread Independent subprocess in a program. In contrast to a process, a thread does not have its own address space: it is subordinate to the process it stems from.

VLIW computer Very large instruction word computer. A computer in which long instruction words cause many functional units to execute their instructions in parallel.

OVER the years many definitions for the notion of "supercomputer" have been given. Some of the most well-known are "the fastest existing computer at any point in time" and "a computer that has a performance level that is typically hundreds of times higher than that of normal commodity computers." Both definitions have their drawbacks. In the first definition the object in question is a moving target because of the fast rate at which new computers are concieved and built. Therefore, with this definition it is hard to know whether a certain computer is still *the* supercomputer or a new, even faster one has just emerged.

The second definition is vague because it presupposes that one can easily determine the performance level of a computer, which is by no means true, and furthermore, the performance factor that should discriminate between supercomputers and commodity computers is also not easily established. Indeed, it is not even straightforward to define what is meant by the term "commodity computer." Should a supercomputer be measured against a PC, used mainly for word processing, or against a workstation used for technical computations?

Still, it is obvious that, whatever definition is used, one expects supercomputers to be significantly faster on any task than the computers to which one is normally exposed. In that sense the second definition is more appropriate. Therefore, we adhere mainly to this rather vague definition, with the addition that supercomputers have a special *architecture* to enable them to be faster than the standard computing equipment we use every day. The architecture, that is, the high-level structure in terms of its processors, its memory modules, and the interconnection network between these elements, largely determines its performance and, as such, whether or not it is a supercomputer. Other defining features of the architecture are the instruction set of the computer and the accessibility of the components in the architecture from the programmer's point of view.

It is good to realize that even for commodity computers the speed is continuously increasing because of the processor speed, which, according to Moore's law, is doubling every 18 months. Therefore, supercomputers need to be at least at the same technology curve with respect to the processor speed and, in addition, employ their architectural advantage to stay ahead of commodity computers. This is also evident from the clock cycle in both commodity computers and supercomputers: in both types of systems the clock cycle is in the range of 1–3 nsec, and it is not likely that future supercomputers will contain processors with significantly faster processors because of the enormous additional costs incurred in the development and fabrication.

As it stands, nowadays the architectural advantage of supercomputers is due almost entirely to *parallelism*, i.e., many processors in a supercomputer are commonly involved in a single computational task. Ideally, the speedup that is achieved increases linearly with the number of processors that are contributing to such a computational task. Because of the time spent in the coordination of the processors, this linear increase in speed is seldomly observed. Nevertheless, parallelism enables us to tackle computational problems that would be simply unthought of without it.

As early as 1972 Flynn made a classification of computer architectures that largely determines if and how parallelism can be employed within these architectures. Commodity computers are of the SISD type. SISD stands here for single-instruction stream, single-data stream computer. Parallel computers are either of the single-instruction stream, multiple-data stream (SIMD) type, in which a single instruction gives rise to executing this instruction on many data items in parallel, or of the multiple-instruction stream, multiple-data stream (MIMD) type, in which various instructions can be executed at the same time, each operating on its own data items. Another very important distinction can be made with respect to the organization of the memory: largely it can be either *shared* or *distributed*. In the former case all data reside in a common memory that is accessible by all processors in the computer. In the latter case one or several processors can normally access only their local part of the memory. In the case that such a processor or group of processors needs data that do not reside in the local part of the memory, it has explcitly to import these data from a nonlocal part of the memory. This has a huge impact on the way programs are written for the respective types of machines. According to Flynn's classification and the way memory is organized, the terms SM-SIMD, DM-SIMD, SM-MIMD, and DM-MIMD computers are often used, where SM stands for shared memory and DM for distributed memory.

In the following we discuss both the use and the workings of supercomputers.

I. THE USE OF SUPERCOMPUTERS

The first electronic computers were designed for military use around 1950 and were used to calculate ballistic orbits.

Soon afterward (1951) the first commercial computers emerged, to be employed for all kinds of scientific and technical computations. This is still the most important realm for today's supercomputers, although the application field has diversified greatly over the years.

An area in which supercomputers have become indispensible is weather forecasting and research in climatology. The quality of weather forecasts has been increasing steadily with that of the numerical models that describe the motions of air, moisture, and driving forces such as sunshine and temperature differences. The price to be paid for these more refined models is the increased amount of computation to be done. In particular, for weather forecasting the timeliness of the results is of obvious importance, so for more intricate weather models, computers have to be faster to meet the time requirements.

Weather models are only one of the manifestations of computer models that deal with phemonema of the flow of gases and fluids, free, in pipes or in ocean beds, or around bodies such as aircraft or cars. This large family of models is the realm of computational fluid dynamics (CFD). In the CFD field vast amounts of supercomputer power are used to investigate climate change, the optimal shape of an aircraft wing (the computer model constituting a "numerical windtunnel"), or the behavior of heated plasma in the Sun's corona.

Also, safety issues for complicated building structures and, again, aircraft and cars are amenable to computer modeling. This area is called structural analysis. The structures under investigation are divided into thousands to millions of subregions, each of which is subjected to forces, temperature changes, etc., that cause the deformation of these regions and stresses in the materials used. Car crash analysis is, in this respect, an obviously important topic that requires supercomputer power to model the deformation of a car's body at impact with other bodies of various sizes, at different speeds, and under different angles. A subject where the CFD and structural analysis fields are combined in a highly complicated way is the Nuclear Stockpile Stewardship, in which it is attempted to replace actual testing of nuclear weapons, sometimes by explosion, by simulation of the testing circumstances with numerical models that integrate the computational components. Especially, the stockpile stewardship has been an enormous incentive for the development of new and faster supercomputers via the Accelerated Strategic Computer Initiative (ASCI) in the United States.

At a more fundamental level, computer models are used to investigate the structure of materials to help understand superconductivity and other phenomena that require knowledge about how electrons behave in semiconducting and conducting materials. Also, the reaction mechanisms in organic and bioorganic molecules and, indeed, their three-dimensional shape depend on the electronic structure in these molecules and, ultimately, their activity in biological systems as building material for living cells or as the key components for medicines. The computer models describing and evaluating all these aspects on an atomic molecular scale are in the field of numerical quantum physics and quantum chemistry. Vast amounts of supercomputing time are spent in this broad field, looking for new medicines, unraveling the structure of viruses, or trying to find higher-temperature superconductors.

In recent years parallel supercomputers have penetrated in areas where computing formerly had no or only a marginal role, for instance, in analysis and prediction of stock exchange rates. Large amounts of computing power are used here to evaluate the many time series and differential equations that model these rates. Furthermore, massive data processing in the form of advanced database processing and data mining relies these days on parallel supercomputers to deliver the timely results required in controlling stock, building customer profiles, and building new knowledge based on hidden patterns in data that were almost-inaccessible before because of their shear size. A subfield that depends critically on this massive data processing is the Human Genome Project, which adds vast amounts of DNA data to the part of the genome that has already been mapped. The amount of data is so large that databases of databases are necessary to handle it. In addition, not all these data are entirely reliable, and they have to be screened and reinserted as the knowledge in this huge project grows. Still, the raw data thus becoming available are just the starting point of all that can be known and done with it. Often quantum chemical and molecular dynamics techniques are used to discover the function and importance of the DNA sequences, and matching with known, almost-identical but subtly different sequences has to be done to assess the influence that, for instance, the exact form has on their activity and function.

Although this is by no means an exhaustive recounting of all the situations that call for the use of supercomputers, it illustrates that supercomputers are wonderfully flexible research instruments. Two general statements about their use can be safely made. First, the number of application fields will expand still more in the near-future, and, second, there is an insatiable need for even higher computing speeds. So we may assume that the notion of supercomputers will be with us for the forseeable future.

II. THE EVOLUTION OF SUPERCOMPUTERS

It is not straightforward to determine how and when the era of supercomputing began. Perhaps it was when the realization of computer architectures that were not necessarily the most cost-effective, but that had the highest possible performance as their primary goal, was explicitly chosen. This was certainly true for the Burroughs ILLIAC IV, a machine that originally was to have 256 processors divided into four 64-processor quadrants. Only one quadrant was ever built and delivered to NASA Ames in 1972. Its 64 processors were executing the same instruction in parallel on data of which each of the processors had a part, and as such, it was of the DM-SIMD machine type. With its 80-nsec clock cycle it was able to reach a speed of 50 Mflop/sec, an impressive performance for that time. A few years later, in 1979, the first Cray-1 vector supercomputer was delivered to Lawrence Livemore National Laboratories. With its conveyor belt-like processing of operands and its 12.5-nsec clock, it was able to produce two results per clock cycle in the right circumstances, at a peak speed of 160 Mflop/sec. This machine was designed by Seymour Cray, formerly of Control Data Corporation, where he was responsible for its predecessor, the STAR-100. With the advent of the Cray machines, the supercomputer era had really begun. A very good account of the early days of supercomputing and the developments that made it possible is given by Hockney and Jesshope (1987). We now turn to the main players in the supercomputing field in the beginning of this period.

A. Vector Processors

In the supercomputing field the early days were dominated by vector processors. Strictly speaking, the results that are produced on a single-processor vector system are not parallel but "almost parallel": after a startup phase each functional unit can deliver a result every cycle. This is brought about by pipelining their operations (see Glossary). Although vector processors are not the only ones that employ pipelining, in a vector processor everything is geared to using it with the highest possible efficiency, such as vector registers and vector instructions, which operate on the data items in these registers. Figure 1 is generic block diagram of a vector processor.

The single-processor vector machine will have only one of the vector processors depicted, and the system may even have its scalar floating-point capability shared with the vector processor. The early vector processors indeed possessed only one VPU, while present-day models can house up to 64 feeding on the same shared memory. It may be noted that the VPU in Fig. 1 does not show a cache. The majority of vector processors do not employ a cache anymore. In many cases the vector unit cannot take advantage of it and the execution speed may even be unfavorably affected because of frequent cache overflow.

Although vector processors have existed that loaded their operands directly from memory and stored the results

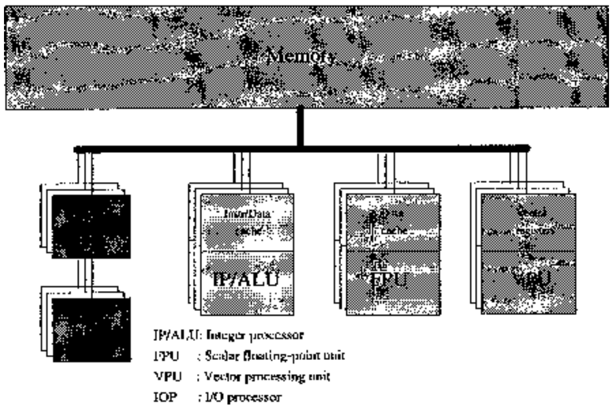

Memory
Instr/Data cache
IP/ALU
Data cache
FPU
Vector registers
VPU
IP/ALU: Integer processor
FPU : Scalar floating-point unit
VPU : Vector processing unit
IOP : I/O processor

FIGURE 1 Block diagram of a vector processor.

again immediately in memory (CDC Cyber 205, ETA-10), all present-day vector processors use vector registers. This usually does not impair the speed of operations, while it provides much more flexibility in gathering of operands and manipulation with intermediate results.

Because of the generic nature of Fig. 1, no details of the interconnection between the VPU and the memory are shown. Still, these details are very important for the effective speed of a vector operation: when the bandwidth between the memory and the VPU is too small, it is not possible to take full advantage of the VPU because it has to wait for operands and/or has to wait before it can store results. When the ratio of arithmetic-to-load/store operations is not high enough to compensate for such situations, severe performance losses may be incurred. The influence of the number of load/store paths for the dyadic vector operation $c = a + b$ (a, b, and c vectors) is depicted in Fig. 2.

Because of the high costs of implementing these data paths between the memory and the VPU, compromises are often sought, and the number of systems that have the full required bandwidth (i.e., two load operations and

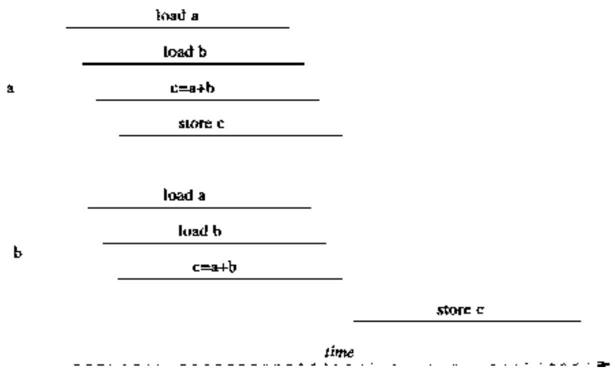


FIGURE 2 Schematic diagram of a vector addition: (a) when two load pipes and one store pipe are available; (b) when two load/store pipes are available.

one store operation at the *same* time) is limited. In fact, the vector systems marketed today no longer have this large bandwidth. Vendors rather rely on additional caches and other tricks to hide the lack of bandwidth.

The VPUs are shown as a single block in Fig. 1, yet there is considerable diversity in the structure of VPUs. Every VPU consists of a number of vector functional units, or "pipes," that fulfill one or several functions in the VPU. Every VPU has pipes that are designated to perform memory access functions, thus assuring the timely delivery of operands to the arithmetic pipes and storing of the results in memory again. Usually there are several arithmetic functional units for integer/logical arithmetic, for floating-point addition, for multiplication, and sometimes for a combination of these, a so-called compound operation. Division is performed by an iterative procedure, table lookup, or a combination of both, using the add and multiply pipe. In addition, there is almost always a mask pipe to enable operation on a selected subset of elements in a vector of operands. Finally, such sets of vector pipes can be replicated within one VPU (2- to 16-fold replication occurs). Ideally, this will increase the performance per VPU by the same factor, provided that the bandwidth to memory is adequate.

The proportion of vector processors in the present-day supercomputer arena is declining rapidly. The reason is the relatively small number of these systems, with their specialized processor architecture, that can be sold. This makes it impossible to amortize the high development and fabrication costs over a large user community. Therefore, nowadays these systems are often replaced by RISC-based parallel machines with a lower effective performance per processor but with more less costly processors.

B. Processor-Array Machines

In processor-array systems all the processors operate in lock-step, i.e., all the processors execute the same instruction at the same time (but on different data items), and no synchronization between processors is required. This greatly simplifies the design of such systems. A *control processor* issues the instructions that are to be executed by the processors in the processor array. All currently available DM-SIMD machines use a front-end processor to which they are connected by a data path to the control processor. Operations that cannot be executed by the processor array or by the control processor are offloaded to the front-end system. For instance, I/O may be through the front-end system, by the processor array machine itself, or both. Figure 3 shows a generic model of a DM-SIMD machine, from which actual models will deviate to some degree. Figure 3 might suggest that all processors in such systems are connected in a two-dimensional (2-D) grid,

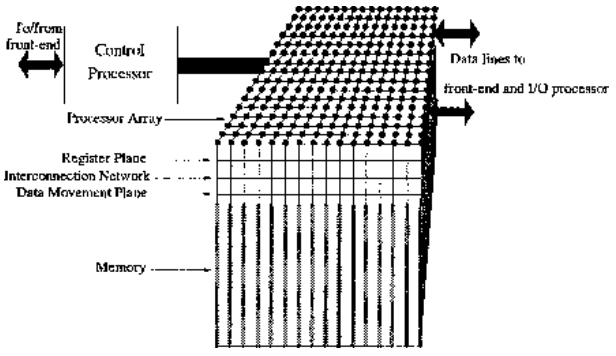


FIGURE 3 A generic block diagram of a distributed-memory SIMD machine.

and indeed, the interconnection topology of this type of machine always includes a 2-D grid. As the opposing ends of each grid line are also always connected, the topology is rather that of a torus. For several machines this is not the only interconnection scheme: they might also be connected in 3-D, diagonal, or more complex structures.

It is possible to exclude processors in the array from executing an instruction under certain logical conditions, but this means that for the time of this instruction these processors are idle (a direct consequence of the SIMD-type operation), which immediately lowers the performance. Another situation that may adversely affect the speed occurs when data required by processor i reside in the memory of processor j—in fact, as this occurs for all processors at the same time, this effectively means that data will have to be permuted across the processors. To access the data in processor j, the data will have to be fetched by this processor and then sent through the routing network to processor i. This may be fairly time-consuming. For both reasons mentioned, DM-SIMD machines are rather specialized in their use when one wants to employ their full parallelism. Generally, they perform excellently on digital signal and image processing and on certain types of Monte Carlo simulations where virtually no data exchange between processors is required and exactly the same type of operations is done on massive data sets of a size that can be made to fit comfortably in these machines.

The control processor as depicted in Fig. 3 may be more or less intelligent. It issues the instruction sequence that will be executed by the processor array. In the worst case (which means a less autonomous control processor), when an instruction is not fit for execution in the processor array (e.g., a simple print instruction), it might be offloaded to the front-end processor, which may be much slower than execution in the control processor. In the case of a more autonomous control processor, this can be avoided, thus saving processing interrupts in both the front-end and the control processor. Most DM-SIMD systems have the capability to handle I/O independently from the front-end processors. This is favorable not only because the communication between the front-end and the back-end systems is avoided. The (specialized) I/O devices for the processor-array system are generally much more efficient in providing the necessary data directly to the memory of the processor array. Especially for very data-intensive applications such as radar and image processing, such I/O systems are very important.

Processor-array machines were first introduced in the 1980s and have not seen much development beyond the first models, except in the overall technology speedup that applies to all computer systems. They fit well in their particular application niche and they will not easily be replaced by radically new architectures.

C. MPP Systems

MPP systems, where MPP stands for massively parallel processors, can be of both the shared-memory type and the distributed-memory type. In both of these types, one is confronted with the problem of how to deliver the data from the memory to the processors. An *interconnection network* is needed that connects the memory or memories, in the case of a distributed memory machine, to the processors. Through the years many types of networks have been devised. Figure 4 shows some types of networks used in present-day MPP systems.

When more CPUs are added, the collective bandwidth to the memory ideally should increase linearly with the number of processors. Unfortunately, full interconnection is quite costly, growing with $\mathcal{O}(n^2)$, while the number of processors increases with $\mathcal{O}(n)$. As shown in Fig. 4, this is exactly the case for a crossbar: it uses n^2 connections, and an Ω-network uses $n \log_2 n$ connections, while with the central bus there is only one connection. This is reflected in the use of each connection path for the different types of interconnections: for a crossbar each data path is direct and does not have to be shared with other elements. In the case of the Ω-network there are $\log_2 n$ switching stages and as many data items may have to compete for any path. For the central data bus all data have to share the same bus, so n data items may have to compete at any time.

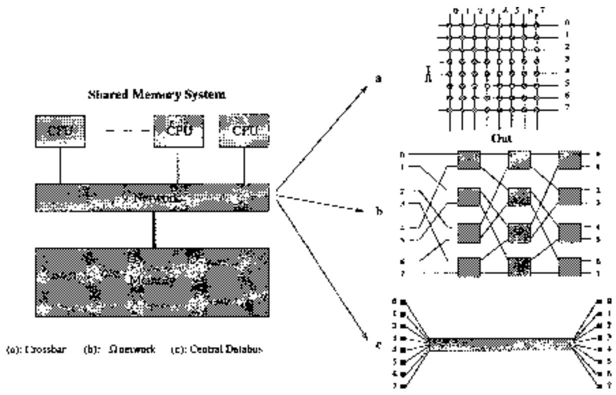


FIGURE 4 Some examples of interconnection structures, used here in a shared-memory MIMD system. The same networks may be applied in DM-MIMD systems.

The bus connection is the least expensive solution, but it has the obvious drawback that bus contention may occur, thus slowing down the traffic between the begin and end points of the communication. Various intricate strategies have been devised using caches associated with the CPUs to minimize the bus traffic. This leads, however, to a more complicated bus structure, which raises the costs. In practice it has proved to be very hard to design buses that are fast enough, especially with the speed of processors increasing very quickly; this imposes an upper bound on the number of processors thus connected, which appears not to exceed 10–20.

A multistage crossbar is a network with logarithmic complexity, and it has a structure which is situated somewhere between that of a bus and that of a crossbar with respect to potential capacity and costs. The Ω-network as depicted in Fig. 4 is an example. Commercially available machines such as the IBM RS/6000 SP and the SGI Origin2000 use such a network structure. For a large number of processors the $n \log_2 n$ connections quickly become more attractive than the n^2 used in crossbars. Of course, the switches at the intermediate levels should be sufficiently fast to cope with the bandwidth required.

Whichever network is used, the type of processors in principle could be arbitrary for any topology. In practice, however, bus-structured machines do not have vector processors, as the speeds of these would grossly mismatch any bus that could be constructed at a reasonable cost. All available bus-oriented systems use RISC processors. The local caches of the processors can sometimes alleviate the bandwidth problem if the data access can be satisfied by the caches, thus avoiding references to the memory.

DM-MIMD MPP machines are undoubtedly the fastest-growing class in the family of supercomputers, although this type of machine is more difficult to deal with than shared-memory machines and processor-array machines. For shared-memory systems the data distribution is completely transparent to the user. This is quite different for DM-MIMD systems, where the user has to distribute the data over the processors, and also the data exchange between processors has to be performed explicitly. The initial reluctance to use DM-MIMD machines has decreased lately. This is partly due to the now-existing standards for communication software such as MPI (message passing interface) and PVM (parallel virtual machine) and is partly because, at least theoretically, this class of systems is able to outperform all other types of machines.

DM-MIMD systems have several advantages: the bandwidth problem that haunts shared-memory systems is avoided because the bandwidth scales up automatically with the number of processors. Furthermore, the speed of the memory, which is another critical issue with shared-memory systems (to get a peak performance that is comparable to that of DM-MIMD systems, the processors of shared-memory machines should be very fast and the speed of the memory should match it), is less important for DM-MIMD machines because more processors can be configured without the aforementioned bandwidth problems.

Of course, DM-MIMD systems also have their disadvantages: the communication between processors is much slower than in SM-MIMD systems, and so, the synchronization overhead for communicating tasks is generally orders of magnitude higher than in shared-memory machines. Moreover, the access to data that are not in the local memory belonging to a particular processor have to be obtained from nonlocal memory (or memories). This is, again, slow in most systems slow compared to local data access. When the structure of a problem dictates a frequent exchange of data between processors and/or requires many processor synchronisations, it may well be that only a very small fraction of the theoretical peak speed can be obtained. As already mentioned, the data and task decompositions are factors that mostly have to be dealt with explicitly, which may be far from trivial.

Nowadays, processors are mostly off-the-shelf RISC processors. A problem for DM-MIMD MPP systems is that the speed of these processors increases at a fast rate, doubling in speed every 18 months. This is not so easily attained for the interconnection network. So a mismatch of communication vs computation speed may occur, thus turning a computation-bound problem into a communication-bound problem.

D. Clustered Systems

Recently a trend can be observed toward building systems that have a rather small number (up to 16) of RISC processors that are tightly integrated in a cluster, a symmetric multiprocessing (SMP) node. The processors in such a node are virtually always connected by a one-stage crossbar, while the clusters themselves are connected by a less costly network. Such a system may look like that depicted in Fig. 5. Note that in Fig. 5 all CPUs in a cluster are connected to a common part of the memory.

Some vendors have included hardware assistence such that all of the processors can access all of the address space. Therefore, such systems can be considered SM-MIMD machines. On the other hand, because the memory is physically distributed, it cannot be guaranteed that a data access operation will always be satisfied within the same time. Therefore such machines are called ccNUMA systems, where ccNUMA stands for cache coherent non-uniform memory access. The term "cache coherent" refers to the fact that for all CPUs any variable that is to be used must have a consistent value. Therefore, it must be assured that the caches that provide these variables are also consistent in this respect by additional hardware and operating system functions.

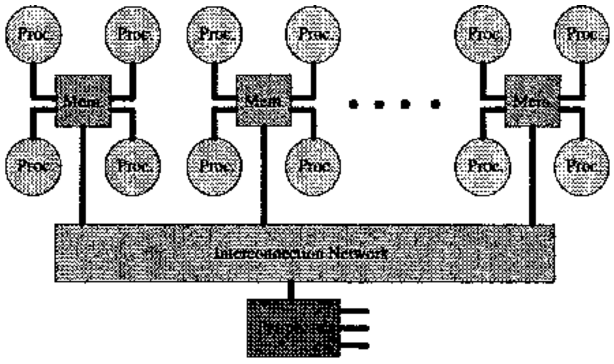


FIGURE 5 Block diagram of a system with a "hybrid" network: clusters of four CPUs are connected by a crossbar. The clusters are connected by a less expensive network; e.g., a butterfly network.

For all practical purposes we can classify these systems as SM-MIMD machines also because special assisting hardware/software (such as a directory memory) has been incorporated to establish a single system image, although the memory is physically distributed.

III. NEW DEVELOPMENTS

The need for an ever-higher computing speed is insatiable and therefore new ways are continuously sought to increase. As may be clear from the discussion in the former section, the main problem that must be overcome is the growing difference in speed between CPUs and memories. A possible way of hiding the speed difference lies in *multithreading*. A thread can be seen as an independent subtask in a program that requires its own data and produces its own distinct results. Most programs contain many threads that can be executed independently. In a multithreaded machine many threads are executed in parallel, and when the data for a certain thread are not yet available another thread can become active within one or a few clock cycles, thus giving the abandoned thread the opportunity to get its data in place. In this way the slowness or *latency* of memory requests can be hidden as long as enough independent threads are available. The Tera Corporation has actually built such a system, the Tera MTA, which is presently evaluated. Also, the miniaturization of the various parts on chips makes it possible to include a limited amount of multithreading supporting hardware on the chip. So, this latency hiding mechanism will very probably turn up in many more processors in the coming years.

Another way to approach the latency problem is by not shipping the data from memory at all: some intelligence could be built into the memory itself, thus allowing for processing-in-memory (PIM). Research into PIM is quite active, and speed gains of factors of several hundreds and more have been demonstrated. One may expect that this kind of processing will appear in supercomputers within the next 5 years.

A different development that already existed in past supercomputer models is getting renewed attention: the principle of the very long instruction word (VLIW) machine. In these systems a large number of functional units is activated at the same time by many instructions packed into a long instruction word. The late multiflow trace system, which was of this type, used instruction words of 1024 bits that, in optimal circumstances, caused 28 functional units to execute. Note that in this approach the memory latency is not addressed, but one tries to speed up the computation by executing more instructions at the same time. The renewed insterest stems from the fact that in VLIW systems the instructions are scheduled statically, i.e., the CPUs are not involved in deciding which instruction should be executing at what moment. This requires extensive hardware support and the results of this dynamic scheduling are not always satisfactory. Furthermore, when the number of functional units increases beyond five or six, the complexity of dynamic scheduling becomes so high that for most programs no reasonably optimal schedule can be found. Also, because the dynamic scheduling is an intricate process, the clock cycle of the processor may be lengthened to accommodate it. In static scheduling the decision process has already taken place by means of the compiler, which produces the instruction schedule. Consequently, the clock cycle may be lowered and more functional units can be put to work at the same time. The downside of this is that very high demands on the quality of the compilers for VLIW systems must be made to assure that the instruction schedules employ the functional units efficiently. The Intel IA-64 chip family can be seen as a modest form of a VLIW chip, using instruction words of 64 bits, although the concept is renamed EPIC (Explicitly Parallel Instruction Computer) by Intel.

IV. SOFTWARE FOR SUPERCOMPUTERS

To take advantage of the speed that supercomputers can offer, one must be able to put to work as much of an available system as possible for the largest possible part of the computational task at hand. For some system types this is done largely automatically, while for others this is very inefficient or very hard. Over the years much effort has been invested in vectorizing and autoparallelizing compilers that can take off much of the burden that otherwise would befall the user of the machine. Fortunately, in the last few years many standardization efforts have also been finished, the outcomes of which have been widely accepted. This means that programs for one supercomputer can be ported to other ones with minimal effort once a certain progamming model has been chosen. Below we discuss some software developments that complement the supercomputers that are presently marketed.

A. Software for Shared-Memory Systems

Parallelization for shared-memory systems is a relatively easy task, at least compared to that for distributed-memory systems. The reason lies in the fact that in shared-memory systems the user does not have to keep track of where the data items of a program are stored: they all reside in the same shared memory. For such machines often an important part of the work in a program can be parallelized, vectorized, or both in an automatic fashion. Consider, for instance, the simple multiplication of two rows of numbers several thousand elements long. This is an operation that is abundant in the majority of technical/scientific programs. Expressed in the programming language Fortran 90, this operation would look like

```
do i = 1,  10000
   a(i) = b(i)*c(i)
end do
```

and would cause rows `b` and `c`, each 10,000 elements long, to be multiplied and the row of 10,000 results to be named `a`. Most parallel systems have a Fortran 90 compiler that is able to divide the 10,000 multiplications in an even way over all available processors, which would result, e.g., in a 50-processor machine, in a reduction of the computing time of almost a factor of 50 (there is some overhead involved in dividing the work over the processors). Not all compilers have this ability. However, by giving some *directives* to the compiler, one still may induce the compiler to spread the work as desired. These directives are defined by the OpenMP Consortium [4] and they are accepted by all major parallel shared-memory system vendors. The program fragment above would look like

```
!$omp  parallel  do
do i = 1, 10000
   a(i)  = b(i)*c(i)
end do
!$omp end parallel do
```

informing the compiler that this fragment should be parallelized. The lines starting with `!$omp` are OpenMP directive lines that guide the parallelization process. There are many less simple situations where OpenMP directives may be applied, sometimes helping the compiler, because it does not have sufficient knowledge to judge whether a certain part of a program can safely be parallelized or not. Of course this requires an intimate knowledge of the program by the user, to know where to use the appropriate directives. A nice feature of the directives is that they have exactly the same form as the commentary in a normal nonparallel program. This commentary is ignored by compilers that do not have OpenMP features. Therefore, programs with directives can be run on parallel and nonparallel systems without altering the program itself.

Apart from what the programmer can do to parallelize his or her programs, most vendors also offer *libraries* of subprograms for operations that will often occur in various application areas. These subprograms are made to run very efficiently in parallel in the vendor's computers and, because every vendor has about the same collection of subprograms available, does not restrict the user of these programs to one computer. A foremost example is `LAPACK`, which provides all kinds of linear algebra operations and is available for all shared-memory parallel systems.

B. Software for Distributed-Memory Systems

As remarked in the former section, the paralellization of applications in distributed-memory systems is less simple than in their shared-memory counterparts. The reason is that in distributed-memory systems not all data items will reside in the same memory, and the user must be aware where they are and explicitly move or copy them to other memories if this is required by the computation. As for shared-memory systems, there has been a significant standardization effort to ensure that programs written for one computer also work in systems of other vendors. The most important ones are the communication libraries MPI (Message Passing Interface) and PVM (Parallel Virtual Machine). Both define subprograms that enable copying data from one memory to another, broadcasting data to the memories in a predefined collection of processors, or gathering data from all or a subset of processors to a specified processor. As in the former section we imagine, again, that rows `b` and `c`, of length 10,000, should be multiplied and the result should be stored in row `a`. This time, however, rows `b` and `c` are scattered over the memories of 50 processors. For each of the processors the program fragment is very similar to that of the shared-memory version, but each processor contains only 200 of the 10,000 elements of `b` and `c`, and therefore the program fragment would read

```
 do i = 1, 200
    a(i) = b(i)*c(i)
end do
```

However, when we need to have the total result in one processor, say processor 0, we have to collect all these subresults explictly in this processor. We can achieve this by using the MPI communication library, and the program fragment should be modified in the following way:

```
 do i = 1, 200
    a(i) = b(i)*c(i)
end do
call mpi_gather(a, 200, mpi_real, &
                a, 200, mpi_real, &
                0, mpi_world_comm, ierr)
```

The last line causes all processors in the processor set `mpi_world_comm` to send their partial results to processor 0, where they will be placed in a row named `a` in the correct order. One can imagine that in many programs the sending and receiving of data from other processors can quickly become complicated. Furthermore, in contrast with the shared-memory program, we have to alter the program with respect to the nonparallel version to obtain the desired result.

Apart from MPI and PVM, there are programming models that attempt to hide the distributed nature of the machine at hand from the user. One of these is HPF (High Performance Fortran). In this Fortran 90-like language one can specify how the data are spread out over the processors, and all communication that results from this is taken care of by the HPF compiler and run time system. HPF, however, can be applied only in relatively easy and very regular cases. Such applications are characterized as *data parallel*. Therefore, the use of HPF is fairly limited.

As with shared-memory systems, also for distributed-memory systems application software libraries have been, and are being, developed. ScaLAPACK is a distributed-memory version of the LAPACK library mentioned above. Other application libraries may in turn rely on ScaLAPACK, for instance, in solving partial differential equations as in the PETSc package. Much of this software can be found on the World Wide Web, e.g., via http://www.netlib.org.

SEE ALSO THE FOLLOWING ARTICLES

Computer Architecture • Computer Networks • Databases • Image Processing • Intelligent Control • Molecular Electronics • Parallel

COMPUTING • QUANTUM CHEMISTRY • SIGNAL PROCESSING, DIGITAL • SOFTWARE ENGINEERING • WEATHER PREDICTION, NUMERICAL

BIBLIOGRAPHY

Culler, D. E., Singh, J. P., and Gupta, A. (1998). "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann, San Francisco, CA.

Hockney, R. W., and Jesshope, C. R. (1987). "Parallel Computers II," Adam Hilger, Bristol.

Hwang, K., and Xu, Z. (1998). "Scalable Parallel Computing: Technology, Architecture, Programming," WCB/McGraw–Hill, New York.

OpenMP Forum (1997). "Fortran Language Specification, Version 1.0," www.openmp.org/, Oct.

van der Steen, A. J. (2000). "Overview of Recent Supercomputers," Yearly updated report for the Dutch National Science Foundation, 10th ed., Feb. [The compressed PostScript version can be downloaded from www.euroben.nl/reports. The web version can be viewed via www.euroben.nl/.]