Web server via the telephone system using ADSL, which was discussed in Chap. 2. (Alternatively, cable TV can be used, in which case the left-hand part of Fig. 8-14 is slightly different and the cable company is the ISP.) The user's computer breaks the data to be sent to the server into packets and sends these packets to the user's **ISP** (**Internet Service Provider**), a company that offers Internet access to its customers. The ISP has a high-speed (fiber-optic) connection to one of the regional or backbone networks that comprise the Internet. The user's packets are forwarded hop-by-hop across the Internet until they arrive at the Web server.

Most companies offering Web service have a specialized computer called a **firewall** that filters all incoming traffic in an attempt to remove unwanted packets (e.g., from hackers trying to break in). The firewall is connected to the local LAN, typically an Ethernet switch, which routes packets to the desired server. Of course, reality is a lot more complicated than we have shown, but the basic idea of Fig. 8-14 is still valid.

Network software consists of multiple **protocols**, each one being a set of formats, exchange sequences, and rules about what the packets mean. For example, when a user wants to fetch a Web page from a server, the user's browser sends a packet containing a *GET PAGE* request using the **HTTP** (**HyperText Transfer Protocol**) to the server, which understands how to process such requests. Many protocols are in use and often combined. In most situations, protocols are structured as a series of layers. Upper layers hand packets to lower layers for processing, with the bottom layer doing the actual transmission. At the receiving side, the packets work their way up the layers in the reverse order.

Since protocol processing is what network processors do for a living, it is necessary to explain a little bit about protocols before looking at the network processors themselves. Let us go back for a moment to the *GET PAGE* request. How is that sent to the Web server? The browser first establishes a connection to the Web server using a protocol called **TCP** (**Transmission Control Protocol**). The software that implements this protocol checks that all packets have been correctly received and in the proper order. If a packet gets lost, the TCP software assures that it is retransmitted as often as need be until it is received.

In practice, what happens is that the Web browser formats the *GET PAGE* request as a correct HTTP message and then hands it to the TCP software to transmit over the connection. The TCP software adds a header in front of the message containing a sequence number and other information. This header is naturally called the **TCP header**.

When it is done, the TCP software takes the TCP header and payload (containing the *GET PAGE* request) and passes it to another piece of software that implements the **IP protocol** (**Internet Protocol**). This software attaches an **IP header** to the front containing the source address (the machine the packet is coming from), the destination address (the machine the packet is supposed to go to), how many more hops the packet may live (to prevent lost packets from living forever), a checksum (to detect transmission and memory errors), and other fields.

Next the resulting packet (now consisting of the IP header, TCP header, and *GET PAGE* request) is passed down to the data link layer, where a data link header is attached to the front for actual transmission. The data link layer also adds a checksum to the end called a **CRC** (**Cyclic Redundancy Code**) used to detect transmission errors. It might seem that having checksums in both the data link layer and the IP layer is redundant, but it improves reliability. At each hop, the CRC is checked and the header and CRC stripped and regenerated, with a format being chosen that is appropriate for the outgoing link. Figure 8-15 shows what the packet looks like when on the Ethernet. On a telephone line (for ADSL) it is similar except with a "telephone-line header" instead of an Ethernet header. Header management is important and is one of the things network processors can do. Needless to say, we have only scratched the surface of the subject of computer networking. For a more comprehensive treatment, see Tanenbaum and Wetherall (2011).

Figure 8-15. A packet as it appears on the Ethernet.

Introduction to Network Processors

Many kinds of devices are connected to networks. End users have personal computers (desktop and notebook), of course, but increasingly also game machines, PDAs (palmtops), and smartphones. Companies have PCs and servers as end systems. However, there are also numerous devices that function as intermediate systems in networks, including routers, switches, firewalls, Web proxies, and load balancers. Interestingly enough, the intermediate systems are the most demanding, since they are expected to move the largest number of packets per second. Servers are also demanding but the user machines are not.

Depending on the network and the packet itself, an incoming packet may need various kinds of processing before being forwarded to either the outgoing line or the application program. This processing may include deciding where to send the packet, fragmenting it or reassembling its pieces, managing its quality of service (especially for audio and video streams), managing security (e.g., encryption or decryption), compression/decompression, and so on.

With LAN speeds approaching 40 gigabits/sec and 1-KB packets, a networked computer might have to process almost 5 million packets/sec. With 64-byte packets, the number of packets that have to be processed per second rises to nearly 80 million. Performing the various functions mentioned above in 12–200 nsec (in addition to making the multiple copies of the packet that are invariably needed) is just not doable in software. Hardware assistance is essential.

One kind of hardware solution for fast packet processing is to use a custom **ASIC** (**Application-Specific Integrated Circuit**). Such a chip is like a hardwired program that does whatever set of processing functions it was designed for. Many current routers uses ASICs. ASICs have many problems, however. First, they take a long time to design and manufacture. They are also rigid, so if new functionality is needed, a new chip is needed. Furthermore, bug management is a nightmare, since the only way to fix one is to design, manufacture, ship, and install new chips. They are also expensive unless the volume is so large as to allow amortizing the development effort over a substantial number of chips.

A second solution is the **FPGA** (**Field Programmable Gate Array**), a collection of gates that can be organized into the desired circuit by rewiring them in the field. These chips have a much shorter time to market than ASICs and can be rewired in the field by removing them from the system and inserting them into a special reprogramming device. On the other hand, they are complex, slow, and expensive, making them unattractive except for niche applications.

Finally, we come to **network processors**, programmable devices that can handle incoming and outgoing packets at wire speed (i.e., in real time). A common design is a plug-in board containing a network processor on a chip along with memory and support logic. One or more network lines connect to the board and are routed to the network processor. There packets are extracted, processed, and either sent out on a different network line (e.g., for a router) or are sent out onto the main system bus (e.g., the PCI bus) in the case of end-user device such as a PC. A typical network processor board and chip are illustrated in Fig. 8-16.

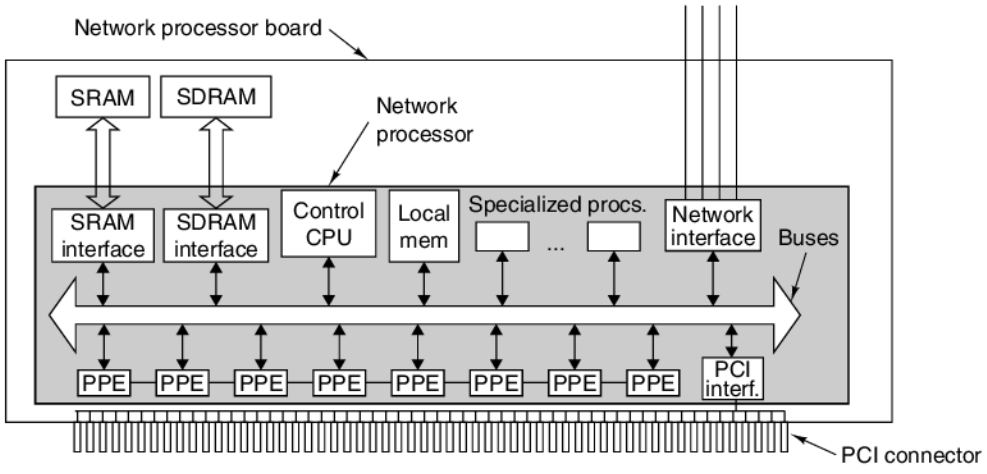


Figure 8-16. A typical network processor board and chip.

Both SRAM and SDRAM are provided on the board and typically used in different ways. SRAM is faster, but more expensive, than SDRAM, so there is only a relatively small amount of it. SRAM is used to hold routing tables and other key

data structures, whereas SDRAM holds the actual packets being processed. Making the SRAM and SDRAM external to the network processor chip gives the board designers the flexibility to determine how much of each to supply. In this way, low-end boards with a single network line (e.g., for a PC or server) can be equipped with a small amount of memory whereas a high-end board for a large router can be equipped with much more.

Network processor chips are optimized for quickly processing large numbers of incoming and outgoing packets. Millions of packets per second per network line is the norm and a router could easily have half a dozen lines. The only way to achieve such processing rates is to build network processors that are highly parallel inside. Indeed, all network processors consist of multiple **PPE**s, variously called **Protocol/Programmable/Packet Processing Engines**. Each one consists of a (possibly modified) RISC core and a small amount of internal memory for holding the program and some variables.

The PPEs can be organized in two ways. The simplest organization is having all the PPEs identical. When a packet arrives at the network processor, either an incoming packet from a network line or an outgoing packet from the bus, it is handed to an idle PPE for processing. If no PPE is idle, the packet is queued in the on-board SDRAM until a PPE frees up. When this organization is used, the horizontal connections shown between the PPEs in Fig. 8-16 do not exist because the PPEs have no need to communicate with one another.

The other PPE organization is the pipeline. In this one, each PPE performs one processing step and then feeds a pointer to its output packet to the next PPE in the pipeline. In this way, the PPE pipeline acts very much like the CPU pipelines we studied in Chap. 2. In both organizations, the PPEs are completely programmable.

In advanced designs, the PPEs have multithreading, meaning that they have multiple register sets and a hardware register indicating which one is currently in use. This feature is used to run multiple programs at the same time by allowing a program (i.e., thread) switch by just changing the "current register set" variable. Most commonly, when a PPE stalls, for example, when it references the SDRAM (which takes multiple clock cycles), it can instantaneously switch to a runnable thread. In this manner, a PPE can achieve a high utilization even when frequently blocking to access the SDRAM or perform some other slow external operation.

In addition to the PPEs, all network processors contain a control processor, usually just a standard general-purpose RISC CPU, for performing all work not related to packet processing, such as updating the routing tables. Its program and data are in the local on-chip memory. Furthermore, many network-processor chips also contain one or more specialized processors for doing pattern matching or other critical operations. These processors are really small ASICs that are good at one simple operation, such as looking up a destination address in the routing table. All the components of the network processor communicate over one or more on-chip, parallel buses that run at multigigabit/sec speeds.

Packet Processing

When a packet arrives, it goes through a number of processing stages, independent of whether the network processor has a parallel or pipeline organization. Some network processors divide these steps into operations performed on incoming packets (either from a network line or from the system bus), called **ingress processing**, and operations performed on outgoing packets, called **egress processing**. When this distinction is made, every packet goes first through ingress processing, then through egress processing. The boundary between ingress and egress processing is flexible because some steps can be done in either part (e.g., collecting traffic statistics).

Below we will discuss a potential ordering of the various steps, but note that not all packets need all steps and that many other orderings are equally valid

1. **Checksum verification.** If the incoming packet is arriving from the Ethernet, the CRC is recomputed so it can be compared with the one in the packet to make sure there was no transmission error. If the Ethernet CRC is correct or not present, the IP checksum is recomputed and compared to the one in the packet to make sure the IP packet was not damaged by a faulty bit in the sender's memory after the IP checksum was computed there. If all checksums are correct, the packet is accepted for further processing; otherwise, it is simply discarded.
2. **Field extraction.** The relevant header is parsed and key fields are extracted. In an Ethernet switch, only the Ethernet header is examined, whereas in an IP router, it is the IP header that is inspected. The key fields are stored in registers (parallel PPE organization) or SRAM (pipeline organization).
3. **Packet classification.** The packet is classified according to a series of programmable rules. The simplest classification is to distinguish data packets from control packets, but usually much finer distinctions are made.
4. **Path selection.** Most network processors have a special fast path optimized for handling plain old garden-variety data packets, with all other packets being treated differently, often by the control processor. Consequently, either the fast or the slow path has to be selected.
5. **Destination network determination.** IP packets contain a 32-bit destination address. It is not possible (or even desirable) to have a 2^{32} entry table to look up the destination of each IP packet, so the leftmost part of each IP address is the network number and the rest specifies a machine on that network. Network numbers can be of any

length, so determining the destination network number is nontrivial and made worse by the fact that multiple matches are possible and the longest one counts. Often a custom ASIC is used in this step.

6. **Route lookup.** Once the number of the destination network is known, the outgoing line to use can be looked up in a table in the SRAM. Again, a custom ASIC may be used in this step.
7. **Fragmentation and reassembly.** Programmers like to present large payloads to the TCP layer to reduce the number of system calls needed, but TCP, IP, and Ethernet all have maximum sizes for the packets they can handle. As a consequence of these limits, payloads and packets may have to be fragmented at the sending side and the pieces reassembled at the receiving side. These are tasks the network processor can perform.
8. **Computation.** Heavy-duty computation on the payload is sometimes required, for example, data compression/decompression and encryption/decryption. These are tasks a network processor can perform.
9. **Header management.** Sometimes headers have to be added, removed, or have some of their fields modified. For example, the IP header has a field that counts the number of hops the packet may yet make before being discarded. Every time it is retransmitted, this field must be decremented, something the network processor can do.
10. **Queue management.** Incoming and outgoing packets often have to be queued while waiting their turn at being processed. Multimedia applications may need a certain interpacket spacing in time to avoid jitter. A firewall or router may need to distribute the incoming load among multiple outgoing lines according to certain rules. All of these tasks can be done by the network processor.
11. **Checksum generation.** Outgoing packets need to be checksummed. The IP checksum can be generated by the network processor, but the Ethernet CRC is generally computed by hardware.
12. **Accounting.** In some cases, accounting for packet traffic is needed, especially when one network is forwarding traffic for other networks as a commercial service. The network processor can do the accounting.
13. **Statistics gathering.** Finally, many organizations like to collect statistics about their traffic. They want to know how many packets came and and how many went out, at what times of day, and more. The network processor is a good place to collect them.

Improving Performance

Performance is the name of the game for network processors. What can be done to improve it? But before improving it, we have to define what performance means. One metric is the number of packets forwarded per second. A second one is the number of bytes forwarded per second. These are different measures, and a scheme that works well with small packets may not work as well with large ones. In particular, with small packets, improving the number of destination lookups per second may help a lot, but with large packets it may not.

The most straightforward way to improve performance is to increase the speed of the network processor clock. Of course, performance is not linear with clock speed, since memory cycle time and other factors also influence it. Also, a faster clock means more heat must be dissipated.

Introducing more PPEs and parallelism is often a winner, especially with an organization consisting of parallel PPEs. A deeper pipeline can also help, but only if the job of processing a packet can be split up into smaller pieces.

Another technique is to add specialized processors or ASICs to handle specific, time-consuming operations that are performed repeatedly and that can be done faster in hardware than in software. Lookups, checksum computations, and cryptography are among the many candidates.

Adding more internal buses and widening existing buses may help gain speed by moving packets through the system faster. Finally, replacing SDRAM by SRAM can usually be counted to improve performance, but at a price, of course.

There is much more that can be said about network processors, of course. Some references are Freitas et al. (2009), Lin et al. (2010), and Yamamoto and Nakao (2011).

8.2.2 Graphics Processors

A second area in which coprocessors are used is for handling high-resolution graphics processing, such as 3D rendering. Ordinary CPUs are not especially good at the massive computations needed to process the large amounts of data required for these applications. For this reason, most PCs and many future processors will be equipped with **GPU**s (**Graphics Processing Units**) to which they can offload large portions of overall processing.

The NVIDIA Fermi GPU

We will study this increasingly important area by means of an example: the NVIDIA **Fermi GPU**, an architecture used in a family of graphics processing chips that are available at several speeds and sizes. The architecture of the Fermi GPU is shown in Fig. 8-17. It is organized into 16 **SM**s (**Streaming Multiprocessors**) having its own high-bandwidth private level-1 cache. Each of the streaming

multiprocessors contain 32 CUDA cores, for a total of 512 CUDA cores per Fermi GPU. A **CUDA** (**Compute Unified Device Architecture**) core is a simple processor supporting single-precision integer and floating-point computations. A single SM with 32 CUDA cores is illustrated in Fig. 2-7. The 16 SMs share access to a single unified 768-KB level 2 cache, which is connected to a multiported DRAM interface. The host processor interface provides a communication path between the host system and the GPU via a shared DRAM bus interface, typically through a PCI-Express interface.

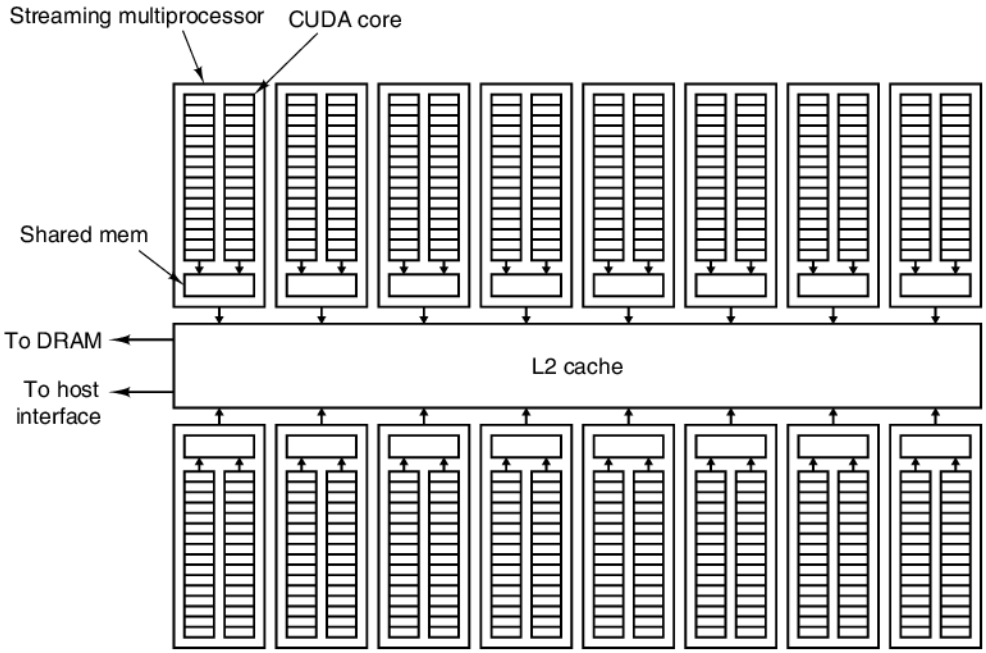


Figure 8-17. The Fermi GPU architecture.

The Fermi architecture is designed to efficiently execute graphics, video, and image processing codes, which typically have many redundant computations spread across many pixels. Because of this redundancy, the streaming multiprocessors, while capable of executing 16 operations at a time, require that all of the operations executed in a single cycle be identical. This style of processing is called **SIMD** (**Single-Instruction Multiple Data**) computation, and it has the important advantage that each SM fetches and decodes only a single instruction each cycle. Only by sharing the instruction processing across all of the cores in an SM can NVIDIA cram 512 cores onto a single silicon die. If programmers can harness all of the computation resources (always a very big and uncertain "if"), then the system provides significant computational advantages over traditional scalar architectures, such as the Core i7 or OMAP4430.

The SIMD processing requirements within the SMs place constraints on the kind of code programmers can run on these units. In fact, each CUDA core must be running the same code in lock-step to achieve 16 operations simultaneously. To ease this burden on programmer, NVIDIA developed the CUDA programming language. The CUDA language specifies the program parallelism using threads. Threads are then grouped into blocks, which are assigned to streaming processors. As long as every thread in a block executes exactly the same code sequence (that is, all branches make the same decision), up to 16 operations will execute simultaneously (assuming there are 16 threads ready to execute). When threads on an SM make different branch decisions, a performance-degrading effect called branch divergence will occur that forces threads with differing code paths to execute serially on the SM. Branch divergence reduces parallelism and slows GPU processing. Fortunately, there is a wide range of activities in graphics and image processing that can avoid branch divergence and achieve good speed-ups. Many other codes have also been shown to benefit from the SIMD-style architecture on graphics processors, such as medical imaging, proof solving, financial prediction, and graph analysis. This widening of potential applications for GPUs has earned them the new moniker of **GPGPU**s (**General-Purpose Graphics Processing Units**).

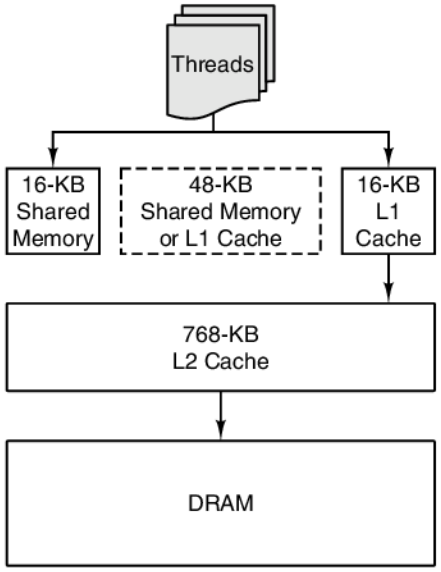


Figure 8-18. The Fermi GPU memory hierarchy.

With 512 CUDA cores, the Fermi GPU would grind to a halt without significant memory bandwidth. To provide this bandwidth, the Fermi GPU implements a modern memory hierarchy as illustrated in Fig. 8-18. Each SM has both a dedicated shared memory and a private level 1 data cache. The dedicated shared memory

is directly addressed by the CUDA cores, and it provides fast sharing of data between threads within a single SM. The level 1 cache speeds up accesses to DRAM data. To accommodate the wide range of program data usage, the SMs can be configured with 16-KB shared memory and 48-KB level 1 cache or 48-KB shared memory and 16-KB level 1 cache. All SMs share a single unified 768-KB level 2 cache. The level 2 cache provides faster access to DRAM data that do not fit in the level 1 caches. The level 2 cache also provides sharing between SMs, although this mode of sharing is much slower than the intra-SM sharing that occurs within an SM's shared memory. Beyond the level 2 cache is the DRAM, which holds the remaining data, imagery, and textures used by programs running on the Fermi GPU. Efficient programs will try to avoid accessing DRAM at all costs, as a single access can take hundreds of cycles to complete.

For a savvy programmer, the Fermi GPU represents one of the most computationally capable platforms ever created. A single Fermi-based GTX 580 GPU running at 772 MHz with 512 CUDA cores can achieve a sustained computational rate of 1.5 teraflops while consuming 250 watts of power. This statistic is even more impressive when one considers that the street price of a GTX 580 GPU is less than 600 U.S. dollars. By way of historical comparison, in 1990, the fastest computer in the world, the Cray-2, had a performance of 0.002 teraflops and a price tag (in inflation-adjusted dollars) of $30 million. It also filled a modest-sized room and came with its own liquid-cooling system to dissipate the 150 kW of power it consumed. The GTX 580 has 750 times more horsepower for 1/50000 of the price while consuming 1/600th as much energy. Not a bad deal.

8.2.3 Cryptoprocessors

A third area in which coprocessors are popular is security, especially network security. When a connection is established between a client and a server, in many cases they must first authenticate each other. Then a secure, encrypted connection has to be established between them so data can be transferred in a secure way to foil any snoopers who may tap the line.

The problem with security is that to achieve it, cryptography has to be used, and cryptography is very compute intensive. Cryptography comes in two general flavors, called **symmetric key cryptography** and **public-key cryptography**. The former is based on mixing up the bits very thoroughly, sort of the electronic equivalent of throwing a message into an electric blender. The latter is based on multiplication and exponentiation of large (e.g., 1024-bit) numbers and is extremely time consuming.

To handle the computation needed to encrypt data securely for transmission or storage and then decrypt them later, various companies have produced crypto coprocessors, sometimes as PCI bus plug-in cards. These coprocessors have special hardware that enables them to do the necessary cryptography much faster than an ordinary CPU can do it. Unfortunately, a detailed discussion of how they work

would first require explaining quite a bit about cryptography itself, which is beyond the scope of this book. For more information about crypto coprocessors, see Gaspar et al. (2010), Haghighizadeh et al. (2010), and Shoufan et al. (2011).

8.3 SHARED-MEMORY MULTIPROCESSORS

We have now seen how parallelism can be added to single chips and to individual systems by adding a coprocessor. The next step is to see how multiple full-blown CPUs can be combined into larger systems. Systems with multiple CPUs can be divided into multiprocessors and multicomputers. After taking a close look at what these terms actually mean, we will first study multiprocessors and then multicomputers.

8.3.1 Multiprocessors vs. Multicomputers

In any parallel computer system, CPUs working on different parts of the same job must communicate to exchange information. Precisely how they should do this is the subject of much debate in the architectural community. Two distinct designs, multiprocessors and multicomputers, have been proposed and implemented. The key difference between the two is the presence or absence of shared memory. This difference permeates how they are designed, built, and programmed, as well as their scale and price.

Multiprocessors

A parallel computer in which all the CPUs share a common memory is called a **multiprocessor**, as indicated symbolically in Fig. 8-19. All processes working together on a multiprocessor can share a single virtual address space mapped onto the common memory. Any process can read or write a word of memory by just executing a LOAD or STORE instruction. Nothing else is needed. The hardware does the rest. Two processes can communicate by simply having one of them write data to memory and having the other one read them back.

The ability for two (or more) processes to communicate by just reading and writing memory is the reason multiprocessors are popular. It is an easy model for programmers to understand and is applicable to a wide range of problems. Consider, for example, a program that inspects a bit-map image and lists all the objects in it. One copy of the image is kept in memory, as shown in Fig. 8-19(b). Each of the 16 CPUs runs a single process, which has been assigned one of the 16 sections to analyze. Nevertheless, each process has access to the entire image, which is essential, since some objects may occupy multiple sections. If a process discovers that one of its objects extends over a section boundary, it just follows the object

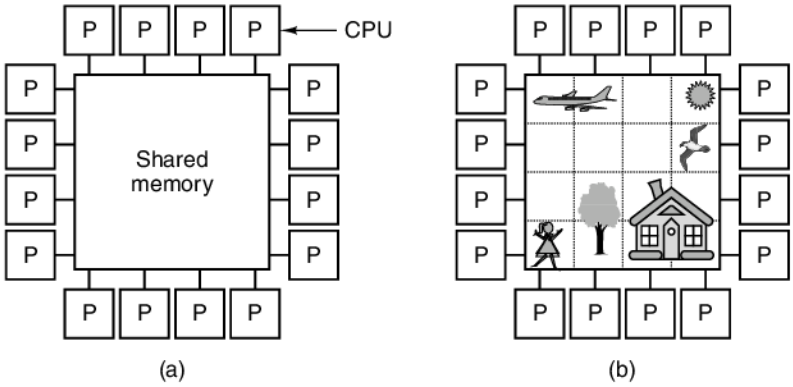


Figure 8-19. (a) A multiprocessor with 16 CPUs sharing a common memory. (b) An image partitioned into 16 sections, each being analyzed by a different CPU.

into the next section by reading the words of that section. In this example, some objects will be discovered by multiple processes, so some coordination is needed at the end to determine how many houses, trees, and airplanes there are.

Because all CPUs in a multiprocessor see the same memory image, there is only one copy of the operating system. Consequently, there is only one page map and one process table. When a process blocks, its CPU saves its state in the operating-system tables, then looks in those tables to find another process to run. It is this single-system image that distinguishes a multiprocessor from a multicomputer, in which each computer has its own copy of the operating system.

A multiprocessor, like all computers, must have I/O devices, such as disks, network adapters, and other equipment. In some multiprocessor systems, only certain CPUs have access to the I/O devices, and thus have a special I/O function. In other ones, every CPU has equal access to every I/O device. When every CPU has equal access to all the memory modules and all the I/O devices, and is treated as interchangeable with the others by the operating system, the system is called an **SMP** (**Symmetric MultiProcessor**).

Multicomputers

The second possible design for a parallel architecture is one in which each CPU has its own private memory, accessible only to itself and not to any other CPU. Such a design is called a **multicomputer**, or sometimes a **distributed memory system**, and is illustrated in Fig. 8-20(a). The key aspect of a multicomputer that distinguishes it from a multiprocessor is that each CPU in a multicomputer has its own private, local memory that it can access by just executing LOAD and STORE instructions but that no other CPU can access using LOAD and STORE instructions.

Thus multiprocessors have a single physical address space shared by all the CPUs, whereas multicomputers have one physical address space per CPU.

Since the CPUs on a multicomputer cannot communicate by just reading and writing the common memory, they need a different communication mechanism. What they do is pass messages back and forth using the interconnection network. Examples of multicomputers include the IBM BlueGene/L, Red Storm, and the Google cluster.

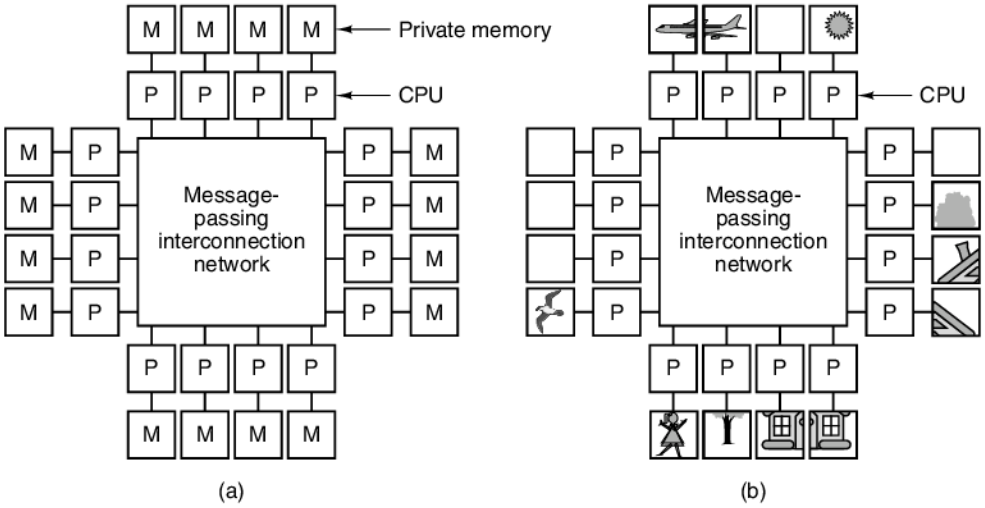


Figure 8-20. (a) A multicomputer with 16 CPUs, each with its own private memory. (b) The bit-map image of Fig. 8-19 split up among the 16 memories.

The absence of hardware shared memory on a multicomputer has important implications for the software structure. Having a single virtual address space with all processes being able to read and write all of memory by just executing LOAD and STORE instructions is impossible on a multicomputer. For example, if CPU 0 (the one in the upper left-hand corner) of Fig. 8-19(b) discovers that part of its object extends into the section assigned to CPU 1, it can nevertheless just continue reading memory to access the tail of the airplane. On the other hand, if CPU 0 in Fig. 8-20(b) makes the same discovery, it cannot just read CPU 1's memory. It has to do something quite different to get the data it needs.

In particular, it has to discover (somehow) which CPU has the data it needs and send that CPU a message requesting a copy of the data. Typically it will then block until the request is answered. When the message arrives at CPU 1, software there has to analyze it and send back the needed data. When the reply message gets back to CPU 0, the software is unblocked and can continue executing.

On a multicomputer, communication between processes often uses software primitives such as send and receive. This gives the software a different, and far more complicated, structure than on a multiprocessor. It also means that correctly

dividing up the data and placing them in the optimal locations is a major issue on a multicomputer. It is less of an issue on a multiprocessor since placement does not affect correctness or programmability although it may affect performance. In short, programming a multicomputer is much more difficult than programming a multiprocessor.

Under these conditions, why would anyone build multicomputers, when multiprocessors are easier to program? The answer is simple: large multicomputers are much simpler and cheaper to build than multiprocessors with the same number of CPUs. Implementing a memory shared by even a few hundred CPUs is a substantial undertaking, whereas building a multicomputer with 10,000 CPUs or more is straightforward. Later in this chapter we will study a multicomputer with over 50,000 CPUs.

Thus we have a dilemma: multiprocessors are hard to build but easy to program whereas multicomputers are easy to build but hard to program. This observation has led to a great deal of effort to construct hybrid systems that are relatively easy to build and relatively easy to program. This work has led to the realization that shared memory can be implemented in various ways, each with its own set of advantages and disadvantages. In fact, much research in parallel architectures these days relates to the convergence of multiprocessor and multicomputer architectures into hybrid forms that combine the strengths of each. The holy grail here is to find designs that are **scalable**, that is, continue to perform well as more and more CPUs are added.

One approach to building hybrid systems is based on the fact that modern computer systems are not monolithic but are constructed as a series of layers—the theme of this book. This insight opens the possibility of implementing the shared memory at any one of several layers, as shown in Fig. 8-21. In Fig. 8-21(a) we see the shared memory being implemented by the hardware as a true multiprocessor. In this design, there is a single copy of the operating system with a single set of tables, in particular, the memory allocation table. When a process needs more memory, it traps to the operating system, which then looks in its table for a free page and maps the page into the called's address space. As far as the operating system is concerned, there is a single memory and it keeps track of which process owns which page in software. There are many ways to implement hardware shared memory, as we will see later.

A second possibility is to use multicomputer hardware and have the operating system simulate shared memory by providing a single system-wide paged shared virtual address space. In this approach, called **DSM** (**Distributed Shared Memory**) (Li and Hudak, 1989), each page is located in one of the memories of Fig. 8-20(a). Each machine has its own virtual memory and its own page tables. When a CPU does a LOAD or STORE on a page it does not have, a trap to the operating system occurs. The operating system then locates the page and asks the CPU currently holding it in its memory to unmap the page and send it over the interconnection network. When it finally arrives, the page is mapped in and the faulting

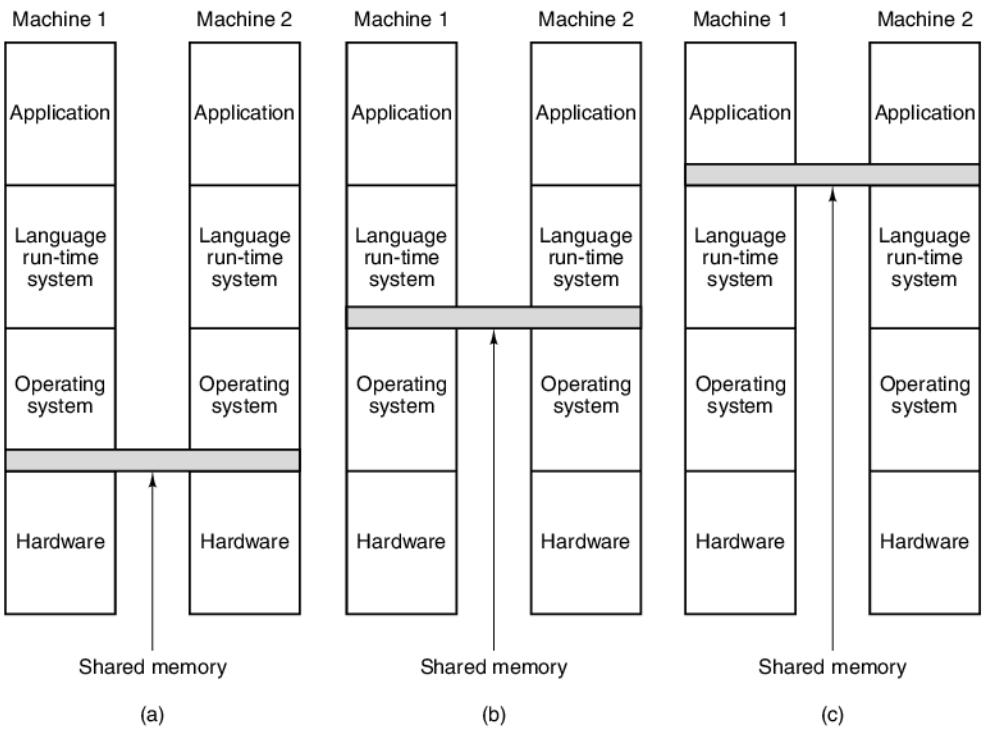


Figure 8-21. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) The language run-time system.

instruction restarted. In effect, the operating system is just satisfying page faults from remote memory instead of from disk. To the user, the machine looks as if it has shared memory. We will examine DSM later in this chapter.

A third possibility is to have a user-level run-time system implement a (possibly language-specific) form of shared memory. In this approach, the programming language provides some kind of shared-memory abstraction, which is then implemented by the compiler and run-time system. For example, the Linda model is based on the abstraction of a shared space of tuples (data records containing a collection of fields). Processes on any machine can input a tuple from the shared tuple space or output a tuple to the shared tuple space. Because access to the tuple space is controlled entirely in software (by the Linda run-time system), no special hardware or operating system support is needed.

Another example of a language-specific shared memory implemented by the run-time system is the Orca model of shared data objects. In Orca, processes share generic objects rather than just tuples and can execute object-specific methods on them. When a method call changes the internal state of an object, it is up to the run-time system to make sure all copies of the object on all machines are updated

simultaneously. Again, because objects are a strictly software concept, the implementation can be done by the run-time system without help from the operating system or hardware. We will look at both Linda and Orca later in this chapter.

Taxonomy of Parallel Computers

Now let us get back to our main topic, the architecture of parallel computers. Many kinds of parallel computers have been proposed and built over the years, so it is natural to ask if there is some way of categorizing them into a taxonomy. Many researchers have tried, with mixed results (Flynn, 1972, and Treleaven, 1985). Unfortunately, the Carolus Linnaeus† of parallel computing is yet to emerge. The only scheme that is used much is Flynn's, and even his is, at best, a very crude approximation. It is given in Fig. 8-22.

Instruction streams	Data streams	Name	Examples
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

Figure 8-22. Flynn's taxonomy of parallel computers.

Flynn's classification is based on two concepts—instruction streams and data streams. An instruction stream corresponds to a program counter. A system with n CPUs has n program counters, hence n instruction streams.

A data stream consists of a set of operands. For example, in a weather-forecasting system, each of a large number of sensors might emit a stream of temperatures at regular intervals.

The instruction and data streams are, to some extent, independent, so four combinations exist, as listed in Fig. 8-22. SISD is just the classical, sequential von Neumann computer. It has one instruction stream, one data stream, and does one thing at a time. SIMD machines have a single control unit that issues one instruction at a time, but they have multiple ALUs to carry it out on multiple data sets simultaneously. The ILLIAC IV (Fig. 2-7) is the prototype of SIMD machines. Mainstream SIMD machines are increasingly rare, but conventional computers sometimes have some SIMD instructions for processing audiovisual material. The Core i7 SSE instructions are SIMD. Nevertheless, there is one new area in which some of the ideas from the SIMD world are playing a role: stream processors.

† Carolus Linnaeus (1707–1778) was the Swedish biologist who devised the system now used for classifying all plants and animals into kingdom, phylum, class, order, family, genus, and species.

These machines are specifically designed to handle the demands of multimedia rendering and may become important in the future (Kapasi et al., 2003).

MISD machines are a somewhat strange category, with multiple instructions operating on the same piece of data. It is not clear whether any such machines exist, although some people regard pipelined machines as MISD.

Finally, we have MIMD, which are just multiple independent CPUs operating as part of a larger system. Most parallel processors fall into this category. Both multiprocessors and multicomputers are MIMD machines.

Flynn's taxonomy stops here, but we have extended it in Fig. 8-23. SIMD has been split into two subgroups. The first one is for numeric supercomputers and other machines that operate on vectors, performing the same operation on each vector element. The second one is for parallel-type machines, such as the ILLIAC IV, in which a master control unit broadcasts instructions to many independent ALUs.

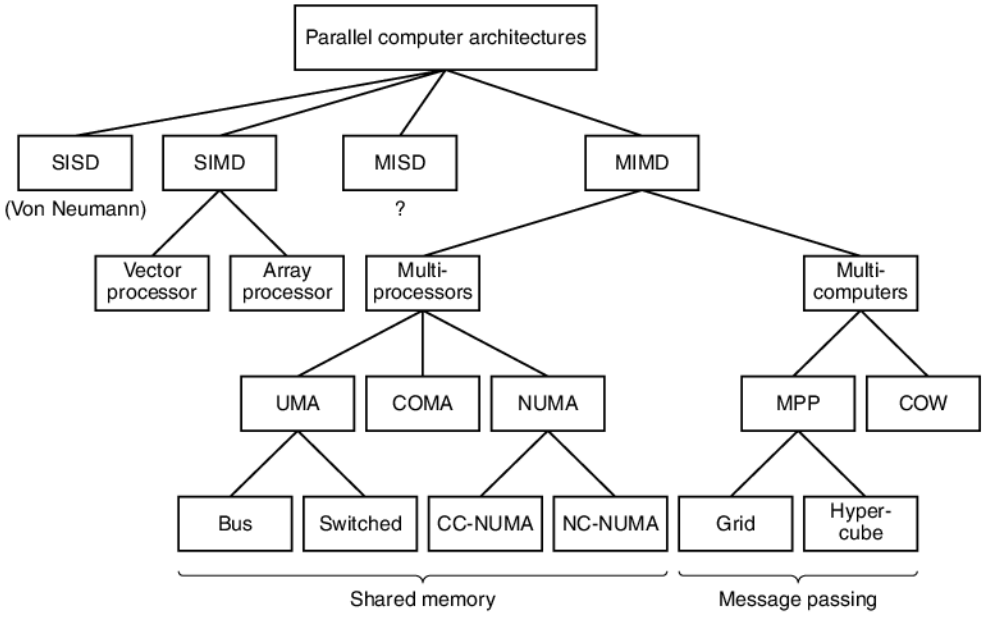


Figure 8-23. A taxonomy of parallel computers.

In our taxonomy, the MIMD category has been split into multiprocessors (shared-memory machines) and multicomputers (message-passing machines). Three kinds of multiprocessors exist, distinguished by the way the shared memory is implemented on them. They are called **UMA** (**Uniform Memory Access**), **NUMA** (**NonUniform Memory Access**), and **COMA** (**Cache Only Memory Access**). These categories exist because in large multiprocessors, the memory is usually split up into multiple modules. UMA machines have the property that each CPU

has the same access time to every memory module. In other words, every memory word can be read as fast as every other memory word. If this is technically impossible, the fastest references are slowed down to match the slowest ones, so programmers do not see the difference. This is what "uniform" means here. This uniformity makes the performance predictable, an important factor for writing efficient code.

In contrast, in a NUMA multiprocessor, this property does not hold. Often there is a memory module close to each CPU and accessing that memory module is much faster than accessing distant ones. The result is that for performance reasons, it matters where code and data are placed. COMA machines are also nonuniform, but in a different way. We will study each of these types and their subcategories in detail later.

The other main category of MIMD machines consists of the multicomputers, which, unlike the multiprocessors, do not have shared primary memory at the architectural level. In other words, the operating system on a multicomputer CPU cannot access memory attached to a different CPU by just executing a LOAD instruction. It has to send an explicit message and wait for an answer. The ability of the operating system to read a distant word by just doing a LOAD is what distinguishes multiprocessors from multicomputers. As we mentioned before, even on a multicomputer, user programs may have the ability to access remote memory by using LOAD and STORE instructions, but this illusion is supported by the operating system, not the hardware. This difference is subtle, but very important. Because multicomputers do not have direct access to remote memory, they are sometimes called **NORMA** (**NO Remote Memory Access**) machines.

Multicomputers can be roughly divided into two categories. The first contains the **MPP**s (**Massively Parallel Processors**), which are expensive supercomputers consisting of many CPUs tightly coupled by a high-speed proprietary interconnection network. The IBM SP/3 is a well-known commercial example.

The other category consists of regular PCs, workstations, or servers, possibly rack mounted, and connected by commercial off-the-shelf interconnection technology. Logically, there is not much difference, but huge supercomputers costing many millions of dollars are used differently than networks of PCs assembled by the users for a fraction of the price of an MPP. These home-brew machines go by various names, including **NOW** (**Network of Workstations**), **COW** (**Cluster of Workstations**), or sometimes just **cluster**.

8.3.2 Memory Semantics

Even though all multiprocessors present the CPUs with the image of a single shared address space, often many memory modules are present, each holding some portion of the physical memory. The CPUs and memories are often connected by a complex interconnection network, as discussed in Sec. 8.1.2. Several CPUs may be attempting to read a memory word at the same time several other CPUs are

attempting to write the same word, and some of the request messages may pass each other in transit and be delivered in a different order than they were issued. Add to this problem the existence of multiple copies of some blocks of memory (e.g., in caches), and the result can easily be chaos unless strict measures are taken to prevent it. In this section we will see what shared memory really means and look at how memories can reasonably respond under these circumstances.

One view of memory semantics is to see it as a contract between the software and the memory hardware (Adve and Hill, 1990). If the software agrees to abide by certain rules, the memory agrees to deliver certain results. The discussion then centers around what the rules are. These rules are called **consistency models**, and many different ones have been proposed and implemented (Sorin et al., 2011).

To give an idea of what the problem is, suppose that CPU 0 writes the value 1 to some memory word and a little later CPU 1 writes the value 2 to the same word. Now CPU 2 reads the word and gets the value 1. Should the computer owner bring the computer to the repair shop to get it fixed? That depends on what the memory promised (its contract).

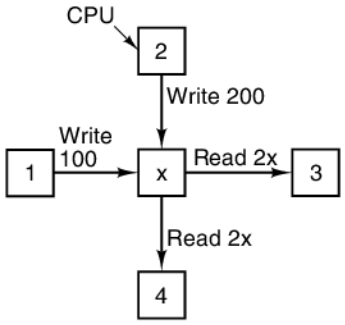
Strict Consistency

The simplest model is **strict consistency**. With this model, any read to a location x always returns the value of the most recent write to x. Programmers love this model, but it is effectively impossible to implement in any way other than having a single memory module that simply services all requests first-come, first-served, with no caching and no data replication. Such an implementation would make memory an enormous bottleneck and is thus not a serious candidate, unfortunately.

Sequential Consistency

Next best is a model called **sequential consistency** (Lamport, 1979). The idea here is that in the presence of multiple read and write requests, some interleaving of all the requests is chosen by the hardware (nondeterministically), but all CPUs see the same order.

To see what this means, consider an example. Suppose that CPU 1 writes the value 100 to word x, and 1 nsec later CPU 2 writes the value 200 to word x. Now suppose that 1 nsec after the second write was issued (but not necessarily completed yet) two other CPUs, 3 and 4, read word x twice in rapid succession, as shown in Fig. 8-24(a). Three possible orderings of the six events (two writes and four reads) are shown in Fig. 8-24(b)–(d), respectively. In Fig. 8-24(b), CPU 3 gets (200, 200) and CPU 4 gets (200, 200). In Fig. 8-24(c), they get (100, 200) and (200, 200), respectively. In Fig. 8-24(d), they get (100, 100) and (200, 100), respectively. All of these are legal, as well as some other possibilities that are not shown. Note that there is not single "correct" value.



(a)	(b)	(c)	(d)
	W100	W100	W200
	W200	R3 = 100	R4 = 200
	R3 = 200	W200	W100
	R3 = 200	R4 = 200	R3 = 100
	R4 = 200	R3 = 200	R4 = 100
	R4 = 200	R4 = 200	R3 = 100

Figure 8-24. (a) Two CPUs writing and two CPUs reading a common memory word. (b)–(d) Three possible ways the two writes and four reads might be interleaved in time.

However—and this is the essence of sequential consistency—no matter what, a sequentially consistent memory will never allow CPU 3 to get (100, 200) while CPU 4 gets (200, 100). If this were to occur, it would mean that according to CPU 3, the write of 100 by CPU 1 completed after the write of 200 by CPU 2. That is fine. But it would also mean that according to CPU 4, the write of 200 by CPU 2 completed before the write of 100 by CPU 1. By itself, this result is also possible. The problem is that sequential consistency guarantees that there is a single global ordering of all writes that is visible to all CPUs. If CPU 3 observes that 100 was written first, then CPU 4 must also see this order.

While sequential consistency is not as powerful a rule as strict consistency, it is still very useful. In effect, it says that when multiple events are happening concurrently, there is some true order in which they occur. Possibly it is determined by timing and chance, but a true ordering exists and all processors observe this same order. Although this statement may seem obvious, below we will discuss consistency models that do not guarantee even this much.

Processor Consistency

A looser consistency model, but one that is easier to implement on large multiprocessors, is **processor consistency** (Goodman, 1989). It has two properties:

1. Writes by any CPU are seen by all in the order they were issued.
2. For every memory word, all CPUs see writes to it in the same order.

Both of these points are important. The first point says that if CPU 1 issues writes with values 1A, 1B, and 1C to some memory location in that sequence, then all

other processors see them in that order, too. In other words, any other processor in a tight loop observing 1A, 1B, and 1C by reading the words written will never see the value written by 1B and then see the value written by 1A, and so on. The second point is needed to require every memory word to have an unambiguous value after several CPUs write to it and finally stop. Everyone must agree on who went last.

Even with these constraints, the designer has a lot of flexibility. Consider what happens if CPU 2 issues writes 2A, 2B, and 2C concurrently with CPU 1's three writes. Other CPUs that are busily reading memory will observe some interleaving of the six writes, such as 1A, 1B, 2A, 2B, 1C, 2C or 2A, 1A, 2B, 2C, 1B, 1C or many others. Processor consistency does *not* guarantee that every CPU sees the same ordering (unlike sequential consistency, which does make this guarantee). Thus it is perfectly legitimate for the hardware to behave in such a way that some CPUs see the first ordering above, some see the second, and some see yet other ones. What *is* guaranteed is that no CPU will see a sequence in which 1B comes before 1A, and so on. The order each CPU does its writes is observed everywhere.

It is worth noting that some authors define processor consistency differently and do not require the second condition.

Weak Consistency

Our next model, **weak consistency**, does not even guarantee that writes from a single CPU are seen in order (Dubois et al., 1986). In a weakly consistent memory, one CPU might see 1A before 1B and another CPU might see 1A after 1B. However, to add some order to the chaos, weakly consistent memories have synchronization variables or a synchronization operation. When a synchronization is executed, all pending writes are finished and no new ones are started until all the old ones are done and the synchronization itself is done. In effect, a synchronization "flushes the pipeline" and brings the memory to a stable state with no operations pending. Synchronization operations are themselves sequentially consistent, that is, when multiple CPUs issue them, some order is chosen, but all CPUs see the same order.

In weak consistency, time is divided into well-defined epochs delimited by the (sequentially consistent) synchronizations, as illustrated in Fig. 8-25. No relative order is guaranteed for 1A and 1B, and different CPUs may see the two writes in different order, that is, one CPU may see 1A then 1B and another CPU may see 1B then 1A. This situation is permitted. However, all CPUs see 1B before 1C because the first synchronization operation forces 1A, 1B, and 2A to complete before 1C, 2B, 3A, or 3B is allowed to start. Thus by doing synchronization operations, software can force some order on the sequence of events, although not at zero cost since flushing the memory pipeline does take time and thus slows the machine down somewhat. Doing it too often can be a problem.

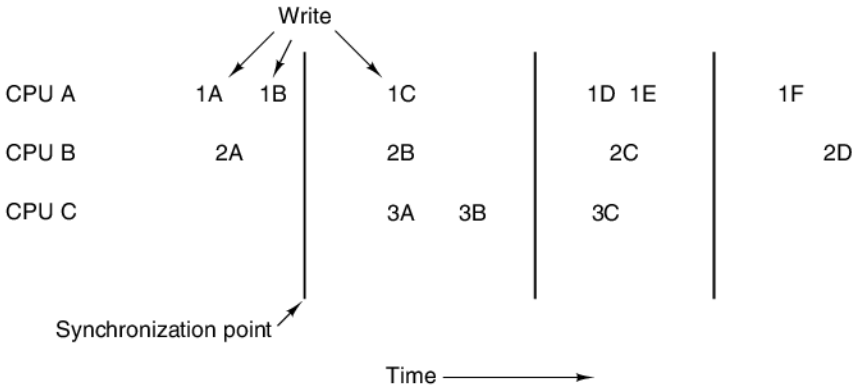


Figure 8-25. Weakly consistent memory uses synchronization operations to divide time into sequential epochs.

Release Consistency

Weak consistency has the problem that it is quite inefficient because it must finish off all pending memory operations and hold all new ones until the current ones are done. **Release consistency** improves matters by adopting a model akin to critical sections (Gharachorloo et al., 1990). The idea behind this model is that when a process exits a critical region it is not necessary to force all the writes to complete immediately. It is only necessary to make sure that they are done before any process enters that critical region again.

In this model, the synchronization operation offered by weak consistency is split into two different operations. To read or write a shared data variable, a CPU (i.e., its software) must first do an acquire operation on the synchronization variable to get exclusive access to the shared data. Then the CPU can use them as it wishes, reading and writing them at will. When it is done, the CPU does a release operation on the synchronization variable to indicate that it is finished. The release does not force pending writes to complete, but it itself does not complete until all previously issued writes are done. Furthermore, new memory operations are not prevented from starting immediately.

When the next acquire is issued, a check is made to see whether all previous release operations have completed. If not, the acquire is held up until they are all done (and hence all the writes done before them are all completed). In this way, if the next acquire occurs sufficiently long after the most recent release, it does not have to wait before starting and the critical region can be entered without delay. If it occurs too soon after a release, the acquire (and all the instructions following it) will be delayed until all pending releases are completed, thus guaranteeing that the variables in the critical section have been updated. This scheme is slightly more complicated than weak consistency, but it has the significant advantage of not delaying instructions as often in order to maintain consistency.

Memory consistency is not a done deal. Researchers are still proposing new models (Naeem et al., 2011, Sorin et al., 2011, and Tu et al., 2010).

8.3.3 UMA Symmetric Multiprocessor Architectures

The simplest multiprocessors are based on a single bus, as illustrated in Fig. 8-26(a). Two or more CPUs and one or more memory modules all use the same bus for communication. When a CPU wants to read a memory word, it first checks to see whether the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.

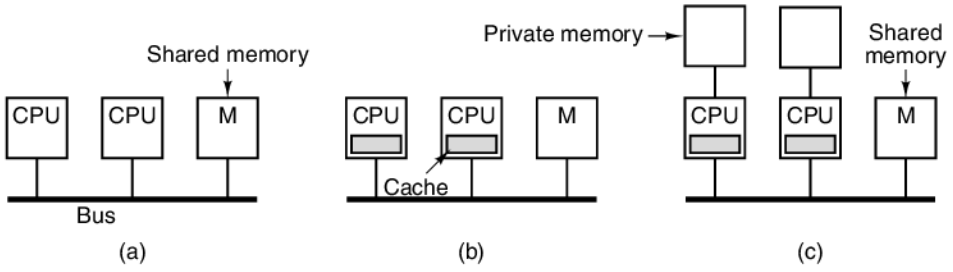


Figure 8-26. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle. Herein lies the problem with this design. With two or three CPUs, contention for the bus will be manageable; with 32 or 64 it will be unbearable. The system will be totally limited by the bandwidth of the bus, and most of the CPUs will be idle most of the time.

The solution is to add a cache to each CPU, as depicted in Fig. 8-26(b). The cache can be inside the CPU chip, next to the CPU chip, on the processor board, or some combination of all three. Since many reads can now be satisfied out of the local cache, there will be much less bus traffic, and the system can support more CPUs. Thus caching is a big win here. However, as we shall see in a moment, keeping the caches consistent with one another is not trivial.

Yet another possibility is the design of Fig. 8-26(c), in which each CPU has not only a cache but also a local, private memory which it accesses over a dedicated (private) bus. To use this configuration optimally, the compiler should place all the program text, strings, constants and other read-only data, stacks, and local variables in the private memories. The shared memory is then used only for writable shared variables. In most cases, this careful placement will greatly reduce bus traffic, but it does require active cooperation from the compiler.

Snooping Caches

While the performance arguments given above are certainly true, we have glossed a bit too quickly over a fundamental problem. Suppose that memory is sequentially consistent. What happens if CPU 1 has a line in its cache, and then CPU 2 tries to read a word in the same cache line? In the absence of any special rules, it, too, would get a copy in its cache. In principle, having the same line cached twice is acceptable. Now suppose that CPU 1 modifies the line and then, immediately thereafter, CPU 2 reads its copy of the line from its cache. It will get **stale data**, thus violating the contract between the software and memory. The program running on CPU 2 will not be happy.

This problem, known in the literature as the **cache coherence** or **cache consistency** problem, is extremely serious. Without a solution, caching cannot be used, and bus-oriented multiprocessors would be limited to two or three CPUs. As a consequence of its importance, many solutions have been proposed over the years (e.g., Goodman, 1983, and Papamarcos and Patel, 1984). Although all these caching algorithms, called **cache coherence protocols**, differ in the details, all of them prevent different versions of the same cache line from appearing simultaneously in two or more caches.

In all solutions, the cache controller is specially designed to allow it to eavesdrop on the bus, monitoring all bus requests from other CPUs and caches and taking action in certain cases. These devices are called **snooping caches** or sometimes **snoopy caches** because they "snoop" on the bus. The set of rules implemented by the caches, CPUs, and memory for preventing different versions of the data from appearing in multiple caches forms the cache coherence protocol. The unit of transfer and storage for a cache is called a **cache line** and is typically 32 or 64 bytes.

The simplest cache coherence protocol is called **write through**. It can best be understood by distinguishing the four cases shown in Fig. 8-27. When a CPU tries to read a word that is not in its cache (i.e., a read miss), its cache controller loads the line containing that word into the cache. The line is supplied by the memory, which in this protocol is always up to date. Subsequent reads (i.e., read hits) can be satisfied out of the cache.

Action	Local request	Remote request
Read miss	Fetch data from memory	
Read hit	Use data from local cache	
Write miss	Update data in memory	
Write hit	Update cache and memory	Invalidate cache entry

Figure 8-27. The write-through cache coherence protocol. The empty boxes indicate that no action is taken.

On a write miss, the word that has been modified is written to main memory. The line containing the word referenced is *not* loaded into the cache. On a write hit, the cache is updated and the word is written through to main memory in addition. The essence of this protocol is that all write operations result in the word being written going through to memory to keep memory up to date at all times.

Now let us look at all these actions again, but this time from the snooper's point of view, shown in the right-hand column of Fig. 8-27. Let us call the cache performing the actions cache 1 and the snooping cache cache 2. When cache 1 misses on a read, it makes a bus request to fetch a line from memory. Cache 2 sees this but does nothing. When cache 1 has a read hit, the request is satisfied locally, and no bus request occurs, so cache 2 is not aware of cache 1's read hits.

Writes are more interesting. If CPU 1 does a write, cache 1 will make a write request on the bus, both on misses and on hits. On all writes, cache 2 checks to see whether it has the word being written. If not, from its point of view this is a remote request/write miss and it does nothing. (To clarify a subtle point, note that in Fig. 8-27 a remote miss means that the word is not present in the snooper's cache; it does not matter whether it was in the originator's cache or not. Thus a single request may be a hit locally and a miss at the snooper, or vice versa.)

Now suppose that cache 1 writes a word that *is* present in cache 2's cache (remote request/write hit). If cache 2 does nothing, it will have stale data, so it marks the cache entry containing the newly modified word as being invalid. In effect, it removes the item from the cache. Because all caches snoop on all bus requests, whenever a word is written, the net effect is to update it in the originator's cache, update it in memory, and purge it from all the other caches. In this way, inconsistent versions are prevented.

Of course, cache 2's CPU is free to read the same word on the very next cycle. In that case, cache 2 will read the word from memory, which is up to date. At that point, cache 1, cache 2, and the memory will all have identical copies of it. If either CPU does a write now, the other one's cache will be purged, and memory will be updated.

Many variations on this basic protocol are possible. For example, on a write hit, the snooping cache normally invalidates its entry containing the word being written. Alternatively, it could accept the new value and update its cache instead of marking it as invalid. Conceptually, updating the cache is the same as invalidating it followed by reading the word from memory. In all cache protocols, a choice must be made between an **update strategy** and an **invalidate strategy**. These protocols perform differently under different loads. Update messages carry payloads and are thus larger than invalidates but may prevent future cache misses.

Another variant is loading the snooping cache on write misses. The correctness of the algorithm is not affected by loading it, only the performance. The question is: "What is the probability that a word just written will be written again soon?" If it is high, there is something to be said for loading the cache on write misses, known as a **write-allocate policy**. If it is low, it is better not to update on

write misses. If the word is *read* soon, it will be loaded by the read miss anyway; little is gained by loading it on the write miss.

As with many simple solutions, this one is inefficient. Every write operation goes to memory over the bus, so with a modest number of CPUs, the bus will still become a bottleneck. To keep the bus traffic within bounds, other cache protocols have been devised. They all have the property that not all writes go directly through to memory. Instead, when a cache line is modified, a bit is set inside the cache noting that the cache line is correct but memory is not. Eventually, such a dirty line has to be written back to memory, but possibly after many writes have been made to it. This type of protocol is known as a **write-back protocol**.

The MESI Cache Coherence Protocol

One popular write-back cache coherence protocol is called **MESI**, after the initials of the names of the four states (M, E, S, and I) that it uses (Papamarcos and Patel, 1984). It is based on the earlier **write-once protocol** (Goodman, 1983). The MESI protocol is used by the Core i7 and many other CPUs for snooping on the bus. Each cache entry can be in one of the following four states:

1. Invalid – The cache entry does not contain valid data.
2. Shared – Multiple caches may hold the line; memory is up to date.
3. Exclusive– No other cache holds the line; memory is up to date.
4. Modified – The entry is valid; memory is invalid; no copies exist.

When the CPU is initially booted, all cache entries are marked invalid. The first time memory is read, the line referenced is fetched into the cache of the CPU reading memory and marked as being in the E (exclusive) state, since it is the only copy in a cache, as illustrated in Fig. 8-28(a) for the case of CPU 1 reading line *A*. Subsequent reads by that CPU use the cached entry and do not go over the bus. Another CPU may also fetch the same line and cache it, but by snooping, the original holder (CPU 1) sees that it is no longer alone and announces on the bus that it also has a copy. Both copies are marked as being in the S (shared) state, as shown in Fig. 8-28(b). In other words, the S state means that the line is in one or more caches for reading and memory is up to date. Subsequent reads by a CPU to a line it has cached in the S state do not use the bus and do not cause the state to change.

Now consider what happens if CPU 2 writes to the cache line it is holding in S state. It puts out an invalidate signal on the bus, telling all other CPUs to discard their copies. The cached copy now goes to M (modified) state, as shown in Fig. 8-28(c). The line is not written to memory. It is worth noting that if a line is in E state when it is written, no bus signal is needed to invalidate other caches because it is known that no other copies exist.

Next consider what happens if CPU 3 reads the line. CPU 2, which now owns the line, knows that the copy in memory is not valid, so it asserts a signal on the

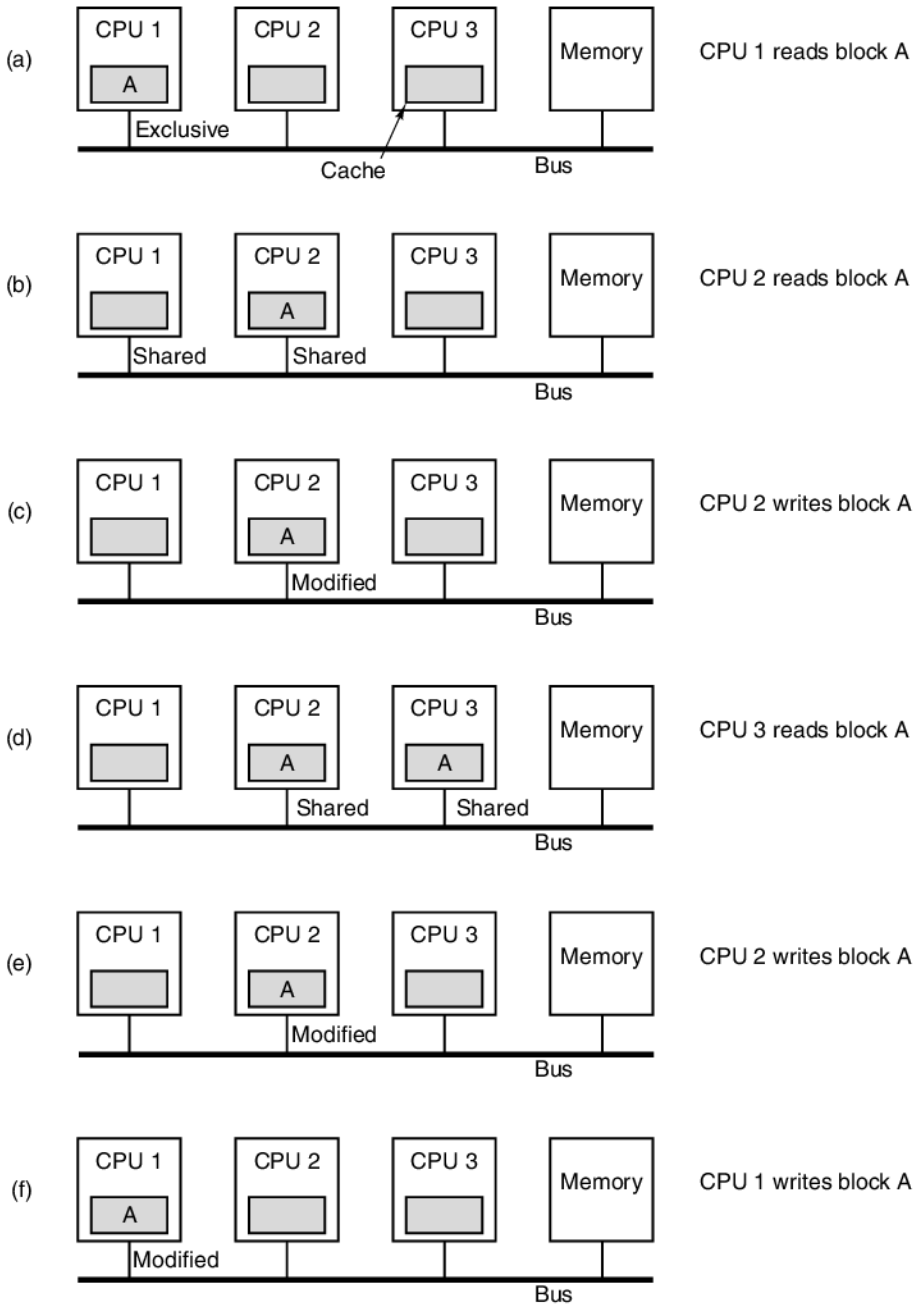


Figure 8-28. The MESI cache coherence protocol.

bus telling CPU 3 to please wait while it writes its line back to memory. When it is finished, CPU 3 fetches a copy, and the line is marked as shared in both caches, as shown in Fig. 8-28(d). After that, CPU 2 writes the line again, which invalidates the copy in CPU 3's cache, as shown in Fig. 8-28(e).

Finally, CPU 1 writes to a word in the line. CPU 2 sees that a write is being attempted and asserts a bus signal telling CPU 1 to please wait while it writes its line back to memory. When it is finished, it marks its own copy as invalid, since it knows another CPU is about to modify it. At this point we have the situation in which a CPU is writing to an uncached line. If the write-allocate policy is in use, the line will be loaded into the cache and marked as being in the M state, as shown in Fig. 8-28(f). If the write-allocate policy is not in use, the write will go directly to memory and the line will not be cached anywhere.

UMA Multiprocessors Using Crossbar Switches

Even with all possible optimizations, the use of a single bus limits the size of a UMA multiprocessor to about 16 or 32 CPUs. To go beyond that, a different kind of interconnection network is needed. The simplest circuit for connecting n CPUs to k memories is the **crossbar switch**, shown in Fig. 8-28. Crossbar switches have been used for decades within telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way.

At each intersection of a horizontal (incoming) and vertical (outgoing) line is a **crosspoint**. A crosspoint is a small switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 8-29(a) we see three crosspoints closed simultaneously, allowing connections between the (CPU, memory) pairs (001, 000), (101, 101), and (110, 010) at the same time. Many other combinations are also possible. In fact, the number of combinations is equal to the number of different ways eight rooks can be safely placed on a chess board.

One of the nicest properties of the crossbar switch is that it is a **nonblocking network**, meaning that no CPU is ever denied the connection it needs because some crosspoint or line is already occupied (assuming the memory module itself is available). Furthermore, no advance planning is needed. Even if seven arbitrary connections are already set up, it is always possible to connect the remaining CPU to the remaining memory. We will later see interconnection schemes that do not have these properties.

One of the worst properties of the crossbar switch is that the number of crosspoints grows as n^2. For medium-sized systems, a crossbar design is workable. We will discuss one such design, the Sun Fire E25K, later in this chapter. However, with 1000 CPUs and 1000 memory modules, we need a million crosspoints. Such a large crossbar switch is not feasible. We need something quite different.

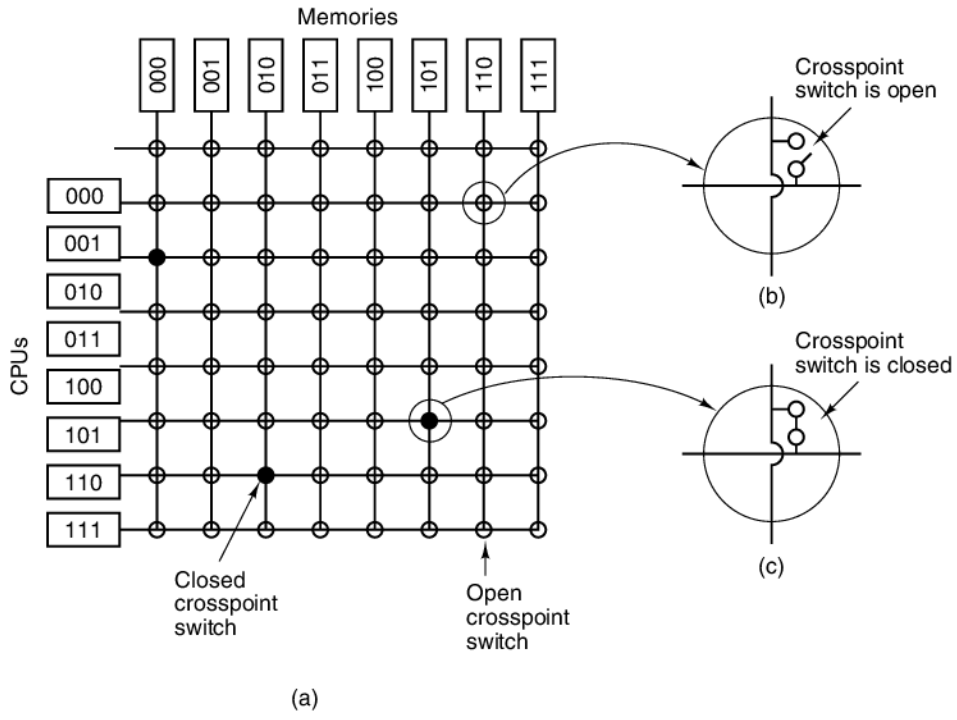


Figure 8-29. (a) An 8×8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

UMA Multiprocessors Using Multistage Switching Networks

That "something quite different" can be based on the humble 2×2 switch shown in Fig. 8-30(a). This switch has two inputs and two outputs. Messages arriving on either input line can be switched to either output line. For our purposes here, messages will contain up to four parts, as shown in Fig. 8-30(b). The *Module* field tells which memory to use. The *Address* specifies an address within a module. The *Opcode* gives the operation, such as READ or WRITE. Finally, the optional *Value* field may contain an operand, such as a 32-bit word to be written on a WRITE. The switch inspects the *Module* field and uses it to determine if the message should be sent on *X* or on *Y*.

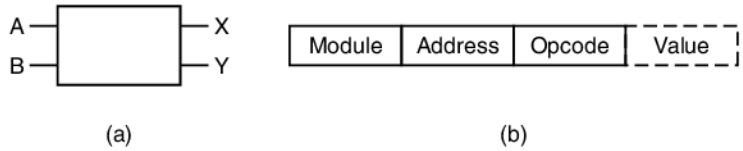


Figure 8-30. (a) A 2×2 switch. (b) A message format.

Our 2×2 switches can be arranged in many ways to build larger **multistage switching networks**. One option is the no-frills, economy class **omega network**,

illustrated in Fig. 8-31. Here we have connected eight CPUs to eight memories using 12 switches. More generally, for n CPUs and n memories we would need $\log_2 n$ stages, with $n/2$ switches per stage, for a total of $(n/2)\log_2 n$ switches, which is a lot better than n^2 crosspoints, especially for large values of n.

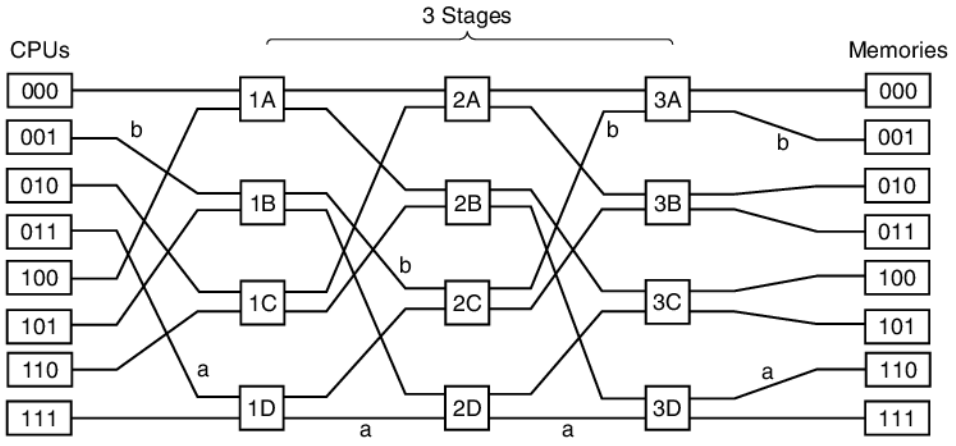


Figure 8-31. An omega switching network.

The wiring pattern of the omega network is often called the **perfect shuffle**, since the mixing of the signals at each stage resembles a deck of cards being cut in half and then mixed card-for-card. To see how the omega network works, suppose that CPU 011 wants to read a word from memory module 110. The CPU sends a READ message to switch 1D containing 110 in the *Module* field. The switch takes the first (i.e., leftmost) bit of 110 and uses it for routing. A 0 routes to the upper output and a 1 routes to the lower one. Since this bit is a 1, the message is routed via the lower output to 2D.

All the second-stage switches, including 2D, use the second bit for routing. This, too, is a 1, so the message is now forwarded via the lower output to 3D. Here the third bit is tested and found to be a 0. Consequently, the message goes out on the upper output and arrives at memory 110, as desired. The path followed by this message is marked in Fig. 8-31 by the letter *a*.

As the message moves through the switching network, the bits at the left-hand end of the module number are no longer needed. They can be put to good use by recording the incoming line number there, so the reply can find its way back. For path *a*, the incoming lines are 0 (upper input to 1D), 1 (lower input to 2D), and 1 (lower input to 3D), respectively. The reply is routed back using 011, only reading it from right to left this time.

While all this is going on, CPU 001 wants to write a word to memory module 001. An analogous process happens here, with the message routed via the upper, upper, and lower outputs, respectively, marked by the letter *b*. When it arrives, its

Module field reads 001, representing the path it took. Since these requests do not use any of the same switches, lines, or memory modules, they can go in parallel.

Now consider what would happen if CPU 000 simultaneously wanted to access memory module 000. Its request would come into conflict with CPU 001's request at switch 3A. One of them would have to wait. Unlike the crossbar switch, the omega network is a **blocking network**. Not every set of requests can be processed simultaneously. Conflicts can occur over the use of a wire or a switch, as well as between requests *to* memory and replies *from* memory.

It is clearly desirable to spread the memory references uniformly across the modules. One common technique is to use the low-order bits as the module number. Consider, for example, a byte-oriented address space for a computer that mostly accesses 32-bit words. The 2 low-order bits will usually be 00, but the next 3 bits will be uniformly distributed. By using these 3 bits as the module number, consecutively addressed words will be in consecutive modules. A memory system in which consecutive words are in different modules is said to be **interleaved**. Interleaved memories maximize parallelism because most memory references are to consecutive addresses. It is also possible to design switching networks that are nonblocking and that offer multiple paths from each CPU to each memory module, to spread the traffic better.

8.3.4 NUMA Multiprocessors

It should be clear by now that single-bus UMA multiprocessors are generally limited to no more than a few dozen CPUs and crossbar or switched multiprocessors need a lot of (expensive) hardware and are not that much bigger. To get to more than 100 CPUs, something has to give. Usually, what gives is the idea that all memory modules have the same access time. This concession leads to the idea of **NUMA** (**NonUniform Memory Access**) multiprocessors. Like their UMA cousins, they provide a single address space across all the CPUs, but unlike the UMA machines, access to local memory modules is faster than access to remote ones. Thus all UMA programs will run without change on NUMA machines, but the performance will be worse than on a UMA machine at the same clock speed.

All NUMA machines have three key characteristics that together distinguish them from other multiprocessors:

1. There is a single address space visible to all CPUs.
2. Access to remote memory done using LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.

When the access time to remote memory is not hidden (because there is no caching), the system is called **NC-NUMA**. When coherent caches are present, the

system is called **CC-NUMA** (at least by the hardware people). The software people often call it **hardware DSM** because it is basically the same as software distributed shared memory but implemented by the hardware using a small page size.

One of the first NC-NUMA machines (although the name had not yet been coined) was the Carnegie-Mellon Cm*, illustrated in simplified form in Fig. 8-32 (Swan et al., 1977). It consisted of a collection of LSI-11 CPUs, each with some memory addressed over a local bus. (The LSI-11 was a single-chip version of the DEC PDP-11, a minicomputer popular in the 1970s.) In addition, the LSI-11 systems were connected by a system bus. When a memory request came into the (specially modified) MMU, a check was made to see if the word needed was in the local memory. If so, a request was sent over the local bus to get the word. If not, the request was routed over the system bus to the system containing the word, which then responded. Of course, the latter took much longer than the former. While a program could run happily out of remote memory, it took 10 times longer to execute than the same program running out of local memory.

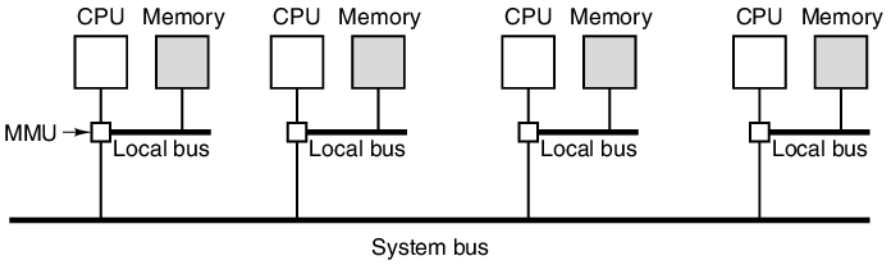


Figure 8-32. A NUMA machine based on two levels of buses. The Cm* was the first multiprocessor to use this design.

Memory coherence is guaranteed in an NC-NUMA machine because no caching is present. Each word of memory lives in exactly one location, so there is no danger of one copy having stale data: there are no copies. Of course, it now matters a great deal which page is in which memory because the performance penalty for being in the wrong place is so high. Consequently, NC-NUMA machines use elaborate software to move pages around to maximize performance.

Typically, a daemon process called a **page scanner** runs every few seconds. Its job is to examine the usage statistics and move pages around in an attempt to improve performance. If a page appears to be in the wrong place, the page scanner unmaps it so that the next reference to it will cause a page fault. When the fault occurs, a decision is made about where to place the page, possibly in a different memory. To prevent thrashing, usually there is some rule saying that once a page is placed, it is frozen in place for a time ΔT. Various algorithms have been studied, but the conclusion is that no one algorithm performs best under all circumstances (LaRowe and Ellis, 1991). Best performance depends on the application.

Cache Coherent NUMA Multiprocessors

Multiprocessor designs such as that of Fig. 8-32 do not scale well because they do not do caching. Having to go to the remote memory every time a nonlocal memory word is accessed is a major performance hit. However, if caching is added, then cache coherence must also be added. One way to provide cache coherence is to snoop on the system bus. Technically, doing this is not difficult, but beyond a certain number of CPUs, it becomes infeasible. To build really large multiprocessors, a fundamentally different approach is needed.

The most popular approach for building large **CC-NUMA** (**Cache Coherent NUMA**) multiprocessors currently is the **directory-based multiprocessor**. The idea is to maintain a database telling where each cache line is and what its status is. When a cache line is referenced, the database is queried to find out where it is and whether it is clean or dirty (modified). Since this database must be queried on every single instruction that references memory, it must be kept in extremely fast special-purpose hardware that can respond in a fraction of a bus cycle.

To make the idea of a directory-based multiprocessor somewhat more concrete, let us consider a simple (hypothetical) example, a 256-node system, each node consisting of one CPU and 16 MB of RAM connected to the CPU via a local bus. The total memory is 2^{32} bytes, divided up into 2^{26} cache lines of 64 bytes each. The memory is statically allocated among the nodes, with 0–16M in node 0, 16–32M in node 1, and so on. The nodes are connected by an interconnection network, as shown in Fig. 8-33(a). This network could be a grid, hypercube, or other topology. Each node also holds the directory entries for the 2^{18} 64-byte cache lines comprising its 2^{24}-byte memory. For the moment, we will assume that a line can be held in at most one cache.

To see how the directory works, let us trace a LOAD instruction from CPU 20 that references a cached line. First the CPU issuing the instruction presents it to its MMU, which translates it to a physical address, say, 0x24000108. The MMU splits this address into the three parts shown in Fig. 8-33(b). In decimal, the three parts are node 36, line 4, and offset 8. The MMU sees that the memory word referenced is from node 36, not node 20, so it sends a request message through the interconnection network to the line's home node, 36, asking whether its line 4 is cached, and if so, where.

When the request arrives at node 36 over the interconnection network, it is routed to the directory hardware. The hardware indexes into its table of 2^{18} entries, one for each of its cache lines and extracts entry 4. From Fig. 8-33(c) we see that the line is not cached, so the hardware fetches line 4 from the local RAM, sends it back to node 20, and updates directory entry 4 to indicate that the line is now cached at node 20.

Now let us consider a second request, this time asking about node 36's line 2. From Fig. 8-33(c) we see that this line is cached at node 82. At this point the hardware could update directory entry 2 to say that the line is now at node 20 and then

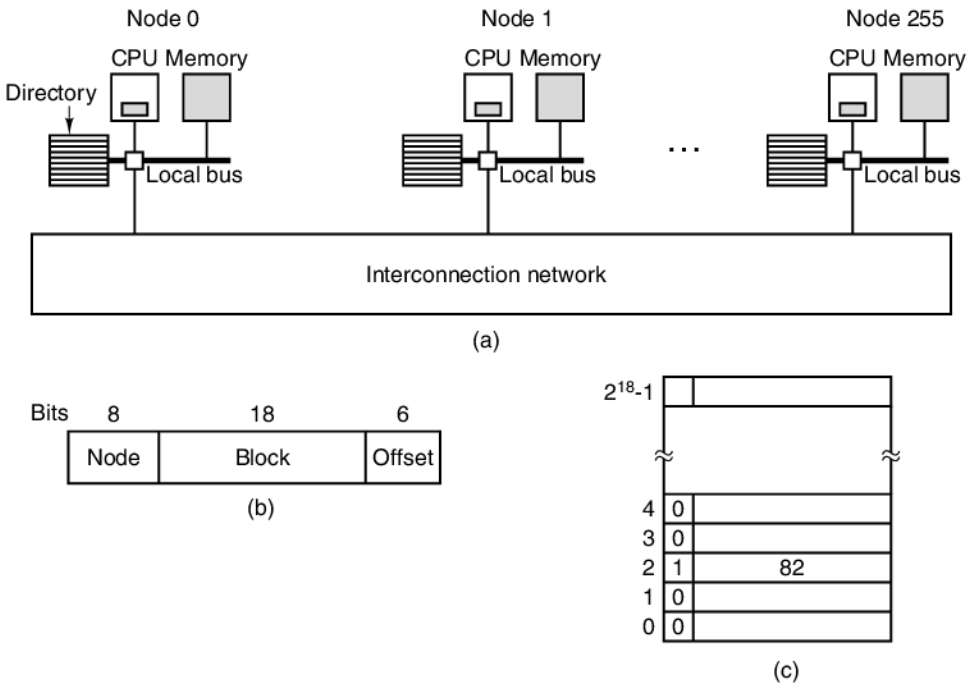


Figure 8-33. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

send a message to node 82 instructing it to pass the line to node 20 and invalidate its cache. Note that even a so-called "shared-memory multiprocessor" has a lot of message passing going on under the hood.

As a quick aside, let us calculate how much memory is being taken up by the directories. Each node has 16 MB of RAM and 2^{18} 9-bit entries to keep track of that RAM. Thus the directory overhead is about 9×2^{18} bits divided by 16 MB or about 1.76 percent, which is generally acceptable (although it has to be high-speed memory, which increases its cost). Even with 32-byte cache lines the overhead would only be 4 percent. With 128-byte cache lines, it would be under 1 percent.

An obvious limitation of this design is that a line can be cached at only one node. To allow lines to be cached at multiple nodes, we would need some way of locating all of them, for example, to invalidate or update them on a write. Various options are possible to allow caching at several nodes at the same time.

One possibility is to give each directory entry k fields for specifying other nodes, thus allowing each line to be cached at up to k nodes. A second possibility is to replace the node number in our simple design with a bit map, with one bit per node. In this option there is no limit on how many copies there can be, but there is a substantial increase in overhead. Having a directory with 256 bits for each

64-byte (512-bit) cache line implies an overhead of over 50 percent. A third possibility is to keep one 8-bit field in each directory entry and use it as the head of a linked list that threads all the copies of the cache line together. This strategy requires extra storage at each node for the linked list pointers, and it also requires following a linked list to find all the copies when that is needed. Each possibility has its own advantages and disadvantages, and all three have been used in real systems.

Another improvement to the directory design is to keep track of whether the cache line is clean (home memory is up to date) or dirty (home memory is not up to date). If a read request comes in for a clean cache line, the home node can satisfy the request from memory, without having to forward it to a cache. A read request for a dirty cache line, however, must be forwarded to the node holding the cache line because only it has a valid copy. If only one cache copy is allowed, as in Fig. 8-33, there is no real advantage to keeping track of its cleanliness, because any new request requires a message to be sent to the existing copy to invalidate it.

Of course, keeping track of whether each cache line is clean or dirty implies that when a cache line is modified, the home node has to be informed, even if only one cache copy exists. If multiple copies exist, modifying one of them requires the rest to be invalidated, so some protocol is needed to avoid race conditions. For example, to modify a shared cache line, one of the holders might have to request exclusive access *before* modifying it. Such a request would cause all other copies to be invalidated before permission was granted. Other performance optimizations for CC-NUMA machines are discussed in Cheng and Carter (2008).

The Sun Fire E25K NUMA Multiprocessor

As an example of a shared-memory NUMA multiprocessor, let us study the Sun Microsystems Sun Fire family. Although it contains various models, we will focus on the E25K, which has 72 UltraSPARC IV CPU chips. An UltraSPARC IV is essentially a pair of UltraSPARC III processors that share a common cache and memory. The E15K is essentially the same system except with uniprocessor instead of dual-processor CPU chips. Smaller members exist as well, but from our point of view, what is interesting is how the one with the most CPUs works.

The E25K system consists of up to 18 boardsets, each boardset consisting of a CPU-memory board, an I/O board with four PCI slots, and an expander board that couples the CPU-memory board with the I/O board and joins the pair to the centerplane, which holds the boards and contains the switching logic. Each CPU-memory board contains four CPU chips and four 8-GB RAM modules. Consequently, each CPU-memory board on the E25K holds eight CPUs and 32 GB of RAM (four CPUs and four 32 GB of RAM on the E15K). A full E25K thus contains 144 CPUs, 576 GB of RAM, and 72 PCI slots. It is illustrated in Fig. 8-34. Interestingly enough, the number 18 was chosen due to packaging constraints: a system with 18 boardsets was the largest one that could fit through a doorway in one piece.

While programmers just think about 0s and 1s, engineers have to worry about things like how the customer will get the product through the door and into the building.

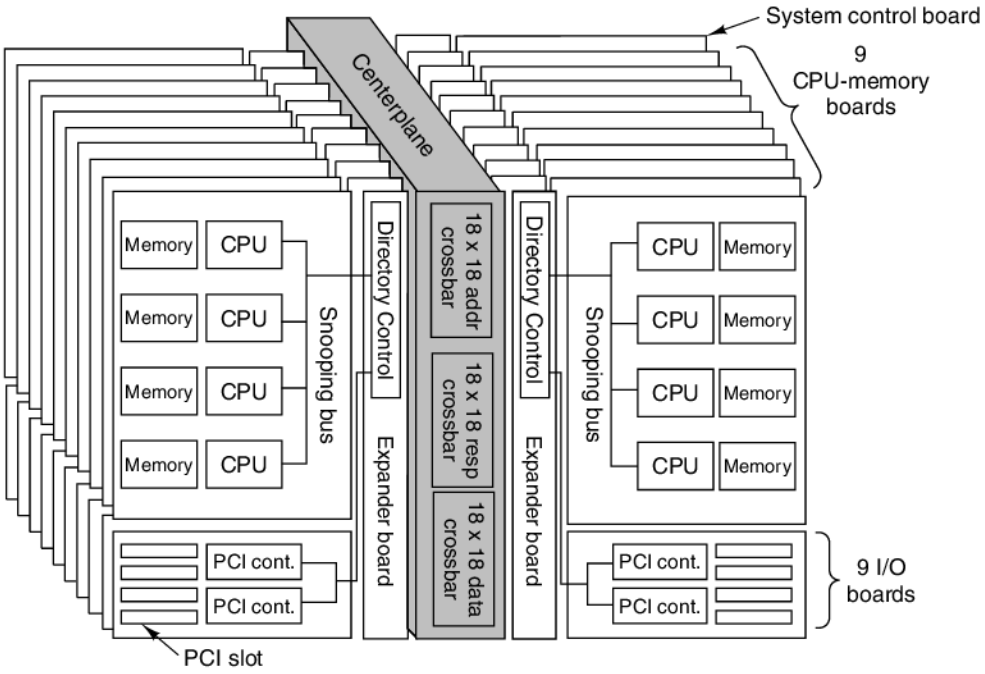


Figure 8-34. The Sun Microsystems E25K multiprocessor.

The centerplane is composed of a set of three 18 × 18 crossbar switches for connecting the 18 boardsets. One crossbar is for the address lines, one is for responses, and one is for data transfer. In addition to the 18 expander boards, the centerplane also has a system control boardset plugged into it. This boardset has a single CPU but also interfaces to the CD-ROM, tape, serial lines, and other peripheral devices needed for booting, maintaining, and controlling the system.

The heart of any multiprocessor is the memory subsystem. How does one connect 144 CPUs to the distributed memory? The straightforward ways—a big shared snooping bus or a 144 × 72 crossbar switch—do not work well. The former fails due to the bus being a bottleneck and the latter fails because the switch is too difficult and expensive to build. Thus large multiprocessors such as the E25K are forced to use a more complex memory subsystem.

At the boardset level, snooping logic is used so all local CPUs can check all memory requests coming from the boardset to references to blocks they currently have cached. Thus when a CPU needs a word from memory, it first converts the virtual address to a physical address and checks its cache. (Physical addresses are 43 bits, but packaging restrictions limit memory to 576 GB.) If the cache block it

needs is in its own cache, the word is returned. Otherwise, the snooping logic checks if a copy of that word is available somewhere else on the boardset. If so, the request is satisfied. If not, the request is passed on via the 18×18 address crossbar switch as described below. The snooping logic can do one snoop per clock cycle. The system clock runs at 150 MHz, so it is possible to perform 150 million snoops/sec per boardset or 2.7 billion snoops/sec system wide.

Although the snooping logic is logically a bus, as portrayed in Fig. 8-34, physically it is a device tree, with commands being relayed up and down the tree. When a CPU or PCI board puts out an address, it goes to an address repeater via a point-to-point connection, as shown in Fig. 8-35. The two address repeaters converge on the expander board, where the addresses are sent back down the tree for each device to check for hits. This arrangement is used to avoid having a bus that involves three boards.

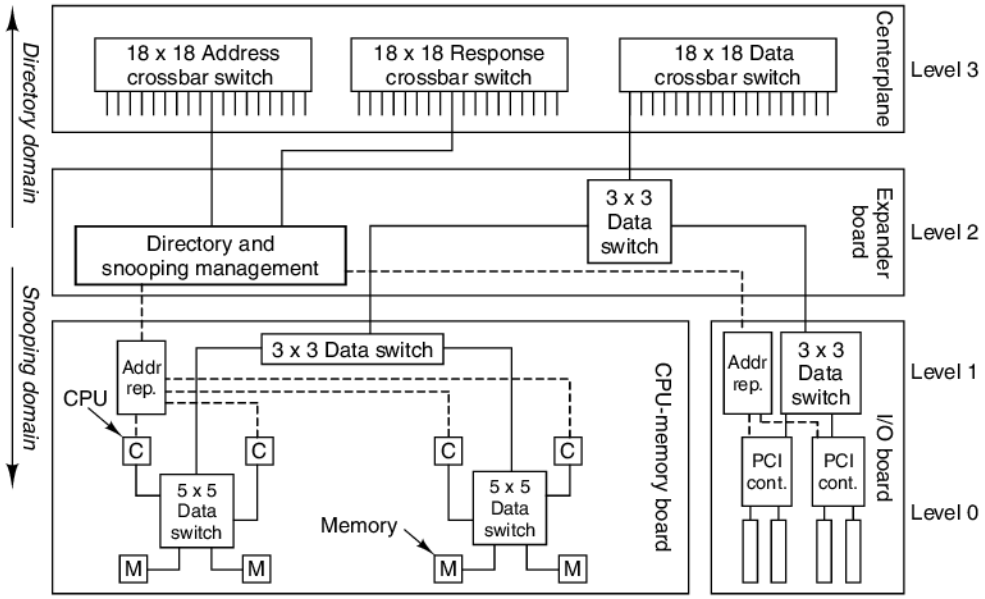


Figure 8-35. The Sun Fire E25K uses a four-level interconnect. Dashed lines are address paths. Solid lines are data paths.

Data transfers use a four-level interconnect as depicted in Fig. 8-35. This design was chosen for high performance. At level 0, pairs of CPU chips and memories are connected by a small crossbar switch that also has a connection to level 1. The two groups of CPU-memory pairs are connected by a second crossbar switch at level 1. The crossbar switches are custom ASICs. For all of them, all the inputs are available on both the rows and the columns, although not all combinations are used (or even make sense). All the switching logic on the boards is built from 3×3 crossbars.

Each boardset consists of three boards: the CPU-memory board, the I/O board, and the expander board, which connects the other two. The level 2 interconnect is another 3×3 crossbar switch (on the expander board) that joins the actual memory to the I/O ports (which are memory mapped on all UltraSPARCs). All data transfers to or from the boardset, whether to memory or to an I/O port, pass through the level 2 switch. Finally, data that have to be transferred to or from a remote board pass through an 18×18 data crossbar switch at level 3. Data transfers are done 32 bytes at a time, so it takes two clock cycles to transfer 64 bytes, the usual transfer unit.

Having looked at how the components are arranged, let us now consider how the shared memory operates. At the bottom level, the 576 GB of memory is split into 2^{29} blocks of 64 bytes each. These blocks are the atomic units of the memory system. Each block has a home board where it lives when not in use elsewhere. Most blocks are on their home board most of the time. However, when a CPU needs a memory block, either from its own board or one of the 17 remote ones, it first requests a copy for its own cache, then accesses the cached copy. Although each CPU chip on the E25K contains two CPUs, they share a single physical cache and thus share all the blocks contained in it.

Each memory block and cache line of each CPU chip can be in one of three states:

1. Exclusive access (for writing).
2. Shared access (for reading).
3. Invalid (i.e., empty).

When a CPU needs to read or write a memory word, it first checks its own cache. Failing to find the word there, it issues a local request for the physical address that is broadcast only on its own boardset. If a cache on the boardset has the needed line, the snooping logic detects the hit and responds to the request. If the line is in exclusive mode, it is transferred to the requester and the original copy marked invalid. If it is in shared mode, the cache does not respond since memory always responds when a cache line is clean.

If the snooping logic cannot find the cache line or it is present and shared, it sends a request over the centerplane to the home board asking where the memory block is. The state of each memory block is stored in the block's ECC bits, so the home board can immediately determine its state. If the block is either unshared or shared with one or more remote boards, the home memory will be up to date, and the request can be satisfied from the home board's memory. In this case, a copy of the case line is transmitted over the data crossbar switch in two clock cycles, eventually arriving at the requesting CPU.

If the request was for reading, an entry is made in the directory at the home board noting that a new customer is sharing the cache line and the transaction is

finished. However, if the request was for writing, an invalidation message must be sent to all other boards (if any) holding a copy of it. In this way, the board making the write request ends up with the only copy.

Now consider the case in which the requested block is in exclusive state located on a different board. When the home board gets the request, it looks up the location of the remote board in the directory and sends the requester a message telling where the cache line is. The requester now sends the request to the correct boardset. When the request arrives, the board sends back the cache line. If it was a read request, the line is marked shared and a copy sent back to the home board. If it was a write request, the responder invalidates its copy so the new requester has an exclusive copy.

Since each board has 2^{29} memory blocks, it would take a directory with 2^{29} entries to keep track of them all in the worst case. Since the directory is much smaller than 2^{29}, it could happen that there is no room in the directory (which is searched associatively) for some entries. In this case, the home directory has to locate the block by broadcasting a request for it to all the other 17 boards. The response crossbar switch plays a role in the directory coherence and update protocol by handling much of the reverse traffic back to the sender. Splitting the protocol traffic over two buses (address and response) and the data over a third bus increases the throughput of the system.

By distributing the load over multiple devices on different boards, the Sun Fire E25K is able to achieve very high performance. In addition to the 2.7 billion snoops/sec mentioned above, the centerplane can handle up to nine simultaneous transfers, with nine boards sending and nine boards receiving. Since the data crossbar is 32 bytes wide, on every clock cycle 288 bytes can be moved through the centerplane. At a clock rate of 150 MHz, this gives a peak aggregate bandwidth of 40 GB/sec when all accesses are remote. If the software can place pages in such a way to ensure that most accesses are local, then the system bandwidth can be appreciably higher than 40 GB/sec.

For more technical information about the Sun Fire, see Charlesworth (2002) and Charlesworth (2001).

In 2009 Oracle purchased Sun Microsystems, and they have continued development of SPARC-based servers. The SPARC Enterprise M9000 is the successor to the E25K. The M9000 incorporates faster quad-core SPARC processors, plus additional memory and PCIe slots. A fully equipped M9000 server contains 256 SPARC processors, 4 TB of DRAM, and 128 PCIe I/O interfaces.

8.3.5 COMA Multiprocessors

NUMA and CC-NUMA machines have the disadvantage that references to remote memory are much slower than those to local memory. In CC-NUMA, this performance difference is hidden to some extent by the caching. Nevertheless, if

the amount of remote data needed greatly exceeds the cache capacity, cache misses will occur constantly and performance will be poor.

Thus we have a situation that UMA machines have excellent performance but are limited in size and are quite expensive. NC-NUMA machines scale to somewhat larger sizes but require manual or semi-automated placement of pages, often with mixed results. The problem is that it is hard to predict which pages will be needed where, and in any case, a page is often too large a unit to move around. CC-NUMA machines, such as the Sun Fire E25K, may experience poor performance if many CPUs need a lot of remote data. All in all, each of these designs has serious limitations.

An alternative kind of multiprocessor tries to get around all these problems by using each CPU's main memory as a cache. In this design, called **COMA** (**Cache Only Memory Access**), pages do not have fixed home machines, as they do in NUMA and CC-NUMA machines. In fact, pages are not significant at all.

Instead, the physical address space is split into cache lines, which migrate around the system on demand. Blocks do not have home machines. Like nomads in some Third World countries, home is where you are right now. A memory that just attracts lines as needed is called an **attraction memory**. Using the main RAM as a big cache greatly increases the hit rate, hence the performance.

Unfortunately, as usual, there is no such thing as a free lunch. COMA systems introduce two new problems:

1. How are cache lines located?
2. When a line is purged, what happens if it is the last copy?

The first problem relates to the fact that after the MMU has translated a virtual address to a physical address, if the line is not in the true hardware cache, there is no easy way to tell if it is in main memory at all. The paging hardware does not help here at all because each page is made up of many individual cache lines that wander around independently. Furthermore, even if it is known that a line is not in main memory, where is it then? It is not possible to just ask the home machine, because there is no home machine.

Some solutions to the location problem have been proposed. To see if a cache line is in main memory, new hardware could be added to keep track of the tag for each cached line. The MMU could then compare the tag for the line needed to the tags for all the cache lines in memory to look for a hit. This solution needs additional hardware.

A somewhat different solution is to map entire pages in but not require that all the cache lines be present. In this solution, the hardware would need a bit map per page, giving one bit per cache line indicating the line's presence or absence. In this design, called **simple COMA** if a cache line is present, it must be in the right position in its page, but if it is not present, any attempt to use it causes a trap to allow the software to go find it and bring it in.

This leads us to finding lines that are really remote. One solution is to give each page a home machine in terms of where its directory entry is, but not where the data are. Then a message can be sent to the home machine to at least locate the cache line. Other schemes involve organizing memory as a tree and searching upward until the line is found.

The second problem in the list above relates to not purging the last copy. As in CC-NUMA, a cache line may be at multiple nodes at once. When a cache miss occurs, a line must be fetched, which usually means a line must be thrown out. What happens if the line chosen happens to be the last copy? In that case, it cannot be thrown out.

One solution is to go back to the directory and check to see if there are other copies. If so, the line can be safely thrown out. Otherwise, it has to be migrated somewhere else. Another solution is to label one copy of each cache line as the master copy and never throw it out. This solution avoids the need to check with the directory. All in all, COMA offers promise to provide better performance than CC-NUMA, but few COMA machines have been built, so more experience is needed. The first two COMA machines built were the KSR-1 (Burkhardt et al., 1992) and the Data Diffusion Machine (Hagersten et al., 1992). More recent papers on COMA are Vu et al. (2008) and Zhang and Jesshope (2008).

8.4 MESSAGE-PASSING MULTICOMPUTERS

As we saw in Fig. 8-23, the two kinds of MIMD parallel processors are multiprocessors and multicomputers. In the previous section we studied multiprocessors. We saw that they appear to the operating system as having shared memory that can be accessed using ordinary LOAD and STORE instructions. This shared memory can be implemented in many ways as we have seen, including snooping buses, data crossbars, multistage switching networks, and various directory-based schemes. Nevertheless, programs written for a multiprocessor can just access any location in memory without knowing anything about the internal topology or implementation scheme. This illusion is what makes multiprocessors so attractive and why programmers like this programming model.

On the other hand, multiprocessors also have their limitations, which is why multicomputers are important, too. First and foremost, multiprocessors do not scale to large sizes. We saw the enormous amount of hardware Sun had to use to get the E25K to scale to 72 CPUs. In contrast, we will study a multicomputer below that has 65,536 CPUs. It will be years before anyone builds a commercial 65,536-node multiprocessor. By then million-node multicomputers will be in use.

In addition, memory contention in a multiprocessor can severely affect performance. If 100 CPUs are all trying to read and write the same variables constantly, contention for the various memories, buses, and directories can cause an enormous performance hit.

As a consequence of these and other factors, there is a great deal of interest in building and using parallel computers in which each CPU has its own private memory, not directly accessible to any other CPU. These are the multicomputers. Programs on multicomputer CPUs interact using primitives like send and receive to explicitly pass messages because they cannot get at each other's memory with LOAD and STORE instructions. This difference completely changes the programming model.

Each node in a multicomputer consists of one or a few CPUs, some RAM (conceivably shared among the CPUs at that node only), a disk and/or other I/O devices, and a communication processor. The communication processors are connected by a high-speed interconnection network of the types we discussed in Sec. 8.3.3. Many different topologies, switching schemes, and routing algorithms are used. What all multicomputers have in common is that when an application program executes the send primitive, the communication processor is notified and transmits a block of user data to the destination machine (possibly after first asking for and getting permission). A generic multicomputer is shown in Fig. 8-36.

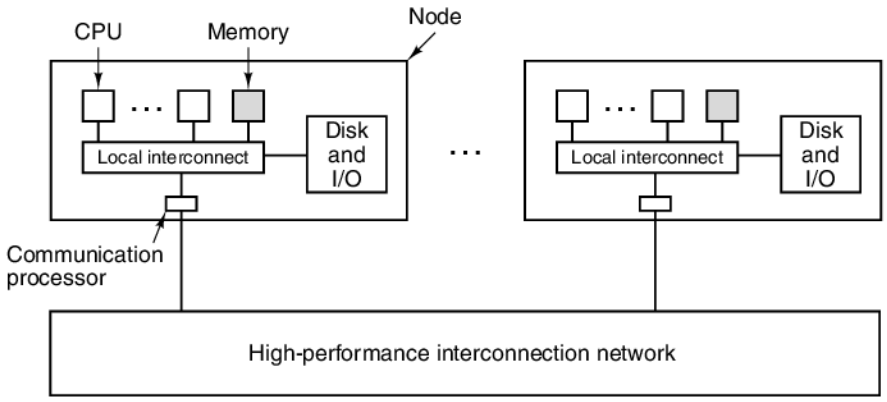


Figure 8-36. A generic multicomputer.

8.4.1 Interconnection Networks

In Fig. 8-36 we see that multicomputers are held together by interconnection networks. Now it is time to look more closely at these interconnection networks. Interestingly enough, multiprocessors and multicomputers are surprisingly similar in this respect because multiprocessors often have multiple memory modules that must also be interconnected with one another and with the CPUs. Thus the material in this section frequently applies to both kinds of systems.

The fundamental reason why multiprocessor and multicomputer interconnection networks are similar is that at the very bottom both of them use message

passing. Even on a single-CPU machine, when the processor wants to read or write a word, what it typically does is assert certain lines on the bus and wait for a reply. This action is fundamentally like message passing: the initiator sends a request and waits for a response. In large multiprocessors, communication between CPUs and remote memory almost always consists of the CPU sending an explicit message, called a **packet**, to memory requesting some data, and the memory sending back a reply packet.

Topology

The topology of an interconnection network describes how the links and switches are arranged, for example, as a ring or as a grid. Topological designs can be modeled as graphs, with the links as arcs and the switches as nodes, as shown in Fig. 8-37. Each node in an interconnection network (or its graph) has some number of links connected to it. Mathematicians call the number of links the **degree** of the node; engineers call it the **fanout**. In general, the greater the fanout, the more routing choices there are and the greater the fault tolerance, that is, the ability to continue functioning even if a link fails by routing around it. If every node has k arcs and the wiring is done right, it is possible to design the network so that it remains fully connected even if $k - 1$ links fail.

Another property of an interconnection network (or its graph) is its **diameter**. If we measure the distance between two nodes by the number of arcs that have to be traversed to get from one to the other, then the diameter of a graph is the distance between the two nodes that are the farthest apart (i.e., have the greatest distance between them). The diameter of an interconnection network is related to the worst-case delay when sending packets from CPU to CPU or from CPU to memory because each hop across a link takes a finite amount of time. The smaller the diameter, the better the worst-case performance. Also important is the average distance between two nodes, since this relates to the average packet transit time.

Yet another important property of an interconnection network is its transmission capacity, that is, how much data it can move per second. One useful measure of this capacity is the **bisection bandwidth**. To compute this quantity, we first have to (conceptually) partition the network into two equal (in terms of number of nodes) but unconnected parts by removing a set of arcs from its graph. Then we compute the total bandwidth of the arcs that have been removed. There may be many different ways to partition the network into two equal parts. The bisection bandwidth is the minimum of all the possible partitions. The significance of this number is that if the bisection bandwidth is, say, 800 bits/sec, then if there is a lot of communication between the two halves, the total throughput may be limited to only 800 bits/sec, in the worst case. Many designers believe bisection bandwidth is the most important metric of an interconnection network. Many interconnection networks are designed with the goal of maximizing the bisection bandwidth.

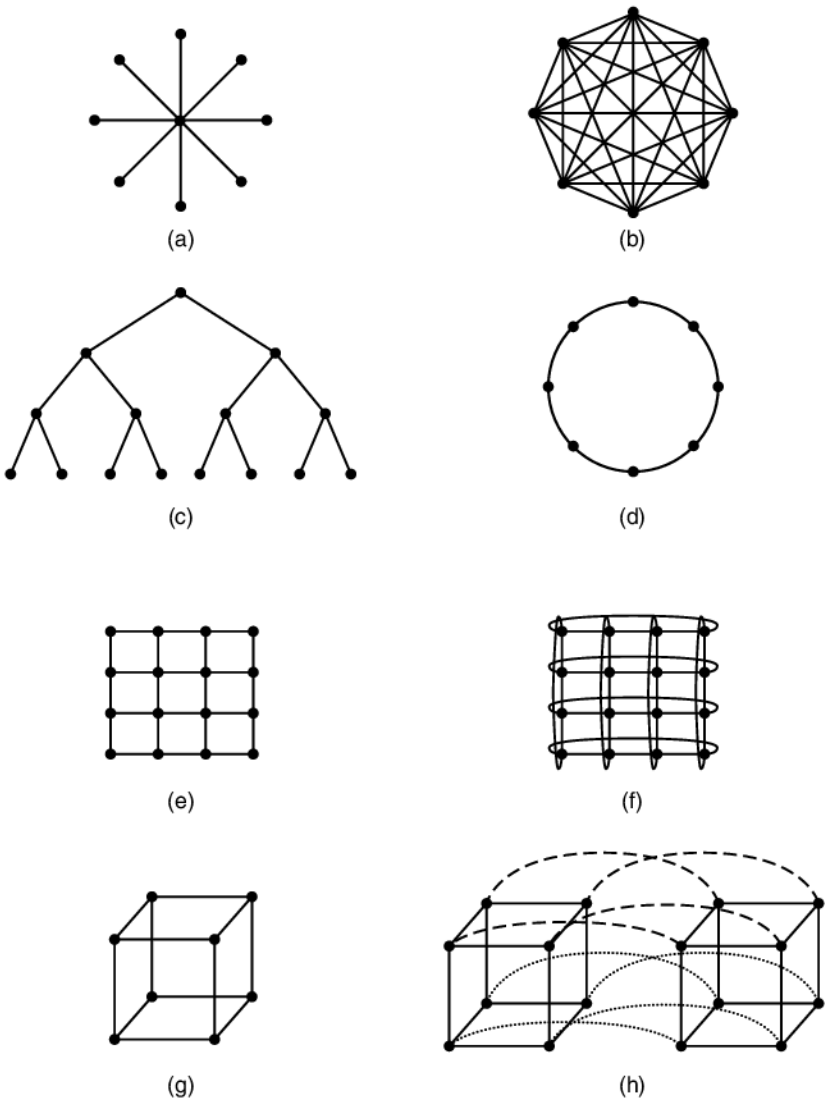


Figure 8-37. Various topologies. The heavy dots represent switches. The CPUs and memories are not shown. (a) A star. (b) A complete interconnect. (c) A tree. (d) A ring. (e) A grid. (f) A double torus. (g) A cube. (h) A 4D hypercube.

Interconnection networks can be characterized by their **dimensionality**. For our purposes, the dimensionality is determined by the number of choices there are to get from the source to the destination. If there is never any choice (i.e., there is only one path from each source to each destination), the network is zero dimensional. If there is one dimension in which a choice can be made, for example, go

east or go west, the network is one dimensional. If there are two axes, so a packet can go east or west, or alternatively, go north or south, the network is two dimensional, and so on.

Several topologies are shown in Fig. 8-37. Only the links (lines) and switches (dots) are shown here. The memories and CPUs (not shown) would typically be attached to the switches by interfaces. In Fig. 8-37(a), we have a zero-dimensional **star** configuration, in which the CPUs and memories would be attached to the outer nodes, with the central one just doing switching. Although a simple design, for a large system, the central switch is likely to be a major bottleneck. Also, from a fault-tolerance perspective, this is a poor design since a single failure at the central switch completely destroys the system.

In Fig. 8-37(b), we have another zero-dimensional design that is at the other end of the spectrum, a **full interconnect**. Here every node has a direct connection to every other node. This design maximizes the bisection bandwidth, minimizes the diameter, and is exceedingly fault tolerant (it can lose any six links and still be fully connected). Unfortunately, the number of links required for k nodes is $k(k-1)/2$, which quickly gets out of hand for large k.

Another topology is the **tree**, illustrated in Fig. 8-37(c). A problem with this design is that the bisection bandwidth is equal to the link capacity. Since there will normally be a lot of traffic near the top of the tree, the top few nodes will become bottlenecks. One way around this problem is to increase the bisection bandwidth by giving the upper links more bandwidth. For example, the lowest-level links might have a capacity b, those at the next level might have a capacity $2b$ and the top-level links might each have $4b$. Such a design is called a **fat tree** and has been used in commercial multicomputers, such as the (now-defunct) Thinking Machines' CM-5.

The **ring** of Fig. 8-37(d) is a one-dimensional topology by our definition because every packet sent has a choice of going left or going right. The **grid** or **mesh** of Fig. 8-37(e) is a two-dimensional design that has been used in many commercial systems. It is highly regular, easy to scale up to large sizes, and has a diameter that increases only as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig. 8-37(f), which is a grid with the edges connected. Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops.

Yet another popular topology is the three-dimensional torus. It consists of a 3D-structure with nodes at the points (i, j, k) where all coordinates are integers in the range from (1, 1, 1) to (l, m, n). Each node has six neighbors, two along each axis. The nodes at the edges have links that wrap around to the opposite edge, just as with the 2D torus.

The **cube** of Fig. 8-37(g) is a regular three-dimensional topology. We have illustrated a $2 \times 2 \times 2$ cube, but in the general case it could be a $k \times k \times k$ cube. In Fig. 8-37(h) we have a four-dimensional cube constructed from two three-dimensional cubes with the corresponding nodes connected. We could make a five-

dimensional cube by cloning the structure of Fig. 8-37(h) and connecting the corresponding nodes to form a block of four cubes. To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An n-dimensional cube formed this way is called a **hypercube**. Many parallel computers use this topology because the diameter grows linearly with the dimensionality. Put in other words, the diameter is the base 2 logarithm of the number of nodes, so, for example, a 10-dimensional hypercube has 1024 nodes but a diameter of only 10, giving excellent delay properties. Note that in contrast, 1024 nodes arranged as a 32×32 grid has a diameter of 62, more than six times worse than the hypercube. The price paid for the smaller diameter is that the fanout and thus the number of links (and the cost) is much larger for the hypercube. Nevertheless, the hypercube is a common choice for high-performance systems.

Multicomputers come in all shapes and sizes, so it is hard to give a clean taxonomy of them. Nevertheless, two general "styles" stand out: the MPPs and the clusters. We will now study each of these in turn.

8.4.2 MPPs—Massively Parallel Processors

The first category consists of the **MPP**s (**Massively Parallel Processors**), which are huge multimillion-dollar supercomputers. These are used in science, in engineering, and in industry for very large calculations, for handling very large numbers of transactions per second, or for data warehousing (storing and managing immense databases). Initially, MPPs were primarily used as scientific supercomputers, but now most of them are used in commercial environments. In a sense, these machines are the successors to the mighty mainframes of the 1960s (but the connection is tenuous, sort of like a paleontologist claiming that a flock of sparrows is the successor to the *Tyrannosaurus Rex*). To a large extent, the MPPs have displaced SIMD machines, vector supercomputers, and array processors at the top of the digital food chain.

Most of these machines use standard CPUs as their processors. Popular choices are Intel processors, the Sun UltraSPARC, and the IBM PowerPC. What sets the MPPs apart is their use of a very high-performance proprietary interconnection network designed to move messages with low latency and at high bandwidth. Both of these are important because the vast majority of all messages are small (well under 256 bytes), but most of the total traffic is caused by large messages (more than 8 KB). MPPs also come with extensive proprietary software and libraries.

Another point that characterizes MPPs is their enormous I/O capacity. Problems big enough to warrant using MPPs invariably have massive amounts of data to be processed, often terabytes. These data must be distributed among many disks and need to be moved around the machine at great speed.

Finally, another issue specific to MPPs is their attention to fault tolerance. With thousands of CPUs, several failures per week are just inevitable. Having an

18-hour run aborted because one CPU crashed is unacceptable, especially when one such failure is to be expected every week. Thus large MPPs always have special hardware and software for monitoring the system, detecting failures, and recovering from them smoothly.

While it would be nice to study the general principles of MPP design now, in truth, there are not many principles. When you come right down to it, an MPP is a collection of more-or-less standard computing nodes connected by a very fast interconnect of the types we have already examined. So instead, we will now look at two examples of MPPs: BlueGene/P and Red Storm.

BlueGene

As a first example of a massively parallel processor, we will now examine the IBM BlueGene system. IBM conceived this project in 1999 as a massively parallel supercomputer for solving computationally intensive problems in, among other fields, the life sciences. For example, biologists believe that the three-dimensional structure of a protein determines its functionality, yet computing the 3D structure of one small protein from the laws of physics took years on the supercomputers of that period. The number of proteins found in human beings is over half a million. Many of them are extremely large and their misfolding is known to be responsible for certain diseases (e.g., cystic fibrosis). Clearly, determining the 3D structure of all the human proteins would require increasing the world's computing power by many orders of magnitude, and modeling protein folding is only one problem that BlueGene was designed to handle. Equally complex challenges in molecular dynamics, climate modeling, astronomy, and even financial modeling also require orders of magnitude improvement in supercomputing.

IBM felt that there was enough of a market for massive supercomputing that it invested $100 million to design and build BlueGene. In November 2001, Livermore National Laboratory, run by the U.S. Department of Energy, signed up as a partner and first customer for the first version of the BlueGene family, called **BlueGene/L**. In 2007, IBM deployed the second generation of the BlueGene supercomputer, called the **BlueGene/P**, which we detail here.

The goal of the BlueGene project was not just to produce the world's fastest MPP, but to also to produce the most efficient one in terms of teraflops/dollar, teraflops/watt, and teraflops/m^3. For this reason, IBM rejected the philosophy behind previous MPPs, which was to use the fastest components money could buy. Instead, a decision was made to produce a custom system-on-a-chip component that was to run at a modest speed and low power in order to produce a very large machine with a high packing density. The first BlueGene/P was delivered to a German university in November 2007. The system contained 65,536 processors, and it was capable of 167 teraflops/sec. When deployed it was the fastest computer in Europe, and the sixth fastest computer in the world. The system was also regarded as one of the most computationally power-efficient supercomputers in the world,

able to produce 371 megaflops/W, making it nearly twice as power efficient as its predecessor the BlueGene/L. This first BlueGene/P deployment was upgraded in 2009 to include 294,912 processors, giving it a computational punch of 1 petaflop/sec.

The heart of the BlueGene/P system is the custom node chip illustrated in Fig. 8-38. It consists of four PowerPC 450 cores running at 850 MHz. The PowerPC 450 is a pipelined dual-issue superscalar processor popular in embedded systems. Each core has a pair of dual-issue floating-point units, which together can issue four floating-point instructions per clock cycle. The floating-point units have been augmented with a number of SIMD-type instructions sometimes useful in scientific computations on arrays. While no performance slouch, this chip is clearly not a top-of-the-line multiprocessor.

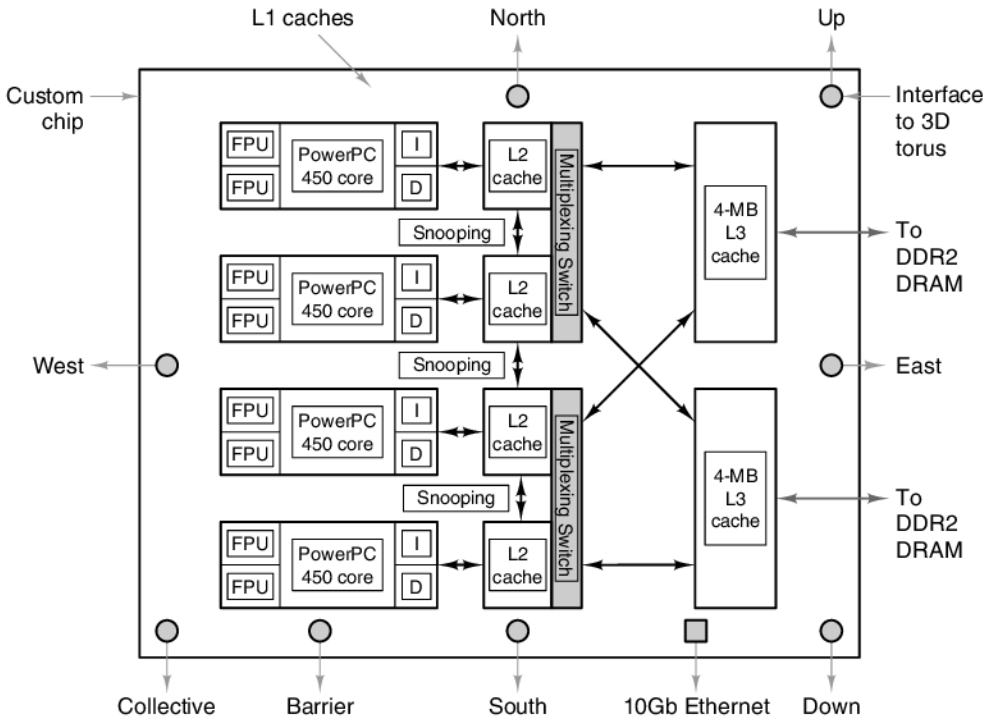


Figure 8-38. The BlueGene/P custom processor chip.

Three levels of cache are present on the chip. The first consists of a split L1 cache with 32 KB for instructions and 32 KB for data. The second is a unified cache consisting of a unified 2-KB cache. The L2 caches are really prefetch buffers rather than true caches. They snoop on each other and are cache consistent. The third level is a unified 4-MB shared cache that feeds data to the L2 caches. The four processors share access to two 4-MB L3 cache modules. There is cache

coherency between the L1 caches on the four CPUs. Thus when a shared piece of memory resides in more than one cache, accesses to that storage by one processor will be immediately visible to the other three processors. A memory reference that misses on the L1 cache but hits on the L2 cache takes about 11 clock cycles. A miss on L2 that hits on L3 takes about 28 cycles. Finally, a miss on L3 that has to go to the main DRAM takes about 75 cycles.

The four CPUs are connected via a high-bandwidth bus to a 3D torus network, which requires six connections: up, down, north, south, east, and west. In addition, each processor has a port to the collective network, used for broadcasting data to all processors. The barrier port is used to speed up synchronization operations, giving each processor fast access to a specialized synchronization network.

At the next level up, IBM designed a custom card that holds one of the chips shown in Fig. 8-38 along with 2 GB of DDR2 DRAM. The chip and the card are shown in Fig. 8-39(a)–(b) respectively.

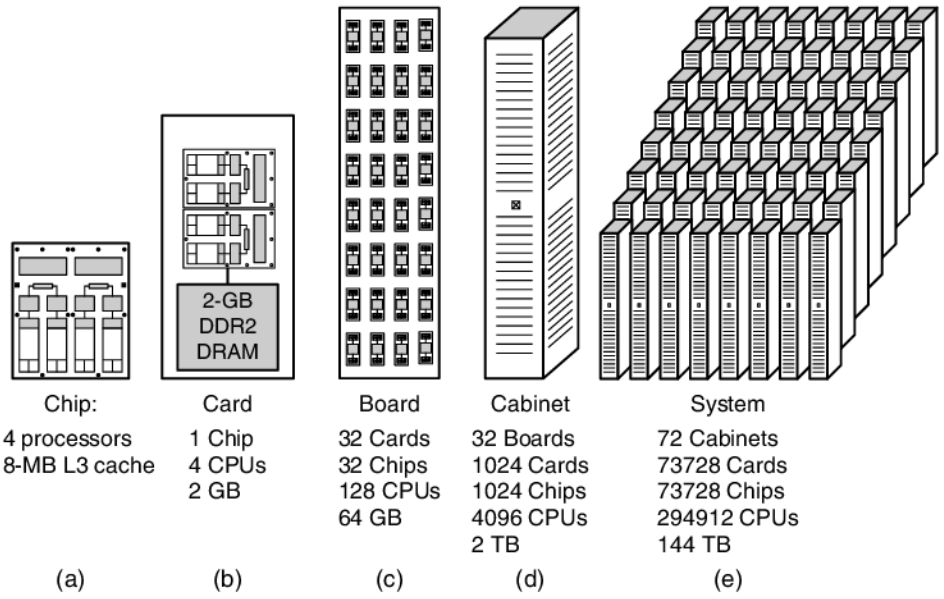


Figure 8-39. The BlueGene/P: (a) chip. (b) card. (c) board. (d) cabinet. (e) system.

The cards are mounted on plug-in boards, with 32 cards per board for a total of 32 chips (and thus 128 CPUs) per board. Since each card contains 2 GB of DRAM, the boards contain 64 GB apiece. One board is illustrated in Fig. 8-39(c). At the next level, 32 of these boards are plugged into a cabinet, packing 4096 CPUs into a single cabinet. A cabinet is illustrated in Fig. 8-39(d).

Finally, a full system, consisting of up to 72 cabinets with 294,912 CPUs, is depicted in Fig. 8-39(e). A PowerPC 450 can issue up to 6 instructions/cycle, thus

a full BlueGene/P system could conceivably issue up to 1,769,472 instructions per cycle. At 850 MHz, this gives the system a possible performance of 1.504 petaflops/sec. However, data hazards, memory latency, and lack of parallelism work together to ensure that the actual throughput of the system is much less. Real programs running on the BlueGene/P have demonstrated performance rates of up to 1 petaflop/sec.

The system is a multicomputer in the sense that no CPU has direct access to any memory except the 2 GB on its own card. While CPUs within a processor chip have shared memory, processors at the board, rack, and system level do not share the same memory. In addition, there is no demand paging because there are no local disks to page off. Instead, the system has 1152 I/O nodes, which are connected to disks and the other peripheral devices.

All in all, while the system is extremely large, it is also quite straightforward with little new technology except in the area of high-density packaging. The decision to keep it simple was no accident since a major goal was high reliability and availability. Consequently, a great deal of careful engineering went into the power supplies, fans, cooling, and cabling with the goal of a mean-time-to-failure of at least 10 days.

To connect all the chips, a scalable, high-performance interconnect is needed. The design used is a three-dimensional torus measuring $72 \times 32 \times 32$. As a consequence, each CPU needs only six connections to the torus network, two to other CPUs logically above and below it, north and south of it, and east and west of it. These six connections are labeled east, west, north, south, up, and down, respectively in Fig. 8-38. Physically, each 1024-node cabinet is an $8 \times 8 \times 16$ torus. Pairs of neighboring cabinets form an $8 \times 8 \times 32$ torus. Four pairs of cabinets in the same row form an $8 \times 32 \times 32$ torus. Finally, all 9 rows form a $72 \times 32 \times 32$ torus.

All links are thus point-to-point and operate at 3.4 Gbps. Since each of the 73,728 nodes has three links to "higher" numbered nodes, one in each dimension, the total bandwidth of the system is 752 terabits/sec. The information content of this book is about 300 million bits, including all the art in encapsulated PostScript format, so BlueGene/P could move 2.5 million copies of this book per second. Where they would go and who would want them is left as an exercise for the reader.

Communication on the 3D torus is done in the form of **virtual cut through** routing. This technique is somewhat akin to store-and-forward packet switching, except that entire packets are not stored before being forwarded. As soon as a byte has arrived at one node, it can be forwarded to the next one along the path, even before the entire packet has arrived. Both dynamic (adaptive) and deterministic (fixed) routing are possible. A small amount of special-purpose hardware on the chip is used to implement the virtual cut through.

In addition to the main 3D torus used for data transport, four other communication networks are present. The second one is the collective network in the form

of a tree. Many of the operations performed on highly parallel systems such as BlueGene/P require the participation of all the nodes. For example, consider finding the minimum value of a set of 65,536 values, one held in each node. The collective network joins all the nodes in a tree. Whenever two nodes send their respective values to a higher-level node, it selects out the smallest one and forwards it upward. In this way, far less traffic reaches the root than if all 65,636 nodes sent a message there.

The third network is the barrier network, used to implement global barriers and interrupts. Some algorithms work in phases with each node required to wait until all the others have completed the phase before starting the next phase. The barrier network allows the software to define these phases and provide a way to suspend all compute CPUs that reach the end of a phase until all of them have reached the end, at which time they are all released. Interrupts also use this network.

The fourth and fifth networks both use 10-gigabit Ethernet. One of them connects the I/O nodes to the file servers, which are external to BlueGene/P, and to the Internet beyond. The other one is used for debugging the system.

Each CPU node runs a small, custom, lightweight kernel that supports a single user and a single process. This process has at most four threads, one running on each CPU in the node. This simple structure was designed for high performance and high reliability.

For additional reliability, application software can call a library procedure to make a checkpoint. Once all outstanding messages have been cleared from the network, a global checkpoint can be made and stored so that in the event of a system failure, the job can be restarted from the checkpoint, rather than from the beginning. The I/O nodes run a traditional Linux operating system and support multiple processes.

Work is continuing to develop the next generation of BlueGene system, called the BlueGene/Q. This system is expected to go online in 2012, and it will have 18 processors per compute chip, which also feature simultaneous multithreading. These two features should greatly increase the number of instructions per cycle the system can execute. The system is expected to reach speeds of 20 petaflops/sec. For more information about BlueGene see Adiga et al. (2002), Alam et al., 2008, Almasi et al. (2003a, 2003b), Blumrich et al. (2005), and IBM (2008).

Red Storm

As our second example of an MPP, let us consider the Red Storm machine (also called Thor's Hammer) at Sandia National Laboratory. Sandia is operated by Lockheed Martin and does classified and unclassified work for the U.S. Department of Energy. Some of the classified work concerns the design and simulation of nuclear weapons, which is highly compute intensive.

Sandia has been in this business for a long time and over the decades has had many leading-edge supercomputers over the years. For decades, it favored vector

supercomputers, but eventually technology and economics made MPPs more cost effective. By 2002, the then-current MPP, called ASCI Red, was getting a bit creaky. Although it had 9460 nodes, collectively they had a mere 1.2 TB of RAM and 12.5 TB of disk space, and the system could barely crank out 3 teraflops/sec. So in the summer of 2002, Sandia selected Cray Research, a long-time supercomputer vendor, to build it a replacement for ASCI Red.

The replacement was delivered in August 2004, a remarkably short design and implementation cycle for such a large machine. The reason it could be designed and delivered so quickly is that Red Storm uses almost entirely off-the-shelf parts, except for one custom chip used for routing. In 2006, the system was updated with new processors; we detail this version of Red Storm here.

The CPU selected for Red Storm was the AMD 2.4-GHz dual-core Opteron. The Opteron has several key characteristics that made it the first choice. The first is that it has three operating modes. In legacy mode, it runs standard Pentium binary programs unmodified. In compatibility mode, the operating system runs in 64-bit mode and can address 2^{64} bytes of memory, but application programs run in 32-bit mode. Finally, in 64-bit mode, the entire machine is 64 bits and all programs can address the full 64-bit address space. In 64-bit mode, it is possible to mix and match software: both 32-bit and 64-bit programs can run at the same time, allowing an easy upgrade path.

The Opteron's second key characteristic is its attention to the memory bandwidth problem. In recent years, CPUs have been getting faster and faster and memory has not been keeping pace, resulting in a massive penalty when there is a level 2 cache miss. AMD integrated the memory controller into the Opteron so it can run at the speed of the processor clock instead of the speed of the memory bus, which improves memory performance. The controller can handle eight DIMMS of 4 GB each, for a maximum total memory of 32 GB per Opteron. In the Red Storm system, each Opteron has only 2–4 GB. However, as memory gets cheaper, no doubt more will be added in the future. By utilizing dual-core Opterons, the system was able to double the raw compute power.

Each Opteron has its own dedicated custom network processor called the **Seastar**, manufactured by IBM. The Seastar is a critical component since nearly all the data traffic between the processors goes over the Seastar network. Without the very high-speed interconnect provided by these custom chips, the system would quickly bog down in data.

Although the Opterons are commercially available off the shelf, the Red Storm packaging is custom built. Each Red Storm board contains four Opterons, 4 GB of RAM, four Seastars, a RAS (Reliability, Availability, and Service) processor, and a 100-Mbps Ethernet chip, as shown in Fig. 8-40.

A set of eight boards is plugged into a backplane and inserted into a card cage. Each cabinet holds three card cages for a total of 96 Opterons, plus the necessary power supplies and fans. The full system consists of 108 cabinets for compute nodes, giving a total of 10,368 Opterons (20,736 processors) with 10 TB of

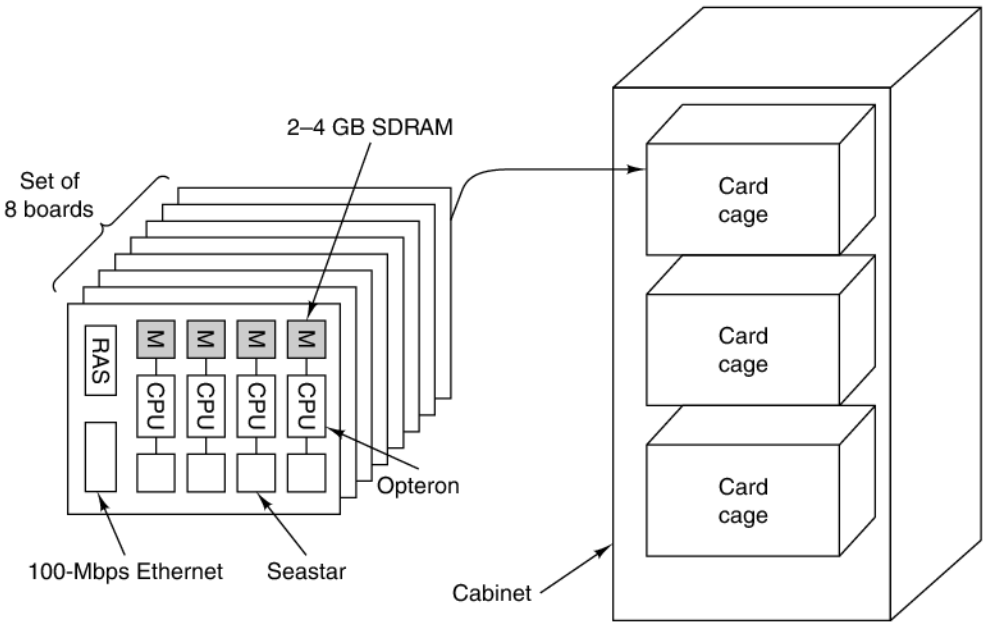


Figure 8-40. Packaging of the Red Storm components.

SDRAM. Each CPU has access only to its own SDRAM. There is no shared memory. The theoretical computing power of the system is 41 teraflops/sec.

The interconnection between the Opteron CPUs is done by the custom Seastar routers, one router per Opteron CPU. They are connected in a 3D torus of size $27 \times 16 \times 24$ with one Seastar at each mesh point. Each Seastar has seven bidirectional 24-Gbps links, going north, east, south, west, up, down, and to the Opteron. The transit time between adjacent mesh points is 2 microsec. Across the entire set of compute nodes it is only 5 microsec. A second network using 100-Mbps Ethernet is used for service and maintenance.

In addition to the 108 compute cabinets, the system also contains 16 cabinets for I/O and service processors. Each cabinet holds 32 Opterons. These 512 CPUs are split: 256 for I/O and 256 for service. The rest of the space is for disks, which are organized as RAID 3 and RAID 5, each with a parity drive and a hot spare. The total disk space is 240 TB. The combined disk bandwidth is 50 GB/sec.

The system is partitioned into classified and unclassified sections, with switches between the parts so they can be mechanically coupled or decoupled. A total of 2688 are always in the classified section and another 2688 Opterons are always in the unclassified section. The remaining 4992 Opterons are switchable into either section, as depicted in Fig. 8-41. The 2688 classified Opterons each have 4 GB of RAM; all the rest have 2 GB each. Apparently classified work is memory intensive. The I/O and service processors are split between the two parts.

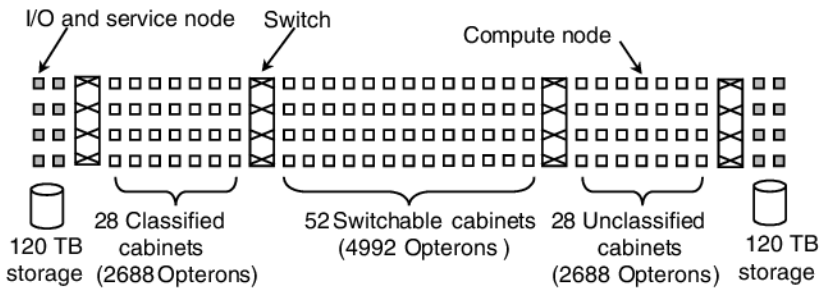


Figure 8-41. The Red Storm system as viewed from above.

Everything is housed in a new 2000-m^2 custom building. The building and site have been designed so that the system can be upgraded to as many as 30,000 Opterons in the future if required. The compute nodes draw 1.6 megawatts of power; the disks draw another megawatt. Adding in the fans and air conditioning, the whole thing uses 3.5 MW.

The computer hardware and software cost $90 million. The building and cooling cost another $9 million, so the total cost came in at just under $100 million, although some of that was nonrecurring engineering cost. If you want to order an exact clone, $60 million would be a good number to keep in mind. Cray intends to sell smaller versions of the system to other government and commercial customers under the name X3T.

The compute nodes run a lightweight kernel called **catamount**. The I/O and service nodes run plain vanilla Linux with a small addition to support MPI (discussed later in this chapter). The RAS nodes run a stripped-down Linux. Extensive software from ASCI Red is available for use on Red Storm, including CPU allocators, schedulers, MPI libraries, math libraries, as well as the application programs.

With such a large system, achieving high reliability is essential. Each board has a RAS processor for doing system maintenance and there are special hardware facilities as well. The goal is an MTBF (Mean Time Between Failures) of 50 hours. ASCI Red had a hardware MTBF of 900 hours but was plagued by an operating-system crash every 40 hours. Although the new hardware is much more reliable than the old, the weak point remains the software.

For more information about Red Storm, see Brightwell et al. (2005, 2010).

A Comparison of BlueGene/P and Red Storm

Red Storm and BlueGene/P are comparable in some ways but different in others, so it is interesting to put some of the key parameters next to each other, as presented in Fig. 8-42.

Item	BlueGene/P	Red Storm
CPU	32-Bit PowerPC	64-Bit Opteron
Clock	850 MHz	2.4 GHz
Compute CPUs	294,912	20,736
CPUs/board	128	8
CPUs/cabinet	4096	192
Compute cabinets	72	108
Teraflops/sec	1000	124
Memory/CPU	512 MB	2–4 GB
Total memory	144 TB	10 TB
Router	PowerPC	Seastar
Number of routers	73,728	10,368
Interconnect	3D torus $72 \times 32 \times 32$	3D torus $27 \times 16 \times 24$
Other networks	Gigabit Ethernet	Fast Ethernet
Partitionable	No	Yes
Compute OS	Custom	Custom
I/O OS	Linux	Linux
Vendor	IBM	Cray Research
Expensive	Yes	Yes

Figure 8-42. A comparison of BlueGene/P and Red Storm.

The two machines were built in the same time frame, so their differences are due not to technology but to designers' different visions and to some extent to the differences between the builders, IBM and Cray. BlueGene/P was designed from the beginning as a commercial machine, which IBM hopes to sell in large numbers to biotech, pharmaceutical, and other companies. Red Storm was built on special contract with Sandia, although Cray plans to make a smaller version for sale, too.

IBM's vision is clear: combine existing cores to produce a custom chip that can be mass produced cheaply, run it at a low speed, and hook together a very large number of them using a modest-speed communication network. Sandia's vision is equally clear, but different: use a powerful off-the-shelf 64-bit CPU, design a very fast custom router chip, and throw in a lot of memory to produce a far more powerful node than BlueGene/P so fewer will be needed and communication between them will be faster.

These decisions had consequences for the packaging. Because IBM built a custom chip combining the processor and router, it achieved a higher packing density: 4096 CPUs/cabinet. Because Sandia went for an unmodified off-the-shelf CPU chip and 2–4 GB of RAM per node, it could get only 192 compute processors in a cabinet. Consequently, Red Storm takes up more floor space and consumes more power than BlueGene/P.

In the exotic world of national laboratory computing, the bottom line is performance. In this respect, BlueGene/P wins, 1000 TF/sec to 124 TF/sec, but Red Storm was designed to be expandable, so by throwing more Opterons at the problem, Sandia could probably up its performance significantly. IBM could respond by cranking the clock up a bit (850 MHz is not really pushing the state-of-the-art very hard). In short, MPP supercomputers have not even come close to any physical limits yet and will continue growing for years to come.

8.4.3 Cluster Computing

The other style of multicomputer is the **cluster computer** (Anderson et al., 1995, and Martin et al., 1997). It typically consists of hundreds or thousands of PCs or workstations connected by a commercially available network board. The difference between an MPP and a cluster is analogous to the difference between a mainframe and a PC. Both have a CPU, both have RAM, both have disks, both have an operating system, and so on. The mainframe just has faster ones (except maybe the operating system). Yet qualitatively they feel different and are used and managed differently. This same difference holds for MPPs vs. clusters.

Historically, the key element that made MPPs special was their high-speed interconnect, but the recent arrival of commercial, off-the-shelf, high-speed interconnects has begun to close the gap. All in all, clusters are likely to drive MPPs into ever tinier niches, just as PCs have turned mainframes into esoteric specialty items. The main niche for MPPs is high-budget supercomputers, where peak performance is everything and if you have to ask the price you cannot afford one.

While many kinds of clusters exist, two species dominate: centralized and decentralized. A centralized cluster is a cluster of workstations or PCs mounted in a big rack in a single room. Sometimes they are packaged in a much more compact way than usual to reduce physical size and cable length. Typically, the machines are homogeneous and have no peripherals other than network cards and possibly disks. Gordon Bell, the designer of the PDP-11 and VAX, has called such machines **headless workstations** (because they have no owners). We were tempted to call them headless COWs, but feared such a term would gore too many holy cows, so we refrained.

Decentralized clusters consist of the workstations or PCs spread around a building or campus. Most of them are idle many hours a day, especially at night. Usually, these are connected by a LAN. Typically, they are heterogeneous and have a full complement of peripherals, although having a cluster with 1024 mice is really not much better than a cluster with 0 mice. Most importantly, many of them have owners who have emotional attachments to their machines and tend to frown upon some astronomer trying to simulate the big bang on theirs. Using idle workstations to form a cluster invariably means having some way to migrate jobs off machines when their owners want to reclaim them. Job migration is possible but adds software complexity.

Clusters are often smallish affairs, ranging from a dozen to perhaps 500 PCs. However, it is also possible to build very large ones from off-the-shelf PCs. Google has done this in an interesting way that we will now look at.

Google

Google is a popular search engine for finding information on the Internet. While its popularity is due, in part, to its simple interface and fast response time, its design is anything but simple. From Google's point of view, the problem is that it has to find, index, and store the entire World Wide Web (an estimated 40 billion pages), be able to search the whole thing in under 0.5 sec, and handle tens of thousands of queries/sec coming from all over the world 24 hours a day. In addition, it must never go down, not even in the face of earthquakes, electrical power failures, telecom outages, hardware failures and software bugs. And, of course, it has to do all of this as cheaply as possible. Building a Google clone is definitely not an exercise for the reader.

How does Google do it? To start with, Google operates multiple data centers around the world. Not only does this approach provide backups in case one of them is swallowed by an earthquake, but when *www.google.com* is looked up, the sender's IP address is inspected and the address of the nearest data center is supplied. The browser then sends the query there.

Each data center has at least one OC-48 (2.488-Gbps) fiber-optics connection to the Internet, on which it receives queries and sends answers, as well as an OC-12 (622-Mbps) backup connection from a different telecom provider, in case the primary ones go down. Uninterruptable power supplies and emergency diesel generators are available at all data centers to keep the show going during power failures. Consequently, during a major natural disaster, performance will suffer, but Google will keep running.

To get a better understanding of why Google chose the architecture it did, it is useful to briefly describe how a query is processed once it hits its designated data center. After arriving at the data center (step 1 in Fig. 8-43), the load balancer routes the query to one of the many query handlers (2), and to the spelling checker (3) and ad server (4) in parallel. Then the search words are looked up on the index servers (5) in parallel. These servers contain an entry for each word on the Web. Each entry lists all the documents (Web pages, PDF files, PowerPoint presentations, etc.) containing the word, sorted in page-rank order. Page rank is determined by a complicated (and secret) formula, but the number of links to a page and their own ranks play a large role.

To get higher performance, the index is divided into pieces called **shards** that can be searched in parallel. Conceptually, at least, shard 1 contains all the words in the index, with each one followed by the IDs of the n highest-ranked documents containing that word. Shard 2 contains all the words and the IDs of the n next-

highest-ranked documents, and so on. As the Web grows, each of these shards may later be split with the first k words in one set of shards, the next k words in a second set of shards and so forth, in order to achieve even more search parallelism.

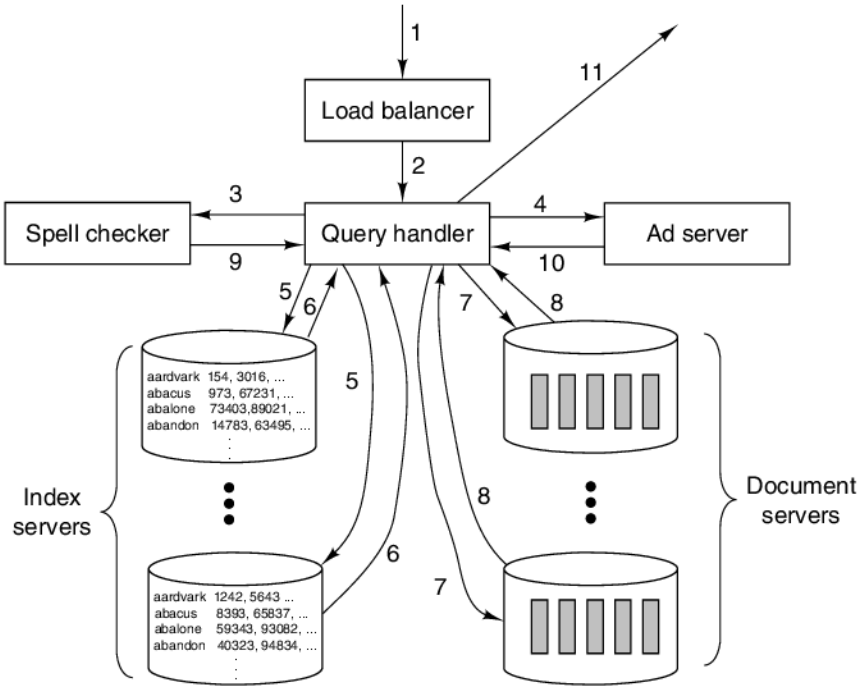


Figure 8-43. Processing of a Google query.

The index servers return a set of document identifiers (6) that are then combined according to the Boolean properties of the query. For example, if the search was for +digital +capybara +dance, then only document identifiers appearing in all three sets are used in the next step. In this step (7), the documents themselves are referenced to extract their titles, URLs, and snippets of text surrounding the search terms. The document servers contain many copies of the entire Web at each data center, hundreds of terabytes at present. The documents are also divided into shards to enhance parallel search. While processing a query does not require reading the whole Web (or even reading the tens of terabytes on the index servers), having to process 100 MB per query is normal.

When the results are returned to the query handler (8), the pages found are collated into page-rank order. If potential spelling errors are detected (9), they are announced and relevant ads are added (10). Displaying ads for advertisers interested in buying specific search terms (e.g., "hotel" or "camcorder") is how Google makes its money. Finally, the results are formatted in HTML (HyperText Markup Language) and sent to the user as a Web page.

With this background, we can now examine the Google architecture. Most companies, when faced with a huge data base, massive transaction rate, and the need for high reliability, would buy the biggest, fastest, and most reliable equipment on the market. Google did just the opposite. It bought cheap, modest-performance PCs. Lots of them. And with them, it built the world's largest off-the-shelf cluster. The driving principle behind this decision was simple: optimize price/performance.

The logic behind this decision lies in economics: commodity PCs are very cheap. High-end servers are not and large multiprocessors are even less so. Thus while a high-end server might have two or three times the performance of a midrange desktop PC, it will typically be 5–10 times the price, which is not cost effective.

Of course, cheap PCs fail more often than top-of-the-line servers, but the latter fail, too, so the Google software had to be designed to work with failing hardware no matter what kind of equipment it was using. Once the fault-tolerance software was written, it did not really matter whether the failure rate was 0.5% per year or 2% per year, failures had to be dealt with. Google's experience is that about 2% of the PCs fail each year. More than half the failures are due to faulty disks, followed by power supplies and then RAM chips. After burn-in, CPUs never fail. Actually, the biggest source of crashes is not hardware at all; it is software. The first response to a crash is just to reboot, which often solves the problem (the electronic equivalent of the doctor saying: "Take two aspirins and go to bed.").

A typical modern Google PC consists of a 2-GHz Intel processor, 4 GB of RAM, and a disk of around 2 TB, the kind of thing a grandmother might buy for checking her email occasionally. The only specialty item is an Ethernet chip. Not exactly state of the art, but very cheap. The PCs are housed in 1u-high cases (about 5 cm thick) and stacked 40 high in 19-inch racks, one stack in front and one stack in back for a total of 80 PCs per rack. The PCs in a rack are connected by switched Ethernet, with the switch inside the rack. The racks in a data center are also connected by switched Ethernet, with two redundant switches per data center used to survive switch failures.

The layout of a typical Google data center is illustrated in Fig. 8-44. The incoming high-bandwidth OC-48 fiber is routed to each of two 128-port Ethernet switches. Similarly, the backup OC-12 fiber is also routed to each of the two switches. The incoming fibers use special input cards and do not occupy any of the 128 Ethernet ports.

Each rack has four Ethernet links coming out of it, two to the left switch and two to the right switch. In this configuration, the system can survive the failure of either switch. Since each rack has four connections to the switch (two from the front 40 PCs and two from the back 40 PCs), it takes four link failures or two link failures and a switch failure to take a rack offline. With a pair of 128-port switches and four links from each rack, up to 64 racks can be supported. With 80 PCs per rack, a data center can have up to 5120 PCs. But, of course, racks do not have to

hold exactly 80 PCs and switches can be larger or smaller than 128 ports; these are just typical values for a Google cluster.

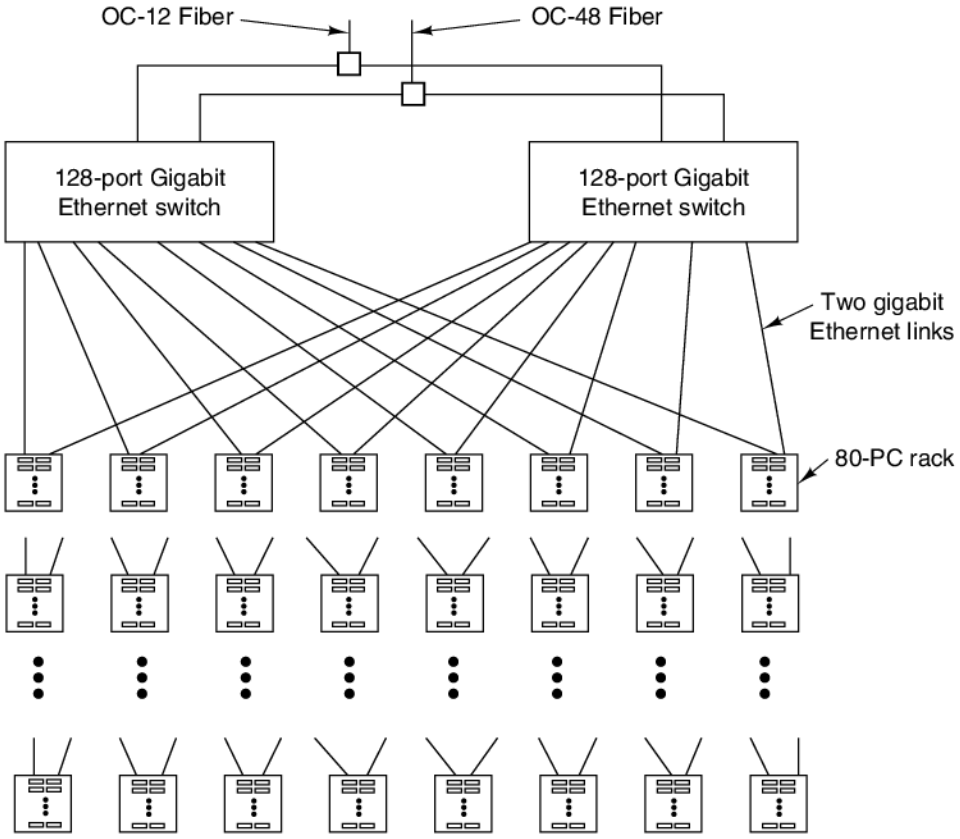


Figure 8-44. A typical Google cluster.

Power density is also a key issue. A typical PC burns about 120 watts or about 10 kW per rack. A rack needs about 3 m^2 so that maintenance personnel can install and remove PCs and for the air conditioning to function. These parameters give a power density of over 3000 watts/m^2. Most data centers are designed for 600–1200 watts/m^2, so special measures are required to cool the racks.

Google has learned three key things about running massive Web servers that bear repeating.

1. Components will fail so plan for it.
2. Replicate everything for throughput and availability.
3. Optimize price/performance.

The first item says that you need to have fault-tolerant software. Even with the best of equipment, when you have a massive number of components, some will fail and the software has to be able to handle it. Whether you have one failure a week or two, the software has to be able to handle failures.

The second item points out that both hardware and software have to be highly redundant. Doing so not only improves the fault-tolerance properties, but also the throughput. In the case of Google, the PCs, disks, cables, and switches are all replicated many times over. Furthermore, the index and the documents are broken into shards and the shards are heavily replicated in each data center and the data centers are themselves replicated.

The third item is a consequence of the first two. If the system has been properly designed to deal with failures, buying expensive components such as RAIDs with SCSI disks is a mistake. Even they will fail, but spending 10 times as much to cut the failure rate in half is a bad idea. Better to buy 10 times as much hardware and deal with the failures when they occur. At the very least, having more hardware will give better performance when everything is working.

For more information about Google, see Barroso et al. (2003), and Ghemawat et al. (2003).

8.4.4 Communication Software for Multicomputers

Programming a multicomputer requires special software, usually libraries, for handling interprocess communication and synchronization. In this section we will say a few words about this software. For the most part, the same software packages run on MPPs and clusters, so applications can be easily ported between platforms.

Message-passing systems have two or more processes running independently of one another. For example, one process may be producing some data and one or more others may be consuming it. There is no guarantee that when the sender has more data the receiver(s) will be ready for it, as each one runs its own program.

Most message-passing systems provide two primitives (usually library calls), send and receive, but several different kinds of semantics are possible. The three main variants are

1. Synchronous message passing.
2. Buffered message passing.
3. Nonblocking message passing.

In **synchronous message passing**, if the sender executes a send and the receiver has not yet executed a receive, the sender is blocked (suspended) until the receiver executes a receive, at which time the message is copied. When the sender gets control back after the call, it knows that the message has been sent and correctly received. This method has the simplest semantics and does not require any

buffering. It has the severe disadvantage that the sender remains blocked until the receiver has gotten and acknowledged receipt of the message.

In **buffered message passing**, when a message is sent before the receiver is ready, the message is buffered somewhere, for example, in a mailbox, until the receiver takes it out. Thus in buffered message passing, a sender can continue after a send, even if the receiver is busy with something else. Since the message has actually been sent, the sender is free to reuse the message buffer immediately. This scheme reduces the time the sender has to wait. Basically, as soon as the system has sent the message the sender can continue. However, the sender now has no guarantee that the message was correctly received. Even if communication is reliable, the receiver may have crashed before getting the message.

In **nonblocking message passing**, the sender is allowed to continue immediately after making the call. All the library does is tell the operating system to do the call later, when it has time. As a consequence, the sender is hardly blocked at all. The disadvantage of this method is that when the sender continues after the send, it may not reuse the message buffer as the message may not yet have been sent. Somehow it has to find out when it can reuse the buffer. One idea is to have it poll the system to ask. The other is to get an interrupt when the buffer is available. Neither of these makes the software any simpler.

Below we will briefly discuss a popular message-passing system available on many multicomputers: MPI.

MPI—Message-Passing Interface

For quite a few years, the most popular communication package for multicomputers was **PVM** (**Parallel Virtual Machine**) (Geist et al., 1994, and Sunderram, 1990). In recent years it has been largely replaced by **MPI** (**Message-Passing Interface**). MPI is much richer and more complex than PVM, with many more library calls, many more options, and many more parameters per call. The original version of MPI, now called MPI-1, was augmented by MPI-2 in 1997. Below we will give a very cursory introduction to MPI-1 (which contains all the basics), then say a little about what was added in MPI-2. For more information about MPI, see Gropp et al. (1994) and Snir et al. (1996).

MPI-1 does not deal with process creation or management, as PVM does. It is up to the user to create processes using local system calls. Once they have been created, they are arranged into static, unchanging process groups. It is with these groups that MPI works.

MPI is based on four major concepts: communicators, message data types, communication operations, and virtual topologies. A **communicator** is a process group plus a context. A context is a label that identifies something, such as a phase of execution. When messages are sent and received, the context can be used to keep unrelated messages from interfering with one another.

Messages are typed and many data types are supported, including characters, short, regular, and long integers, single- and double-precision floating-point numbers, and so on. It is also possible to construct other types derived from these.

MPI supports an extensive set of communication operations. The most basic one is used to send messages as follows:

```
MPI_Send(buffer, count, data_type, destination, tag, communicator)
```

This call sends a buffer with *count* number of items of the specified data type to the destination. The *tag* field labels the message so the receiver can say it only wants to receive a message with that tag. The last field tells which process group the destination is in (the *destination* field is just an index into the list of processes for the specified group). The corresponding call for receiving a message is

```
MPI_Recv(&buffer, count, data_type, source, tag, communicator, &status)
```

which announces that the receiver is looking for a message of a certain type from a certain source with a certain tag.

MPI supports four basic communication modes. Mode 1 is synchronous, in which the sender may not begin sending until the receiver has called MPI_Recv. Mode 2 is buffered, in which this restriction does not hold. Mode 3 is standard, which is implementation dependent and can be either synchronous or buffered. Mode 4 is ready, in which the sender claims the receiver is available (as in synchronous), but no check is made. Each of these primitives comes in a blocking and a nonblocking version, leading to eight primitives in all. Receiving has only two variants: blocking and nonblocking.

MPI supports collective communication, including broadcast, scatter/gather, total exchange, aggregation, and barrier. For all forms of collective communication, all the processes in a group must make the call and with compatible arguments. Failure to do this is an error. A typical form of collective communication is for processes organized in a tree, in which values propagate up from the leaves to the root, undergoing some processing at each step, for example, adding up the values or taking the maximum.

A basic concept in MPI is the **virtual topology**, in which the processes can be arranged in a tree, ring, grid, torus, or other topology by the user per application. Such an arrangement provides a way to name communication paths and facilitates communication.

MPI-2 adds dynamic processes, remote memory access, nonblocking collective communication, scalable I/O support, real-time processing, and many other new features that are beyond the scope of this book. In the scientific community, a battle raged for years between the MPI and PVM camps. The PVM side said that PVM was easier to learn and simpler to use. The MPI side said the MPI does more and also points out that MPI is a formal standard with a standardization committee and an official defining document. The PVM side agreed but claimed that the lack

of a full-blown standardization bureaucracy is not necessarily a drawback. When all was said and done, it appears that MPI won.

8.4.5 Scheduling

MPI programmers can easily create jobs requesting multiple CPUs and running for substantial periods of time. When multiple independent requests are available from different users, each needing a different number of CPUs for different time periods, the cluster needs a scheduler to determine which job gets to run when.

In the simplest model, the job scheduler requires each job to specify how many CPUs it needs. Jobs are then run in strict FIFO order, as shown in Fig. 8-45(a). In this model, after a job is started, a check is made to see if enough CPUs are available to start the next job in the input queue. If so, it is started, and so on. Otherwise, the system waits until more CPUs become available. As an aside, although we have suggested that this cluster has eight CPUs, it might well have 128 CPUs that are allocated in units of 16 (giving eight CPU groups), or another combination.

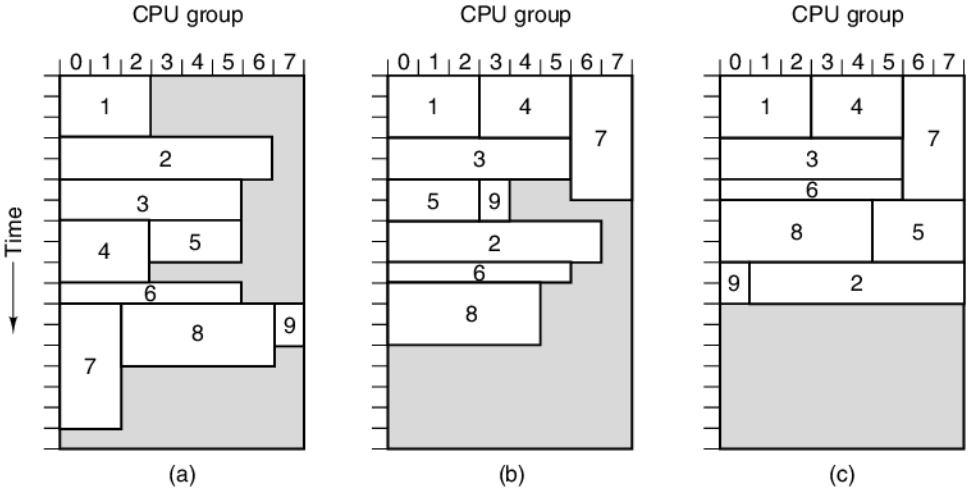


Figure 8-45. Scheduling a cluster. (a) FIFO. (b) Without head-of-line blocking. (c) Tiling. The shaded areas indicate idle CPUs.

A better scheduling algorithm avoids head-of-line blocking by skipping over jobs that do not fit and picking the first one that does fit. Whenever a job finishes, the queue of remaining jobs is checked in FIFO order. This algorithm gives the result of Fig. 8-45(b).

A still more sophisticated scheduling algorithm requires each submitted job to specify its shape, that is, how many CPUs for how many minutes. With that information, the job scheduler can attempt to tile the CPU-time rectangle. Tiling is

especially effective when jobs are submitted during the daytime for execution at night, so the job scheduler has all the information about all the jobs in advance and can run them in optimal order, as illustrated in Fig. 8-45(c).

8.4.6 Application-Level Shared Memory

That multicomputers scale to larger sizes than multiprocessors should be clear from our examples. This reality has led to the development of message-passing systems like MPI. Many programmers do not like this model and would like to have the illusion of shared memory, even if it is not really there. Achieving this goal would be the best of both worlds: large, inexpensive hardware (at least, per node) plus ease of programming. This is the holy grail of parallel computing.

Many researchers have concluded that while shared memory at the architectural level may not scale well, there may be other ways to achieve the same goal. From Fig. 8-21, we see that there are other levels at which a shared memory can be introduced. In the following sections, we will look at some ways that shared memory can be introduced into the programming model on a multicomputer, without it being present at the hardware level.

Distributed Shared Memory

One class of application-level shared-memory system is the page-based system. It goes under the name of **DSM** (**Distributed Shared Memory**). The idea is simple: a collection of CPUs on a multicomputer share a common paged virtual address space. In the simplest version, each page is held in the RAM of exactly one CPU. In Fig. 8-46(a), we see a shared virtual address space consisting of 16 pages, spread over four CPUs.

When a CPU references a page in its own local RAM, the read or write just happens without any further delay. However, when a CPU references a page in a remote memory, it gets a page fault. Instead of having the missing page being brought in from disk, though, the run-time system or operating system sends a message to the node holding the page to unmap it and send it over. After it has arrived, it is mapped in and the faulting instruction restarted, just as with a normal page fault. In Fig. 8-46(b), we see the situation after CPU 0 has faulted on page 10: it is moved from CPU 1 to CPU 0.

This basic idea was first implemented in IVY (Li and Hudak, 1989). It provides a fully shared, sequentially consistent memory on a multicomputer. However, many optimizations are possible to improve the performance. The first optimization, present in IVY, is to allow pages that are marked as read-only to be present at multiple nodes at the same time. Thus when a page fault occurs, a copy of the page is sent to the faulting machine, but the original stays where it is since there is no danger of conflicts. The situation of two CPUs sharing a read-only page (page 10) is illustrated in Fig. 8-46(c).

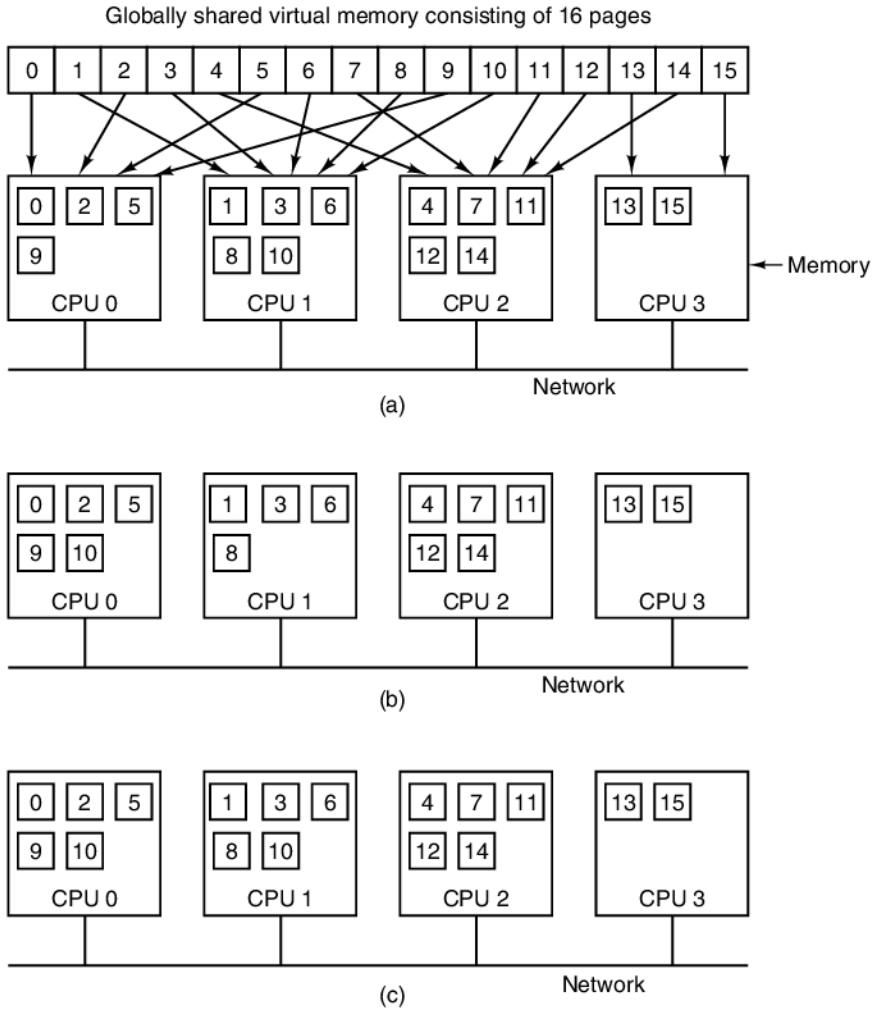


Figure 8-46. A virtual address space consisting of 16 pages spread over four nodes of a multicomputer. (a) The initial situation. (b) After CPU 0 references page 10. (c) After CPU 1 references page 10, here assumed to be a read-only page.

Even with this optimization, performance is frequently unacceptable, especially when one process is actively writing a few words at the top of some page and another process on a different CPU is actively writing a few words at the bottom of the page. Since only one copy of the page exists, the page must be ping-ponged back and forth constantly, a situation known as **false sharing**.

The problem of false sharing can be attacked in several ways. In the Treadmarks system, for example, sequentially consistent memory is abandoned in favor

of release consistency (Amza, 1996). Potentially writable pages may be present at multiple nodes at the same time, but before doing a write, a process must first do an acquire operation to signal its intention. At that point, all copies but the most recent one are invalidated. No other copies may be made until the corresponding release is done, at which time the page can be shared again.

A second optimization done in Treadmarks is to initially map each writable page in read-only mode. When the page is first written to, a protection fault occurs and the system makes a copy of the page, called the **twin**. Then the original page is mapped in as read-write and subsequent writes can go at full speed. When a remote page fault happens later and the page has to be shipped over there, a word-by-word comparison is done between the current page and the twin. Only those words that have been changed are sent, reducing the size of the messages.

When a page fault occurs, the missing page has to be located. Various solutions are possible, including those used in NUMA and COMA machines, such as (home-based) directories. In fact, many of the solutions used in DSM are also applicable to NUMA and COMA because DSM is really just a software implementation of NUMA or COMA with each page being treated like a cache line.

DSM is a hot area of research. Interesting systems include CASHMERE (Kontothanassis, et al., 1997 and Stets et al., 1997), CRL (Johnson et al., 1995), Shasta (Scales et al., 1996), and Treadmarks (Amza, 1996 and Lu et al., 1997).

Linda

Page-based DSM systems like IVY and Treadmarks use the MMU hardware to trap accesses to missing pages. While making and sending differences instead of whole pages helps, the fact remains that pages are an unnatural unit for sharing, so other approaches have been tried.

One such approach is Linda, which provides processes on multiple machines with a highly structured distributed shared memory (Carriero and Gelernter, 1989). This memory is accessed through a small set of primitive operations that can be added to existing languages, such as C and FORTRAN, to form parallel languages, in this case, C-Linda and FORTRAN-Linda.

The unifying concept behind Linda is that of an abstract **tuple space**, which is global to the entire system and accessible to all processes in it. Tuple space is like a global shared memory, only with a certain built-in structure. The tuple space contains some number of **tuples**, each consisting of one or more fields. For C-Linda, field types include integers, long integers, and floating-point numbers, as well as composite types such as arrays (including strings) and structures (but not other tuples). Figure 8-47 shows three tuples as examples.

Four operations are provided on tuples. The first one, out, puts a tuple into the tuple space. For example,

```
out("abc", 2, 5);
```

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is sister", Carolyn, Elinor)
```

Figure 8-47. Three Linda tuples.

puts the tuple ("abc", 2, 5) into the tuple space. The fields of out are normally constants, variables, or expressions, as in

```
out("matrix–1", i, j, 3.14);
```

which outputs a tuple with four fields, the second and third of which are determined by the current values of the variables *i* and *j*.

Tuples are retrieved from the tuple space by the in primitive. They are addressed by content rather than by name or address. The fields of in can be expressions or formal parameters. Consider, for example,

```
in("abc", 2, ? i);
```

This operation "searches" the tuple space for a tuple consisting of the string "abc", the integer, 2, and a third field containing any integer (assuming that *i* is an integer). If found, the tuple is removed from the tuple space and the variable *i* is assigned the value of the third field. The matching and removal are atomic, so if two processes execute the same in operation simultaneously, only one of them will succeed, unless two or more matching tuples are present. The tuple space may even contain multiple copies of the same tuple.

The matching algorithm used by in is straightforward. The fields of the in primitive, called the **template**, are (conceptually) compared to the corresponding fields of every tuple in the tuple space. A match occurs if the following three conditions are all met:

1. The template and the tuple have the same number of fields.
2. The types of the corresponding fields are equal.
3. Each constant or variable in the template matches its tuple field.

Formal parameters, indicated by a question mark followed by a variable name or type, do not participate in the matching (except for type checking), although those containing a variable name are assigned after a successful match.

If no matching tuple is present, the calling process is suspended until another process inserts the needed tuple, at which time the called is automatically revived and given the new tuple. The fact that processes block and unblock automatically means that if one process is about to output a tuple and another is about to input it, it does not matter which goes first.

In addition to out and in, Linda also has a primitive read, which is the same as in except that it does not remove the tuple from the tuple space. There is also a primitive eval, which causes its parameters to be evaluated in parallel and the resulting tuple to be deposited in the tuple space. This mechanism can be used to perform an arbitrary computation. This is how parallel processes are created in Linda.

A common programming paradigm in Linda is the **replicated worker model**. This model is based on the idea of a **task bag** full of jobs to be done. The main process starts out by executing a loop containing

```
out("task-bag", job);
```

in which a different job description is output to the tuple space on each iteration. Each worker starts out by getting a job-description tuple using

```
in("task-bag", ?job);
```

which it then carries out. When it is done, it gets another. New work may also be put into the task bag during execution. In this simple way, work is dynamically divided among the workers, and each worker is kept busy all the time, all with relatively little overhead.

Various implementations of Linda on multicomputer systems exist. In all of them, a key issue is how to distribute the tuples among the machines and how to locate them when needed. Various possibilities include broadcasting and directories. Replication is also an important issue. These points are discussed in Bjornson (1993).

Orca

A somewhat different approach to application-level shared memory on a multicomputer is to use objects instead of just tuples as the unit of sharing. An object consists of internal (hidden) state plus methods for operating on that state. By not allowing the programmer to access the state directly, many possibilities are opened to allow sharing over machines that do not have physical shared memory.

One object-based system that gives the illusion of shared memory on multicomputer systems is Orca (Bal, 1991, Bal et al., 1992, and Bal and Tanenbaum, 1988). Orca is a traditional programming language (based on Modula 2) to which two new features have been added: objects and the ability to create new processes. An Orca object is an abstract data type, analogous to an object in Java or a package in Ada. It encapsulates internal data structures and user-written methods, called **operations**. Objects are passive, that is, they do not contain threads to which messages can be sent. Instead, processes access an object's internal data by invoking its methods.

Each Orca method consists of a list of (guard, block-of-statements) pairs. A guard is a Boolean expression that does not contain any side effects, or the empty

guard, which is the same as the value *true*. When an operation is invoked, all of its guards are evaluated in an unspecified order. If all of them are *false*, the invoking process is delayed until one becomes *true*. When a guard is found that evaluates to *true*, the block of statements following it is executed. Figure 8-48 depicts a *stack* object with two operations, *push* and *pop*.

```
Object implementation stack;
   top:integer;                          # storage for the stack
   stack: array [integer 0..N–1] of integer;

   operation push(item: integer);        # function returning nothing
   begin
      guard top < N – 1 do
         stack[top] := item;             # push item onto the stack
         top := top + 1;                 # increment the stack pointer
      od;
   end;

   operation pop( ): integer;            # function returning an integer
   begin
      guard top > 0 do                   # suspend if the stack is empty
         top := top – 1;                 # decrement the stack pointer
         return stack[top];              # return the top item
      od;
   end;

begin
   top := 0;                             # initialization
end;
```

Figure 8-48. A simplified ORCA stack object, with internal data and two operations.

Once a *stack* has been defined, variables of this type can be declared, as in

```
s, t: stack;
```

which creates two stack objects and initializes the *top* variable in each to 0. The integer variable *k* can be pushed onto the stack *s* by the statement

```
s$push(k);
```

and so forth. The *pop* operation has a guard, so an attempt to pop a variable from an empty stack will suspend the called until another process has pushed something on the stack.

Orca has a **fork** statement to create a new process on a user-specified processor. The new process runs the procedure named in the **fork** statement. Parameters, including objects, may be passed to the new process, which is how objects become distributed among machines. For example, the statement

```
for i in 1 .. n do fork foobar(s) on i; od;
```

generates one new process on each of machines 1 through n, running the program *foobar* in each of them. As these n new processes (and the parent) execute in parallel, they can all push and pop items onto the shared stack s as though they were all running on a shared-memory multiprocessor. It is the job of the run-time system to sustain the illusion of shared memory where it really does not exist.

Operations on shared objects are atomic and sequentially consistent. The system guarantees that if multiple processes perform operations on the same shared object nearly simultaneously, the system chooses some order and all processes see the same order of events.

Orca integrates shared data and synchronization in a way not present in page-based DSM systems. Two kinds of synchronization are needed in parallel programs. The first kind is mutual-exclusion synchronization, to keep two processes from executing the same critical region at the same time. In Orca, each operation on a shared object is effectively like a critical region because the system guarantees that the final result is the same as if all the critical regions were executed one at a time (i.e., sequentially). In this respect, an Orca object is like a distributed form of a monitor (Hoare, 1975).

The other kind of synchronization is condition synchronization, in which a process blocks waiting for some condition to hold. In Orca, condition synchronization is done with guards. In the example of Fig. 8-48, a process trying to pop an item from an empty stack will be suspended until the stack is no longer empty. After all, you cannot pop a word from an empty stack.

The Orca run-time system handles object replication, migration, consistency, and operation invocation. Each object can be in one of two states: single copy or replicated. An object in single-copy state exists on only one machine, so all requests for it are sent there. A replicated object is present on all machines containing a process using it, which makes read operations easier (since they can be done locally), at the expense of making updates more expensive. When an operation that modifies a replicated object is executed, it must first get a sequence number from a centralized process that issues them. Then a message is sent to each machine holding a copy of the object, telling it to execute the operation. Since all such updates bear sequence numbers, all machines just carry out the operations in sequence order, which guarantees sequential consistency.

8.4.7 Performance

The point of building a parallel computer is to make it go faster than a uniprocessor machine. If it does not achieve that simple goal, it is not worth having. Furthermore, it should achieve the goal in a cost-effective manner. A machine that is twice as fast as a uniprocessor at 50 times the cost is not likely to be a big seller. In this section we will examine some of the performance issues associated with parallel computer architectures, starting with how you even measure it.

Hardware Metrics

From a hardware perspective, the performance metrics of interest are the CPU and I/O speeds and the performance of the interconnection network. The CPU and I/O speeds are the same as in the uniprocessor case, so the key parameters of interest in a parallel system are those associated with the interconnect. There are two key items: latency and bandwidth, which we will now look at in turn.

The roundtrip latency is the time it takes for a CPU to send a packet and get a reply. If the packet is sent to a memory, then the latency measures the time to read or write a word or block of words. If it is sent to another CPU, it measures the interprocessor communication time for packets of that size. Usually, the latency of interest is for minimal packets, often one word or a small cache line.

The latency is built up from several factors and is different for circuit-switched, store-and-forward, virtual cut through, and wormhole-routed interconnects. For circuit switching, the latency is the sum of the setup time and the transmission time. To set up a circuit, a probe packet has to be sent out to reserve the resources and then report back. Once that has happened, the data packet has to be assembled. When it is ready, bits can flow at full speed, so if the total setup time is T_s, the packet size is p bits, and the bandwidth b bits/sec, the one-way latency is $T_s + p/b$. If the circuit is full duplex, then there is no setup time for the reply, so the minimum latency for sending a p-bit packet and getting a p-bit reply is $T_s + 2p/b$ sec.

For packet switching, it is not necessary to send a probe packet to the destination in advance, but there is still some internal setup time to assemble the packet, T_a. Here the one-way transmission time is $T_a + p/b$, but this is only the time to get the packet into the first switch. There is a finite delay within the switch, say T_d and then the process is repeated to the next switch and so on. The T_d delay is composed of both processing time and queueing delay, waiting for the output port to become free. If there are n switches, then the total one-way latency is given by the formula $T_a + n(p/b + T_d) + p/b$, where the final term is due to the copy from the last switch to the destination.

The one-way latencies for virtual cut through and wormhole routing in the best case are close to $T_a + p/b$ because there is no probe packet to set up a circuit, and no store-and-forward delay either. Basically, it is the initial setup time to assemble the packet, plus the time to push the bits out the door. In all cases, propagation delay has to be added, but that is usually small.

The other hardware metric is bandwidth. Many parallel programs, especially in the natural sciences, move a lot of data around, so the number of bytes/sec that the system can move is critical to performance. Several metrics for bandwidth exist. We have seen one of them—bisection bandwidth—already. Another is **aggregate bandwidth**, which is computed by simply adding up the capacities of all the links. This number gives the maximum number of bits that can be in transit at once. Yet another important metric is the average bandwidth out of each CPU.

If each CPU is capable of outputting 1 MB/sec, it does little good that the interconnect has a bisection bandwidth of 100 GB/sec. Communication will be limited by how much data each CPU can output.

In practice, actually achieving anything even close to the theoretical bandwidth is very difficult. Many sources of overhead work to reduce the capacity. For example, there is always some per-packet overhead associated with each packet: assembling it, building its header, and getting it going. Sending 1024 4-byte packets will never achieve the same bandwidth as sending one 4096-byte packet. Unfortunately, for achieving low latencies, using small packets is better, since large ones block the lines and switches too long. Thus there is an inherent conflict between achieving low average latencies and high-bandwidth utilization. For some applications, one is more important than the other and for other applications it is the other way around. It is worth noting, however, that you can always buy more bandwidth (by putting in more or wider wires), but you cannot buy lower latencies. Thus it is generally better to err on the side of making latencies as short as possible, and worry about bandwidth later.

Software Metrics

Hardware metrics like latency and bandwidth look at what the hardware is capable of doing. However, users have a different perspective. They want to know how much faster their programs are going to run on a parallel computer than on a uniprocessor. For them, the key metric is speed-up: how much faster a program runs on an n-processor system than on a one-processor system. Typically these results are shown in graphs like those of Fig. 8-49. Here we see several different parallel programs run on a multicomputer consisting of 64 Pentium Pro CPUs. Each curve shows the speed-up of one program with k CPUs as a function of k. Perfect speed-up is indicated by the dotted line, in which using k CPUs makes the program go k times faster, for any k. Few programs achieve perfect speed-up, but some come close. The N-body problem parallelizes extremely well; awari (an African board game) does reasonably well; but inverting a certain skyline matrix does not go more than five times faster no matter how many CPUs are available. The programs and results are discussed in Bal et al. (1998).

Part of the reason that perfect speed-up is nearly impossible to achieve is that almost all programs have some sequential component, often the initialization phase, reading in the data, or collecting the results. Having many CPUs does not help here. Suppose that a program runs for T sec on a uniprocessor, with a fraction f of this time being sequential code and a fraction $(1 - f)$ being potentially parallelizable, as shown in Fig. 8-50(a). If the latter code can be run on n CPUs with no overhead, its execution time can be reduced from $(1 - f)T$ to $(1 - f)T/n$ at best, as shown in Fig. 8-50(b). This gives a total execution time for the sequential and parallel parts of $fT + (1 - f)T/n$. The speed-up is just the execution time of the original program, T, divided by this new execution time:

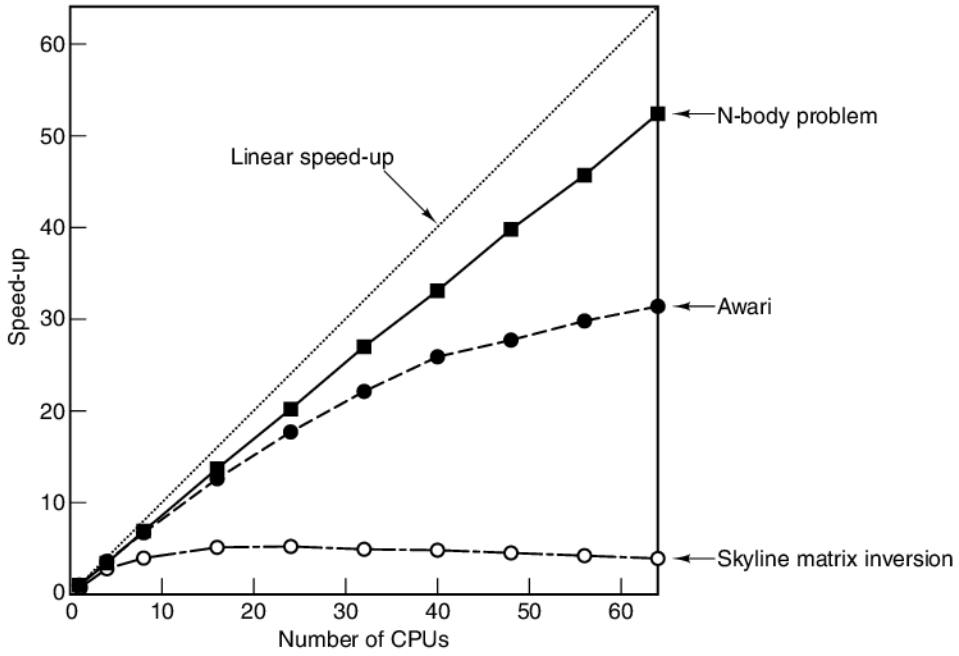


Figure 8-49. Real programs achieve less than linear speed-up.

$$Speed-up = \frac{n}{1 + (n-1)f}$$

For $f = 0$ we can get linear speed-up, but for $f > 0$, perfect speed-up is not possible due to the sequential component. This result is known as **Amdahl's law**.

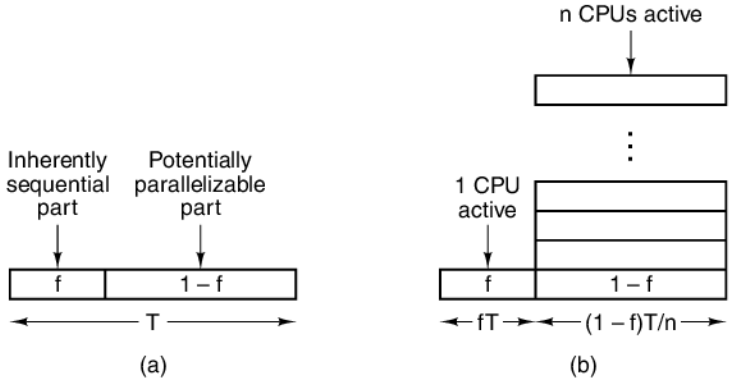


Figure 8-50. (a) A program has a sequential part and a parallelizable part. (b) Effect of running part of the program in parallel.

Amdahl's law is not the only reason perfect speed-up is nearly impossible to achieve. Nonzero communication latencies, finite communication bandwidths, and algorithmic inefficiencies can also play a role. Also, even if 1000 CPUs were available, not all programs can be written to make use of so many CPUs, and the overhead in getting them all started may be significant. Furthermore, sometimes the best-known algorithm does not parallelize well, so a suboptimal algorithm must be used in the parallel case. This all said, for many applications, having the program run n times faster is highly desirable, even if it takes $2n$ CPUs to do it. CPUs are not that expensive, after all, and many companies live with considerably less than 100% efficiency in other parts of their businesses.

Achieving High Performance

The most straightforward way to improve performance is to add more CPUs to the system. However, this addition must be done in such a way as to avoid creating any bottlenecks. A system in which one can add more CPUs and get correspondingly more computing power is said to be **scalable**.

To see some of the implications of scalability, consider four CPUs connected by a bus, as illustrated in Fig. 8-51(a). Now imagine scaling the system to 16 CPUs by adding 12 more, as shown in Fig. 8-51(b). If the bandwidth of the bus is b MB/sec, then by quadrupling the number of CPUs, we have also reduced the available bandwidth per CPU from b/4 MB/sec to b/16 MB/sec. Such a system is not scalable.

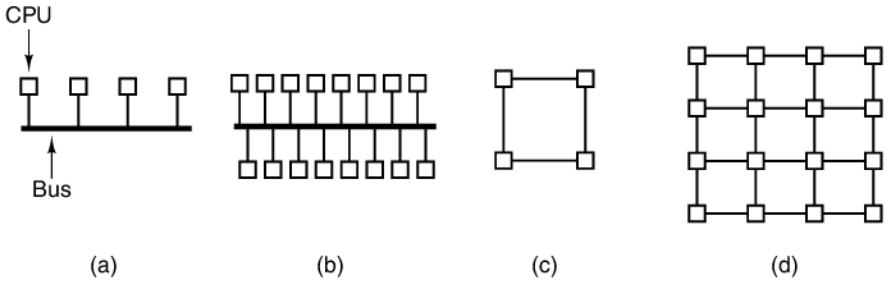


Figure 8-51. (a) A 4-CPU bus-based system. (b) A 16-CPU bus-based system. (c) A 4-CPU grid-based system. (d) A 16-CPU grid-based system.

Now we do the same thing with a grid-based system, as shown in Fig. 8-51(c) and Fig. 8-51(d). With this topology, adding new CPUs also adds new links, so scaling the system up does not cause the aggregate bandwidth per CPU to drop, as it does with a bus. In fact, the ratio of links to CPUs increases from 1.0 with 4 CPUs (4 CPUs, 4 links) to 1.5 with 16 CPUs (16 CPUs, 24 links), so adding CPUs improves the aggregate bandwidth per CPU.

Of course, bandwidth is not the only issue. Adding CPUs to the bus does not increase the diameter of the interconnection network or latency in the absence of traffic, whereas adding them to the grid does. For an $n \times n$ grid, the diameter is $2(n - 1)$, so the worst (and average) case latency increases roughly as the square root of the number of CPUs. For 400 CPUs, the diameter is 38, whereas for 1600 CPUs it is 78, so quadrupling the number of CPUs approximately doubles the diameter and thus the average latency.

Ideally, a scalable system should maintain the same average bandwidth per CPU and a constant average latency as more and more CPUs are added. In practice, however, keeping enough bandwidth per CPU is doable, but in all practical designs, latency grows with size. Having it grow logarithmically, as in a hypercube, is about the best that can be done.

The problem with having latency grow as the system scales up is that latency is often fatal to performance in fine- and medium-grained applications. If a program needs data that are not in its local memory, there is often a substantial delay in getting them, and the bigger the system, the longer the delay, as we have just seen. This problem is equally true of multiprocessors as of multicomputers, since in both cases the physical memory is invariably divided up into far-flung modules.

As a consequence of this observation, system designers often go to great lengths to reduce, or at least hide, the latency, using several techniques we will now mention. The first latency-hiding technique is data replication. If copies of a block of data can be kept at multiple locations, accesses from those locations can be speeded up. One such replication technique is caching, in which one or more copies of data blocks are kept close to where they are being used, as well as where they "belong." However, another strategy is to maintain multiple peer copies—copies that have equal status—as opposed to the asymmetric primary/secondary relationship used in caching. When multiple copies are maintained, in whatever form, key issues are where the data blocks are placed, when, and by whom. Answers range from dynamic placement on demand by the hardware, to intentional placement at load time following compiler directives. In all cases, managing consistency is an issue.

A second technique for hiding latency is **prefetching**. If a data item can be fetched before it is needed, the fetching process can be overlapped with normal execution, so that when the item is needed, it will be there. Prefetching can be automatic or under program control. When a cache loads not only the word being referenced, but an entire cache line containing the word, it is gambling that the succeeding words are also likely to be needed soon.

Prefetching can also be controlled explicitly. When the compiler realizes that it will need some data, it can put in an explicit instruction to go get them, and put that instruction sufficiently far in advance that the data will be there in time. This strategy requires that the compiler has a complete knowledge of the underlying machine and its timing, as well as control over where all data are placed. Such speculative LOAD instructions work best when it is known for sure that the data will

be needed. Getting a page fault on a LOAD for a path that is ultimately not taken is very costly.

A third technique that can hide latency is multithreading, as we have seen. If switching between processes can be made fast enough, for example, by giving each one its own memory map and hardware registers, then when one thread blocks waiting for remote data to arrive, the hardware can quickly switch to another one that is able to continue. In the limiting case, the CPU runs the first instruction from thread one, the second instruction from thread two, and so on. In this way, the CPU can be kept busy, even in the face of long memory latencies for the individual threads.

A fourth technique for hiding latency is using nonblocking writes. Normally, when a STORE instruction is executed, the CPU waits until the STORE has completed before continuing. With nonblocking writes, the memory operation is started, but the program just continues anyway. Continuing past a LOAD is harder, but with out-of-order execution, even that is possible.

8.5 GRID COMPUTING

Many of today's challenges in science, engineering, industry, the environment, and other areas are large scale and interdisciplinary. Solving them requires expertise, skills, knowledge, facilities, software, and data from multiple organizations, often in different countries. Some examples are as follows:

1. Scientists developing a mission to Mars.
2. A consortium building a complex product (e.g., a dam or aircraft).
3. An international relief team coordinating aid after a natural disaster.

Some of these are long-term cooperations, others are more short term, but they all share the common thread of requiring separate organizations with their own resources and procedures to work together to achieve a common goal.

Until recently, having different organizations, with different computers, operating systems, databases, and protocols work together to share resources and data has been very difficult. However, the growing need for large-scale interorganizational cooperation has led to the development of systems and technology for connecting widely separated computers into what is called the **grid**. In a sense, the grid is the next step along the axis of Fig. 8-1. It can be thought of as a very large, international, loosely coupled, heterogeneous, cluster.

The goal of the grid is to provide a technical infrastructure to allow a group of organizations that share a common goal to form a **virtual organization**. This virtual organization has to be flexible, with a large and changing membership, permitting the members to work together in areas they deem appropriate, while allowing them to maintain control over their own resources to whatever degree they wish.

To this end, grid researchers are developing services, tools, and protocols to enable these virtual organizations to function.

The grid is inherently multilateral with many participants who are peers. It can be contrasted with existing computing frameworks. In the client-server model, a transaction involves two parties: the server, who offers some service, and the client, who wants to use the service. A typical example of the client-server model is the Web, in which users go to Web servers to find information. The grid also differs from peer-to-peer applications, in which pairs of individuals exchange files. Email is a common example of a peer-to-peer application. Because the grid is different from these models, it requires new protocols and technology.

The grid needs access to a wide variety of resources. Each resource has a specific system and organization that owns it and that decides how much of the resource to make available to the grid, during which hours, and to whom. In an abstract sense, what the grid is about is resource access and management.

One way to model the grid is the layered hierarchy of Fig. 8-52. The **fabric layer** at the bottom is the set of components from which the grid is built. It includes CPUs, disks, networks, and sensors on the hardware side, and programs and data on the software side. These are the resources that the grid makes available in a controlled way.

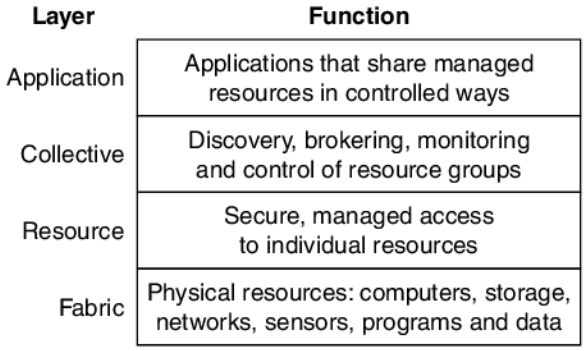


Figure 8-52. The grid layers.

One level higher is the **resource layer**, which manages the individual resources. In many cases, a resource participating in the grid has a local process that manages it and allows controlled access to it by remote users. This layer provides a uniform interface to higher layers for inquiring about the characteristics and status of individual resources, monitoring them, and using them in a secure way.

Next is the **collective layer**, which handles groups of resources. One of its functions is resource discovery, by which a user can locate available CPU cycles, disk space, or specific data. The collective layer may maintain directories or other databases to provide this information. It may also offer a brokering service by which the providers and users of services are matched up, possibly allocating scarce resources among competing users. The collective layer is also responsible

for replicating data, managing the admission of new members and resources to the grid, accounting, and maintaining the policy databases of who can use what.

Still further up is the **application layer**, where the user applications reside. It uses the lower layers to acquire credentials proving its right to use certain resources, submit usage requests, monitor the progress of these requests, deal with failures, and notify the user of the results.

Security is the key to a successful grid. Resource owners nearly always insist on maintaining tight control of their resources and want to determine who gets to use them, for how long, and how much. Without good security, no organization would make its resources available to the grid. On the other hand, if a user had to have a login account and password on every computer he wanted to use, using the grid would be unbearably cumbersome. Consequently, the grid has had to develop a security model to handle these concerns.

A key characteristic of the security model is the single sign-on. The first step in using the grid is to be authenticated and acquire a credential, a digitally signed document specifying on whose behalf the work is to be done. Credentials can be delegated, so that when a computation needs to create subcomputations, the child processes can also be identified. When a credential is presented at a remote machine, it has to be mapped onto the local security mechanism. On UNIX systems, for example, users are identified by 16-bit user IDs, but other systems have other schemes. Finally, the grid needs mechanisms to allow access policies to be stated, maintained, and updated.

In order to provide interoperability between different organizations and machines, standards are needed, in terms both of the services offered and of the protocols used to access them. The grid community has created an organization, the Global Grid Forum, to manage the standardization process. It has come up with a framework called **OGSA** (**Open Grid Services Architecture**) for positioning the various standards it is developing. Wherever possible, the standards utilize existing standards, for example, using WSDL (Web Services Definition Language) for describing OGSA services. The services being standardized currently fall into eight broad categories as follows, but no doubt new ones will be created later.

1. Infrastructure services (enable communication between resources).
2. Resource management services (reserve and deploy resources).
3. Data services (move and replicate data to where it is needed).
4. Context services (describe required resources and usage policies).
5. Information services (get information about resource availability).
6. Self-management services (support a stated quality of service).
7. Security services (enforce security policies).
8. Execution management services (manage workflow).

Much more could be said about the grid, but space limitations prevent us from pursuing this topic further. For more information about the grid, see Abramson (2011), Balasangameshwara and Raju (2012), Celaya and Arronategui (2011), Foster and Kesselman (2003), and Lee et al. (2011).

8.6 SUMMARY

It is getting increasingly difficult to make computers go faster by just revving up the clock due to increased heat dissipation problems and other factors. Instead, designers are looking to parallelism for speed-up. Parallelism can be introduced at many different levels, from very low, where the processing elements are very tightly coupled, to very high, where they are very loosely coupled.

At the bottom level is on-chip parallelism, in which parallel activities occur on a single chip. One form of on-chip parallelism is instruction-level parallelism, in which one instruction or a sequence of instructions issues multiple operations that can be executed in parallel by different functional units. A second form of on-chip parallelism is multithreading, in which the CPU can switch back and forth among multiple threads on an instruction-by-instruction basis, creating a virtual multiprocessor. A third form of on-chip parallelism is the single-chip multiprocessor, in which two or more cores are placed on the same chip to allow them to run at the same time.

One level up we find the coprocessors, typically plug-in boards that add extra processing power in some specialized area such as network protocol processing or multimedia. These extra processors relieve the main CPU of work, allowing it to do other things while they are performing their specialized tasks.

At the next level, we find the shared-memory multiprocessors. These systems contain two or more full-blown CPUs that share a common memory. UMA multiprocessors communicate via a shared (snooping) bus, a crossbar switch, or a multistage switching network. They are characterized by having a uniform access time to all memory locations. In contrast, NUMA multiprocessors also present all processes with the same shared address space, but here remote accesses take appreciably longer than local ones. Finally, COMA multiprocessors are yet another variation, in which cache lines move around the machine on demand but have no real home as in the other designs.

Multicomputers are systems with many CPUs that do not share a common memory. Each has its own private memory, with communication by message passing. MPPs are large multicomputers with specialized communication networks such as IBM's BlueGene/L. Clusters are simpler systems using off-the-shelf components, such as the engine that powers Google.

Multicomputers are often programmed using a message-passing package such as MPI. An alternative approach is to use application-level shared memory such as a page-based DSM system, the Linda tuple space, or Orca or Globe objects. DSM

simulates shared memory at the page level, making it similar to a NUMA machine, except with a much greater penalty for remote references.

Finally, at the highest level, and the most loosely coupled, are the grids. These are systems in which entire organizations are hooked together over the Internet to share compute power, data, and other resources.

PROBLEMS

1. Intel x86 instructions can be as long as 17 bytes. Is the x86 a VLIW CPU?
2. As process-design technology allows engineers to put ever more transistors on a chip, Intel and AMD have chosen to increase the number of cores on each chip. Are there any other feasible choices they could have made instead?
3. What are the clipped values of 96, −9, 300, and 256 when the clipping range is 0–255?
4. Are the following TriMedia instructions allowed, and if not, why not?

   a. Integer add, integer subtract, load, floating add, load immediate
   b. Integer subtract, integer multiply, load immediate, shift, shift
   c. Load immediate, floating add, floating multiply, branch, load immediate

5. Figure 8-7(d) and (e) show 12 cycles of instructions. For each one, tell what happens in the following three cycles.
6. On a particular CPU, an instruction that misses the level 1 cache but hits the level 2 cache takes k cycles in total. If multithreading is used to mask level 1 cache misses, how many threads must be run at once using fine-grained multithreading to avoid dead cycles?
7. The NVIDIA Fermi GPU is similar in spirit to one of the architectures we studied in Chap. 2. Which one?
8. One morning, the queen bee of a certain beehive calls in all her worker bees and tells them that today's assignment is to collect marigold nectar. The workers then fly off in different directions looking for marigolds. Is this an SIMD or an MIMD system?
9. During our discussion of memory consistency models, we said that a consistency model is a kind of contract between the software and the memory. Why is such a contract needed?
10. Consider a multiprocessor using a shared bus. What happens if two processors try to access the global memory at exactly the same instant?
11. Consider a multiprocessor using a shared bus. What happens if three processors try to access the global memory at exactly the same instant?
12. Suppose that for technical reasons it is possible for a snooping cache to snoop only on address lines, not on data lines. Would this change affect the write through protocol?

13. As a simple model of a bus-based multiprocessor system without caching, suppose that one instruction in every four references memory, and that a memory reference occupies the bus for an entire instruction time. If the bus is busy, the requesting CPU is put into a FIFO queue. How much faster will a 64-CPU system run than a 1-CPU system?

14. The MESI cache coherence protocol has four states. Other write-back cache coherence protocols have only three states. Which of the four MESI states could be sacrificed, and what would the consequences of each choice be? If you had to pick only three states, which would you pick?

15. Are there any situations with the MESI cache coherence protocol in which a cache line is present in the local cache but for which a bus transaction is nevertheless needed? If so, explain.

16. Suppose that there are n CPUs on a common bus. The probability that any CPU tries to use the bus in a given cycle is p. What is the chance that

a. The bus is idle (0 requests).
b. Exactly one request is made.
c. More than one request is made.

17. Name the major advantage and the major disadvantage of a crossbar switch.

18. How many crossbar switches does a full Sun Fire E25K have?

19. Suppose that the wire between switch 2A and switch 3B in the omega network of Fig. 8-31 breaks. Who is cut off from whom?

20. Hot spots (heavily referenced memory locations) are clearly a major problem in multistage switching networks. Are they also a problem in bus-based systems?

21. An omega switching network connects 4096 RISC CPUs, each with a 60-nsec cycle time, to 4096 infinitely fast memory modules. The switching elements each have a 5-nsec delay. How many delay slots are needed by a LOAD instruction?

22. Consider a machine using an omega switching network, like the one shown in Fig. 8-31. Suppose that the program and stack for processor i are kept in memory module i. Propose a slight change in the topology that makes a large difference in the performance (the IBM RP3 and BBN Butterfly use this modified topology). What disadvantage does your new topology have compared to the original?

23. In a NUMA multiprocessor, local memory references take 20 nsec and remote references 120 nsec. A certain program makes a total of N memory references during its execution, of which 1 percent are to a page P. That page is initially remote, and it takes C nsec to copy it locally. Under what conditions should the page be copied locally in the absence of significant use by other processors?

24. Consider a CC-NUMA multiprocessor like that of Fig. 8-33 except with 512 nodes of 8 MB each. If the cache lines are 64 bytes, what is the percentage overhead for the directories? Does increasing the number of nodes increase the overhead, decrease the overhead, or leave it unchanged?

25. What is the difference between NC-NUMA and CC-NUMA?

26. For each topology shown in Fig. 8-37, compute the diameter of the network.

27. For each topology shown in Fig. 8-37, determine the degree of fault tolerance each one has, defined as the maximum number of links that can be lost without partitioning the network in two.

28. Consider the double-torus topology of Fig. 8-37(f) but expanded to a size of $k \times k$. What is the diameter of the network? (*Hint*: Consider odd k and even k separately).

29. An interconnection network is in the form of an $8 \times 8 \times 8$ cube. Each link has a full-duplex bandwidth of 1 GB/sec. What is the bisection bandwidth of the network?

30. Amdahl's law limits the potential speed-up achievable on a parallel computer. Compute, as a function of f, the maximum possible speed-up as the number of CPUs approaches infinity. What are the implications of this limit for $f = 0.1$?

31. Figure 8-51 shows how scaling fails with a bus but succeeds with a grid. Assuming that each bus or link has a bandwidth b, compute the average bandwidth per CPU for each of the four cases. Then scale each system to 64 CPUs and repeat the calculations. What is the limit as the number of CPUs goes to infinity?

32. In the text, three variations of `send` were discussed: synchronous, blocking, and nonblocking. Give a fourth method that is similar to a blocking `send` but has slightly different properties. Give an advantage and a disadvantage of your method as compared to blocking `send`.

33. Consider a multicomputer running on a network with hardware broadcasting, such as Ethernet. Why does the ratio of read operations (those not updating internal state variables) to write operations (those updating internal state variables) matter?

9

BIBLIOGRAPHY

This chapter is an alphabetical bibliography of all books and articles cited in this book.

ABRAMSON, D.: "Mixing Cloud and Grid Resources for Many Task Computing," *Proc. Int'l Workshop on Many Task Computing on Grids and Supercomputers*, ACM, pp. 1–2, 2011.

ADAMS, M., and DULCHINOS, D.: "OpenCable," *IEEE Commun. Magazine*, vol. 39, pp. 98–105, June 2001.

ADIGA, N.R. et al.: "An Overview of the BlueGene/L Supercomputer," *Proc. Supercomputing 2002*, ACM, pp. 1–22, 2002.

ADVE, S.V., and HILL, M.: "Weak Ordering: A New Definition," *Proc. 17th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 2–14, 1990.

AGERWALA, T., and COCKE, J.: "High Performance Reduced Instruction Set Processors," IBM T.J. Watson Research Center Technical Report RC12434, 1987.

AHMADINIA, A., and SHAHRABI, A.: "A Highly Adaptive and Efficient Router Architecture for Network-on-Chip," *Computer J.*, vol. 54, pp. 1295–1307, Aug. 2011.

ALAM, S., BARRETT, R., BAST, M., FAHEY, M.R., KUEHN, J., McCURDY, ROGERS, J., ROTH, P., SANKARAN, R., VETTER, J.S., WORLEY, P., and YU, W.: "Early Evaluation of IBM BlueGene/P," *Proc. ACM/IEEE Conf. on Supercomputing*, ACM/IEEE, 2008.

ALAMELDEEN, A.R., and WOOD, D.A.: "Adaptive Cache Compression for High-Performance Processors," *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 212–223, 2004.

ALMASI, G.S. et al.: "System Management in the BlueGene/L Supercomputer," *Proc. 17th Int'l Parallel and Distr. Proc. Symp.*, IEEE, 2003a.

ALMASI, G.S. et al.: "An Overview of the Bluegene/L System Software Organization," *Par. Proc. Letters*, vol. 13, 561–574, April 2003b.

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., and ZWAENEPOEL, W.: "TreadMarks: Shared Memory Computing on a Network of Workstations," *IEEE Computer Magazine*, vol. 29, pp. 18–28, Feb. 1996.

ANDERSON, D.: *Universal Serial Bus System Architecture*, Reading, MA: Addison-Wesley, 1997.

ANDERSON, D., BUDRUK, R., and SHANLEY, T.: *PCI Express System Architecture*, Reading, MA: Addison-Wesley, 2004.

ANDERSON, T.E., CULLER, D.E., PATTERSON, D.A., and the NOW team: "A Case for NOW (Networks of Workstations)," *IEEE Micro Magazine*, vol. 15, pp. 54–64, Jan. 1995.

AUGUST, D.I., CONNORS, D.A., MAHLKE, S.A., SIAS, J.W., CROZIER, K.M., CHENG, B.-C., EATON, P.R., OLANIRAN, Q.B., and HWU, W.-M.: "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proc. 25th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 227–237, 1998.

BAL, H.E.: *Programming Distributed Systems*, Hemel Hempstead, England: Prentice Hall Int'l, 1991.

BAL, H.E., BHOEDJANG, R., HOFMAN, R, JACOBS, C., LANGENDOEN, K., RUHL, T., and KAASHOEK, M.F.: "Performance Evaluation of the Orca Shared Object System," *ACM Trans. on Computer Systems*, vol. 16, pp. 1–40, Jan.–Feb. 1998.

BAL, H.E., KAASHOEK, M.F., and TANENBAUM, A.S.: "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. on Software Engineering*, vol. 18, pp. 190–205, March 1992.

BAL, H.E., and TANENBAUM, A.S.: "Distributed Programming with Shared Data," *Proc. 1988 Int'l Conf. on Computer Languages*, IEEE, pp. 82–91, 1988.

BALASANGAMESHWARA, J., and RAJU, N.: "A Hybrid Policy for Fault Tolerant Load Balancing in Grid Computing Environments," *J. Network and Computer Applications*, vol. 35, pp. 412–422, Jan. 2012.

BARROSO, L.A., DEAN, J., and HOLZLE, U.: "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro Magazine*, vol. 23, pp. 22–28, March–April 2003.

BECHINI, A., CONTE, T.M., and PRETE, C.A.: "Opportunities and Challenges in Embedded Systems," *IEEE Micro Magazine*, vol. 24, pp. 8–9, July–Aug. 2004.

BHAKTHAVATCHALU, R., DEEPTHY, G.R.; and SHANOOJA, S.: "Implementation of Reconfigurable Open Core Protocol Compliant Memory System Using VHDL," *Proc. Int'l Conf. on Industrial and Information Systems*, pp. 213–218, 2010.

BJORNSON, R.D.: "Linda on Distributed Memory Multiprocessors," Ph.D. Thesis, Yale Univ., 1993.

BLUMRICH, M., CHEN, D., CHIU, G., COTEUS, P., GARA, A., GIAMPAPA, M.E., HARING, R.A., HEIDELBERGER, P., HOENICKE, D., KOPCSAY, G.V., OHMACHT, M., STEINMACHER-BUROW, B.D., TAKKEN, T., VRANSAS, P., and LIEBSCH, T.: "An Overview of the BlueGene/L System," *IBM J. Research and Devel.*, vol. 49, March–May, 2005.

BOSE, P.: "Computer Architecture Research: Shifting Priorities and Newer Challenges," *IEEE Micro Magazine*, vol. 24, p. 5, Nov.–Dec. 2004.

BOUKNIGHT, W.J., DENENBERG, S.A., MCINTYRE, D.E., RANDALL, J.M., SAMEH, A.H., and SLOTNICK, D.L.: "The Illiac IV System," *Proc. IEEE*, pp. 369–388, April 1972.

BRADLEY, D.: "A Personal History of the IBM PC," *IEEE Computer*, vol. 44, pp. 19–25, Aug. 2011.

BRIDE, E.: "The IBM Personal Computer: A Software-Driven Market," *IEEE Computer*, vol. 44, pp. 34–39, Aug. 2011.

BRIGHTWELL, R., CAMP, W., COLE, B., DEBENEDICTIS, E., LELAND, R., TOMPKINS, H, and MACCABE, A.B.: "Architectural Specification for Massively Parallel Supercomputers: An Experience-and-Measurement-Based Approach," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1–46, 2005.

BRIGHTWELL, R., UNDERWOOD, K.D., VAUGHAN, C., and STEVENSON, J.: "Performance Evaluation of the Red Storm Dual-Core Upgrade," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 175–190, Feb. 2010.

BURKHARDT, H., FRANK, S., KNOBE, B., and ROTHNIE, J.: "Overview of the KSR-1 Computer System," Technical Report KSR-TR-9202001, Kendall Square Research Corp., Cambridge, MA, 1992.

CARRIERO, N., and GELERNTER, D.: "Linda in Context," *Commun. of the ACM*, vol. 32, pp. 444–458, April 1989.

CELAYA, J., and ARRONATEGUI, U.: "A Highly Scalable Decentralized Scheduler of Tasks with Deadlines," *Proc. 12th Int'l Conf. on Grid Computing*, IEEE/ACM, pp. 58–65, 2011.

CHARLESWORTH, A.: "The Sun Fireplane Interconnect," *IEEE Micro Magazine*, vol. 22, pp. 36–45, Jan.–Feb. 2002.

CHARLESWORTH, A.: "The Sun Fireplane Interconnect," *Proc. Conf. on High Perf. Networking and Computing*, ACM, 2001.

CHEN, L., DROPSHO, S., and ALBONESI, D.H.: "Dynamic Data Dependence Tracking and Its Application to Branch Prediction," *Proc. Ninth Int'l Symp. on High-Performance Computer Arch.*, IEEE, pp. 65–78, 2003.

CHENG, L., and CARTER, J.B.: "Extending CC-NUMA Systems to Support Write Update Optimizations," *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, ACM/IEEE, 2008.

CHOU, Y., FAHS, B., and ABRAHAM, S.: " Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 76–77, 2004.

COHEN, D.: "On Holy Wars and a Plea for Peace," *IEEE Computer Magazine*, vol. 14, pp. 48–54, Oct. 1981.

CORBATO, F.J., and VYSSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. FJCC*, pp. 185–196, 1965.

DENNING, P.J.: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323–333, May 1968.

DIJKSTRA, E.W.: "GOTO Statement Considered Harmful," *Commun. of the ACM*, vol. 11, pp. 147–148, March 1968a.

DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, F. Genuys (ed.), New York: Academic Press, 1968b.

DONALDSON, G., and JONES, D.: "Cable Television Broadband Network Architectures," *IEEE Commun. Magazine*, vol. 39, pp. 122–126, June 2001.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 434–442, 1986.

DULONG, C.: "The IA-64 Architecture at Work," *IEEE Computer Magazine*, vol. 31, pp. 24–32, July 1998.

DUTTA-ROY, A.: "An Overview of Cable Modem Technology and Market Perspectives," *IEEE Commun. Magazine*, vol. 39, pp. 81–88, June 2001.

FAGGIN, F., HOFF, M.E., Jr., MAZOR, S., and SHIMA, M.: "The History of the 4004," *IEEE Micro Magazine*, vol. 16, pp. 10–20, Nov. 1996.

FALCON, A., STARK, J., RAMIREZ, A., LAI, K., and VALERO, M.: "Prophet/Critic Hybrid Branch Prediction," *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 250–261, 2004.

FISHER, J.A., and FREUDENBERGER, S.M.: "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. Fifth Int'l Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 85–95, 1992.

FLYNN, D.: "AMBA: Enabling Reusable On-Chip Designs," *IEEE Micro Magazine*, vol. 17, pp. 20–27, July 1997.

FLYNN, M.J.: "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers*, vol. C-21, pp. 948–960, Sept. 1972.

FOSTER, I., and KESSELMAN, C.: *The Grid 2: Blueprint for a New Computing Infrastructure*, San Francisco: Morgan Kaufman, 2003.

FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435–436, Oct. 1961.

FREITAS, H.C., MADRUGA, F.L., ALVES, M., and NAVAUX, P.: "Design of Interleaved Multithreading for Network Processors on Chip," *Proc. Int'l Symp. on Circuits and Systems*, IEEE, 2009.

GASPAR, L., FISCHER, V., BERNARD, F., BOSSUET, L., and COTRET, P.: "HCrypt: A Novel Concept of Crypto-processor with Secured Key Management," *Int'l Conf. on Reconfigurable Computing and FPGAs*, 2010.

GAUR, J., CHAUDHURI, C., and SUBRAMONEY, S.: "Bypass and Insertion Algorithms for Exclusive Last-level Caches," *Proc. 38th Int'l Symp. on Computer Arch.*, ACM, 2011.

GEBHART, M., JOHNSON, D.R., TARJAN, D., KECKLER, S.W., DALLY, W.J., LINDHOLM, E., and SKADRON, K.: "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," *Proc. 38th Int'l Symp. on Computer Arch.* ACM, 2011.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., and SUNDERRAM, V.: *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing*, Cambridge, MA: MIT Press, 1994.

GEPNER, P., GAMAYUNOV, V., and FRASER, D.L.: "The 2nd Generation Intel Core Processor. Architectural Features Supporting HPC," *Proc. 10th Int'l Symp. on Parallel and Dist. Computing*, pp. 17–24, 2011.

GERBER, R., and BINSTOCK, A.: *Programming with Hyper-Threading Technology*, Santa Clara, CA: Intel Press, 2004.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P.B., GUPTA, A., and HENNESSY, J.L.: "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. on Comp. Arch.*, ACM, pp. 15–26, 1990.

GHEMAWAT, S., GOBIOFF, H., and LEUNG, S.-T.: "The Google File System," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 29–43, 2003.

GOODMAN, J.R.: "Using Cache Memory to Reduce Processor Memory Traffic," *Proc. 10th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 124–131, 1983.

GOODMAN, J.R.: "Cache Consistency and Sequential Consistency," Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

GOTH, G.: "IBM PC Retrospective: There Was Enough Right to Make It Work," *IEEE Computer*, vol. 44, pp. 26–33, Aug. 2011.

GROPP, W., LUSK, E., and SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: MIT Press, 1994.

GUPTA, N., MANDAL, S., MALAVE, J., MANDAL, A., and MAHAPATRA, R.N.: "A Hardware Scheduler for Real Time Multiprocessor System on Chip," *Proc. 23rd Int'l Conf. on VLSI Design,* IEEE, 2010.

GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., and FRANKE, H.: "Reducing Disk Power Consumption in Servers with DRPM," *IEEE Computer Magazine*, vol. 36, pp. 59–66, Dec. 2003.

HAGERSTEN, E., LANDIN, A., and HARIDI, S.: "DDM—A Cache-Only Memory Architecture," *IEEE Computer Magazine*, vol. 25, pp. 44–54, Sept. 1992.

HAGHIGHIZADEH, F., ATTARZADEH, H., and SHARIFKHANI, M.: "A Compact 8-Bit AES Crypto-processor," *Proc. Second. Int'l Conf. on Computer and Network Tech.*, IEEE, 2010.

HAMMING, R.W.: "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, vol. 29, pp. 147–160, April 1950.

HENKEL, J., HU, X.S., and BHATTACHARYYA, S.S.: "Taking on the Embedded System Challenge," *IEEE Computer Magazine*, vol. 36, pp. 35–37, April 2003.

HENNESSY, J.L.: "VLSI Processor Architecture," *IEEE Trans. on Computers*, vol. C-33, pp. 1221–1246, Dec. 1984.

HERRERO, E., GONZALEZ, J., and CANAL, R.: "Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors," *Proc. 23rd Int'l Conf. on VLSI Design,* IEEE, 2010.

HOARE, C.A.R.: "Monitors: An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549–557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.

HWU, W.-M.: "Introduction to Predicated Execution," *IEEE Computer Magazine*, vol. 31, pp. 49–50, Jan. 1998.

JIMENEZ, D.A.: "Fast Path-Based Neural Branch Prediction," *Proc. 36th Int'l Symp. on Microarchitecture*, IEEE, pp. 243–252, 2003.

JOHNSON, K.L., KAASHOEK, M.F., and WALLACH, D.A.: "CRL: High-Performance All-Software Distributed Shared Memory," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213–228, 1995.

KAPASI, U.J., RIXNER, S., DALLY, W.J., KHAILANY, B., AHN, J.H., MATTSON, P., and OWENS, J.D.: "Programmable Stream Processors," *IEEE Computer Magazine*, vol. 36, pp. 54–62, Aug. 2003.

KAUFMAN, C., PERLMAN, R., and SPECINER, M.: *Network Security,* 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2002.

KIM, N.S., AUSTIN, T., BLAAUW, D., MUDGE, T., FLAUTNER, K., HU, J.S., IRWIN, M.J., KANDEMIR, M., and NARAYANAN, V.: "Leakage Current: Moore's Law Meets Static Power," *IEEE Computer Magazine*, vol. 36, 68–75, Dec. 2003.

KNUTH, D.E.: *The Art of Computer Programming: Fundamental Algorithms,* 3rd ed., Reading, MA: Addison-Wesley, 1997.

KONTOTHANASSIS, L., HUNT, G., STETS, R., HARDAVELLAS, N., CIERNIAD, M., PARTHASARATHY, S., MEIRA, W., DWARKADAS, S., and SCOTT, M.: "VM-Based Shared Memory on Low Latency Remote Memory Access Networks," *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 157–169, 1997.

LAMPORT, L.: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, vol. C-28, pp. 690–691, Sept. 1979.

LAROWE, R.P., and ELLIS, C.S.: "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 319–363, Nov. 1991.

LEE, J., KELEHER, P., and SUSSMAN, A.: "Supporting Computing Element Heterogeneity in P2P Grids," *Proc. IEEE Int'l Conf. on Cluster Computing*, IEEE, pp. 150–158, 2011.

LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

LIN, Y.-N., LIN, Y.-D., and LAI, Y.-C.: "Thread Allocation in CMP-based Multithreaded Network Processors," *Parallel Computing*, vol. 36, pp. 104–116, Feb. 2010.

LU, H., COX, A.L., DWARKADAS, S., RAJAMONY, R., and ZWAENEPOEL, W.: "Software Distributed Shared Memory Support for Irregular Applications," *Proc. Sixth Conf. on Prin. and Practice of Parallel Progr.*, pp. 48–56, June 1997.

LUKASIEWICZ, J.: *Aristotle's Syllogistic,* 2nd ed., Oxford: Oxford University Press, 1958.

LYYTINEN, K., and YOO, Y.: "Issues and Challenges in Ubiquitous Computing," *Commun. of the ACM*, vol. 45, pp. 63–65, Dec. 2002.

MARTIN, R.P., VAHDAT, A.M., CULLER, D.E., and ANDERSON, T.E.: "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 85–97, 1997.

MAYHEW, D., and KRISHNAN, V.: "PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects," *Proc. 11th Symp. on High Perf. Interconnects*, IEEE, pp. 21–29, Aug. 2003.

McKUSICK, M.K., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181–197, Aug. 1984.

McNAIRY, C., and SOLTIS, D.: "Itanium 2 Processor Microarchitecture," *IEEE Micro Magazine*, vol. 23, pp. 44–55, March-April 2003.

MISHRA, A.K., VIJAYKRISHNAN, N., and DAS, C.R.: "A Case for Heterogeneous On-Chip Interconnects for CMPs," *Proc. 38th Int'l Symp. on Computer Arch.* ACM, 2011.

MORGAN, C.: *Portraits in Computing*, New York: ACM Press, 1997.

MOUDGILL, M., and VASSILIADIS, S.: "Precise Interrupts," *IEEE Micro Magazine*, vol. 16, pp. 58–67, Jan. 1996.

MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software—Practice and Experience*, vol. 14, pp. 365–368, 1984.

NAEEM, A., CHEN, X., LU, Z., and JANTSCH, A.: "Realization and Performance Comparison of Sequential and Weak Memory Consistency Models in Network-On-Chip Based Multicore Systems," *Proc. 16th Design Automation Conf. Asia and South Pacific*, IEEE, pp. 154–159, 2011.

ORGANICK, E.: *The MULTICS System*, Cambridge, MA: MIT Press, 1972.

OSKIN, M., CHONG, F.T., and CHUANG, I.L.: "A Practical Architecture for Reliable Quantum Computers," *IEEE Computer Magazine*, vol. 35, pp. 79–87, Jan. 2002.

PAPAMARCOS, M., and PATEL., J.: "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. 11th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 348–354, 1984.

PARIKH, D., SKADRON, K., ZHANG, Y., and STAN, M.: "Power-Aware Branch Prediction: Characterization and Design," *IEEE Trans. on Computers*, vol. 53, 168–186, Feb. 2004.

PATTERSON, D.A.: "Reduced Instruction Set Computers," *Commun. of the ACM*, vol. 28, pp. 8–21, Jan. 1985.

PATTERSON, D.A., GIBSON, G., and KATZ, R.: "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109–166, 1988.

PATTERSON, D.A., and SEQUIN, C.H.: "A VLSI RISC," *IEEE Computer Magazine*, vol. 15, pp. 8–22, Sept. 1982.

POUNTAIN, D.: "Pentium: More RISC than CISC," *Byte*, vol. 18, pp. 195–204, Sept. 1993.

RADIN, G.: "The 801 Minicomputer," *Computer Arch. News*, vol. 10, pp. 39–47, March 1982.

RAMAN, S.K., PENTKOVSKI, V., and KESHAVA, J.: "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro Magazine*, vol. 20, pp. 47–57, July-Aug. 2000.

RITCHIE, D.M.: "Reflections on Software Research," *Commun. pf the ACM*, vol. 27, pp. 758–760, Aug. 1984.

RITCHIE, D.M., and THOMPSON, K.: "The UNIX Time-Sharing System," *Commun. of the ACM*, vol. 17, pp. 365–375, July 1974.

ROBINSON, G.S.: "Toward the Age of Smarter Storage," *IEEE Computer Magazine*, vol. 35, pp. 35–41, Dec. 2002.

ROSENBLUM, M., and OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System," *Proc. Thirteenth Symp. on Operating System Principles*, ACM, pp. 1–15, 1991.

RUSSINOVICH, M.E., and SOLOMON, D.A.: *Microsoft Windows Internals,* 4th ed., Redmond, WA: Microsoft Press, 2005.

RUSU, S., MULJONO, H., and CHERKAUER, B.: "Itanium 2 Processor 6M," *IEEE Micro Magazine*, vol. 24, pp. 10–18, March–April 2004.

SAHA, D., and MUKHERJEE, A.: "Pervasive Computing: A Paradigm for the 21st Century," *IEEE Computer Magazine*, vol. 36, pp. 25–31, March 2003.

SAKAMURA, K.: "Making Computers Invisible," *IEEE Micro Magazine*, vol. 22, pp. 7–11, 2002.

SANCHEZ, D., and KOZYRAKIS, C.: "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. 38th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 57–68, 2011.

SCALES, D.J., GHARACHORLOO, K., and THEKKATH, C.A.: "Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory," *Proc. Seventh Int'l Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, ACM, pp. 174–185, 1996.

SELTZER, M., BOSTIC, K., McKUSICK, M.K., and STAELIN, C.: "An Implementation of a Log-Structured File System for UNIX," *Proc. Winter 1993 USENIX Technical Conf.*, pp. 307–326, 1993.

SHANLEY, T., and ANDERSON, D.: *PCI System Architecture,* 4th ed., Reading, MA: Addison-Wesley, 1999.

SHOUFAN, A., HUBER, N., and MOLTER, H.G.: "A Novel Cryptoprocessor Architecture for Chained Merkle Signature Schemes," *Microprocessors and Microsystems*, vol. 35, pp. 34–47, Feb. 2011.

SINGH, G.: "The IBM PC: The Silicon Story," *IEEE Computer*, vol. 44, pp. 40–45, Aug. 2011.

SLATER, R.: *Portraits in Silicon*, Cambridge, MA: MIT Press, 1987.

SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D.W., and DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: MIT Press, 1996.

SOLARI, E., and CONGDON, B.: *PCI Express Design & System Architecture*, Research Tech, Inc., 2005.

SOLARI, E., and WILLSE, G.: *PCI and PCI-X Hardware and Software,* 6th ed., San Diego, CA: Annabooks, 2004.

SORIN, D.J., HILL, M.D., and WOOD, D.A.: *A Primer on Memory Consistency and Cache Coherence*, San Francisco: Morgan & Claypool, 2011.

STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., and SCOTT, M.: "CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170–183, 1997.

SUMMERS, C.K.: *ADSL: Standards, Implementation, and Architecture*, Boca Raton, FL: CRC Press, 1999.

SUNDERRAM, V.B..: "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, Dec. 1990.

SWAN, R.J., FULLER, S.H., and SIEWIOREK, D.P.: "Cm*—A Modular Multiprocessor," *Proc. NCC*, pp. 645–655, 1977.

TAN, W.M.: *Developing USB PC Peripherals*, San Diego, CA: Annabooks, 1997.

TANENBAUM, A.S., and WETHERALL, D.J.: *Computer Networks,* 5th ed., Upper Saddle River, NJ: Prentice Hall, 2011.

THOMPSON, K.: "Reflections on Trusting Trust," *Commun. of the ACM*, vol. 27, pp. 761–763, Aug. 1984.

THOMPSON, J., DREISIGMEYER, D.W., JONES, T., KIRBY, M., and LADD, J.: "Accurate Fault Prediction of BlueGene/P RAS Logs via Geometric Reduction," IEEE, pp. 8–14, 2010.

TRELEAVEN, P.: "Control-Driven, Data-Driven, and Demand-Driven Computer Architecture," *Parallel Computing*, vol. 2, 1985.

TU, X., FAN, X., JIN, H., ZHENG, L., and PENG, X.: "Transactional Memory Consistency: A New Consistency Model for Distributed Transactional Memory," *Proc. Third Int'l Joint Conf. on Computational Science and Optimization*, IEEE, 2010.

VAHALIA, U.: *UNIX Internals*, Upper Saddle River, NJ: Prentice Hall, 1996.

VAHID, F.: "The Softening of Hardware," *IEEE Computer Magazine*, vol. 36, pp. 27–34, April 2003.

VETTER, P., GODERIS, D., VERPOOTEN, L., and GRANGER, A.: "Systems Aspects of APON/VDSL Deployment," *IEEE Commun. Magazine*, vol. 38, pp. 66–72, May 2000.

VU, T.D., ZHANG, L., and JESSHOPE, C.: "The Verification of the On-Chip COMA Cache Coherence Protocol," *Proc. 12th Int'l Conf. on Algebraic Methodology and Software Technology*, Springer-Verlag, pp. 413–429, 2008.

WEISER, M.: "The Computer for the 21st Century," *IEEE Pervasive Computing*, vol. 1, pp. 19–25, Jan.–March 2002; originally published in *Scientific American*, Sept. 1991.

WILKES, M.V.: "Computers Then and Now," *J. ACM*, vol. 15, pp. 1–7, Jan. 1968.

WILKES, M.V.: "The Best Way to Design an Automatic Calculating Machine," *Proc. Manchester Univ. Computer Inaugural Conf.*, 1951.

WING-KEI, Y., HUANG, R., XU, S., WANG, S.-E., KAN, E., and SUH, G.E.: "SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading Architectures," *Proc. 38th Int'l Symp. on Computer Arch.* ACM, 2011.

YAMAMOTO, S., and NAKAO, A.: "Fast Path Performance of Packet Cache Router Using Multi-core Network Processor," *Proc. Seventh Symp. on Arch. for Network and Comm. Sys.*, ACM/IEEE, 2011.

ZHANG, L., and JESSHOPE, C.: "On-Chip COMA Cache-Coherence Protocol for Micro-grids of Microthreaded Cores," *Proc. of 2007 European Conf. on Parallel Processing*, Springer-Verlag, pp. 38–48, 2008.

A

BINARY NUMBERS

The arithmetic used by computers differs in some ways from the arithmetic used by people. The most important difference is that computers perform operations on numbers whose precision is finite and fixed. Another difference is that most computers use the binary rather than the decimal system for representing numbers. These topics are the subject of this appendix.

A.1 FINITE-PRECISION NUMBERS

While doing arithmetic, one usually gives little thought to the question of how many decimal digits it takes to represent a number. Physicists can calculate that there are 10^{78} electrons in the universe without being bothered by the fact that it requires 79 decimal digits to write that number out in full. Someone calculating the value of a function with pencil and paper who needs the answer to six significant digits simply keeps intermediate results to seven, or eight, or however many are needed. The problem of the paper not being wide enough for seven-digit numbers never arises.

With computers, matters are quite different. On most computers, the amount of memory available for storing a number is fixed at the time that the computer is designed. With a certain amount of effort, the programmer can represent numbers two, or three, or even many times larger than this fixed amount, but doing so does not change the nature of this difficulty. The finite nature of the computer forces us

to deal only with numbers that can be represented in a fixed number of digits. We call such numbers **finite-precision numbers**.

In order to study properties of finite-precision numbers, let us examine the set of positive integers representable by three decimal digits, with no decimal point and no sign. This set has exactly 1000 members: 000, 001, 002, 003, ..., 999. With this restriction, it is impossible to express certain kinds of numbers, such as

1. Numbers larger than 999.
2. Negative numbers.
3. Fractions.
4. Irrational numbers.
5. Complex numbers.

One important property of arithmetic on the set of all integers is **closure** with respect to the operations of addition, subtraction, and multiplication. In other words, for every pair of integers i and j, $i + j$, $i - j$, and $i \times j$ are also integers. The set of integers is not closed with respect to division, because there exist values of i and j for which i/j is not expressible as an integer (e.g., 7/2 and 1/0).

Finite-precision numbers are not closed with respect to any of these four basic operations, as shown below, using three-digit decimal numbers as an example:

600 + 600 = 1200	(too large)
003 − 005 = −2	(negative)
050 × 050 = 2500	(too large)
007 / 002 = 3.5	(not an integer)

The violations can be divided into two mutually exclusive classes: operations whose result is larger than the largest number in the set (overflow error) or smaller than the smallest number in the set (underflow error), and operations whose result is neither too large nor too small but is simply not a member of the set. Of the four violations above, the first three are examples of the former, and the fourth is an example of the latter.

Because computers have finite memories and therefore must of necessity perform arithmetic on finite-precision numbers, the results of certain calculations will be, from the point of view of classical mathematics, just plain wrong. A calculating device that gives the wrong answer even though it is in perfect working condition may appear strange at first, but the error is a logical consequence of its finite nature. Some computers have special hardware that detects overflow errors.

The algebra of finite-precision numbers is different from normal algebra. As an example, consider the associative law:

$$a + (b - c) = (a + b) - c$$

Let us evaluate both sides for $a = 700$, $b = 400$, $c = 300$. To compute the left-hand

side, first calculate $(b - c)$, which is 100, and then add this amount to a, yielding 800. To compute the right-hand side, first calculate $(a + b)$, which gives an overflow in the finite arithmetic of three-digit integers. The result may depend on the machine being used but it will not be 1100. Subtracting 300 from some number other than 1100 will not yield 800. The associative law does not hold. The order of operations is important.

As another example, consider the distributive law:

$$a \times (b - c) = a \times b - a \times c$$

Let us evaluate both sides for $a = 5$, $b = 210$, $c = 195$. The left-hand side is 5×15, which yields 75. The right-hand side is not 75 because $a \times b$ overflows.

Judging from these examples, one might conclude that although computers are general-purpose devices, their finite nature renders them especially unsuitable for doing arithmetic. This conclusion is, of course, not true, but it does serve to illustrate the importance of understanding how computers work and what limitations they have.

A.2 RADIX NUMBER SYSTEMS

An ordinary decimal number with which everyone is familiar consists of a string of decimal digits and, possibly, a decimal point. The general form and its usual interpretation are shown in Fig. A-1. The choice of 10 as the base for exponentiation, called the **radix**, is made because we are using decimal, or base 10, numbers. When dealing with computers, it is frequently convenient to use radices other than 10. The most important radices are 2, 8, and 16. The number systems based on these radices are called **binary**, **octal**, and **hexadecimal**, respectively.

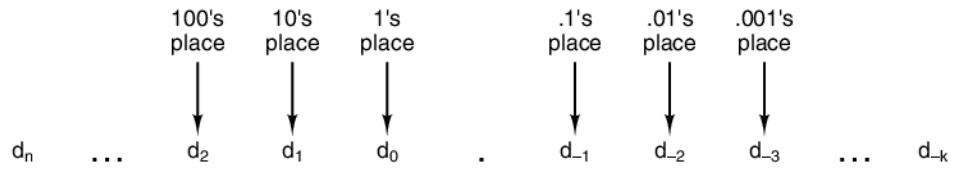


Figure A-1. The general form of a decimal number.

A radix k number system requires k different symbols to represent the digits 0 to $k - 1$. Decimal numbers are built up from the 10 decimal digits

0 1 2 3 4 5 6 7 8 9

In contrast, binary numbers do not use these ten digits. They are all constructed exclusively from the two binary digits

0 1

Octal numbers are built up from the eight octal digits

0 1 2 3 4 5 6 7

For hexadecimal numbers, 16 digits are needed. Thus six new symbols are required. It is conventional to use the uppercase letters A through F for the six digits following 9. Hexadecimal numbers are then built up from the digits

0 1 2 3 4 5 6 7 8 9 A B C D E F

The expression "binary digit" meaning a 1 or a 0 is usually referred to as a **bit**. Figure A-2 shows the decimal number 2001 expressed in binary, octal, decimal, and hexadecimal form. The number 7B9 is obviously hexadecimal, because the symbol B can only occur in hexadecimal numbers. However, the number 111 might be in any of the four number systems discussed. To avoid ambiguity, people use a subscript of 2, 8, 10, or 16 to indicate the radix when it is not obvious from the context.

Binary	1	1	1	1	1	0	1	0	0	0	1
	1×2^{10}	$+\ 1\times2^{9}$	$+\ 1\times2^{8}$	$+\ 1\times2^{7}$	$+\ 1\times2^{6}$	$+\ 0\times2^{5}$	$+\ 1\times2^{4}$	$+\ 0\times2^{3}$	$+\ 0\times2^{2}$	$+\ 0\times2^{1}$	$+\ 1\times2^{0}$
	1024	+ 512	+ 256	+ 128	+ 64	+ 0	+16	+ 0	+ 0	+ 0	+ 1
Octal	3	7	2	1							
	3×8^{3}	$+\ 7\times8^{2}$	$+\ 2\times8^{1}$	$+\ 1\times8^{0}$							
	1536	+ 448	+ 16	+ 1							
Decimal	2	0	0	1							
	2×10^{3}	$+\ 0\times10^{2}$	$+\ 0\times10^{1}$	$+\ 1\times10^{0}$							
	2000	+ 0	+ 0	+ 1							
Hexadecimal	7	D	1								
	7×16^{2}	$+\ 13\times16^{1}$	$+\ 1\times16^{0}$								
	1792	+ 208	+ 1								

Figure A-2. The number 2001 in binary, octal, and hexadecimal.

As an example of binary, octal, decimal, and hexadecimal notation, consider Fig. A-3, which shows a collection of nonnegative integers expressed in each of these four different systems. Perhaps some archaeologist thousands of years from now will discover this table and regard it as the Rosetta Stone to late twentieth century and early twenty-first century number systems.

Decimal	Binary	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Figure A-3. Decimal numbers and their binary, octal, and hexadecimal equivalents.

A.3 CONVERSION FROM ONE RADIX TO ANOTHER

Conversion between octal or hexadecimal numbers and binary numbers is easy. To convert a binary number to octal, divide it into groups of 3 bits, with the 3 bits immediately to the left (or right) of the decimal point (often called a binary point) forming one group, the 3 bits immediately to their left, another group, and so on. Each group of 3 bits can be directly converted to a single octal digit, 0 to 7, according to the conversion given in the first lines of Fig. A-3. It may be necessary to add one or two leading or trailing zeros to fill out a group to 3 full bits. Conversion from octal to binary is equally trivial. Each octal digit is simply replaced by the equivalent 3-bit binary number. Conversion from hexadecimal to binary is just like

octal-to-binary except that each hexadecimal digit corresponds to a group of 4 bits instead of 3 bits. Figure A-4 gives some examples of conversions.

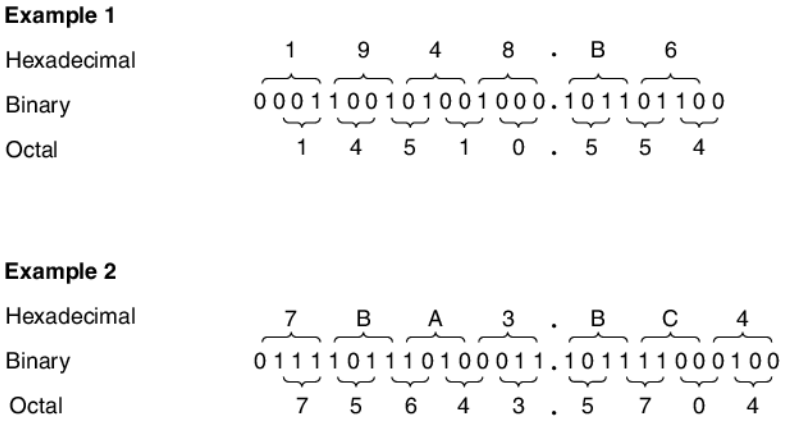


Figure A-4. Examples of octal-to-binary and hexadecimal-to-binary conversion.

Conversion of decimal numbers to binary can be done in two different ways. The first method follows directly from the definition of binary numbers. The largest power of 2 smaller than the number is subtracted from the number. The process is then repeated on the difference. Once the number has been decomposed into powers of 2, the binary number can be assembled with 1s in the bit positions corresponding to powers of 2 used in the decomposition, and 0s elsewhere.

The other method (for integers only) consists of dividing the number by 2. The quotient is written directly beneath the original number and the remainder, 0 or 1, is written next to the quotient. The quotient is then considered and the process repeated until the number 0 has been reached. The result of this process will be two columns of numbers, the quotients and the remainders. The binary number can now be read directly from the remainder column starting at the bottom. Figure A-5 gives an example of decimal-to-binary conversion.

Binary integers can also be converted to decimal in two ways. One method consists of summing up the powers of 2 corresponding to the 1 bits in the number. For example,

$$10110 = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$

In the other method, the binary number is written vertically, one bit per line, with the leftmost bit on the bottom. The bottom line is called line 1, the one above it line 2, and so on. The decimal number will be built up in a parallel column next to the binary number. Begin by writing a 1 on line 1. The entry on line n consists of two times the entry on line $n - 1$ plus the bit on line n (either 0 or 1). The entry on the top line is the answer. Figure A-6 gives an example of this method of binary to decimal conversion.

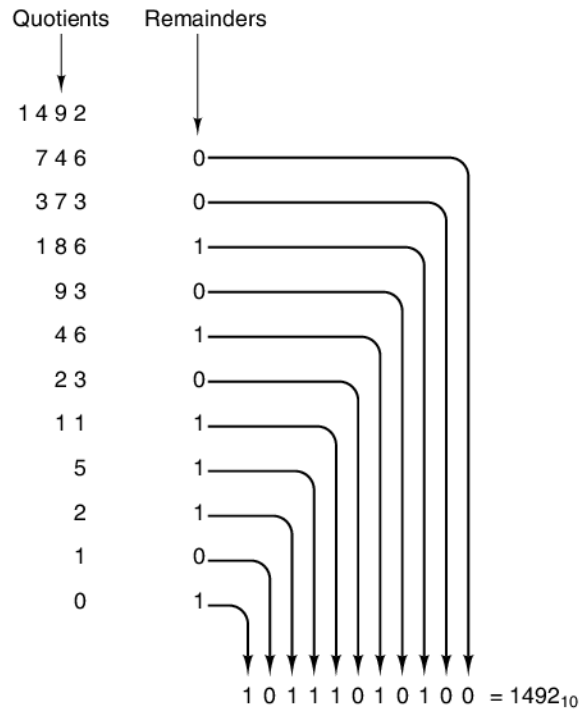


Figure A-5. Conversion of the decimal number 1492 to binary by successive halving, starting at the top and working downward. For example, 93 divided by 2 yields a quotient of 46 and a remainder of 1, written on the line below it.

Decimal-to-octal and decimal-to-hexadecimal conversion can be accomplished either by first converting to binary and then to the desired system or by subtracting powers of 8 or 16.

A.4 NEGATIVE BINARY NUMBERS

Four different systems for representing negative numbers have been used in digital computers at one time or another in history. The first one is called **signed magnitude**. In this system the leftmost bit is the sign bit (0 is + and 1 is −) and the remaining bits hold the absolute magnitude of the number.

The second system, called **one's complement**, also has a sign bit with 0 used for plus and 1 for minus. To negate a number, replace each 1 by a 0 and each 0 by a 1. This holds for the sign bit as well. One's complement is obsolete.

The third system, called **two's complement**, also has a sign bit that is 0 for plus and 1 for minus. Negating a number is a two-step process. First, each 1 is replaced by a 0 and each 0 by a 1, just as in one's complement. Second, 1 is added

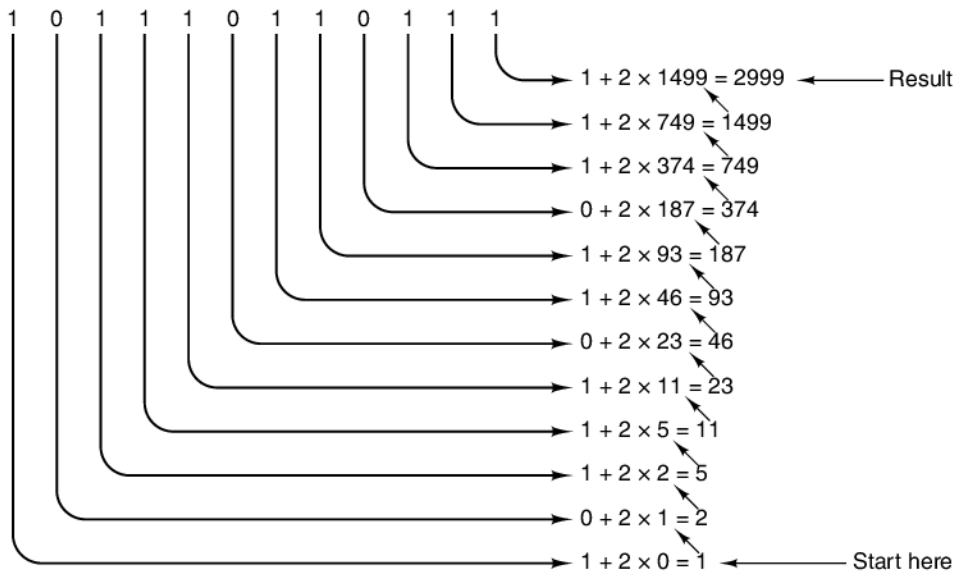


Figure A-6. Conversion of the binary number 101110110111 to decimal by successive doubling, starting at the bottom. Each line is formed by doubling the one below it and adding the corresponding bit. For example, 749 is twice 374 plus the 1 bit on the same line as 749.

to the result. Binary addition is the same as decimal addition except that a carry is generated if the sum is greater than 1 rather than greater than 9. For example, converting 6 to two's complement is done in two steps:

00000110 (+6)
11111001 (–6 in one's complement)
11111010 (–6 in two's complement)

If a carry occurs from the leftmost bit, it is thrown away.

The fourth system, which for m-bit numbers is called **excess 2^{m-1}**, represents a number by storing it as the sum of itself and 2^{m-1}. For example, for 8-bit numbers, $m = 8$, the system is called excess 128 and a number is stored as its true value plus 128. Therefore, –3 becomes –3 + 128 = 125, and –3 is represented by the 8-bit binary number for 125 (01111101). The numbers from –128 to +127 map onto 0 to 255, all of which are expressible as an 8-bit positive integer. Interestingly enough, this system is identical to two's complement with the sign bit reversed. Figure A-7 gives examples of negative numbers in all four systems.

Both signed magnitude and one's complement have two representations for zero: a plus zero, and a minus zero. This situation is undesirable. The two's complement system does not have this problem because the two's complement of plus zero is also plus zero. The two's complement system does, however, have a dif-

N decimal	N binary	–N signed mag.	–N 1's compl.	–N 2's compl.	–N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

Figure A-7. Negative 8-bit numbers in four systems.

ferent singularity. The bit pattern consisting of a 1 followed by all 0s is its own complement. The result is to make the range of positive and negative numbers unsymmetric; there is one negative number with no positive counterpart.

The reason for these problems is not hard to find: we want an encoding system with two properties:

1. Only one representation for zero.
2. Exactly as many positive numbers as negative numbers.

The problem is that any set of numbers with as many positive as negative numbers and only one zero has an odd number of members, whereas m bits allow an even number of bit patterns. There will always be either one bit pattern too many or one bit pattern too few, no matter what representation is chosen. This extra bit pattern can be used for –0 or for a large negative number, or for something else, but no matter what it is used for it will always be a nuisance.

A.5 BINARY ARITHMETIC

The addition table for binary numbers is given in Fig. A-8.

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

Figure A-8. The addition table in binary.

Two binary numbers can be added, starting at the rightmost bit and adding the corresponding bits in the addend and the augend. If a carry is generated, it is carried one position to the left, just as in decimal arithmetic. In one's complement arithmetic, a carry generated by the addition of the leftmost bits is added to the rightmost bit. This process is called an end-around carry. In two's complement arithmetic, a carry generated by the addition of the leftmost bits is merely thrown away. Examples of binary arithmetic are shown in Fig. A-9.

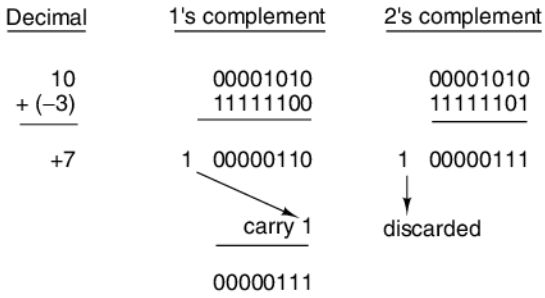


Figure A-9. Addition in one's complement and two's complement.

If the addend and the augend are of opposite signs, overflow error cannot occur. If they are of the same sign and the result is of the opposite sign, overflow error has occurred and the answer is wrong. In both one's and two's complement arithmetic, overflow occurs if and only if the carry into the sign bit differs from the carry out of the sign bit. Most computers preserve the carry out of the sign bit, but the carry into the sign bit is not visible from the answer. For this reason, a special overflow bit is usually provided.

PROBLEMS

1. Convert the following numbers to binary: 1984, 4000, 8192.

2. What is 1001101001 (binary) in decimal? In octal? In hexadecimal?

3. Which of the following are valid hexadecimal numbers? BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD.

4. Express the decimal number 100 in all radices from 2 to 9.

5. How many different positive integers can be expressed in k digits using radix r numbers?

6. Most people can only count to 10 on their fingers; however, computer scientists can do better. If you regard each finger as one binary bit, with finger extended as 1 and finger touching palm as 0, how high can you count using both hands? With both hands and both feet? Now use both hands and both feet, with the big toe on your left foot as a sign bit for two's complement numbers. What is the range of expressible numbers?

7. Perform the following calculations on 8-bit two's complement numbers.

00101101	11111111	00000000	11110111
+ 01101111	+ 11111111	− 11111111	− 11110111

8. Repeat the calculation of the preceding problem but now in one's complement.

9. Consider the following addition problems for 3-bit binary numbers in two's complement. For each sum, state
a. Whether the sign bit of the result is 1.
b. Whether the low-order 3 bits are 0.
c. Whether an overflow occurred.

000	000	111	100	100
+ 001	+ 111	+ 110	+ 111	+ 100

10. Signed decimal numbers consisting of n digits can be represented in $n + 1$ digits without a sign. Positive numbers have 0 as the leftmost digit. Negative numbers are formed by subtracting each digit from 9. Thus the negative of 014725 is 985274. Such numbers are called nine's complement numbers and are analogous to one's complement binary numbers. Express the following as three-digit nine's complement numbers: 6, −2, 100, −14, −1, 0.

11. Determine the rule for addition of nine's complement numbers and then perform the following additions.

0001	0001	9997	9241
+ 9999	+ 9998	+ 9996	+ 0802

12. Ten's complement is analogous to two's complement. A ten's complement negative number is formed by adding 1 to the corresponding nine's complement number, ignoring the carry. What is the rule for ten's complement addition?

13. Construct the multiplication tables for radix 3 numbers.

14. Multiply 0111 and 0011 in binary.

15. Write a program that takes in a signed decimal number as an ASCII string and prints out its representation in two's complement in binary, octal, and hexadecimal.

16. Write a program that takes in two 32-character ASCII strings containing 0s and 1s, each representing a two's complement 32-bit binary number. The program should print their sum as a 32-character ASCII string of 0s and 1s.

B

FLOATING-POINT NUMBERS

In many calculations the range of numbers used is very large. For example, a calculation in astronomy might involve the mass of the electron, 9×10^{-28} grams, and the mass of the sun, 2×10^{33} grams, a range exceeding 10^{60}. These numbers could be represented by

```
000000000000000000000000000000000.000000000000000000000000000009
200000000000000000000000000000000.000000000000000000000000000000
```

and all calculations could be carried out keeping 34 digits to the left of the decimal point and 28 places to the right of it. Doing so would allow 62 significant digits in the results. On a binary computer, multiple-precision arithmetic could be used to provide enough significance. However, the mass of the sun is not even known accurately to five significant digits, let alone 62. In fact few measurements of any kind can (or need) be made accurately to 62 significant digits. Although it would be possible to keep all intermediate results to 62 significant digits and then throw away 50 or 60 of them before printing the final results, doing this is wasteful of both CPU time and memory.

What is needed is a system for representing numbers in which the range of expressible numbers is independent of the number of significant digits. In this appendix, such a system will be discussed. It is based on the scientific notation commonly used in physics, chemistry, and engineering.

B.1 PRINCIPLES OF FLOATING POINT

One way of separating the range from the precision is to express numbers in the familiar scientific notation

$$n = f \times 10^e$$

where f is called the **fraction**, or **mantissa**, and e is a positive or negative integer called the **exponent**. The computer version of this notation is called **floating point**. Some examples of numbers expressed in this form are

$$\begin{array}{llll} 3.14 & = 0.314 & \times 10^1 & = 3.14 \times 10^0 \\ 0.000001 & = 0.1 & \times 10^{-5} & = 1.0 \times 10^{-6} \\ 1941 & = 0.1941 & \times 10^4 & = 1.941 \times 10^3 \end{array}$$

The range is effectively determined by the number of digits in the exponent and the precision is determined by the number of digits in the fraction. Because there is more than one way to represent a given number, one form is usually chosen as the standard. In order to investigate the properties of this method of representing numbers, consider a representation, R, with a signed three-digit fraction in the range $0.1 \le |f| < 1$ or zero and a signed two-digit exponent. These numbers range in magnitude from $+0.100 \times 10^{-99}$ to $+0.999 \times 10^{+99}$, a span of nearly 199 orders of magnitude, yet only five digits and two signs are needed to store a number.

Floating-point numbers can be used to model the real-number system of mathematics, although there are some differences. Figure B-1 gives an exaggerated schematic of the real number line. The real line is divided up into seven regions:

1. Large negative numbers less than -0.999×10^{99}.
2. Negative numbers between -0.999×10^{99} and -0.100×10^{-99}.
3. Small negative numbers with magnitudes less than 0.100×10^{-99}.
4. Zero.
5. Small positive numbers with magnitudes less than 0.100×10^{-99}.
6. Positive numbers between 0.100×10^{-99} and 0.999×10^{99}.
7. Large positive numbers greater than 0.999×10^{99}.

One major difference between the set of numbers representable with three fraction and two exponent digits and the real numbers is that the former cannot be used to express any numbers in regions 1, 3, 5, or 7. If the result of an arithmetic operation yields a number in regions 1 or 7—for example, $10^{60} \times 10^{60} = 10^{120}$—**overflow error** will occur and the answer will be incorrect. The reason for this is due to the finite nature of the representation for numbers and is thus unavoidable. Similarly, a result in regions 3 or 5 cannot be expressed either. This situation is called

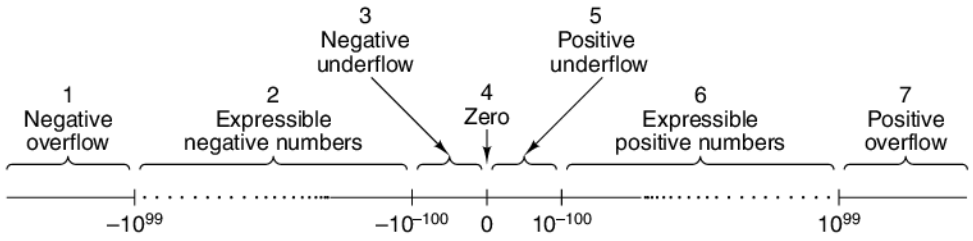


Figure B-1. The real number line can be divided into seven regions.

underflow error. Underflow error is less serious than overflow error, because 0 is often a satisfactory approximation to numbers in regions 3 and 5. A bank balance of 10^{-102} dollars is hardly better than a bank balance of 0.

Another important difference between floating-point numbers and real numbers is their density. Between any two real numbers, x and y, is another real number, no matter how close x is to y. This property comes from the fact that for any distinct real numbers, x and y, $z = (x + y)/2$ is a real number between them. The real numbers form a continuum.

Floating-point numbers, in contrast, do not form a continuum. Exactly 179,100 positive numbers can be expressed in the five-digit, two-sign system used above, 179,100 negative numbers, and 0 (which can be expressed in many ways), for a total of 358,201 numbers. Of the infinite number of real numbers between -10^{+100} and $+0.999 \times 10^{99}$, only 358,201 of them can be specified by this notation. They are symbolized by the dots in Fig. B-1. It is quite possible for the result of a calculation to be one of the other numbers, even though it is in region 2 or 6. For example, $+0.100 \times 10^{3}$ divided by 3 cannot be expressed *exactly* in our system of representation. If the result of a calculation cannot be expressed in the number representation being used, the obvious thing to do is to use the nearest number that can be expressed. This process is called **rounding**.

The spacing between adjacent expressible numbers is not constant throughout region 2 or 6. The separation between $+0.998 \times 10^{99}$ and $+0.999 \times 10^{99}$ is vastly more than the separation between $+0.998 \times 10^{0}$ and $+0.999 \times 10^{0}$. However, when the separation between a number and its successor is expressed as a percentage of that number, there is no systematic variation throughout region 2 or 6. In other words, the **relative error** introduced by rounding is approximately the same for small numbers as large numbers.

Although the preceding discussion was in terms of a representation system with a three-digit fraction and a two-digit exponent, the conclusions drawn are valid for other representation systems as well. Changing the number of digits in the fraction or exponent merely shifts the boundaries of regions 2 and 6 and changes the number of expressible points in them. Increasing the number of digits in the fraction increases the density of points and therefore improves the accuracy

of approximations. Increasing the number of digits in the exponent increases the size of regions 2 and 6 by shrinking regions 1, 3, 5, and 7. Figure B-2 shows the approximate boundaries of region 6 for floating-point decimal numbers for various sizes of fraction and exponent.

Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figure B-2. The approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers.

A variation of this representation is used in computers. For efficiency, exponentiation is to base 2, 4, 8, or 16 rather than 10, in which case the fraction consists of a string of binary, base-4, octal, or hexadecimal digits. If the leftmost of these digits is zero, all the digits can be shifted one place to the left and the exponent decreased by 1, without changing the value of the number (barring underflow). A fraction with a nonzero leftmost digit is said to be **normalized**.

Normalized numbers are generally preferable to unnormalized numbers, because there is only one normalized form, whereas there are many unnormalized forms. Examples of normalized floating-point numbers are given in Fig. B-3 for two bases of exponentiation. In these examples a 16-bit fraction (including sign bit) and a 7-bit exponent using excess 64 notation are shown. The radix point is to the left of the leftmost fraction bit—that is, to the right of the exponent.

B.2 IEEE FLOATING-POINT STANDARD 754

Until about 1980, each computer manufacturer had its own floating-point format. Needless to say, all were different. Worse yet, some of them actually did arithmetic incorrectly because floating-point arithmetic has some subtleties not obvious to the average hardware designer.

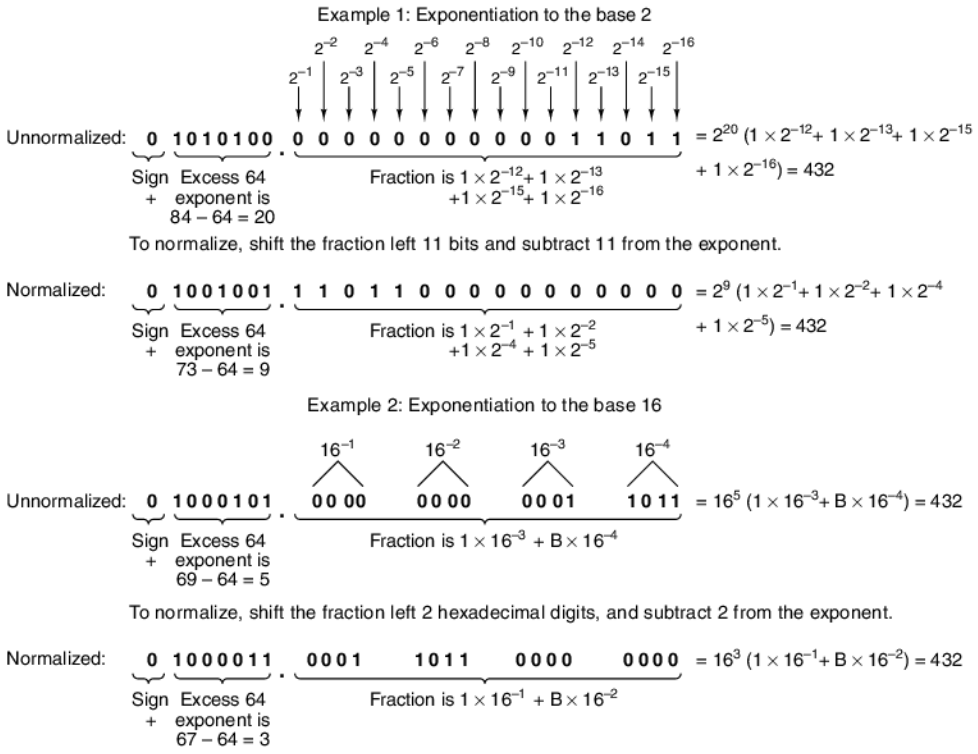


Figure B-3. Examples of normalized floating-point numbers.

To rectify this situation, in the late 1970s IEEE set up a committee to standardize floating-point arithmetic. The goal was not only to permit floating-point data to be exchanged among different computers but also to provide hardware designers with a model known to be correct. The resulting work led to IEEE Standard 754 (IEEE, 1985). Most CPUs these days (including the Intel and JVM ones studied in this book) have floating-point instructions that conform to the IEEE floating-point standard. Unlike many standards, which tend to be wishy-washy compromises that please no one, this one is not bad, in large part because it was primarily the work of one person, Berkeley math professor William Kahan. The standard will be described in the remainder of this section.

The standard defines three formats: single precision (32 bits), double precision (64 bits), and extended precision (80 bits). The extended-precision format is intended to reduce roundoff errors. It is used primarily inside floating-point arithmetic units, so we will not discuss it further. Both the single- and double-precision formats use radix 2 for fractions and excess notation for exponents. The formats are shown in Fig. B-4.

Both formats start with a sign bit for the number as a whole, 0 being positive and 1 being negative. Next comes the exponent, using excess 127 for single

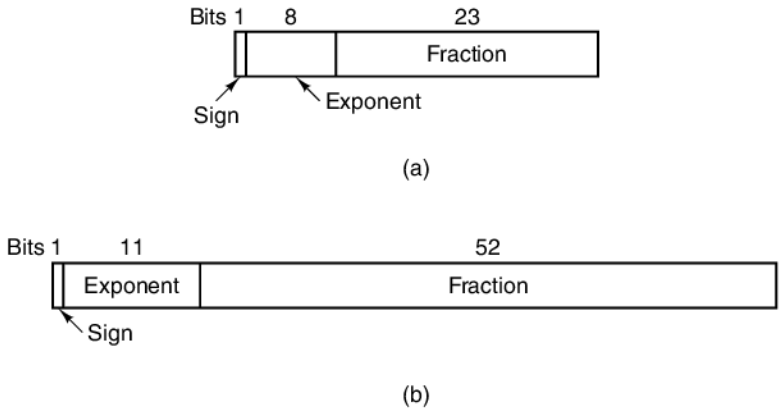


Figure B-4. IEEE floating-point formats. (a) Single precision. (b) Double precision.

precision and excess 1023 for double precision. The minimum (0) and maximum (255 and 2047) exponents are not used for normalized numbers; they have special uses described below. Finally, we have the fractions, 23 and 52 bits, respectively.

A normalized fraction begins with a binary point, followed by a 1 bit, and then the rest of the fraction. Following a practice started on the PDP-11, the authors of the standard realized that the leading 1 bit in the fraction does not have to be stored, since it can just be assumed to be present. Consequently, the standard defines the fraction in a slightly different way than usual. It consists of an implied 1 bit, an implied binary point, and then either 23 or 52 arbitrary bits. If all 23 or 52 fraction bits are 0s, the fraction has the numerical value 1.0; if all of them are 1s, the fraction is numerically slightly less than 2.0. To avoid confusion with a conventional fraction, the combination of the implied 1, the implied binary point, and the 23 or 52 explicit bits is called a **significand** instead of a fraction or mantissa. All normalized numbers have a significand, s, in the range $1 \leq s < 2$.

The numerical characteristics of the IEEE floating-point numbers are given in Fig. B-5. As examples, consider the numbers 0.5, 1, and 1.5 in normalized single-precision format. These are represented in hexadecimal as 3F000000, 3F800000, and 3FC00000, respectively.

One of the traditional problems with floating-point numbers is how to deal with underflow, overflow, and uninitialized numbers. The IEEE standard deals with these problems explicitly, borrowing its approach in part from the CDC 6600. In addition to normalized numbers, the standard has four other numerical types, described below and shown in Fig. B-6.

A problem arises when the result of a calculation has a magnitude smaller than the smallest normalized floating-point number that can be represented in this system. Previously, most hardware took one of two approaches: just set the result to zero and continue, or cause a floating-point underflow trap. Neither of these is

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Figure B-5. Characteristics of IEEE floating-point numbers.

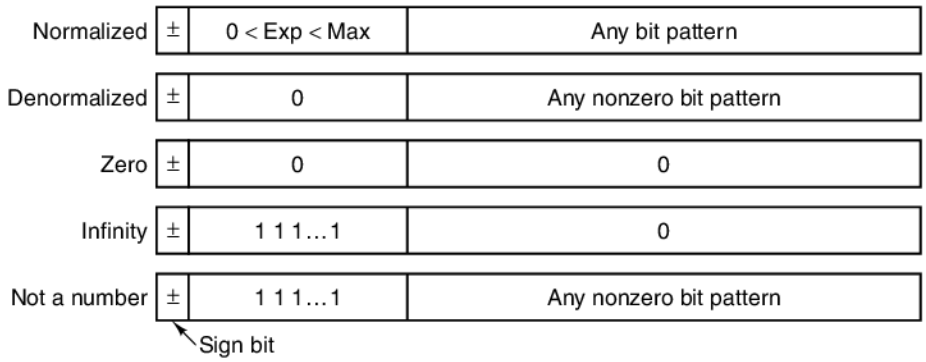


Figure B-6. IEEE numerical types.

really satisfactory, so IEEE invented **denormalized numbers**. These numbers have an exponent of 0 and a fraction given by the following 23 or 52 bits. The implicit 1 bit to the left of the binary point now becomes a 0. Denormalized numbers can be distinguished from normalized ones because the latter are not permitted to have an exponent of 0.

The smallest normalized single precision number has a 1 as exponent and 0 as fraction, and represents 1.0×2^{-126}. The largest denormalized number has a 0 as exponent and all 1s in the fraction, and represents about $0.9999999 \times 2^{-126}$, which is almost the same thing. One thing to note however, is that this number has only 23 bits of significance, versus 24 for all normalized numbers.

As calculations further decrease this result, the exponent stays put at 0, but the first few bits of the fraction become zeros, reducing both the value and the number of significant bits in the fraction. The smallest nonzero denormalized number consists of a 1 in the rightmost bit, with the rest being 0. The exponent represents

2^{-126} and the fraction represents 2^{-23} so the value is 2^{-149}. This scheme provides for a graceful underflow by giving up significance instead of jumping to 0 when the result cannot be expressed as a normalized number.

Two zeros are present in this scheme, positive and negative, determined by the sign bit. Both have an exponent of 0 and a fraction of 0. Here too, the bit to the left of the binary point is implicitly 0 rather than 1.

Overflow cannot be handled gracefully. There are no bit combinations left. Instead, a special representation is provided for infinity, consisting of an exponent with all 1s (not allowed for normalized numbers), and a fraction of 0. This number can be used as an operand and behaves according to the usual mathematical rules for infinity. For example infinity plus anything is infinity, and any finite number divided by infinity is zero. Similarly, any finite number divided by zero yields infinity.

What about infinity divided by infinity? The result is undefined. To handle this case, another special format is provided, called **NaN** (**Not a Number**). It too, can be used as an operand with predictable results.

PROBLEMS

1. Convert the following numbers to IEEE single-precision format. Give the results as eight hexadecimal digits.

a. 9
b. 5/32
c. −5/32
d. 6.125

2. Convert the following IEEE single-precision floating-point numbers from hex to decimal:

a. 42E48000H
b. 3F880000H
c. 00800000H
d. C7F00000H

3. The format of single-precision floating-point numbers on the 370 has a 7-bit exponent in the excess 64 system, and a fraction containing 24 bits plus a sign bit, with the binary point at the left end of the fraction. The radix for exponentiation is 16. The order of the fields is sign bit, exponent, fraction. Express the number 7/64 as a normalized number in this system in hex.

4. The following binary floating-point numbers consist of a sign bit, an excess 64, radix 2 exponent, and a 16-bit fraction. Normalize them.

a. 0 1000000 0001010100000001
b. 0 0111111 0000001111111111
c. 0 1000011 1000000000000000

5. To add two floating-point numbers, you must adjust the exponents (by shifting the fraction) to make them the same. Then you can add the fractions and normalize the result, if need be. Add the single-precision IEEE numbers 3EE00000H and 3D800000H and express the normalized result in hexadecimal.

6. The Tightwad Computer Company has decided to come out with a machine having 16-bit floating-point numbers. The Model 0.001 has a floating-point format with a sign bit, 7-bit, excess 64 exponent, and 8-bit fraction. The Model 0.002 has a sign bit, 5-bit, excess 16 exponent, and 10-bit fraction. Both use radix 2 exponentiation. What are the smallest and largest positive normalized numbers on each model? About how many decimal digits of precision does each have? Would you buy either one?

7. There is one situation in which an operation on two floating-point numbers can cause a drastic reduction in the number of significant bits in the result. What is it?

8. Some floating-point chips have a square root instruction built in. A possible algorithm is an iterative one (e.g., Newton-Raphson). Iterative algorithms need an initial approximation and then steadily improve it. How can one obtain a fast approximate square root of a floating-point number?

9. Write a procedure to add two IEEE single-precision floating-point numbers. Each number is represented by a 32-element Boolean array.

10. Write a procedure to add two single-precision floating-point numbers that use radix 16 for the exponent and radix 2 for the fraction but do not have an implied 1 bit to the left of the binary point. A normalized number has 0001, 0010, ..., 1111 as the leftmost 4 bits of the fraction, but not 0000. A number is normalized by shifting the fraction left 4 bits and subtracting 1 from the exponent.

This page intentionally left blank

C

ASSEMBLY LANGUAGE PROGRAMMING

Evert Wattel
Vrije Universiteit
Amsterdam, The Netherlands

Every computer has an **ISA** (**Instruction Set Architecture**), which is a set of registers, instructions, and other features visible to its low-level programmers. This ISA is commonly referred to as **machine language**, although the term is not entirely accurate. A program at this level of abstraction is a long list of binary numbers, one per instruction, telling which instructions to execute and what their operands are. Programming with binary numbers is very difficult to do, so all machines have an **assembly language**, a symbolic representation of the instruction set architecture, with symbolic names like ADD, SUB, and MUL, instead of binary numbers. This appendix is a tutorial on assembly language programming for one specific machine, the Intel 8088, which was used in the original IBM PC and was the base from which the modern Core i7 grew. The appendix also covers the use of some tools that can be downloaded to help learn about assembly language programming.

The purpose of this appendix is not to turn out polished assembly language programmers, but to help the reader learn about computer architecture through hands-on experience. For this reason, a simple machine—the Intel 8088— has

been chosen as the running example. While 8088s are rarely encountered any more, every Core i7 is capable of executing 8088 programs, so the lessons learned here are still applicable to modern machines. Furthermore, most of the Core i7's basic instructions are the same as the 8088's, only using 32-bit registers instead of 16-bit registers. Thus, this appendix can also be seen as a gentle introduction to Core i7 assembly language programming.

In order to program any machine in assembly language, the programmer must have a detailed knowledge of the machine's instruction set architecture. Accordingly, Sections C.1 through C.4 of this appendix are devoted to the architecture of the 8088, its memory organization, addressing modes, and instructions. Section C.5 discusses the assembler, which is used in this appendix and which is available for free, as described later. The notation used in this appendix is the one used by this assembler. Other assemblers use different notations, so readers already familiar with 8088 assembly programming should be alert for differences. Section C.6 discusses an interpreter/tracer/debugger tool, which can be downloaded to help the beginner programmer get programs debugged. Section C.7 describes the installation of the tools, and how to get started. Section C.8 contain programs, examples, exercises and solutions.

C.1 OVERVIEW

We will start our tour of assembly language programming with a few words on assembly language and then give a small example to illustrate it.

C.1.1 Assembly Language

Every assembler uses **mnemonics**, that is, short words such as ADD, SUB, and MUL for machine instructions such as add, subtract, and multiply, to make them easy to remember. In addition, assemblers allow the use of **symbolic names** for constants and **labels** to indicate instruction and memory addresses. Also, most assemblers support some number of **pseudoinstructions**, which do not translate into ISA instructions, but which are commands to the assembler to guide the assembly process.

When a program in assembly language is fed to a program called an **assembler**, the assembler converts the program into a **binary program** suitable for actual execution. This program can then be run on the actual hardware. However, when beginners start to program in assembly language, they often make errors and the binary program just stops, without any clue as to what went wrong. To make life easier for beginners, it is sometimes possible to run the binary program not on the actual hardware, but on a simulator, which executes one instruction at a time and gives a detailed display of what it is doing. In this way, debugging is much

easier. Programs running on a simulator run slowly, of course, but when the goal is to learn assembly language programming, rather than run a production job, this loss of speed is not important. This appendix is based on a toolkit that includes such a simulator, called the **interpreter** or **tracer**, as it interprets and traces the execution of the binary program step by step as it runs. The terms "simulator," "interpreter," and "tracer" will be used interchangeably throughout this appendix. Usually, when we are talking about just executing a program, we will speak of the "interpreter" and when we are talking about using it as a debugging tool, we will call it the "tracer," but it is the same program.

C.1.2 A Small Assembly Language Program

To make some of these abstract ideas a bit more concrete, consider the program and tracer image of Fig. C-1. An image of the tracer screen is given in Fig. C-1. Fig. C-1(a) shows a simple assembly language program for the 8088. The numbers following the exclamation marks are the source line numbers, to make it easier to refer to parts of the program. A copy of this program can be found in the accompanying material, in the directory *examples* in the source file *HlloWrld.s*. This assembly program, like all assembly programs discussed in this appendix, has the suffix *.s*, which indicates that it is an assembly language source program. The tracer screen, shown in Fig. C-1(b), contains seven windows, each containing different information about the state of the binary program being executed.

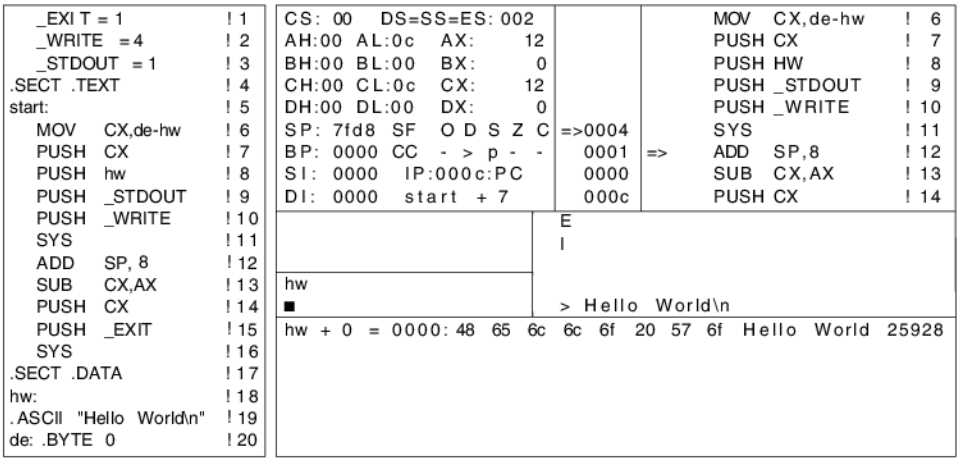


Figure C-1. (a) An assembly language program. (b) The corresponding tracer display.

Let us now briefly examine the seven windows of Fig. C-1(b). On the top are three windows, two larger ones and a smaller one in the middle. The top left

window shows the contents of the processor, consisting of the current values of the segment registers, CS, DS, SS, and ES, the arithmetic registers, AH, AL, AX, and others.

The middle window in the top row contains the stack, an area of memory used for temporary values.

The right-hand window in the top row contains a fragment of the assembly language program, with the arrow showing which instruction is currently being executed. As the program runs, the current instruction changes and the arrow moves to point to it. The strength of the tracer is that by hitting the return key (labeled Enter on PC keyboards), one instruction is executed and all the windows are updated, making it possible to run the program in slow motion.

Below the left window is a window that contains the subroutine call stack, here empty. Below it are commands to the tracer itself. To the right of these two windows is a window for input, output, and error messages.

Below these windows is a window that shows a portion of memory. These windows will be discussed in more detail later, but the basic idea should be clear: the tracer shows the source program, the machine registers, and quite a bit of information about the state of the program being executed. As each instruction is executed the information is updated, allowing the user to see in great detail what the program is doing.

C.2 THE 8088 PROCESSOR

Every processor, including the 8088, has an internal state, where it keeps certain crucial information. For this purpose, the processor has a set of **registers** where this information can be stored and processed. Probably the most important of these is the PC (**program counter**), which contains the memory location, that is, the **address**, of the next instruction to be executed. This register is also called IP (**Instruction Pointer**). This instruction is located in a part of the main memory, called the **code segment**. The main memory on the 8088 may be up to slightly more the 1 MB in size, but the current code segment is only 64 KB. The CS register in Fig. C-1 tells where the 64-KB code segment begins within the 1-MB memory. A new code segment can be activated by simply changing the value of the CS register. Similarly, there is also a 64-KB data segment, which tells where the data begins. In Fig. C-1 its origin is given by the DS register, which can also be changed as needed to access data outside the current data segment. The CS and DS registers are needed because the 8088 has 16-bit registers, so they cannot directly hold the 20-bit addresses needed to reference the entire 1-MB memory. This is why the code and data segment registers were introduced.

The other registers contain data or pointers to data in the main memory. In assembly language programs, these registers can be directly accessed. Apart from

these registers, the processor also contains all the necessary equipment to perform the instructions, but these parts are available to the programmer only through the instructions.

C.2.1 The Processor Cycle

The operation of the 8088 (and all other computers) consists of executing instructions, one after another. The execution of a single instruction can be broken down into the following steps:

1. Fetch the instruction from memory from the code segment using PC.
2. Increment the program counter.
3. Decode the fetched instruction.
4. Fetch the necessary data from memory and/or processor registers.
5. Perform the instruction.
6. Store the results of the instruction in memory and/or registers.
7. Go back to step 1 to start the next instruction.

The execution of an instruction is somewhat like running a very small program. In fact, some machines really do have a little program, called a **microprogram**, to execute their instructions. Microprograms are described in detail in Chap. 4.

From the point of view of an assembly programmer, the 8088 has a set of 14 registers. These registers are in some sense the scratch pad where the instructions operate and are in constant use, although the results stored in them are very volatile. Figure C-2 gives an overview of these 14 registers. It is clear that this figure and the register window of the tracer of Fig. C-1 are very similar because they represent the same information.

The 8088 registers are 16 bits wide. No two registers are completely functionally equivalent, but some of them share certain features, so they are subdivided into groups in Fig. C-2. We will now discuss the different groups.

C.2.2 The General Registers

The registers in the first group, AX, BX, CX, and DX are the **general registers**. The first register of this group, AX, is called the **accumulator register**. It is used to collect results of computations and is the target of many of the instructions. Although each of the registers can perform a host of tasks, in some instructions this AX is the implied destination, for example, in multiplication.

The second register of this group is BX, the **base register**. For many purposes BX can be used in the same way as AX, but it has one power AX does not have. It is

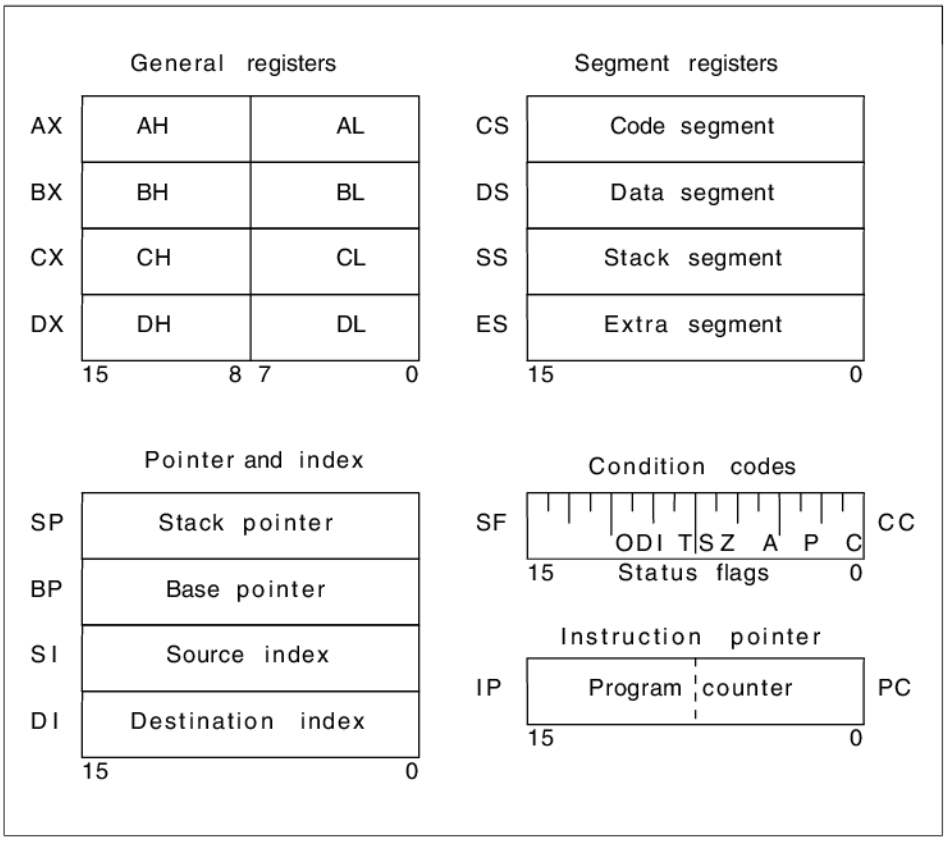


Figure C-2. The 8088 registers.

possible to put a memory address in BX and then execute an instruction whose operand comes from the memory address contained in BX. In other words, BX can hold a pointer to memory, AX cannot. To show this, we compare two instructions. First we have

```
MOV AX,BX
```

which copies to AX the contents of BX. Second we have

```
MOV AX,(BX)
```

which copies to AX the contents of the memory word whose address is contained in BX. In the first example, BX contains the source operand; in the second one it points to the source operand. In both of these examples, note that the MOV

instruction has a source and a destination operand, and that the destination is written before the source.

The next general register is CX, the **counter register**. Besides fulfilling many other tasks, this register is specifically used to contain counters for loops. It is automatically decremented in the LOOP instruction, and loops are usually terminated when CX reaches zero.

The fourth register of the general group is DX, the **data register**. It is used together with AX in double word length (i.e., 32-bit) instructions. In this case, DX contains the high-order 16 bits and AX contains the low-order 16 bits. Usually, 32-bit integers are indicated by the term **long**. The term **double** is usually reserved for 64-bit floating point values, although some people use "double" for 32-bit integers. In this tutorial, there will be no confusion because we will not discuss floating-point numbers at all.

All of these general registers can be regarded either as a 16-bit register or as a pair of 8-bit registers. In this way, the 8088 has precisely eight different 8-bit registers, which can be used in byte and character instructions. None of the other registers can be split into 8-bit halves. Some instructions use an entire register, such as AX, but other instructions use only half of a register, such as AL or AH. In general, instructions doing arithmetic use the full 16-bit registers, but instructions dealing with characters usually use the 8-bit registers. It is important, however, to realize that AL and AH are just names for both halves of AX. When AX is loaded with a new value, both AL and AH are changed to the lower and upper halves of the 16-bit number put in AX, respectively. To see how AX, AH, and AL interact, consider the instruction

```
MOV AX,258
```

which loads the AX register with the decimal value 258. After this instruction, the byte register AH contains the value 1, and the byte register AL contains the number 2. If this instruction is followed by the byte add instruction

```
ADDB AH,AL
```

then the byte register AH is incremented by the value in AL (2) so that it now contains 3. The effect on the register AX of this action is that its value is now 770, which is equivalent to 00000011 00000010 in binary notation or 0x03 0x02 in hexadecimal notation. The eight byte-wide registers are almost interchangeable, with the exception that AL always contains one of the operands in the MULB instruction, and is the implied destination of this operation, together with AH. DIVB also uses the AH : AL pair for the dividend. The lower byte of the counter register CL can be used to hold the number of cycles in shift and rotate instruction.

Section C.8, example 2, shows some of the properties of the general registers by means of a discussion of the program *GenReg.s*.

C.2.3 Pointer Registers

The second group of registers consists of the **pointer and index registers**. The most important register of this group is the **stack pointer**, which is denoted by SP. Stacks are important in most programming languages. The stack is a segment of memory that holds certain context information about the running program. Usually, when a procedure is called, part of the stack is reserved for holding the procedure's local variables, the address to return to when the procedure has finished, and other control information. The portion of the stack relating to a procedure is called its **stack frame**. When a called procedure calls another procedure, an additional stack frame is allocated, usually just below the current one. Additional calls allocate additional stack frames below the current ones. While not mandatory, stacks almost always grow downward, from high addresses to low addresses. Nevertheless, the lowest numerical address occupied on the stack is always called the top of the stack.

In addition to their use for holding local variables, stacks can also hold temporary results. The 8088 has an instruction, PUSH, which puts a 16-bit word on top of the stack. This instruction first decrements SP by 2, then stores its operand at the address SP is now pointing to. Similarly, POP removes a 16-bit word from the top of the stack by fetching the value on top of the stack and then incrementing SP by 2. The SP register points to the top of the stack and is modified by PUSH, POP, and CALL instructions, being decremented by PUSH, incremented by POP, and decremented by CALL.

The next register in this group is BP, the **base pointer**. It usually contains an address in the stack. Whereas SP always points to the top of the stack, BP can point to any location within the stack. In practice, a common use for BP is to point to the beginning of the current procedure's stack frame, in order to make it easy to find the procedure's local variables. Thus, BP often points to the bottom of the current stack frame (the stack frame word with the highest numerical value) and SP points to the top (the stack frame word with the lowest numerical value). The current stack frame is thus delimited by BP and SP.

In this register group, there are two index registers: SI, the **source index,** and DI, the **destination index**. These registers are often used in combination with BP to address data in the stack, or with BX to compute the addresses of data memory locations. More extensive treatment of these registers will be deferred to the section on addressing modes.

One of the most important registers, which is a group by itself, is the **instruction pointer**, which is Intel's name for the program counter (PC). This register is not addressed directly by the instructions, but contains an address in the program code segment of the memory. The processor's instruction cycle starts by fetching the instruction pointed to by PC. This register is then incremented before the rest of the instruction is executed. In this way this program counter points to the first instruction beyond the current one.

The **flag register** or **condition code register** is actually a set of single-bit registers. Some of the bits are set by arithmetic instructions and relate to the result, as follows:

Z - result is zero

S - result is negative (sign bit)

V - result generated an overflow

C - result generated a carry

A - Auxilary carry (out of bit 3)

P - parity of the result

Other bits in this register control operation of certain aspects of the processor. The *I* bit enables interrupts. The *T* bit enables tracing mode, which is used for debugging. Finally, the *D* bit controls the direction of the string operations. Not all 16 bits of this flag register are used; the unused ones are hardwired to zero.

There are four registers in the **segment register group**. Recall that the stack, the data and the instruction codes all reside in main memory, but usually in different parts of it. The segment registers govern these different parts of the memory, which are called **segments**. These registers are called CS for the code segment register, DS for the data segment register, SS for the stack segment register, and ES for the extra segment register. Most of the time, their values are not changed. In practice, the data segment and stack segment use the same piece of memory, with the data being at the bottom of the segment and the stack being at the top. More about these registers will be explained in Sec. C.3.1.

C.3 MEMORY AND ADDRESSING

The 8088 has a somewhat ungainly memory organization due to its combination of a 1-MB memory and 16-bit registers. With a 1-MB memory, it takes 20 bits to represent a memory address. Consequently, it is impossible to store a pointer to memory in any of the 16-bit registers. To get around this problem, memory is organized as segments, each of them 64 KB, so an address within a segment can be represented in 16 bits. We will now go into the 8088 memory architecture in more detail.

C.3.1 Memory Organization and Segments

The memory of the 8088, which consists simply of an array of addressable 8-bit bytes, is used for the storage of instructions as well as for the storage of data and for the stack. In order to separate the parts of the memory which are used for

these different purposes, the 8088 uses **segments** which are chunks of the memory set apart for a certain uses. In the 8088, such a segment consists of 65,536 consecutive bytes. There are four segments:

1. The code segment.
2. The data segment.
3. The stack segment.
4. The extra segment.

The code segment contains the program instructions. The contents of the PC register are always interpreted as a memory address in the code segment. A PC value of 0 refers to the lowest address in the code segment, not absolute memory address zero. The data segment contains the initialized and uninitialized data for the program. When BX contains a pointer, it points to this data segment. The stack segment contains local variables and intermediate results pushed on the stack. Addresses in SP and BP are always in this stack segment. The extra segment is a spare segment register that can be placed anywhere in memory that it is needed.

For each of the segments, there exists a corresponding segment register: the 16-bit registers CS, DS, SS, and ES. The starting address of a segment is the 20-bit unsigned integer which is constructed by shifting the segment register by 4 bits to the left, and putting zero's in the four right-most positions. This means that segment registers always indicate multiples of 16, in a 20-bit address space. The segment register points to the base of the segment. Addresses within the segment can be constructed by converting the 16-bit segment register value to its true 20-bit address by appending four zero bits to the end and adding the offset to that. In effect, an absolute memory address is computed by multiplying the segment register by 16 and then adding the offset to it. For example, if DS is equal to 7, and BX is 12, then the address indicated by BX is $7 \times 16 + 12 = 124$. In other words, the 20-bit binary address implied by DS = 7 is 00000000000001110000. Adding the 16-bit offset 0000000000001100 (decimal 12) to the segment's origin gives the 20-bit address 00000000000001111100 (decimal 124).

For *every* memory reference, one of the segment registers is used to construct the actual memory address. If some instruction contains a direct address without reference to a register, then this address is automatically in the data segment, and DS is used to determine the base of the segment. The physical address is found by adding this bottom to the address in the instruction. The physical address in memory of the next instruction code is obtained by shifting the contents of CS by four binary places and adding the value of the program counter. In other words, the true 20-bit address implied by the 16-bit CS register is first computed, then the 16-bit PC is added to it to form a 20-bit absolute memory address.

The stack segment is made up of 2-byte words and so the stack pointer, SP, should always contain an even number. The stack is filled up from high addresses to low addresses. Thus, the PUSH instruction decreases the stack pointer by 2 and then stores the operand in the memory address computed from SS and SP. The POP command retrieves the value, and increments SP by 2. Addresses in the stack segment which are lower than those indicated by SP are considered free. Stack cleanup is thus achieved by merely increasing SP. In practice, DS and SS are always the same, so a 16-bit pointer can be used to refer to a variable in the shared data/stack segment. If DS and SS were different, a 17th bit would be needed on each pointer to distinguish pointers into the data segment from pointers into the stack segment. In retrospect, having a separate stack segment at all was probably a mistake.

If addresses in the four segment registers are chosen to be far apart, then the four segments will be disjointed, but if the available memory is restricted, it is not necessary to make them disjoint. After compilation, the size of the program code is known. It is then efficient to start the data and stack segments at the first multiple of 16 after the last instruction. This assumes that the code and data segment will never use the same physical addresses.

C.3.2 Addressing

Almost every instruction needs data, either from memory or from the registers. To name this data, the 8088 has a reasonably versatile collection of addressing modes. Many instructions contain two operands, usually called **destination** and **source**. Think, for instance, about the copy instruction, or the add instruction:

```
MOV AX,BX
```

or

```
ADD CX,20
```

In these instructions, the first operand is destination and the second is the source. (The choice of which goes first is arbitrary; the reverse choice could also have been made.) It goes without saying that, in such a case, the destination must be a **left value** that is, it must be a place where something can be stored. This means that constants can be sources, but not destinations.

In its original design, the 8088 required that at least one operand in a two-operand instruction be a register. This was done so that the difference between **word instructions,** and **byte instructions** could be seen by checking whether the addressed register was a **word register** or a **byte register**. In the first release of the processor, this idea was so strictly enforced that it was impossible to push a constant, because neither the source nor the destination was a register in that instruction. Later versions were not as strict, but the idea influenced the design

anyway. In some cases, one of the operands is not mentioned. For example, in the MULB instruction, only the AX register is powerful enough to act as a destination.

There are also a number of one-operand instructions, such as increments, shifts, negates, etc. In these cases, there is no register requirement, and the difference between the word and byte operations has to be inferred from the opcodes (i.e., instruction types) only.

The 8088 supports four basic data types: 1-byte **byte,** the 2-byte **word,** the 4-byte **long**, and **binary coded decimal**, in which two decimal digits are packed into a word. The latter type is not supported by the interpreter.

A memory address always refers to a byte, but in case of a word or a long, the memory locations directly above the indicated byte are implicitly referred to as well. The word at 20 is in the memory locations 20 and 21. The long at address 24 occupies the addresses 24, 25, 26 and 27. The 8088 is **little endian,** meaning that the low-order part of the word is stored at the lower address. In the stack segment, words should be placed at even addresses. The combination AX DX, in which AX holds the low-order word, is the only provision made for longs in the processor registers.

The table of Fig. C-3 gives an overview of the 8088 addressing modes. Let us now briefly discuss them. The topmost horizontal block of the table lists the registers. They can be used as operands in nearly all instructions, both as sources and as destinations. There are eight word registers and eight byte registers.

The second horizontal block, data segment addressing, contains addressing modes for the data segment. Addresses of this type always contain a pair of parentheses, to indicate that the contents of the address instead of the value is meant. The easiest addressing mode of this type is **direct addressing**, in which the data address of the operand is in the instruction itself. Example

```
ADD CX,(20)
```

in which the contents of the memory word at address 20 and 21 is added to CX. Memory locations are usually represented by labels instead of by numerical values in the assembly language, and the conversion is made at assembly time. Even in CALL and JMP instructions, the destination can be stored in a memory location addressed by a label. The parentheses around the labels are essential (for the assembler we are using) because

```
ADD CX,20
```

is also a valid instruction, only it means add the constant 20 to CX, not the contents of memory word 20. In Fig. C-3, the # symbol is used to indicate a numerical constant, label, or constant expression involving a label.

In **register indirect addressing,** the address of the operand is stored in one of the registers BX, SI, or DI. In all three cases the operand is found in the data

Mode	Operand	Examples
Register addressing		
Byte register	Byte register	AH,AL,BH,BL,CH,CL,DH,DL
Word register	Word register	AX,BX,CX,DX,SP,BP,SI,DI
Data segment addressing		
Direct address	Address follows opcode	(#)
Register indirect	Address in register	(SI), (DI), (BX)
Register displacement	Address is register+displ.	#(SI), #(DI), #(BX)
Register with index	Address is BX + SI/DI	(BX)(SI), (BX)(DI)
Register index displacement	BX + SI DI + displacement	#(BX)(SI), #(BX)(DI)
Stack segment address		
Base Pointer indirect	Address in register	(BP)
Base pointer displacement	Address is BP + displ.	#(BP)
Base Pointer with index	Address is BP + SI/DI	(BP)(SI), (BP)(DI)
Base pointer index displ.	BP+SI/DI + displacement	#(BP)(SI), #(BP)(DI)
Immediate data		
Immediate byte/word	Data part of instruction	#
Implied address		
Push/pop instruction	Address indirect (SP)	PUSH, POP, PUSHF, POPF
Load/store flags	status flag register	LAHF, STC, CLC, CMC
Translate XLAT	AL, BX	XLAT
Repeated string instructions	(SI), (DI), (CX)	MOVS, CMPS, SCAS
In / out instructions	AX, AL	IN #, OUT #
Convert byte, word	AL, AX, DX	CBW,CWD

Figure C-3. Operand addressing modes. The symbol # indicates a numerical value or label.

segment. It is also possible to put a constant in front of the register, in which case the address is found by adding the register to the constant. This type of addressing, called **register displacement**, is convenient for arrays. If, for example, SI contains 5, then the fifth character of the string at the label *FORMAT* can be loaded in AL by

```
MOVB AL,FORMAT(SI).
```

The entire string can be scanned by incrementing or decrementing the register in each step. When word operands are used, the register should be changed by two each time.

It is also possible to put the base (i.e., lowest numerical address) of the array in the BX register, and keep the SI or DI register for counting. This is called **register with index** addressing. For example:

```
PUSH (BX)(DI)
```

fetches the contents of the data segment location whose address is given by the

sum of the BX and DI registers. This value is then pushed onto the stack. The last two types of addresses can be combined to get **register with index and displacement** addressing, as in

```
NOT 20(BX)(DI)
```

which complements the memory word at BX + DI + 20 and BX + DI + 21.

All the indirect addressing modes in the data segment also exist for the stack segment, in which case the base pointer BP is used instead of the base register BX. In this way (BP) is the only register indirect stack addressing mode, but more involved modes also exist, up to base pointer indirect with index and displacement –1(BP)(SI). These modes are valuable for addressing local variables and function parameters, which are stored in stack addresses in subroutines. This arrangement is described further in Sec. C.4.5.

All the addresses which comply with the addressing modes discussed up to now can be used as sources and as destinations for operations. Together they are defined to be **effective addresses**. The addressing mode in the remaining two blocks cannot be used as destinations and are not referred to as effective addresses. They can only be used as sources.

The addressing mode in which the operand is a constant byte or word value in the instruction itself is called **immediate addressing**. Thus, for example,

```
CMP AX,50
```

compares AX to the constant 50 and sets bits in the flag register, depending on the results.

Finally, some of the instructions use **implied addressing**. For these instructions, the operand or operands are implicit in the instruction itself. For example, the instruction

```
PUSH AX
```

pushes the contents of AX onto the stack by decrementing SP and then copying AX to the location now pointed to by SP. SP is not named in the instruction itself, however; the mere fact that it is a PUSH instruction implies that SP is used. Similarly, the flag manipulation instructions implicitly use the status flags register without naming it. Several other instructions also have implicit operands.

The 8088 has special instructions for moving (MOVS), comparing (CMPS), and scanning (SCAS) strings. With these string instructions, the index registers SI and DI are automatically changed after the operation. This behavior is called **auto increment** or **auto decrement** mode. Whether SI and DI are incremented or decremented depends on the **direction flag** in the status flags register. A direction flag value of 0 increments, whereas a value of 1 decrements. The change is 1 for byte instructions and 2 for word instructions. In a way, the stack pointer is also auto increment and auto decrement: it is decremented by 2 at the start of a PUSH and incremented by 2 at the end of a POP.

C.4 THE 8088 INSTRUCTION SET

The heart of every computer is the set of instructions it can carry out. To really understand a computer, it is necessary to have a good understanding of its instruction set. In the following sections, we will discuss the most important of the 8088's instructions. Some of them are shown in Fig. C-4, where they are divided into 10 groups.

C.4.1 Move, Copy and Arithmetic

The first group of instructions is the copy and move instructions. By far, the most common is the instruction MOV, which has an explicit source and an explicit destination. If the source is a register, the destination can be an effective address. In this table a register operand is indicated by an *r* and an effective address by an *e*, so this operand combination is denoted by *e*←*r*. This is the first entry in the *Operands* column for MOV. Since, in the instruction syntax, the destination is the first operand and the source is the second operand, the arrow ← is used to indicate the operands. Thus, *e*←*r* means that a register is copied to an effective address.

For the MOV instruction, the source can also be an effective address and the destination a register, which will be denoted by *r*←*e*, the second entry in the *Operands* column of the instruction. The third possibility is immediate data as source, and effective address as destination, which yields *e*←#. Immediate data in the table is indicated by the sharp sign (#). Since both the word move MOV and the byte move MOVB exist, the instruction mnemonic ends with a *B* between parentheses. Thus, the line really represents six different instructions.

None of the flags in the condition code register are affected by a move instruction, so the last four columns have the entry "-". Note that the move instructions do not move data. They make copies, meaning that the source is not modified as would happen with a true move.

The second instruction in the table is XCHG, which exchanges the contents of a register with the contents of an effective address. For the exchange the table uses the symbol ↔. In this case, there exists a byte version as well as a word version. Thus, the instruction is denoted by XCHG and the *Operand* field contains *r*↔*e*. The next instruction is LEA, which stands for Load Effective Address. It computes the numerical value of the effective address and stores it in a register.

Next is PUSH, which pushes its operand onto the stack. The explicit operand can either be a constant (# in the *Operands* column) or an effective address (*e* in the *Operands* column). There is also an implicit operand, SP, which is not mentioned in the instruction syntax. What the instruction does is decrement SP by 2, then store the operand at the location now pointed to by SP.

Then comes POP, which removes an operand from the stack to an effective address. The next two instructions, PUSHF and POPF, also have implied operands, and push and pop the flags register, respectively. This is also the case for XLAT

which loads the byte register AL from the address computed from AL + BX . This instruction allows for rapid lookup in tables of size 256 bytes.

Officially defined in the 8088, but not implemented in the interpreter (and thus not listed in Fig. C-4), are the IN and OUT instructions. These are, in fact, move instructions to and from an I/O device. The implied address is always the AX register, and the second operand in the instruction is the port number of the desired device register.

In the second block of Fig. C-4 are the addition and subtraction instructions. Each of these has the same three operand combinations as MOV: effective address to register, register to effective address, and constant to effective address. Thus, the *Operands* column of the table contains *r*←*e*, *e*←*r*, and *e*←#. In all four of these instructions, the overflow flag, *O*, the sign flag, *S*, the zero flag, *Z*, and the carry flag, *C* are all set, based on the result of the instruction. This means, for example, that *O* is set if the result cannot be correctly expressed in the allowed number of bits, and cleared if it can be. When the largest 16-bit number, 0x7fff (32,767 in decimal), is added to itself, the result cannot be expressed as a 16-bit signed number, so *O* is set to indicate the error. Similar things happen to the other status flags in these operations. If an instruction has an effect on a status flag, an asterisk (*) is shown in the corresponding column. In the instructions ADC and SBB, the carry flag at the start of the operation is used as an extra 1 (or 0), which is seen as a carry or borrow from the previous operation. This facility is especially useful for representing 32-bit or longer integers in several words. For all additions and subtractions, byte versions also exist.

The next block contains the multiplication and division instructions. Signed integer operands require the IMUL and IDIV instructions; unsigned ones use MUL and DIV. The AH : AL register combination is the implied destination in the byte version of these instructions. In the word version, the implied destination is the *AX*: DX register combination. Even if the result of the multiplication is only a word or a byte, the DX or AH register is rewritten during the operation. The multiplication is always possible because the destination contains enough bits. The overflow and carry bits are set when the product cannot be represented in one word, or one byte. The zero and the negative flags are undefined after a multiply.

Division also uses the register combinations DX : AX or AH : AL as the destination. The quotient goes into AX or AL and the remainder into DX or AH. All four flags, carry, overflow, zero and negative, are undefined after a divide operation. If the divisor is 0, or if the quotient does not fit into the register, the operation executes a **trap**, which stops the program unless a trap handler routine is present. Moreover, it is sensible to handle minus signs in software before and after the divide, because in the 8088 definition the sign of the remainder equals the sign of the dividend, whereas in mathematics, a remainder is always nonnegative.

The instructions for binary coded decimals, among which Ascii Adjust for Addition (AAA), and Decimal Adjust for Addition (DAA), are not implemented by the interpreter and not shown in Fig. C-4.

Mnemonic	Description	Operands	Status flags			
			O	S	Z	C
MOV(B)	Move word, byte	r ← e, e ← r, e ← #	-	-	-	-
XCHG(B)	Exchange word	r ↔ e	-	-	-	-
LEA	Load effective address	r ← #e	-	-	-	-
PUSH	Push onto stack	e, #	-	-	-	-
POP	Pop from stack	e	-	-	-	-
PUSHF	Push flags	-	-	-	-	-
POPF	Pop flags	-	-	-	-	-
XLAT	Translate AL	-	-	-	-	-
ADD(B)	Add word	r ← e, e ← r, e ← #	*	*	*	*
ADC(B)	Add word with carry	r ← e, e ← r, e ← #	*	*	*	*
SUB(B)	Subtract word	r ← e, e ← r, e ← #	*	*	*	*
SBB(B)	Subtract word with borrow	r ← e, e ← r, e ← #	*	*	*	*
IMUL(B)	Multiply signed	e	*	U	U	*
MUL(B)	Multiply unsigned	e	*	U	U	*
IDIV(B)	Divide signed	e	U	U	U	U
DIV(B)	Divide unsigned	e	U	U	U	U
CBW	Sign extend byte-word	-	-	-	-	-
CWD	Sign extend word-double	-	-	-	-	
NEG(B)	Negate binary	e	*	*	*	*
NOT(B)	Logical complement	e	-	-	-	-
INC(B)	Increment destination	e	*	*	*	-
DEC(B)	Decrement destination	e	*	*	*	-
AND(B)	Logical and	e ← r, r ← e, e ← #	0	*	*	0
OR(B)	Logical or	e ← r, r ← e, e ← #	0	*	*	0
XOR(B)	Logical exclusive or	e ← r, r ← e, e ← #	0	*	*	0
SHR(B)	Logical shift right	e ← 1, e ← CL	*	*	*	*
SAR(B)	Arithmetic shift right	e ← 1, e ← CL	*	*	*	*
SAL(B) (=SHL(B))	shift left	e ← 1, e ← CL	*	*	*	*
ROL(B)	Rotate left	e ← 1, e ← CL	*	-	-	*
ROR(B)	Rotate right	e ← 1, e ← CL	*	-	-	*
RCL(B)	Rotate left with carry	e ← 1, e ← CL	*	-	-	*
RCR(B)	Rotate right with carry	e ← 1, e ← CL	*	-	-	*
TEST(B)	Test operands	e ↔ r, e ↔ #	0	*	*	0
CMP(B)	Compare operands	e ↔ r, e ↔ #	*	*	*	*
STD	Set direction flag (↓)	-	-	-	-	-
CLD	Clear direction flag (↑)	-	-	-	-	-
STC	Set carry flag	-	-	-	-	1
CLC	Clear carry flag	-	-	-	-	0
CMC	Complement carry	-	-	-	-	*
LOOP	Jump back if decremented CX ≥ 0	label	-	-	-	-
LOOPZ LOOPE	Back if Z=1 and DEC(CX)≥0	label	-	-	-	-
LOOPNZ LOOPNE	Back if Z=0 and DEC(CX)≥0	label	-	-	-	-
REP REPZ REPNZ	Repeat string instruction	string instruction	-	-	-	-
MOVS(B)	Move word string	-	-	-	-	-
LODS(B)	Load word string	-	-	-	-	-
STOS(B)	Store word string	-	-	-	-	-
SCAS(B)	Scan word string	-	*	*	*	*
CMPS(B)	Compare word string	-	*	*	*	*
JCC	Jump according conditions	label	-	-	-	-
JMP	Jump to label	e, label	-	-	-	-
CALL	Jump to subroutine	e, label	-	-	-	-
RET	Return from subroutine	-, #	-	-	-	-
SYS	System call trap	-	-	-	-	-

Figure C-4. Some of the most important 8088 instructions.

C.4.2 Logical, Bit and Shift Operations

The next block contains instructions for sign extension, negation, logical complement, increment and decrement. The sign extend operations have no explicit operands, but act on the DX : AX or the AH : AL register combinations. The single operand for the other operations of this group can be found at any effective address. The flags are affected in the expected way in case of the NEG, INC and DEC, except that the carry is not affected in the increment and decrement, which is quite unexpected and which some people regard as a design error.

The next block of instructions is the two-operand logical group, all of whose instructions behave as expected. In the shift and rotate group, all operations have an effective address as their destination, but the source is either the byte register CL or the number 1. In the shifts, all four flags are affected; in the rotates, only the carry and the overflow are affected. The carry always gets the bit that is shifted or rotated out of the high-order or low-order bit, depending on the direction of the shift or rotate. In the rotates with carry, RCR, RCL, RCRB, and RCLB, the carry together with the operand at the effective address, constitutes a 17-bit or a 9-bit circular shift register combination, which facilitates multiple word shifts and rotates.

The next block of instructions is used to manipulate the flag bits. The main reason for doing this is to prepare for conditional jumps. The double arrow ($\leftrightarrow$) is used to indicate the two operands in compare and test operations, which do not change during the operation. In the TEST operation, the logical AND of the operands is computed to set or clear the zero flag and the sign flag. The computed value itself is not stored anywhere and the operand is unmodified. In the CMP, the difference of the operands is computed and all four flags are set or cleared as a result of the comparison. The direction flag, which determines whether the SI and DI registers should be incremented or decremented in the string instructions, can be set or cleared by STD and CLD, respectively.

The 8088 also has a **parity flag** and an **auxiliary carry flag**. The parity flag gives the parity of the result (odd or even). The auxiliary flag checks whether overflow was generated in the low (4-bit) nibble of the destination. There are also instructions LAHF and SAHF, which copy the low-order byte of the flag register in AH, and vice versa. The overflow flag is in the high-order byte of the condition code register and is not copied in these instructions. These instructions and flags are mainly used for backward compatibility with the 8080 and 8085 processors.

C.4.3 Loop and Repetitive String Operations

The following block contains the instructions for looping. The LOOP instruction decrements the CX register and jumps back to the label indicated if the result is positive. The instructions LOOPZ, LOOPE, LOOPNZ and LOOPNE also test the zero flag to see whether the loop should be aborted before CX is 0.

The destination for all LOOP instructions must be within 128 bytes of the current position of the program counter because the instruction contains an 8-bit signed offset. The number of *instructions* (as opposed to bytes) that can jumped over cannot be calculated exactly because different instructions have different lengths. Usually, the first byte defines the type of an instruction, and so some instructions take only one byte in the code segment. Often, the second byte is used to define the registers and register modes of the instruction, and if the instructions contain displacements or immediate data, the instruction length can increase to four or six bytes. The average instruction length is typically about 2.5 bytes per instruction, so the LOOP cannot jump further back than approximatel 50 instructions.

There also exist some special string instruction looping mechanisms. These are REP, REPZ, and REPNZ. Similarly, the five string instructions in the next block of Fig. C-4 all have implied addresses and all use auto increment or auto decrement mode on the index registers. In all of these instructions, the SI register points into the **data segment**, but the DI register refers to the **extra segment**, which is based on ES. Together with the REP instruction, the MOVSB can be used to move complete strings in one instruction. The length of the string is contained in the CX register. Since the MOVSB instruction does not affect the flags, it is not possible to check for an ASCII zero byte during the copy operation by means of the REPNZ, but this can be fixed by using first a REPNZ SCASB to get a sensible value in CX and later a REP MOVSB. This point will be illustrated by the string copy example in Sec. C.8. For all of these instructions, extra attention should be paid to the segment register ES, unless ES and DS have the same value. In the interpreter a small memory model is used, so that ES = DS = SS.

C.4.4 Jump and Call Instructions

The last block is about conditional and unconditional jumps, subroutine calls, and returns. The simplest operation here is the JMP instruction. It can have a label as destination or the contents of any effective address. A distinction is made between a **near jump** and a **far jump**. In a near jump, the destination is in the current code segment, which does not change during the operation. In a far jump, the CS register is changed during the jump. In the direct version with a label, the new value of the code segment register is supplied in the call after the label, in the effective address version, a long is fetched from memory, such that the low word corresponds to the destination label, and the high word to the new code segment register value.

It is, of course, not surprising, that such a distinction exists. To jump to an arbitrary address within a 20-bit address space, some provision has to be made for specifying more than 16 bits. The way it is done is by giving new values for CS and PC.

Conditional jumps

The 8088 has 15 conditional jumps, a few of which have two names (e.g., JUMP GREATER OR EQUAL is the same instruction as JUMP NOT LESS THAN). They are listed in Fig. C-5. All of these allow only jumps with a distance of up to 128 bytes from the instruction. If the destination is not within this range, a jump over jump construction has to be used. In such a construction, the jump with the opposite condition is used to jump over the next instruction. If the next instruction contains an unconditional jump to the intended destination, then the effect of these two instructions is just a longer-ranging jump of the intended type. for example

```
        JB FARLABEL
```

becomes

```
        JNA 1f
        JMP FARLABEL
1:
```

In other words, if it is not possible to do JUMP BELOW, then a JUMP NOT ABOVE to a nearby label *1* is placed, followed by an unconditional jump to *FARLABEL*. The effect is the same, at a slightly higher cost in time and space. The assembler generates these jump over jumps automatically when the destination is expected to be too distant. Doing the calculation correctly is a bit tricky. Suppose that the distance is close to the edge, but some of the intervening instructions are also conditional jumps. The outer one cannot be resolved until the sizes of the inner ones are known, and so on. To be safe, the assembler errs on the side of caution. Sometimes it generates a jump over jump when it is not strictly necessary. It only generates a direct condition jump when it is certain that the target is within range.

Most conditional jumps depend on the status flags, and are preceded by a compare or test instruction. The CMP instruction subtracts the source from the destination operand, sets the condition codes and discards the result. Neither of the operands is changed. If the result is zero or has the sign bit on (i.e., is negative), the corresponding flag bit is set. If the result cannot be expressed in the allowed number of bits, the overflow flag is set. If there is a carry out of the high-order bit, the carry flag is set. The conditional jumps can test all of these bits.

If the operands are considered to be signed, the instructions using GREATER THAN and LESS THAN should be used. If they are unsigned, the ones using ABOVE and BELOW should be used.

C.4.5 Subroutine Calls

The 8088 has an instruction used to call procedures, usually known in assembly language as **subroutines**. In the same way as in the jump instructions, there exist **near call** instructions and **far call** instructions. In the interpreter, only the

Instruction	Description	When to jump
JNA, JBE	Below or equal	CF=1 or ZF=1
JNB, JAE, JNC	Not below	CF=0
JE, JZ	Zero, equal	ZF=1
JNLE, JG	Greater than	SF=OF and ZF=0
JGE, JNL	Greater equal	SF=OF
JO	Overflow	OF=1
JS	Sign negative	SF=1
JCXZ	CX is zero	CX=0
JB, JNAE, JC	Below	CF=1
JNBE, JA	Above	CF=0&ZF=0
JNE, JNZ	Nonzero, nonequal	ZF=0
JL, JNGE	Less than	SF≠OF
JLE, JNG	Less or equal	SF≠OF or ZF=1
JNO	Nonoverflow	OF=0
JNS	Nonnegative	SF=0

Figure C-5. Conditional jumps.

near call is implemented. The destination is either a label or can be found at an effective address. Parameters needed in the subroutines have to be pushed onto the stack in reverse order first, as illustrated in Fig. C-6. In assembly language, parameters are usually called **arguments**, but the terms are interchangeable. Following these pushes the CALL instruction is executed. The instruction starts by pushing the current program counter onto the stack. In this way the return address is saved. The return address is the address at which the execution of the calling routine has to be resumed when the subroutine returns.

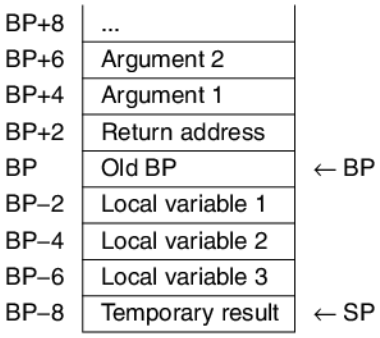


Figure C-6. An example stack.

Next the new program counter is loaded either from the label, or from the effective address. If the call is far, then the CS register is pushed before PC and

both the program counter and the code segment register are either loaded from immediate data or from the effective address. This finishes the CALL instruction.

The return instruction, RET, just pops the return address from the stack, stores it in the program counter and the program continues at the instruction immediately after the CALL instruction. Sometimes the RET instruction contains a positive number as immediate data. This number is assumed to be the number of bytes of arguments that were pushed onto the stack before the call; it is added to SP to clean up the stack. In the far variant, RETF, the code segment register is popped after the program counter, as would be expected.

Inside the subroutine, the arguments need to be accessible. Therefore the subroutine starts often by pushing the base pointer and copying the current value of SP into BP. This means that the base pointer points to its previous value. Now the return address is at BP + 2 and the first and second arguments can be found at the effective addresses BP + 4 and BP + 6, respectively. If the procedure needs local variables, then the required number of bytes can be subtracted from the stack pointer, and those variables can be addressed from the base pointer with negative offsets. In the example of Fig. C-6, there are three single-word local variables, located at BP – 2, BP – 4, and BP – 6, respectively. In this way, the entire set of current arguments and local variables is reachable through the BP register.

The stack is used in the ordinary way to save intermediate results, or for preparing arguments for the next call. Without computing the amount of stack used in the subroutine, the stack can be restored before the return by copying the base pointer into the stack pointer, popping the old BP and finally executing the RET instruction.

During a subroutine call, the values of the processor registers sometimes change. It is good practice to use some type of convention such that the calling routine need not be aware of the registers used by the called routine. The simplest way to do this is to use the same conventions for system calls and ordinary subroutines. It is assumed that the AX and DX can change in the called routine. If one of these registers contains valuable information then it is advisable for the calling routine to stack them before pushing the arguments. If the subroutine uses other registers as well, those can be pushed onto the stack immediately at the start of the subroutine, and popped before the RET instruction. In other words, a good convention is for the caller to save AX and DX if they contain anything important, and for the callee to save any other registers it overwrites.

C.4.6 System Calls and System Subroutines

In order to separate the tasks of opening, closing, reading, and writing files from assembly programming, programs are run on top of an operating system. To allow the interpreter to run on multiple platforms, a set of seven system calls and five functions are supported by the interpreter. They are listed in Fig. C-7.

Nr	Name	Arguments	Return value	Description
5	_OPEN	*name, 0/1/2	file descriptor	Open file
8	_CREAT	*name, *mode	file descriptor	Create file
3	_READ	fd, buf, nbytes	# bytes	Read nbytes in buffer buf
4	_WRITE	fd, buf, nbytes	# bytes	Write nbytes from buffer buf
6	_CLOSE	fd	0 on success	Close file with fd
19	_LSEEK	fd, offset(long), 0/1/2	position (long)	Move file pointer
1	_EXIT	status		Close files Stop process
117	_GETCHAR		read character	Read character from std input
122	_PUTCHAR	char	write byte	Write character to std output
127	_PRINTF	*format, arg		Print formatted on std output
121	_SPRINTF	buf, *format, arg		Print formatted in buffer buf
125	_SSCANF	buf, *format, arg		Read arguments from buffer buf

Figure C-7. Some UNIX system calls and subroutines in the interpreter.

These twelve routines can be activated by the standard calling sequence; first push the necessary arguments on the stack in reverse order, then push the call number, and finally execute the system trap instruction SYS without operands. The system routine finds all the necessary information on the stack, including the call number of the required system service. Return values are put either in the AX register, or in the DX : AX register combination (when the return value is a long).

It is guaranteed that all other registers will keep their values over the SYS instruction. Also, the arguments will still be on the stack after the call. Since they are not needed any more, the stack pointer should be adjusted after the call (by the caller), unless they are needed for a subsequent call.

For convenience, the names of the system calls can be defined as constants at the start of the assembler program, so that they can be called by name instead of by number. In the examples, several system calls will be discussed, so in this section only a minimum of necessary detail is supplied.

In these system calls, files are opened either by the OPEN or by the CREAT call. In both cases, the first argument is the address of the start of a string containing the file name. The second argument in the OPEN call is either 0 (if the file should be opened for reading), 1 (if it should be opened for writing), or 2 (for both). If the file should allow writes, and does not exist, it is created by the call. In the CREAT call an empty file is created, with permission set according to the second argument. Both the OPEN and the CREAT call return a small integer in the AX register, which is called the **file descriptor** and which can be used for reading, writing or closing the file. A negative return value means the call failed. At the start of the program, three files are already opened with file descriptors: 0 for standard input, 1 for standard output, and 2 for standard error output.

The READ and WRITE calls have three arguments: the file descriptor, a buffer to hold the data, and the number of bytes to transfer. Since the arguments are stacked

in reverse order, we first push the number of bytes, then the address of the start of the buffer, then the file descriptor and finally the call number (READ or WRITE). This order of stacking the arguments was chosen to be the same as the standard C language calling sequence in which

```
read(fd, buffer, bytes);
```

is implemented by pushing the parameters in the order *bytes*, *buffer*, and finally *fd*.

The CLOSE call requires just the file descriptor and returns 0 in AX if the file could be closed successfully. The EXIT call requires the exit status on the stack and does not return.

The LSEEK call changes the **read/write pointer** in an open file. The first argument is the file descriptor. Since the second argument is a long, first the high-order word, then the low word should be pushed onto the stack, even when the offset would fit into a word. The third argument indicates whether the new read/write pointer should be computed relative to the start of the file (case 0), relative to the current position (case 1), or relative to the end of the file (case 2). The return value is the new position of the pointer relative to the start of a file, and can be found as a long in the DX : AX register combination.

Now we come to the functions that are not system calls. The GETCHAR function reads one character from standard input, and puts it in AL. AH is set to zero. On failure, AX is set to −1. The call PUTCHAR writes a byte on standard output. The return value for a successful write is the byte written; on failure it is −1.

The call PRINTF outputs formatted information. The first argument to the call is the address of a format string, which tells how to format the output. The string "%d" indicates that the next argument is an integer on the stack, which is converted to decimal notation when printed. In the same way, "%x" converts to hexadecimal and "%o" converts to octal. Furthermore, "%s" indicates that the next argument is a null-terminated string, which is passed to the call through a memory address on the stack. The number of extra arguments on the stack should match the number of conversion indications in the format string.

For example, the call

```
printf("x = %d and y = %d\n", x, y);
```

prints the string with the numerical values of *x* and *y* substituted for the "%d" strings in the format string. Again, for compatibility with C, the order in which the arguments are pushed is "y", "x", and finally, the address of the format string. The reason for this convention is that *printf* has a variable number of parameters, and by pushing them in the reverse order the format string itself is always the last one and thus can be located. If the parameters were pushed from left to right, the format string would be deep in the stack and the *printf* procedure would not know where to find it.

In the call PRINTF, the first argument is the buffer, to receive the output string, instead of standard output. The other arguments are the same as in PRINTF. The

SSCANF call is the converse of the PRINTF in the sense that the first argument is a string, which can contain integers in decimal, octal, or hexadecimal notation, and the next argument is the format string, which contains the conversion indications. The other arguments are addresses of memory words to receive the converted information. These system subroutines are very versatile and an extensive treatment of the possibilities is far beyond the scope of this appendix. In Sec. C.8, several examples show how they can be used in different situations.

C.4.7 Final Remarks on the Instruction Set

In the official definition of the 8088, there exists a **segment override** prefix, which facilitates the possibility of using effective addresses from a different segment; that is, the first memory address following the override is computed using the indicated segment register. For example, the instruction

```
ESEG MOV DX,(BX)
```

first computes the address of BX using the extra segment, and then moves the contents to DX. However, the stack segment, in the case of addresses using SP, and the extra segment, in the case of string instructions with the DI register, cannot be overridden. The segment registers SS, DS and ES can be used in the MOV instruction, but it is impossible to move immediate data into a segment register, and those registers cannot be used in an XCHG operation. Programming with changing segment registers and overrides is quite tricky and should be avoided whenever possible. The interpreter uses fixed segment registers, so these problems do not arise here.

Floating-point instructions are available in most computers, sometimes directly in the processor, sometimes in a separate coprocessor, and sometimes only interpreted in the software through a special kind of floating point trap. Discussion of those features is outside the scope of this appendix.

C.5 THE ASSEMBLER

We have now finished our discussion of the 8088 architecture. The next topic is the software used to program the 8088 in assembly language, in particular the tools we provide for learning assembly language programming. We will first discuss the assembler, then the tracer, and then move on to some practical information for using them.

C.5.1 Introduction

Up until now, we have referred to instructions by their **mnemonics**, that is, by short easy-to-remember symbolic names like ADD and CMP. Registers were also called by symbolic names, such as AX and BP. A program written using symbolic

names for instructions and registers is called an **assembly language program**. To run such a program, it is first necessary to translate it into the binary numbers that the CPU actually understands. The program that converts an assembly language program into binary numbers is the **assembler**. The output of the assembler is called an **object file**. Many programs make calls to subroutines that have been previously assembled and stored in libraries. To run these programs, the newly-assembled object file and the library subroutines it uses (also object files) must be combined into a single **executable binary file** by another program called a **linker**. Only when the linker has built the executable binary file from one or more object files is the translation fully completed. The operating system can then read the executable binary file into memory and execute it.

The first task of the assembler is to build a **symbol table**, which is used to map the names of symbolic constants and labels directly to the binary numbers that they represent. Constants that are directly defined in the program can be put in the symbol table without any processing. This work is done on pass one.

Labels represent addresses whose values are not immediately obvious. To compute their values, the assembler scans the program line by line in what is called the **first pass**. During this pass, it keeps track of a **location counter** usually indicated by the symbol ".", pronounced **dot**. For every instruction and memory reservation that is found in this pass, the location counter is increased by the size of the memory necessary to contain the scanned item. Thus, if the first two instructions are of size 2 and 3 bytes, respectively, then a label on the third instruction will have numerical value 5. For example, if this code fragment is at the start of a program, the value of *L* will be 5.

```
MOV AX,6
MOV BX,500
L:
```

At the start of the **second pass**, the numerical value of every symbol is known. Since the numerical values of the instruction mnemonics are constants, **code generation** can now begin. One at a time, instructions are read again and their binary values are written into the object file. When the last instruction has been assembled, the object file is complete.

C.5.2 The ACK-Based Assembler, as88

This section describes the details of the assembler/linker *as88*, which is provided on the CD-ROM and website and which works with the tracer. This assembler is Amsterdam Compiler Kit (ACK) and is patterned after UNIX assemblers rather than MS-DOS or Windows assemblers. The comment symbol in this assembler is the exclamation mark (!). Anything following an exclamation mark until the end of the line is a comment and does not affect the object file produced. In the same way, empty lines are allowed, but ignored.

This assembler uses three different sections, in which the translated code and data will be stored. Those sections are related to the memory segments of the machine. The first is the **TEXT section**, for the processor instructions. Next is the **DATA section** for the initialization of the memory in the data segment, which is known at the start of the process. The last is the **BSS** (**Block Started by Symbol**), section, for the reservation of memory in the data segment that is not initialized (i.e., initialized to 0). Each of these sections has its own location counter. The purpose of having sections is to allow the assembler to generate some instructions, then some data, then some instructions, then more data, and so on, and then have the linker rearrange the pieces so that all the instructions are together in the text segment and all the data words are together in the data segment. Each line of assembly code produces output for only one section, but code lines and data lines can be interleaved. At run time, the TEXT section is stored in the text segment and the data and BSS sections are stored (consecutively) in the data segment.

An instruction or data word in the assembly language program can begin with a label. A label may also appear all by itself on a line, in which case it is as though it appeared on the next instruction or data word. For example, in

```
    CMP AX,ABC
    JE L
    MOV AX,XYZ
L:
```

L is a label that refers to the instruction of data word following it. Two kinds of labels are allowed. First are the **global labels**, which are alphanumeric identifiers followed by a colon (:). These must all be unique, and cannot match any keyword or instruction mnemonic. Second, in the TEXT section only, we can have **local labels**, each of which consists of a single digit followed by a colon (:). A local label may occur multiple times. When a program contains an instruction such as

```
JE 2f
```

this means JUMP EQUAL forward to the next local label *2*. Similarly,

```
JNE 4b
```

means JUMP NOT EQUAL backward to the closest label *4*.

The assembler allows constants to be given a symbolic name using the syntax

```
identifier = expression
```

in which the identifier is an alphanumeric string, as in

```
BLOCKSIZE = 1024
```

Like all identifiers in this assembly language, only the first eight characters are significant, so *BLOCKSIZE* and *BLOCKSIZZ* are the same symbol, namely, *BLOCKSIZ*. Expressions can be constructed from constants, numerical values, and

operators. Labels are considered to be constants because at the end of the first pass their numerical values are known.

Numerical values can be **octal** (starting with a 0), **decimal**, or **hexadecimal** (starting with 0X or 0x). Hexadecimal numbers use the letters a–f or A–F for the values 10–15. The integer operators are +, –, *, /, and %, for addition, subtraction, multiplication, division and remainder, respectively. The logical operators are &, ^, and ~, for bitwise AND, bitwise OR and logical complement (NOT) respectively. Expressions can use the square brackets, [and] for grouping. Parentheses are *NOT* used, to avoid confusion with the addressing modes.

Labels in expressions should be handled in a sensible way. Instruction labels cannot be subtracted from data labels. The difference between comparable labels is a numerical value, but neither labels nor their differences are allowed as constants in multiplicative or logical expressions. Expressions which are allowed in constant definitions can also be used as constants in processor instructions. Some assemblers have macro facility, by which multiple instructions can be grouped together and given a name, but *as88* does not have this feature.

In every assembly language, there are some directives that influence the assembly process itself but which are not translated into binary code. They are called **pseudoinstructions**. The *as88* pseudoinstructions are listed in Fig. C-8.

Instruction	Description
.SECT .TEXT	Assemble the following lines in the TEXT section
.SECT .DATA	Assemble the following lines in the DATA section
.SECT .BSS	Assemble the following lines in the BSS section
.BYTE	Assemble the arguments as a sequence of bytes
.WORD	Assemble the arguments as a sequence of words
.LONG	Assemble the arguments as a sequence of longs
.ASCII "str"	Store str as an ASCII string without a trailing zero byte
.ASCIZ "str"	Store str as an ASCII string with a trailing zero byte
.SPACE n	Advance the location counter n positions
.ALIGN n	Advance the location counter up to an n-byte boundary
.EXTERN	Identifier is an external name

Figure C-8. The *as88* pseudoinstructions.

The first block of pseudoinstructions determines the section in which the following lines should be processed by the assembler. Usually such a section requirement is made on a separate line and can be put anywhere in the code. For implementation reasons, the first section to be used must be the TEXT section, then the DATA section, then the BSS section. After these initial references, the sections can be used in any order. Furthermore, the first line of a section should have a global label. There are no other restrictions on the ordering of the sections.

The second block of pseudoinstructions contains the data type indications for the data segment. There are four types: .BYTE, .WORD, .LONG, and string. After an optional label and the pseudoinstruction keyword, the first three types expect a comma-separated list of constant expressions on the remainder of the line. For strings there are two keywords, ASCII, and ASCIZ, with the only difference being that the second keyword adds a zero byte to the end of the string. Both require a string between double quotes. Several escapes are allowed in string definitions. These include those of Fig. C-9. In addition to these, any specific character can be inserted by a backslash and an octal representation, for example, \377 (at most three digits, no 0 required here).

Escape symbol	Description
\n	New line (line feed)
\t	Tab
\\	Backslash
\b	Back space
\f	Form feed
\r	Carriage return
\"	Double quote

Figure C-9. Some of the escapes allowed by *as88*.

The SPACE pseudoinstruction simply requires the location pointer to be incremented by the number of bytes given in the arguments. This keyword is especially useful following a label in the BSS segment to reserve memory for a variable. ALIGN keyword is used to advance the location pointer to the first 2-, 4-, or 8-byte boundary in memory to facilitate the assembly of words, longs, etc. at a suitable memory location. Finally, the keyword EXTERN announces that the routine or memory location mentioned will be made available to the linker for external references. The definition need not be in the current file; it can also be somewhere else, as long as the linker can handle the reference.

Although the assembler itself is fairly general, when it is used with the tracer some small points are worth noting. The assembler accepts keywords in either uppercase or lowercase but the tracer always displays them in uppercase. Similarly, the assembler accepts both "\r" (carriage return) and "\n" (line feed) as the new line indication, but the tracer always uses the latter. Moreover, although the assembler can handle programs split over multiple files, for use with the tracer, the entire program must be in a single file with extension ".$". Inside it, include files can be requested by the command

```
#include filename
```

In this case, the required file is also written in the combined ".$" file at the position of the request. The assembler checks whether the include file was already

processed and loads only one copy. This is especially useful if several files use the same header file. In this case, only one copy is included in the combined source file. In order to include the file, the *#include* must be the first token of the line without leading white space, and the file path must be between double quotes.

If there is a single source file, say *pr.s*, then it is assumed that the project name is *pr*, and the combined file will be *pr.$*. If there is more than one source file, then the basename of the first file is taken to be the projectname, and used for the definition of the *.$* file, which is generated by the assembler by concatenating the source files. This behavior can be overridden if the command line contains a "–o projname" flag before the first source file, in which case the combined file will be *projname*. $.

Note that there are some drawbacks in using include files and more than one source. It is necessary that the names of labels, variables and constants are different for all sources. Moreover, the file which is eventually assembled to the load file is the *projname*. $ file, so the line numbers mentioned by the assembler in case of errors and warnings are determined with respect to this file. For very small projects, it is sometimes simplest to put the entire program in one file and avoid *#include*.

C.5.3 Some Differences with Other 8088 Assemblers

The assembler, *as88*, is patterned after the standard UNIX assembler, and, as such, differs in some ways from the Microsoft Macro Assembler MASM and the Borland 8088 assembler TASM. Those two assemblers were designed for the MS-DOS operating system, and in places the assembler issues and the operating system issues are closely interrelated. Both MASM and TASM support all 8088 memory models allowed by MS-DOS. There is, for example, the **tiny** memory model, in which all code and data must fit in 64 KB, the **small** model, in which the code segment and the data segment each can be 64 KB, and **large** models, which contain multiple code and data segments. The difference between those models depends on the use of the segment registers. The large model allows far calls and changes in the DS register. The processor itself puts some restrictions on the segment registers (e.g., the CS register is not allowed as destination in a MOV instruction). To make tracing simpler, the memory model used in *as88* resembles the small model, although the assembler without the tracer can handle the segment registers without additional restrictions.

These other assemblers do not have a .BSS section, and initialize memory only in the DATA sections. Usually the assembler file starts with some header information, then the DATA section, which is indicated by the keyword .data, followed by the program text after the keyword .code. The header has a keyword title to name the program, a keyword .model to indicate the memory model, and a keyword .stack to reserve memory for the stack segment. If the intended binary is a *.com*

file, then the tiny model is used, all segment registers are equal, and at the head of this combined segment 256 bytes are reserved for a "Program Segment Prefix."

Instead of the .WORD, .BYTE, and ASCIZ directives, these assemblers have keywords DW for define word and DB for define byte. After the DB directive, a string can be defined inside a pair of double quotes. Labels for data definitions are not followed by a colon. Large chunks of memory are initialized by the DUP keyword, which is preceded by a count and followed by an initialization. For example, the statement

```
LABEL  DB 1000 DUP (0)
```

initializes 1000 bytes of memory with ASCII zero bytes at the label *LABEL*.

Furthermore, labels for subroutines are not followed by a colon, but by the keyword PROC. At the end of the subroutine, the label is repeated and followed by the keyword ENDP, so the assembler can infer the exact scope of a subroutine. Local labels are not supported.

The keywords for the instructions are identical in MASM, TASM, and *as88*. Also, the source is put after the destination in two operand instructions. However, it is common practice to use registers for the passing of arguments to functions, instead of on the stack. If, however, assembly routines are used inside C or C++ programs, then it is advisable to use the stack in order to comply with the C subroutine calling mechanism. This is not a real difference, since it is also possible to use registers instead of the stack for arguments in *as88*.

The biggest difference between the MASM, TASM and *as88* is in making system calls. The system is called in MASM and TASM by means of a system interrupt INT. The most common one is INT 21H, which is intended for the MS-DOS function calls. The call number is put in AX, so again we have passing of arguments in registers. For different devices there are different interrupt vectors, and interrupt numbers, such as INT 16H for the BIOS keyboard functions and INT 10H for the display. In order to program these functions, the programmer has to be aware of a great deal of device-dependent information. In contrast, the UNIX system calls available in *as88* are much easier to use.

C.6 THE TRACER

The tracer-debugger is meant to run on a 24 × 80 ordinary (VT100) terminal, with the ANSI standard commands for terminals. On UNIX or Linux machines, the terminal emulator in the X-window system usually meets the requirements. On Windows machines, the *ansi.sys* driver usually has to be loaded in the system initialization files as described below. In the tracer examples, we have already seen the layout of the tracer window. As can be seen in Fig. C-10, the tracer screen is subdivided into seven windows.

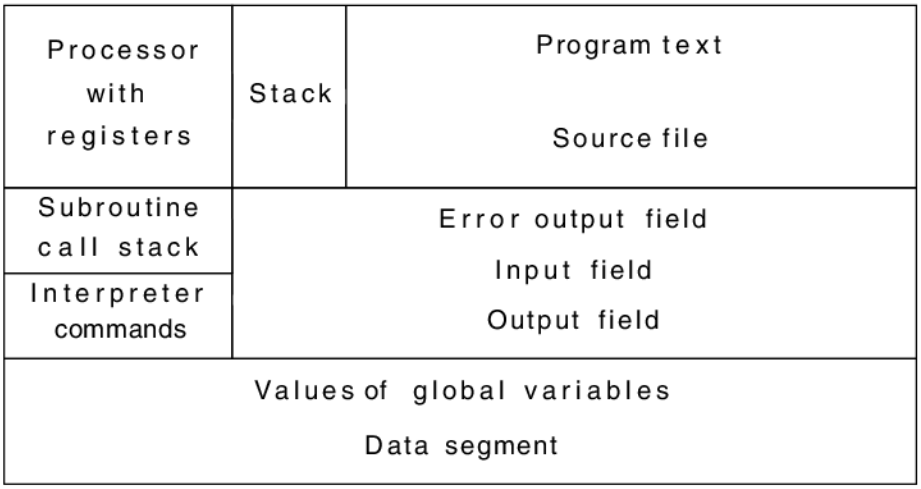


Figure C-10. The tracer's windows.

The upper left window is the processor window, which displays the general registers in decimal notation and the other registers in hexadecimal. Since the numerical value of the program counter is not very instructive, the position in the program source code with respect to the previous global label is supplied on the line below it. Above the program counter field, five condition codes are shown. Overflow is indicated by a "v", the direction flag by ">" for increasing and by "<" for decreasing. The sign flag is either "n", for negative or "p" for zero and positive. The zero flag is "z" if set, and the carry flag set is "c". A "–" indicates a cleared flag.

The upper middle window is used for the stack, displayed in hexadecimal. The stack pointer position is indicated with an arrow =>". Return addresses of subroutines are indicated by a digit in front of the hexadecimal value. The upper right window displays a part of the source file in the neighborhood of the next instruction to be executed. The position of the program counter is also indicated by an arrow "=>".

In the window under the processor, the most recent source code subroutine call positions are displayed. Directly under it is the tracer command window, which has the previously-issued command on top and the command cursor on the bottom. Note that every command needs to be followed by a carriage return (labeled Enter on PC keyboards).

The bottom window can contain six items of global data memory. Every item starts with a position relative to some label, followed by the absolute position in the data segment. Next comes a colon, then eight bytes in hexadecimal. The next 11 positions are reserved for characters, followed by four decimal word representations. The bytes, the characters, and the words each represent the same memory

contents, although for the character representation we have three extra bytes. This is convenient, because it is not clear from the start whether the data will be used as signed or unsigned integers, or as a string.

The middle right window is used for input and output. The first line is for error output of the tracer, the second line for input, and then there are some lines left for output. Error output is preceded by the letter "E", input by an "I", and standard output by a ">". In the input line there is an arrow "->" to indicate the pointer which is to be read next. If the program calls *read* or *getchar*, the next input in the tracer command line is going into the input field. Also, in this case, it is necessary to close the input line with a return. The part of the line which has not yet processed can be found after the "->" arrow.

Usually, the tracer reads both its commands and its input from standard input. However, it is also possible to prepare a file of tracer commands and a file of input lines to be read before the control is passed to the standard input. Tracer command files have extensions *.t* and input files *.i*. In the assembly language, both uppercase and lowercase characters can be used for keywords, system subroutines and pseudoinstructions. During the assembly process, a file with extension *.$* is made in which those lowercase keywords are translated into uppercase and carriage return characters are discarded. In this way, for each project, say, *pr* we can have up to six different files:

1. *pr.s* for the assembly source code.
2. *pr.$* for the composite source file.
3. *pr.88* for the load file.
4. *pr.i* for preset standard input.
5. *pr.t* for preset tracer commands.
6. *pr.#* for linking the assembly code to the load file.

The last file is used by the tracer to fill the upper right window and the program counter field in the display. Also, the tracer checks whether the load file has been created after the last modification of the program source; if not it issues a warning.

C.6.1 Tracer Commands

Figure C-11 lists the tracer commands. The most important ones are the single return command, which is at the first line of the table and which executes exactly one processor instruction, and the quit command *q*, at the bottom line of the table. If a number is given as a command, then that number of instructions is executed. The number *k* is equivalent to typing a return *k* times. The same effect is achieved if the number is followed by an exclamation mark, *!*, or an *X*.

The command *g* can be used to go to a certain line in the source file. There are three versions of this command. If it is preceded by a line number, then the tracer executes until that line is encountered. With a label */T*, with or without *+#*, the line number at which to stop is computed from the instruction label *T*. The *g* command, without any indication preceding it, causes the tracer to execute commands until the current line number is again encountered.

Address	Command	Example	Description
			Execute one instruction
#	, ! , X	24	Execute # instructions
/T+#	g , ! ,	/start+5g	Run until line # after label T
/T+#	b	/start+5b	Put breakpoint on line # after label T
/T+#	c	/start+5c	Remove breakpoint on line # after label T
#	g	108g	Execute program until line #
	g	g	Execute program until current line again
	b	b	Put breakpoint on current line
	c	c	Remove breakpoint on current line
	n	n	Execute program until next line
	r	r	Execute until breakpoint or end
	\&=	\&=	Run program until same subroutine level
	-	-	Run until subroutine level minus 1
	+	+	Run until subroutine level plus 1
/D+#		/buf+6	Display data segment on label+#
/D+#	d , !	/buf+6d	Display data segment on label+#
	R , CTRL L	R	Refresh windows
	q	q	Stop tracing, back to command shell

Figure C-11. The tracer commands. Each command must be followed by a carriage return (the Enter key). An empty box indicates that just a carriage return is needed. Commands with no *Address* field listed above have no address. The # symbol represents an integer offset.

The command */label* is different for an instruction label and a data label. For a data label, a line in the bottom window is filled or replaced with a set of data starting with that label. For an instruction label, it is equivalent to the *g* command. The label may be followed by a plus sign and a number (indicated by # in Fig. C-11), to obtain an offset from the label.

It is possible to set a **breakpoint** at an instruction. This is done with the command *b*, which can be optionally preceded by an instruction label, possibly with an offset. If a line with a breakpoint is encountered during execution, the tracer stops. To start again from a breakpoint, a return or run command is required. If the label and the number are omitted, then the breakpoint is set at the current line. The

breakpoint can be cleared by a breakpoint clear command, *c*, which can be preceded by labels and numbers, like the command *b*. There is a run command, *r*, in which the tracer executes until either a breakpoint, an exit call, or the end of the commands is encountered.

The tracer also keeps track of the subroutine level at which the program is running. This is shown in the window below the processor window and can also be seen through the indication numbers in the stack window. There are three commands that are based on these levels. The – command causes the tracer to run until the subroutine level is one less than the current level. What this command does is execute instructions until the current subroutine is finished. The converse is the + command, which runs the tracer until the next subroutine level is encountered. The = command runs until the same level is encountered, and can be used to execute a subroutine at the *CALL* command. If = is used, the details of the subroutine are not shown in the tracer window. There is a related command, *n*, which runs until the next line in the program is encountered. This command is especially useful when issued as a *LOOP* command; execution stops exactly when the bottom of the loop is executed.

C.7 GETTING STARTED

In this section, we will explain how to use the tools. First of all, it is necessary to locate the software for your platform. We have precompiled versions for Solaris, UNIX, for Linux and for Windows. The tools are located on the CD-ROM and on the Web at *www.prenhall.com/tanenbaum*. Once there, click on the *Companion Web Site* for this book and then click on the link in the left-hand menu. Unpack the selected zip file to a directory *assembler*. This directory and its subdirectories contain all the necessary material. On the CD-ROM, the main directories are *Bigendnx*, *LtlendNx*, and *MSWindos*, and in each there is a subdirectory *assembler* which contains the material. The three top-level directories are for Big-Endian UNIX (e.g. Sun workstations), Little-Endian UNIX (e.g., Linux on PCs), and Windows systems, respectively.

After unpacking or copying, the assembler directory should contain the following subdirectories and files: *READ_ME*, *bin*, *as_src*, *trce_src*, *examples*, and *exercise*. The precompiled sources can be found in the *bin* directory but, for convenience, there is also a copy of the binaries in the *examples* directory.

To get a quick preview of how the system works, go to the *examples* directory and type the command

```
t88 HlloWrld
```

This command corresponds to the first example in Sec. C.8.

The source code for the assembler is in the directory *as_src*. The source code files are in the language C, and the command *make* should recompile the sources.