The Impact of Memory and Architecture on Computer Performance

Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA

Tel: +1 801 581 5254
FAX: +1 801 581 4148

Internet: beebe@math.utah.edu

28 March 1994
Version 1.03

Contents

List of Figures

Abstract

These notes for Mathematics computing courses describe the impact of memory and architecture on numerical performance on modern computers.

The memory hierarchy of registers, cache, main memory, and virtual memory is first described, and an excursion is made into RISC/CISC instruction set issues.

Performance bottlenecks arising at each of these memory levels are discussed and illustrated with numerical programming examples.

The text is augmented with numerous drawings and tables relevant to recent computers, from desktop microcomputers to the most advanced supercomputers.

1 Introduction

> Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.
>
> A. W. Burks, H. H. Goldstine, and J. von Neumann
> 1946 [73, p. 402]

For several decades, high-performance computer systems have incorporated a memory hierarchy [73]. Because central processing unit (CPU) performance has risen much more rapidly than memory performance since the late 1970s, modern computer systems have an increasingly severe performance gap between CPU and memory. Figure 1 illustrates this trend, and Figure 2 shows that performance increases are happening at both the high and low ends of the market. Computer architects have attempted to compensate for this performance gap by designing increasingly complex memory hierarchies.

> Clock increases in speed do not exceed a factor of two every five years (about 14%).
>
> C. Gorden Bell
> 1992 [12, p. 35]

> ... a quadrupling of performance each three years still appears to be possible for the next few years. ... The quadrupling has its basis in Moore's law stating that semiconductor density would quadruple every three years.
>
> C. Gorden Bell
> 1992 [12, p. 40]

> ... (Others) suggest that the days of the traditional computer are numbered. ... Today it is improving in performance faster than at any time in its history, and the improvement in cost and performance since 1950 has been five orders of magnitude. Had the transportation industry kept pace with these advances, we could travel from San Francisco to New York in one minute for one dollar!
>
> John L. Hennessy and David A. Patterson
> 1990 [73, p. 571]

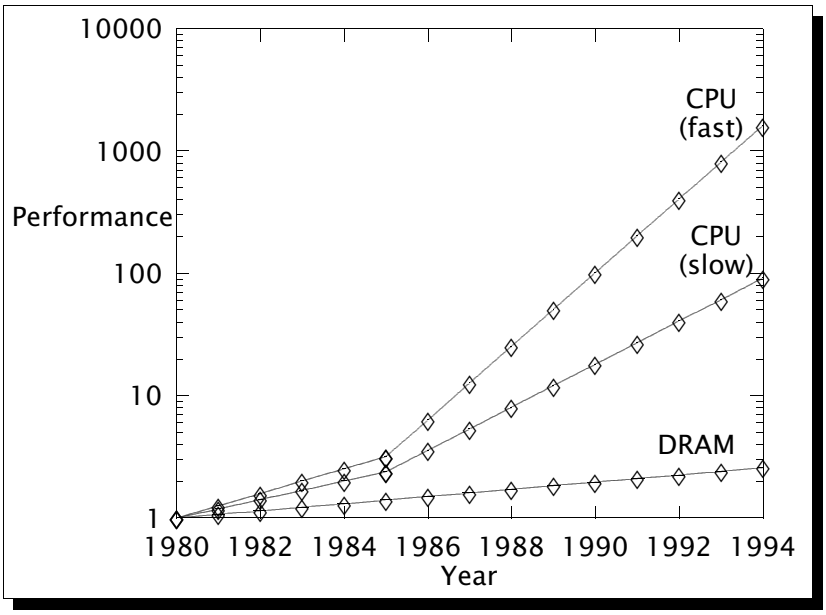

Figure 1: **CPU and memory performance.** This drawing uses 1980 as a baseline. Memory speed (dynamic random-access memory, DRAM) is plotted with an annual 7% increase. The slow CPU line grows at 19% annually until 1985, and at 50% annually since then. The fast CPU line rises at 26% annually until 1985, and at 100% annually since then. The data is taken from [73, Fig. 8.18, p. 427], but extended beyond 1992.

The existence of a memory hierarchy means that a few well-behaved programs will perform almost optimally on a particular system, but alas, most will not. Because the performance difference between the extremes of good and bad behavior can be several orders of magnitude, it is important for programmers to understand the impact of memory access patterns on performance.

Fortunately, once the issues are thoroughly understood, it is usually possible to control memory access in high-level languages, so it is seldom necessary to resort to assembly-language programming, or to delve into details of electronic circuits.

The purpose of these notes is to give the reader a description of the computer memory hierarchy, and then to demonstrate how a programmer working in a high-level language can exploit the hierarchy to achieve near-optimal performance.

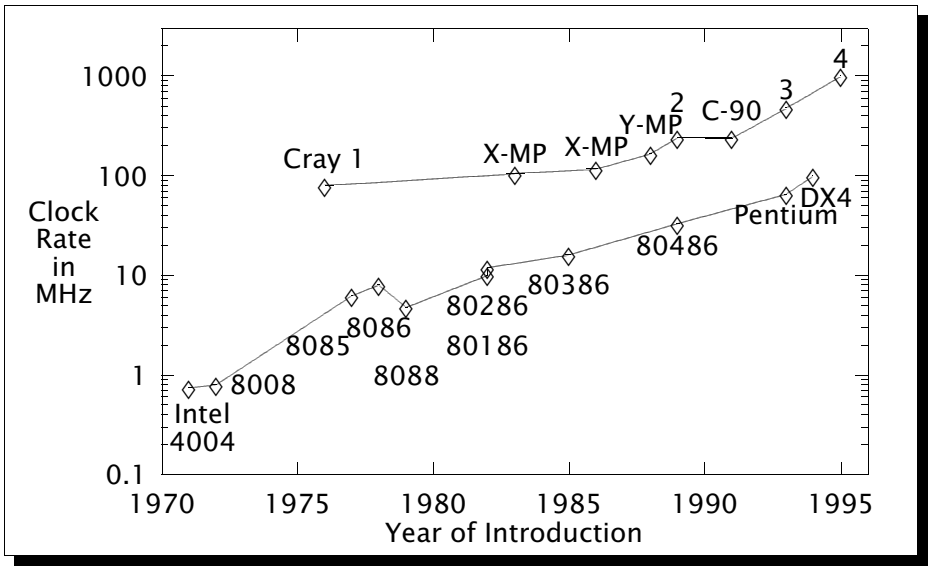

Figure 2: **Rising clock rates of supercomputers and microcomputers.** In microcomputers, each CPU model is generally available in a range of clock rates; the figures shown here are the fastest at the time of introduction.
Chips are fabricated on silicon disks, or wafers, about 15 cm in diameter, with up to several hundred chips per wafer. When they are subsequently cut from the wafer and packaged in ceramic or plastic and tested, some are found to work at higher clock rates than others; the faster ones command a higher price. For photographs and cost details, see [73, pp. 53–66].
Supercomputer clock rates are fixed because of the highly-optimized, and delicate, design balance between CPU, memory, buses, and peripherals.

2 Acronyms and units

Like any other technical field, computing has its own jargon, and numerous acronyms. We define technical terms as we encounter them, italicizing their first occurrence, and include them in the index at the end to make it easy to find the definitions. All of the authors of cited works and displayed quotations are indexed as well, so you can find your way from the bibliography back into the text where the citation is discussed.

Manufacturers commonly designate models of computers and their peripherals by numbers. Some of these, like 360 and 486, become so well known that they are commonly used as nouns, without naming the company. Computer products are indexed both by vendor, and by model number.

Capacities of computer memories span a very wide range. To keep the numbers manageable, it has become conventional to adopt the metric system's size prefixes, but with a slightly different meaning: instead of representing powers of 1000, they stand for powers of 1024, which is 2^{10}. *For all quantities other than memory, the size prefixes have their conventional metric meanings.* Figure 3 shows prefixes that you should become familiar with.

Prefix	Symbol		Memory Size
kilo	K	1024^1	1024
mega	M	1024^2	1,048,576
giga	G	1024^3	1,073,741,824
tera	T	1024^4	1,099,511,627,776
peta	P	1024^5	1,125,899,906,842,624
exa	E	1024^6	1,152,921,504,606,846,976

Figure 3: **Computer memory capacity prefixes.**

When a prefix symbol appear before the lowercase letter *b*, it means that many binary digits, or *bits*: 4 Kb is $(4 \times 1024 =)$ 4096 bits. Before the uppercase letter *B*, it means *bytes*,[1] where a byte is 8 bits: 0.5 KB is 512 bytes. Typical computer networks today transfer data at rates of 10 Mbps (million bits per second); the 'million' here is really 1,048,576.

In case you have forgotten them, or never learned some of them, the metric prefixes for quantities less than one are given in Figure 4. Notice that prefix symbols in uppercase mean something larger than one, and symbols in lowercase mean less than one.

Prefix	Symbol		Metric Value
milli	m	1000^{-1}	0.001
micro	μ	1000^{-2}	0.000,001
nano	n	1000^{-3}	0.000,000,001
pico	p	1000^{-4}	0.000,000,000,001
femto	f	1000^{-5}	0.000,000,000,000,001
atto	a	1000^{-6}	0.000,000,000,000,000,001

Figure 4: **Metric prefixes for small quantities.**

The symbol for *micro* is the Greek letter μ, mu, since the letter *m* was already used for *milli*.

[1] The spelling *byte* was chosen to avoid confusion between *bites* and *bits*. A four-bit quantity is referred to as a *nybble*: one hexadecimal digit is represented by a nybble.

Computer speeds are measured by the number of clock ticks, or cycles, per second, a unit known as *Hertz*. Modern computers are very fast, with clocks ticking millions or billions of times a second, so the common units are *MHz (megaHertz)* and *GHz (gigaHertz)*. The value 50 MHz means fifty million ticks per second, or more conveniently, fifty ticks per microsecond (μs), and 5 GHz means five billion ticks per second, or five ticks per nanosecond (ns).

Clock speeds are also measured by the length of the clock cycle, which is the reciprocal of the value in Hz. A clock rate of 1 MHz corresponds to a cycle time of 1 μs, and 1 GHz to a cycle time of 1 ns.

Modern computer circuits are so densely packed that the distance between the wires is measured in *microns (millionths of a meter)*. One micron is about 1/1000 the diameter of a human hair. The most advanced circuits in early 1994 have 0.25 micron spacing.

3 Memory hierarchy

Closest to the CPU are *registers*, which are on-chip memory locations, each capable of holding a single memory word.

Next to registers is a small *primary cache* (from 1 KB to 64 KB), and on some systems, a larger *secondary cache* (256 KB to 4 MB). One system, the Solbourne Viking/MXCC, has a 16 MB *tertiary cache*.

Cache memories are built from *static random-access memory* (SRAM) chips, which are an order of magnitude more expensive than the *dynamic random-access memory* (DRAM) chips used in main computer memory, and correspondingly faster.

Beyond registers and cache lies *main memory*, sometimes called *random-access memory* (RAM). On some systems (CDC 6600 and 7600, Hitachi S-820, large models of the IBM 360 family and descendants, and NEC SX-3/SX-X), main memory can be further divided into 'fast' and 'slow' memory.

Beyond registers, cache, and RAM lies *virtual memory*, in which disk storage, or slow RAM, is used to provide the illusion of a much larger address space than that offered by main memory.

Figure 5 illustrates the relative sizes and access times of these memory levels, and Figure 6 shows their cost and performance. Figure 7 presents disk storage price and speed trends.

Before we discuss each of these levels in the memory hierarchy, we need to take an excursion into instruction set design. Instruction sets have a major impact on cost and performance, and registers, the fastest level in the memory hierarchy, are intimately connected to the instruction set. We will return to a discussion of registers in Section 3.2 on page 15.

Level	Size (words)	Access Time (cycles)
Register	64–246	< 1
Primary cache	8K	1–2
Secondary cache	256K	5–15
Main memory	4G	40–100
Disk	100G	> 1000

Figure 5: **Memory hierarchy: size and performance.** The disk time is based on a 100 MHz clock, 10 msec average access time, 10 MB/sec transfer rate, and 4 KB disk blocks. The data for all but disk is taken from [8, Fig. 7].

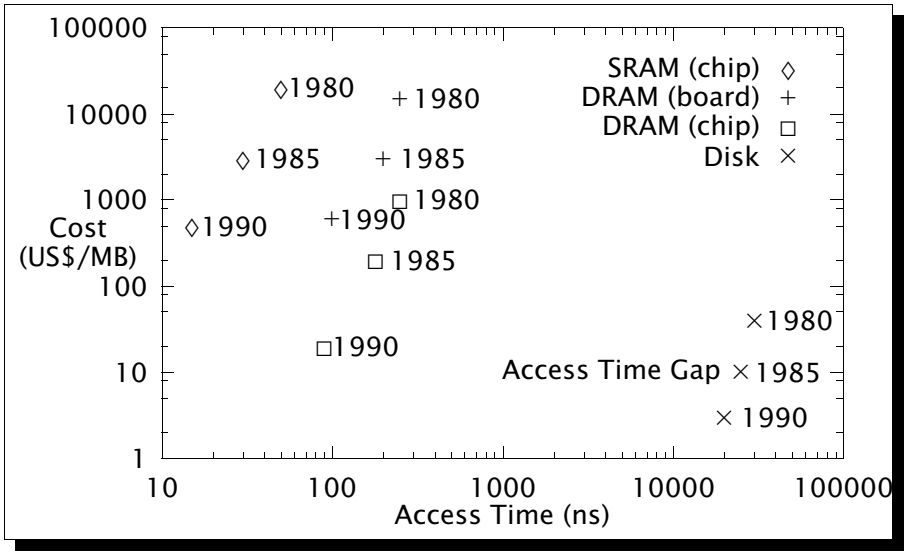

Figure 6: **Cost and access time of memory technologies.** The *access time gap* indicated at the lower right demonstrates a limitation of current memory technologies: there is nothing available between disk and DRAM, despite the rather large difference in access times. The data is taken from [73, Fig. 9.16, p. 518].

3.1 Complex and reduced instruction sets

On some architectures, the data to be operated on must first be loaded into a register from memory, and when the operation is complete, the register value must be stored back into memory before the register can be reused. This was the approach taken on early computers of the 1940s and 1950s, and on the high-performance CDC mainframes of the late 1960s and early 1970s.

Disk Model	Year	Price/MB US$	Ratio	Bytes /sec	Ratio
IBM 350	1956	9760	13555	10,000	1
IBM 1301	1961	3313	4601	90,000	9
IBM 2314	1965	1243	1726	312,000	31
IBM 3330-11	1973	191	265	806,000	81
IBM 3350	1973			1,198,000	120
IBM 3370	1979	41	57	1,859,000	186
DEC 5400	1993	0.64	1	5,532,000	553

Figure 7: **Price and speed of computer disk storage.** The prices are *not* adjusted for inflation; if they were, the price fall would be much steeper (the consumer price index rose by a factor of 5.17 from 1956 to 1993). The IBM price and speed data are taken from [74, p. 646].

The IBM 360 mainframe architecture, introduced on April 7, 1964, with descendants still in wide use today, and copied[2] by Amdahl, RCA, and Wang in the USA, Nixdorf and Siemens in Germany, Fujitsu and Hitachi in Japan, and Ryad in the USSR, took a different approach. It added complicated instructions that allowed arithmetic operations directly between a register and memory, and permitted memory-to-memory string moves and compares.

The Digital Equipment Corporation (DEC) VAX architecture (1978) went even further, with register-register, register-memory, and memory-memory variants of most instructions: the Fortran statement `A = B + C` could be compiled into a *single* triple-memory-address instruction `ADD B,C,A`, instead of the longer sequence

```
LOAD  Reg1,B
LOAD  Reg2,C
ADD   Reg1,Reg2
STORE Reg1,A
```

on a register-register architecture.

In the 1960s and 1970s, this trend of *Complex Instruction Set Computer* (CISC) architectures seemed a good thing, and most vendors followed it. Alliant, Amdahl, Bull, Burroughs, Convex, Data General, DEC, Fujitsu, Gould, Harris, Hewlett-Packard,Hitachi, Honeywell, IBM, Intel, Motorola, Na-

[2]Competing implementations were possible, because IBM published a definitive description of the architecture, known as the *IBM System/360 Principles of Operation*, together with formal mathematical specification of the instruction set. IBM obtained numerous patents on the technology required to implement System/360, so it survived, and indeed, thrived, in the competition. Successive editions of the *Principles of Operation* continue to define the IBM mainframe computer architecture.

tional Semiconductor, NCR, NEC, Norddata, Philips, Prime, RegneCentralen, Siemens, Texas Instruments, Univac, Wang, and Xerox all built and marketed CISC machines, although they weren't called that at the time.

One argument for CISC was that it eased the translation of high-level languages into machine instructions, as in the Fortran example above. Another was that the best way to achieve maximum performance was to put complex operations entirely inside the CPU. A good example of this is a *string move* instruction. In a high-level language, it would require a loop that moved one byte at a time. Implemented on chip, it could require millions of cycles to complete, but all of the loop control overhead could be handled without reference to memory, or further instruction fetch and decoding.

Notably absent from the CISC pack were Control Data Corporation (CDC) and Cray Research, Inc., which built the top-end machines for scientific computing. Seymour Cray was one of the founders of CDC in the 1960s, and one of the main architects of the CDC 1604, 6600 and 7600 machines. In the early 1970s, he left CDC to form Cray Research, Inc., where he was the principal designer of the Cray 1 and Cray 2. In May 1989, Seymour Cray left Cray Research, Inc. to form Cray Computer Corporation, where he now heads a design team working on the Cray 3 and Cray 4, while CRI has gone on to produce the Cray C-90. The Cray 1 was the first commercially successful vector supercomputer, with instructions capable of doing vector operations on up to 64 elements at a time. The Cray 3 and the smaller Convex C-3 are so far the only commercially-available computers constructed with *gallium arsenide* (GaAs) technology, which offers faster switching times than the various silicon circuit technologies used in other computers.

Cray's designs are characterized by small instruction sets that support only register-register operations; load and store instructions are the only path between CPU and memory. Although this type of architecture requires more instructions to do the job, the fact that Cray has been able to produce several of the world's fastest supercomputers[3] since 1976 is evidence that the approach has merit.

In 1975,a design team at IBM Austin under the leadership of John Cocke began development of a new architecture based on some of Cray's principles; the effort became known as the IBM 801 Project, after the number of the building in which they worked [75, 33, 28]. The initial target application of the new processor was telephone switching circuitry, but the scope was later broadened to general-purpose computing. The project operated under industrial secrecy until the first publications about it appeared in 1982.

In 1980, two architects in academia, John L. Hennessy at Stanford University, and David A. Patterson across San Francisco Bay at the Berkeley

[3]You can pick up the latest Cray 3 with 1 CPU and 64 Mwords of memory for about US$3 million. Four CPUs will cost about US$10 million, and sixteen, the architectural limit, about US$32 million. Save your spare change for operations costs; the electricity bill for the Cray X-MP/24 at the San Diego Supercomputer Center ran about US$1 million per year.

campus of the University of California, decided to investigate load-store microprocessor designs, and Patterson coined the name RISC, for *Reduced Instruction Set Computer*.

Today's descendents of these efforts are the MIPS processors used by DEC, Honeywell-Bull, NEC, Siemens, Silicon Graphics (SGI), Stardent, Tandem, and others, from the Stanford work, and the SPARC processors used by AT&T, Cray, Fujitsu, ICL, LSI Logic, Philips, Solbourne, Sun, Texas Instruments, Thinking Machines, Toshiba, Xerox, and others, from the Berkeley work.

The vindication of the RISC design concept came when the Berkeley RISC-I chip ran benchmarks at a substantial fraction of the speed of the popular DEC VAX-780 minicomputer, and the RISC-II chip surpassed it. Both chips were done by student design teams under direction of their professors.

Back at IBM, the 801 Project was finally convinced to look at the workstation market, and in 1986, IBM introduced the IBM RT PC, the first RISC workstation. Sadly, such a long time, eight years, had elapsed between design and product introduction that the performance of the RT was below that of its competitors, and the machine was widely viewed as a failure. Indeed, there was an industry joke that RT stood for *Reduced Taxes*, because IBM gave away more systems than they sold.

Nevertheless, IBM learned many useful lessons from the RT, and started a new project in 1986 to develop their second generation of RISC processors. This work resulted in the introduction in February 1990 of the IBM RS/6000 workstations, based on the RIOS, or POWER (Performance Optimized With Enhanced RISC), CPU.

The IBM RS/6000 systems stunned the world with their floating-point performance. In less than a year following the RS/6000 announcement, IBM had more than US$1 billion in sales revenues from the new machines, far beyond their initial marketing projections, and an excellent showing for a new product in a US$60 billion company. Figure 8 shows the financial rankings of some of the computer companies mentioned in this document.

Half-million dollar large minicomputer systems of the time were delivering 1 to 2 Mflops (millions of floating-point operations per second), and the highest-performance small supercomputer, the Ardent Titan (`graphics.math.utah.edu` is a local example) could reach 15 to 20 Mflops with vector operations and four processors working in parallel. The new single-processor IBM RS/6000-320 could achieve more than 30 Mflops in matrix multiplication and Gaussian elimination, and cost only about US$15,000.

Since early 1990, several other vendors have introduced RISC workstations capable of very high floating-point performance, and by late 1993, workstations based on the DEC Alpha, Hewlett-Packard PA-RISC, and IBM POWER-2 processors were offering 100 to 250 Mflops. For comparison, the multi-million-dollar 1976-vintage Cray 1 vector supercomputer had a peak speed of 160 Mflops.

World Rank			IS Revenue (US$M)		
1992	1991	Company	1992	1991	% Change
1	1	IBM	64,520.0	62,840.0	2.7%
2	2	Fujitsu	20,142.2	19,330.9	4.2%
3	3	NEC	15,395.0	15,317.6	0.5%
4	4	DEC	14,162.0	14,237.8	-0.5%
5	5	Hewlett-Packard	12,688.0	10,646.0	19.2%
6	6	Hitachi	11,352.0	10,310.2	10.1%
7	7	AT&T	10,450.0	8,169.0	27.9%
8	9	Siemens Nixdorf	8,345.1	7.308.6	14.2%
9	8	Unisys	7.832.0	8,000.0	-2.1%
10	13	Toshiba	7,448.7	5,115.9	45.6%
11	10	Apple	7,173.7	6,496.0	10.4%
19	17	Sun	3,832.0	3,454.7	10.9%
20	24	Microsoft	3,253.0	2,275.9	42.9%
42	31	Wang	1,490.0	1,940.0	-23.2%
47	47	Intel	1,230.0	1,100.0	11.8%
55	44	Data General	1,100.8	1,212.0	-9.2%
60	54	Motorola	935.0	987.0	-5.3%
62	72	Silicon Graphics	923.0	805.7	14.6%
68	65	Texas Instruments	800.0	747.9	7.0%
69	58	Cray Research	797.6	862.5	-7.5%
97	46	Control Data	517.0	1,172.6	-55.9%

Figure 8: **Selected entries from the Datamation 100.** Rankings from the top 100 of the world's information technology suppliers from the 1993 annual Datamation survey [2]. Some corporate revenues are excluded from the data: Intel's 1992 gross revenue exceeds US$5 billion.

> In supercomputing, peak or advertising power is the maximum performance that the manufacturer guarantees no program will ever exceed. ... For $\mathcal{O}(1000)$ matrices that are typical of supercomputer applications, scalable multicomputers with several thousand processing elements deliver negligible performance.
>
> C. Gorden Bell
> 1992 [12, p. 30]

Silicon Graphics expects to ship their Power Challenge systems in mid-1994 with 350 Mflops peak performance per processor, and up to 18 processors in a box. Sun has announced plans for new 64-bit generations of SPARC, to be named UltraSPARC, based on the SPARC Version 9 architecture

specification [97] The first two generations should appear in 1995, and the third generation, targeted at 700 to 1000 Mflops, in 1997. Current SPARC processors are capable of 10 to 50 Mflops. Fujitsu, Hitachi, and NEC all have teraflop computer projects with target dates of 1995 or 1996. Just how much of this promised performance will be achievable by real-world applications remains to be seen.

> The irony of the teraflops quest is that programming may not change very much, even though virtually all programs must be rewritten to exploit the very high degree of parallelism required for efficient operation of the coarse-grained scalable computers. Scientists and engineers will use just another dialect of Fortran that supports data parallelism.
>
> C. Gorden Bell
> 1992 [12, p. 29]

> ... a new language, having more inherent parallelism, such as dataflow may evolve. Fortran will adopt it.
>
> C. Gorden Bell
> 1992 [12, p. 43]

One might assume from the words *complex* and *reduced* that RISC computers would have smaller instruction sets than CISC computers. This is not necessarily the case, as Figure 9 shows. The distinguishing characteristic is really that RISC designs restrict memory access to load and store instructions, have uniform instruction widths, and attempt to achieve one cycle (or less) per instruction. Some CISC implementations average 10 or more cycles per instruction.

The DEC Alpha instruction set is particularly large. Of the 451 instructions, 305 (68%) are floating-point instructions. This architecture was designed for high floating-point performance, and has both IEEE 754 and DEC VAX floating-point instructions.

Sophisticated optimizing compilers are essential on RISC architectures, and hand-coded assembly-language programs are expected to be very rare. On CISC architectures, compilers employ only a modest fraction of the instruction sets. RISC designers argue that only instructions that get used should be in the architecture.

Processor	No. of Instr. Sizes	Indirect Memory Address	Combined Load/Store Arithmetic Operations	Max. No. of Memory Accesses	No. of Memory Address Modes	No. of Instr.
Historic CPU						
EDSAC (1949)	??	??	??	??	??	18
IBM 701 (1952)	1	??	??	??	??	33
IBM 704 (1954)	??	??	??	??	??	91
IBM 709 (1957)	??	Yes	??	??	??	> 180
IBM 7090 (1958)	??	Yes	??	??	??	> 180
IBM 1401 (1958)	6	No	??	3	??	34
IBM 7044 (1961)	??	Yes	??	??	??	120
IBM 7094 (1962)	??	Yes	??	??	??	274
IBM 7030 Stretch (1961)	??	Yes	??	??	??	735
CDC 6600 (1964)	1	No	No	1	??	63
CISC-style CPU						
IBM System/360	4	No	Yes	2	2	143
IBM System/370	4	No	Yes	2	2	204
IBM System/370-XA	4	No	Yes	2	2	208
IBM 3090-VF	4	No	Yes	2	2	≥ 379
Intel 4004	??	??	??	??	??	45
Intel 8008	??	??	??	??	??	48
Intel 8085	??	??	??	??	??	74
Intel 8086	??	No	Yes	2	??	133
Intel 80386	12	No	Yes	2	15	135
Intel 80486	12	No	Yes	2	15	151
Intel iAPX 432	many	Yes	No	3	4	221
Motorola 68040	11	Yes	Yes	2	44	114
DEC VAX	54	Yes	Yes	6	22	244
RISC-style CPU						
Cray 1	2	No	No	1	1	128
Cray X-MP	2	No	No	1	1	246
DEC Alpha	1	No	No	1	1	451
HP PA-RISC	1	No	No	1	10	165
IBM 801	1	No	No	1	4	???
IBM POWER	1	No	No	1	4	184
Intel i860	1	No	No	1	1	161
Intel i960	2	No	No	1	7	184
MIPS R2000	1	No	No	1	1	93
MIPS R4000	1	No	No	1	1	154
Motorola 88000	1	No	No	1	3	51
Sun SPARC	1	No	No	1	2	72

Figure 9: **Instruction set sizes.** The IBM 701 was the first IBM machine to use binary, instead of decimal, arithmetic. The IBM 709 followed the IBM 704, the machine on which the first Fortran compiler was released, in April 1957. The 709 got Fortran by mid-1958. A related model, the IBM 705, was the last large vacuum tube computer built by IBM.
The IBM 7030 Stretch was the first attempt at a supercomputer, one that would run over 100 times faster than existing computers, with a design target of 1 μs for a floating-point addition. It was the first machine to use a 64-bit word, the first to use 8-bit bytes, and one of the first to be built with transistors instead of vacuum tubes. Only seven machines were ever made, so Stretch was not a commercial success. In the early 1980s, a Stretch was resurrected and run for several years at Brigham Young University in Provo, UT, USA.
The CDC 6600 was probably the first commercially successful scientific computer, three times faster than the IBM Stretch, and much cheaper. It is the direct ancestor of today's Cray supercomputers.
Some of the data is taken from *Digital Review*, 2 December 1991, p. 52.

> An optimizing compiler is ultimately written by an overworked human being. If the human being can't figure out how to make the compiler profitably use a given instruction, then his compiler is not going to emit it.
>
> Brian Case
> 1992 [85, p. 13]

Constant instruction widths in RISC architectures make it easier to overlap instruction execution, a technique called *pipelining*.

If a pipelined CPU runs internally at a faster clock rate than other components, it is said to be *superpipelined*. The MIPS R4400 is the best current example of this.

If multiple functional units are provided so that for example, a floating-point operation, an integer operation, and a branch operation can all be processed simultaneously, the architecture is said to be *superscalar*. The IBM RS/6000, Intel i960CA, and SuperSPARC processors are all superscalar. The best such architectures are capable of averaging substantially less than one cycle per instruction on numerically-intensive benchmarks.

It is futile to compare different processors by clock rate, since different instructions, particularly on CISC computers, have widely different cycle counts. Equally useless are MIPS (millions of instructions per second) ratings. One CISC instruction can require several RISC instructions. What really matters is the time taken to do the *same* job.

Researchers continue to explore ways of preparing realistic *benchmark* programs for comparison of computer systems. The most reliable benchmark suite at present is that from the System Performance Evaluation Corporation (SPEC). SPEC numbers are based on a large collection of numerical and non-numerical programs, and reported as SPECint92 and SPECfp92 numbers. The latter are approximately the same magnitude as Mflops.

Measuring $1/8$ inch wide by $1/6$ inch long and made of 2,300 MOS transistors,[a] Intel's first microprocessor was equal in computing power to the first electronic computer, ENIAC, which filled 3,000 cubic feet with 18,000 vacuum tubes when it was built in 1946. Intel's 4004 4-bit microprocessor (1971) could execute 60,000 operations in 1 second—primitive by today's standards, but a major breakthrough at the time.

Intel [43, p. 132]
1992

[a] A transistor is a modulation, amplification, and switching device implemented in solid-state logic (i.e. no moving parts). It replaced the older, and very much larger, vacuum tubes, or valves as they are called in Britain.

In 1956, the Nobel Prize in Physics was awarded to William Shockley, John Bardeen, and Walter H. Brattain of AT&T Bell Telephone Laboratories for their invention of the transistor in December 1947 (the year of the EDSAC).

In 1972, John Bardeen shared a second Nobel Prize in Physics with Leon N. Cooper and J. Robert Schrieffer, for their theory of superconductivity.

In the mid-1980s, discoveries of (relatively) high-temperature superconductivity led to a surge of interest in the field. Superconductivity may someday be an important aspect of high-performance computers, because in a superconducting circuit, there is no electrical resistance, and therefore, no heat. It is a combination of resistive heating, and short circuits due to the small wire sizes, that currently limits computer chip size and speed.

...the development of the 16-bit 8086, introduced in 1978...29K transistors[a] ...
In late 1985, Intel introduced the Intel 386 DX microprocessor. It has more than 275,000 transistors on a chip.
In February 1989, Intel also introduced the world's first one-million transistor CPU, the i860 microprocessor.

Intel [43, p. 133–134] 1992

[a] The Intel 4004 had 2300 transistors, the 8008 had 2000, and the 8085 had 6200.

It has recently been observed that the number of transistors in emerging high-density memory chips is much larger than the number in current CPUs. Although memory circuits are much simpler than CPUs because of their regularity, it is nevertheless possible that future large memory chips might contain a CPU without sacrifice of significant chip area. This would have several important effects:

- access time to the on-chip memory would be sharply reduced, with a corresponding increase in performance;
- the cost of a powerful CPU would become negligible;
- every computer could potentially be a parallel computer.

> By the year 2000, ... it should be possible to integrate between 50 and 100 million transistors on a silicon die... a processor... could have as many as four distinct CPUs on-chip, two megabytes of cache memory, a special 20-million transistor unit devoted to advanced human interface incorporating voice I/O, image recognition and motion video, and two vector processing units for three-dimensional graphics and simulation. Each of the four CPUs, each with 5 million transistors, could run at 700 million instructions per second. Together, all the units on this 250-MHz chip might perform at 2 billion instructions per second.
>
> Intel [43, p. 135] 1992

3.2 Registers

Almost all computers have registers to provide a small amount of fast-access on-chip memory. The number of registers is limited because a fixed number of bits in the instruction word encodes the register address: a 3-bit register address permits (2^3 =) 8 registers. Figure 10 illustrates a typical instruction encoding.

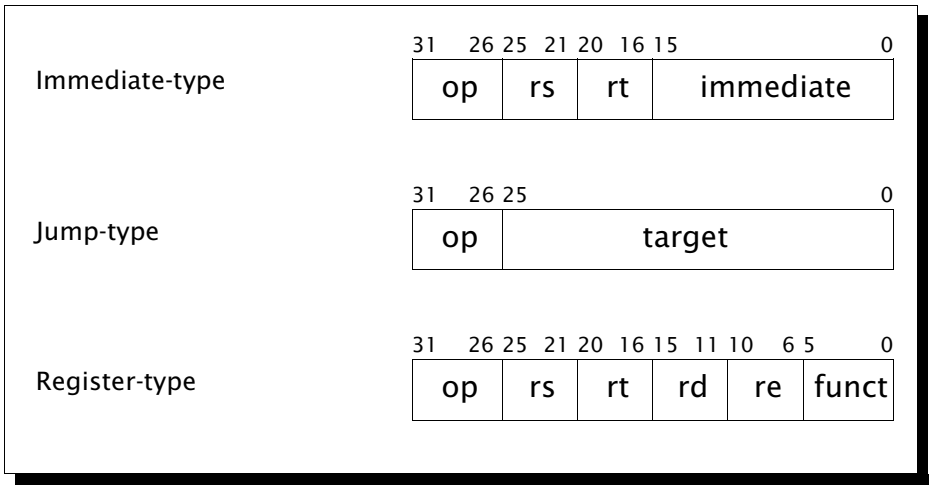

Figure 10: **CPU instruction formats.** These particular encodings are for the MIPS processors, used in DECstation, MIPS, Silicon Graphics, and Stardent computers. The 5-bit rs, rt, rd, and re fields address 32 registers.
In the immediate-type and jump-type encodings, the 6-bit operation code limits the total number of instructions to 64. However, the register-type instructions have a 6-bit function code to supply 64 more instructions for each operation code, so the instruction set could be made quite large.

The number of registers varies among architectures; it can be as few as 1 to 4, but in high-performance systems, is usually larger, with 8 to 64 being typical. Figure 11 presents register characteristics of several well-known architectures.

Registers usually need to be saved and restored at the point of procedure calls, so large register sets can make procedure calls, context switches, and interrupt handling expensive. This is undesirable, because modern programming techniques encourage modularization of code into many small procedures. On the other hand, having too few registers means that excessive numbers of load and store instructions must be issued.

One RISC architecture, the Sun SPARC, supports *overlapping register windows* selected from large register sets, up to 520 in current chip implementations. Each process sees 32 registers at one time, 8 for global values, 8 in the caller's space, 8 for local values, and 8 in the callee's space. When a procedure call is made, the register window is moved over 16 registers, so that the former 8 callee registers now become 8 caller registers, without having to move any data. This scheme reduces the frequency of register saving and restoring at procedure calls and returns, and also simplifies addressing of procedure arguments. When the complete register set is exhausted because of deeply nested procedure calls, the processor interrupts so that a software trap handler can arrange to save the complete register set. In practice, this is a rare occurrence.

Most architectures have separate integer and floating-point registers (e.g. IBM 360, Intel, Motorola 680x0, and almost all RISC architectures, other than the Motorola 88000). A few have general-purpose registers that can hold either integer or floating-point values (e.g. DEC PDP-10, DEC VAX, and Motorola 88000). Separate integer and floating-point registers permits design of systems without any floating-point hardware, which is sometimes cost-effective in specialized applications. Separate register sets also impose a form of type safety, in that an integer operation cannot be applied to a floating-point value without first moving the value to a register of the proper type.

On most architectures, almost any register can be used with any instruction, but on a few, notably the early models of the Intel processors used in the IBM PC, most instructions require use of particular dedicated registers. This inflexibility forces the use of additional register-register or register-memory instructions that would be unnecessary with a general register architecture.

In the case of floating-point registers, each may hold a single word, in which case adjacent even/odd register pairs are required to hold a double-precision value, and if supported, four adjacent registers hold a quadruple-precision value.

Some architectures support floating-point registers that hold double-precision values, in which case single-precision values are extended with

Architecture	General Registers	Floating-point Registers	Vector Registers
AMD 29000	128 × 32-bit local 64 × 32-bit global	0	0
CDC 6400	8 address (A) 18-bit + 8 index (B) 18-bit + 8 data (X) 60-bit	0	0
Convex C-1, 2, 3	8 address 32-bit + 8 data 64-bit	0	8 × 128-element 64-bit + 8 × 128-element 64-bit communication registers
Cray 1, 2, X-MP, Y-MP	8 address (A) 24-bit + 64 index (B) 24-bit + 8 scalar (S) 64-bit + 64 scalar (T) 64-bit	0	8 × 64-element 64-bit
Cray C-90	same as on Cray Y-MP	0	8 × 128-element 64-bit
DEC PDP-10	16 × 36-bit	0	0
DEC PDP-11	8 × 16-bit	6 × 64-bit	0
DEC VAX	15 × 32-bit	0	0
DEC Alpha	32 × 64-bit	32 × 64-bit	0
Fujitsu VP-2600	16 × 32-bit	4 × 64-bit	8 × 2048-element 64-bit or 256 × 64-element 64-bit
Hitachi S-820/80	16 × 32-bit	4 × 64-bit	32 × 512-element 64-bit
HP 9000/8xx	32 × 32-bit	12 × 64-bit	0
HP 9000/7xx	32 × 32-bit	28 × 64-bit	0
IBM 360	16 × 32-bit	4 × 64-bit	0
IBM 3090-VF	16 × 32-bit	4 × 64-bit	16 × 128-element 32-bit or 8 × 128-element 64-bit
Intel 8086 + 8087	8 dedicated 16-bit	8 × 80-bit	0
Intel i860	32 × 32-bit	32 × 32-bit	0
Intel i960	16 × 32-bit local + 16 × 32-bit global	4 × 80-bit	0
MIPS R4000	32 × 32-bit	32 × 32-bit	0
Motorola 68k	8 data (D) 32-bit + 8 address (A) 32-bit	8 × 80-bit	0
Motorola 88k	32 × 32-bit	0	0
NEC SX-3	128 × 64-bit	0	72 × 256-element 64-bit
Sun SPARC	32 × 32-bit	32 × 32-bit	0
Sun SPARC V9	32 × 64-bit	32 × 64-bit	0
Stardent 1520	32 × 32-bit	32 × 32-bit	32 × 32-element 64-bit

Figure 11: **Register counts for various architectures.** Some machines have additional special-purpose registers which are not recorded here. Machines without floating-point registers use the general registers instead.
The DEC PDP-10 registers are mapped into the start of addressable memory, and short code loops can be moved there for faster execution.
The Stardent vector registers can be configured in groups of any power of two, e.g. one 1024-element 64-bit register, two 512-element registers, ..., 1024 1-element registers. However, Stardent compilers default to using 32 32-element registers, because performance studies later demonstrated that memory access speed, not vector length, limited performance.
Like the Stardent, the Fujitsu VP-2600 registers are reconfigurable, and the compilers do this automatically.
Although the Hitachi vector registers have fixed length, the hardware is capable of managing operations on vectors of arbitrary length. On other vector processors, the compiler must do this job.
Where register counts are not a power of two, some registers have been dedicated for special purposes, such as holding flags, or a permanent zero, or the current instruction address.

trailing zero bits in the fraction, and quadruple-precision values, when supported, fill an adjacent register pair.

Finally, a few architectures (but most of the world's computers) have floating-point registers that are *longer* than memory double words: the Honeywell mainframes of the 1970s, and the Intel and Motorola microprocessors. The latter two implement floating-point arithmetic in the IEEE 754 80-bit temporary real format, making conversions between the extended format and the 32-bit or 64-bit formats during memory references. Floating-point computations are done *only* with 80-bit values. The higher precision of intermediate computations is generally beneficial, but it can sometimes lead to surprises.

3.3 Cache memory

Between registers and regular memory lies cache memory, which is constructed with faster, but more expensive, memory technology; see Figure 6 on page 6 for some comparisons.

> A register bank is faster than a cache, both because it is smaller, and because the address mechanism is much simpler. Designers of high performance machines have typically found it possible to read one register and write another in a single cycle, while two cycles [latency] are needed for a cache access. ... Also, since there are not too many registers, it is feasible to duplicate or triplicate them, so that several registers can be read out simultaneously.
>
> Butler W. Lampson
> 1982 [73, p. 483]

The first commercial machine to provide cache memory was the IBM 360/85 in 1968, although the idea had been discussed for several machines built in England in the early 1960s.

In some architectures, the cache is on chip, giving the opportunity for introducing a larger secondary cache off chip.

The goal of cache designs is to keep the most frequently-used values in faster memory, transparently to the programmer and compiler. That is, when a load instruction is issued, the CPU first looks in the cache, and if the required value is found there, loads it. Otherwise, it fetches the word from memory into the cache, and then into the register.

To further enhance performance, most cache designs fetch a group of consecutive memory words (2, 4, 8, 16, or 32) when a word is loaded from memory. This word group is called a *cache line*. The reason for loading multiple words is that adjacent words are likely to be needed soon. This is generally the case in matrix and vector operations in numerical computing.

On a store operation, data is written from a register into the cache, not to memory. If the design is a *write-through cache*, then the cache word is also immediately sent to memory. If it is a *write-back cache*, the cache word is not transferred to memory until the cache location is needed by another load operation.

A write-through policy means that some unnecessary stores to memory are done, but memory always contains the current data values. A write-back policy means that unnecessary stores are avoided, but makes design of multiprocessor systems more difficult, since other processors now have to look in multiple caches, as well as memory. A write-back policy also interferes with immediate visibility of data in video-buffer memory.

Many cache designs have separate caches for instructions and data. The I-cache does not have to support write operations, since instructions are treated as read-only. Separation of instructions and data produces better cache use, and allows the CPU to fetch instructions and data simultaneously from both I- and D-caches. It is also possible to use different sizes and cache-line lengths for the I- and D-caches; a larger D-cache is likely to be advantageous.

Clearly, the introduction of cache introduces substantial complications, particularly in multiprocessor computers. Cache also interposes an additional overhead on memory access that may be undesirable. An internal report from Sun Microsystems tabulated timings of 412 benchmark programs on the same Sun workstation, with and without cache. On 15 benchmarks, the cached system actually ran slower than the non-cached system, usually by about 20%, because little reuse was being made of the data loaded from memory into cache.

Smaller vector machines with caches generally bypass the cache in vector loads and stores. Some larger vector machines have no cache at all: their main memories are constructed with SRAMs instead of DRAMs, so they are equivalent to cache memories in speed. The Kendall Square Research (KSR) parallel processor dispenses instead with main memory: it has a patented *All Cache* architecture.

Cache architectures remain an active area of research, and will likely do so until a memory technology is found that eliminates the ever-widening gap in performance between the CPU and memory.

3.4 Main memory

Today, main memory, or random-access memory (RAM), is constructed with solid-state storage devices. These devices do not retain stored data when power is removed.

3.4.1 Main memory technologies

Dynamic RAM (DRAM) must be continually refreshed to maintain its data. The refresh operation is handled by the memory controller hardware, and is therefore invisible to programs and programmers, but it does impact memory performance.

Static RAM (SRAM) does not require refreshing, but is much more expensive; see Figure 6 on page 6 for some comparisons.

RAM used to be called *core* memory because it was constructed of tiny ferrite cores, small perforated disks (like a washer), each about the diameter of a pencil lead, hand-woven into a matrix of fine wires. If you shook a core memory module, you could hear it faintly jingle. Each core recorded just one bit, so core memory was very expensive compared to solid-state RAM, which replaced it in the 1970s. Core memory first appeared on the MIT Whirlwind computer whose development began in 1947; it had 2048 16-bit words of core, and was in use at MIT until the mid-1960s.

> Among the customers who especially appreciated ferrite-core memories were those who had previous used IBM 701 or 702 computers equipped with cathode ray tube memories. Application programs were frequently run twice on the 701 because the computer provided no other way to know if an error had occurred. On the 702, errors were detected all too frequently by the parity-check, error-detection circuitry. The improvement in reliability with ferrite cores was dramatic.
>
> Emerson W. Pugh, Lyle R. Johnson, and John H. Palmer
> 1991 [74, p. 175]

Because of its laborious construction, core memory was not cheap. Although a mechanical core threading machine was invented in 1956, core main memories for the early IBM 360 machines cost about US$0.02 per bit, and faster core memories were as much as US$0.75 per bit [74, p. 195]. Assuming only one parity bit per byte, this corresponds to prices ranging from US$189,000 to more than US$7 million per megabyte of core memory, compared to 1994 prices of about US$40 for a megabyte of semiconductor memory. When the factor of four from the consumer price index change from 1965 to 1992 is incorporated to account for inflation, a megabyte of fast memory in 1965 would have cost as much as the most expensive supercomputer today!

> ... the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. ... Its cost was reasonable, and because it was reliable, it could in due course be made large.
>
> Maurice V. Wilkes
> *Memoirs of a Computer Pioneer*
> 1985 [98, p. 209] [73, p. 425]

The name *core* lives on, even if the technology does not: a dump of the memory image of a UNIX process is still called a `core` file.

Most computers today also have a small amount of *read-only memory (ROM)* for holding startup code that is executed when the machine is first powered on. ROM chips cannot be rewritten after their first writing.

> ... the first commercial use of LSI MOS silicon-gate technology occurred with the introduction of Intel's 1101 256-bit SRAM in 1969. It was the emergence shortly thereafter of the 1103, the world's first 1 Kbit dynamic RAM (DRAM), that signaled a breakthrough; for the first time, significant amounts of information could be stored on a single chip. The 1103 began to replace core memories and soon became an industry standard. By 1972, it was the largest selling semiconductor memory in the world.
>
> Intel [43, p. 131] 1992

Erasable programmable ROM (EPROM) can be erased by exposure to ultraviolet light.

A variant of EPROM, known as *electrically-erasable programmable ROM (EEPROM)*, can be written a few thousand times under application of higher voltage.

Another variant, *non-volatile RAM (NVRAM)*, can retain stored data provided a small voltage from a backup battery is supplied.

Flash memory, introduced by Intel in 1988, combines the reliability and low cost of EPROM with the electrical erasability of EEPROM.

EPROM and NVRAM are used where small amounts of memory must be maintained for long times without external disk storage. These two memory technologies are used for real-time clocks, computer startup configuration parameters, and access passwords and page counts in PostScript laser printers. They make it possible to have small standalone computers without any external device, such as a magnetic tape, for loading the initial program code. This is extremely important in embedded applications; the average automobile produced in Detroit in the 1990s has more than 20 microprocessors in it.

> The microprocessor and the EPROM developed a symbiotic relationship with one another: The ability of EPROM to be reprogrammed to meet specific application needs allowed the microprocessor to act as a general-purpose processor to handle the logic functions of a virtually unlimited range of applications.
>
> Intel [43, p. 131] 1992

In the early 1990s, *single inline memory modules* (SIMMs) became common. SIMMs are ordinary DRAM chips, but in a different mounting configuration. Instead of plugging in flat on a large memory board, SIMMs are mounted on a small board that plugs in perpendicular to a memory board slot, allowing larger numbers of chips to be attached.

DRAM chip capacities are measured in bits; 8 chips together produce an identical number of bytes. In all but the cheapest designs, more than 8 chips are used: 9 chips provide a single parity bit per byte, and 37 chips provide single-bit error correction, and double-bit error detection, in each 32-bit word. Error detection and recovery are increasingly important as memory capacities increase, because error rates are proportional to capacity.

3.4.2 Cost of main memory

For the last two decades, memory prices have fallen by a factor of about four every three to four years, and succeeding generations of architectures have had increasingly large address spaces. Memory sizes will be treated in the next section; the memory price fall is illustrated in Figure 12.

From 1990 to 1993, memory prices have regrettably been relatively constant, but capacity continues to increase. By 1993, 4 Mb DRAMs were common, and by the end of that year, the per-byte price of 16 Mb DRAM dropped below that of 4 Mb DRAMs. In early 1994, 64 Mb DRAMs became available.

At the International Solid State Circuits Conference (ISSCC'93) held in San Francisco in February 1993, Hitachi, NEC, and Toshiba announced initial prototypes of working 256 Mb DRAMs, with access speeds as low as 30 ns. Sony and Mitsubishi also described 256 Mb DRAM plans. NEC expects to ship sample quantities by 1995, and is also investigating 1 Gb DRAM development.

At the same conference, Hitachi announced 64 Mb SRAM; the densest static RAM to date has been 16 Mb.

Perhaps the most tantalizing prospect for the future of computer memory technology came from the Electrotechnical Laboratory in Japan, which has produced Josephson junction RAM.[4] JJRAM chip capacity is still very

[4] The British physicist Brian Josephson (1940–) received the Nobel Prize in Physics in 1973 for his picosecond switching technology. The discovery was made in 1962 when he was only 22 years old!

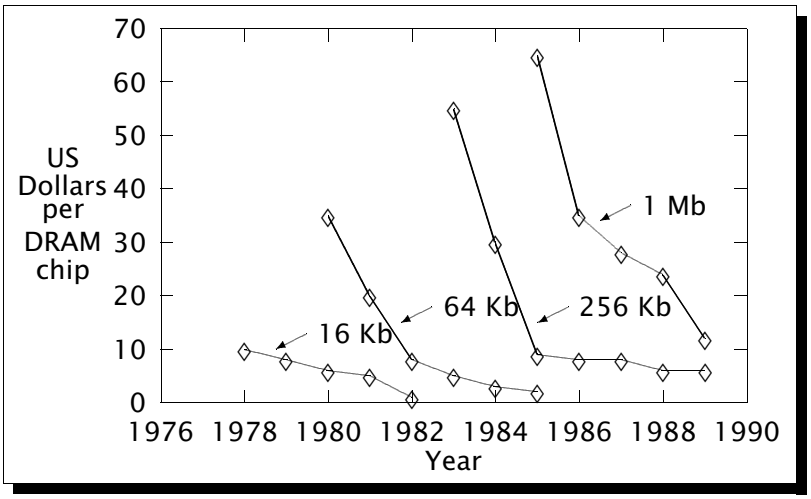

Figure 12: **Memory chip price and capacity.** Prices are *not* adjusted for inflation; if they were, the curves would fall even more sharply (the consumer price index change from 1978 to 1990 is very close to a factor of two). DRAMs drop to a constant price of US$1 to US$2 per chip, independent of capacity. For reasons connected with transistor technology, each successive DRAM chip generation increases capacity 4 times. The data is taken from [73, Fig. 2.9, p. 55].

small—16 Kb, but it is 100 times faster than 256 Mb DRAM, and consumes only 5% of the latter's power.

3.4.3 Main memory size

> In the early 1950s, there was a widely-held belief among theorists that 1000 words or so of memory should suffice. However, practical programmers seemed to need more than that.
>
> Charles J. Bashe et al
> 1981 [6, p. 365]

The size of main memory is limited by the number of address bits in the architecture, and by how much the customer can afford to pay.

> The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses. Unfortunately, the architects didn't reveal their plans to the software people, and the expansion effort was foiled by programmers who stored extra information in the upper eight "unused" address bits.
>
> John L. Hennessy and David A. Patterson
> 1990 [73, p. 486]

In work begun in 1975, the 24-bit memory address of the IBM 360 architecture was increased to 31 bits in the IBM 370-XA (eXtended Architecture) first released in 1981 with the IBM 308x models, but it took some time for operating system support became available, precisely for the reason noted in the quotation above. It was not possible to use all 32 bits for addressing, because an important loop instruction used signed arithmetic on addresses, and because bit 32 was used to mark the last entry in a procedure call address list.

Once the software had been cleaned up, further evolution was easier: the IBM ESA/370 (Enterprise Systems Architecture) was announced in 1988 and implemented in the ES/3090 processor family. It supports a 44-bit address (16 TB), using a technique to be discussed in the next section.

> There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every computer.
>
> C. G. Bell and W. D. Strecker
> 1976 [73, p. 481]

> (7) **Provide a decade of addressing.** Computers never have enough address space. History is full of examples of computers that have run out of memory-addressing space for important applications while still relatively early in their life (e.g. the PDP-8, the IBM System 360, and the IBM PC). Ideally, a system should be designed to last for 10 years without running out of memory-address space for the maximum amount of memory that can be installed. Since dynamic RAM chips tend to quadruple in size every three years, this means that the address space should contain seven bits more than required to address installed memory on the initial system.
>
> C. Gordon Bell
> *Eleven Rules of Supercomputer Design*
> 1989 [81, p. 178]

Processor	Year of Introduction	Address Bits	Memory Size (bytes)	
4004	1971	12	4,096	(4K)
8008	1972	14	16,384	(16K)
8080	1974	16	65,536	(64K)
8085	1976	16	65,536	(64K)
8086	1978	20	1,048,576	(1M)
8088	1978	20	1,048,576	(1M)
80186	1981	20	1,048,576	(1M)
80286	1982	24	16,777,216	(16M)
80386	1985	44	17,592,186,044,416	(16T)
80486	1989	44	17,592,186,044,416	(16T)
Pentium	1993	44	17,592,186,044,416	(16T)
P6	1995 (est)	44	17,592,186,044,416	(16T)
P7	1997 (est)	64	18,446,744,073,709,551,616	(16E)
iAPX 432	1981	40	1,099,511,627,776	(1T)
i960	1985	32	4,294,967,296	(4G)
i860	1989	32	4,294,967,296	(4G)

Figure 13: **Intel processor memory sizes.** The Intel 4004 was the first microprocessor. P6 and P7 are code names for development projects; the final product names will differ.
The last three processors are rather different architectures.
The iAPX 432 was an extremely complex processor, designed for support of the Ada programming language. It was a commercial failure because its complexity severely limited performance. Although its memory space seems large, it is divided into 16M segments, each of 64KB. The instruction set is variable-length bit encoded, and instructions need not start on a byte boundary; they vary in length from 6 to 344 bits. There are no registers; operands reside in memory, or on a stack in a special segment.
The i960 is a curious blend of RISC and CISC. It has been used extensively in device control applications, including many laser printers.
The i860 RISC processor appeared in only one workstation, from the now-defunct Stardent. It is used in Intel multiprocessor supercomputers, the iPSC860, Delta, Sigma, and Paragon. The i860 has complicated instruction sequencing requirements that have made it quite difficult to write correct compilers. This no doubt accounts for the long delay in implementing the UNIX operating system on it.

Figure 13 illustrates the growth in memory capacity of successive models of the most widely-sold CPU architecture, that used in the IBM PC. Figure 14 shows the increasing memory capacity of IBM processors, starting from the early days of digital computers. Memory growth in Cray supercomputers is included in Figure 17 on page 37. Together, the architecture families of these three vendors span much of the history, and most of the performance

Processor Family	Year of Introduction	Address Bits	Memory Size (bytes)	
702	1953	?? (decimal)	10,000	(144K)
704	1954	15 (word)	147,456	(144K)
705-I	1954	?? (decimal)	20,000	(19K)
705-II	1956	?? (decimal)	40,000	(39K)
1460	1963	?? (decimal)	16,000	(15K)
7010	1962	?? (decimal)	100,000	(97K)
7080	1960	?? (decimal)	160,000	(156K)
7074	1960	?? (decimal)	300,000	(292K)
7094	1963	15 (word)	147,456	(144K)
7030 Stretch	1961	24	2,097,152	(2M)
System/360	1964	24	16,777,216	(16M)
System/370	1971	24	16,777,216	(16M)
801	1978	32	4,294,967,296	(4G)
System/370-XA	1981	31	2,147,483,648	(2G)
ESA/370	1988	44	17,592,186,044,416	(16T)
RS/6000	1990	52	4,503,599,627,370,496	(4P)

Figure 14: **IBM processor memory sizes.** The IBM 7030 Stretch was unusual in that it used *bit addressing*; 2^{24} bits is equivalent to 2^{21} (2M) bytes.

range, of computing.

A 64-bit address first became available in 1993 from DEC with the Alpha, and from Silicon Graphics with the MIPS R4000. IBM, Hewlett-Packard, and Sun are expected to have 64-bit addressing about 1995 or 1996, with Intel following somewhat later. The 64-bit address space is so vast that if you were to start writing to it at the substantial rate of 100 MB/sec, it would take 5850 years to fill memory just once!

Of course, with 1994 memory prices of about US$40/MB, it would also cost US$704 trillion to buy a full 64-bit memory; that number is very much larger than the US national debt. By the time you retire, computer memories of this size may just be affordable.

Larger address spaces will significantly impact program design. In particular, it will be possible for even large files to be memory-mapped, and therefore to appear to be byte-addressable, like an array, without any performance hit from I/O, which requires system calls, context switches, real-time waits, and data movement.

> The 960 design goal was a worst-case interrupt latency on the order of 5 μs. Unfortunately, the floating-point remainder instruction, for example, takes up to 75,878 clock cycles to complete (honest!). Even at 20 MHz, that's nearly 4 ms.
>
> John H. Wharton
> 1992 [85, p. 235]

> A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, AMI 6502, Zilog Z80, CRAY-1, and CRAY X-MP.
>
> David A. Patterson and John L. Hennessy
> 1990 [73, p. 481]

3.4.4 Segmented memory

Since the earliest machines in the 1940s, most computer architectures have been designed with a linear address space, with the number of bits in an address constrained by the machine word size. Thus, several 36-bit machines used an 18-bit word address (262,144 words, or slightly more than 1 MB), and most 32-bit machines designed since the DEC VAX was introduced in 1978 have a 32-bit byte address, supporting 4,294,967,296 bytes (4 GB).

One important architecture, the Intel x86 processor family used in the IBM PC, has not followed this model. Instead, it uses *segmented memory*. In the 8086, 8088, and 80186, the 16-bit architecture supports a 20-bit memory address, capable of addressing 1 MB. An address is formed by left-shifting a 16-bit *segment register* by 4 bits, and adding it with unsigned arithmetic to a 16-bit *offset register*. In the 80286, by some complicated tricks, the address space was extended to 24 bits, supporting 16 MB of addressable memory, the same as the IBM 360 mainframe architecture provides. The 80286 also introduced a complicated memory protection mechanism.

> If the 80286 protection model looks harder to build than the VAX model, that's because it is. This effort must be especially frustrating to the 80286 engineers, since most customers just use the 80286 as a fast 8086 and don't exploit the elaborate protection mechanism. Also, the fact that the protection model is a mismatch for the simple paging protection of UNIX means that it will be used only by someone writing an operating system specially for this computer.
>
> John L. Hennessy and David A. Patterson
> 1990 [73, p. 448]

In the Intel 80386 and later processors, registers are 32 bits wide, and the non-segmented address size is accordingly extended to provide a 4 GB linear address space. With the addition of segmentation, the address size can be further extended to 44 bits, supporting 16 TB of memory. This has yet to be relevant for any operating system on the IBM PC. Regrettably, the majority of programs and operating systems that run on the IBM PC are designed for memory segmented into 64 KB chunks.

Pitfall: Extending an address space by adding segments on top of a flat address space.

... From the point of view of marketing, adding segments solves the problems of addressing. Unfortunately, there is trouble any time a programming language wants an address that is larger than one segment, such as indices of large arrays, unrestricted pointers, or reference parameters. ... In the 1990s, 32-bit addresses will be exhausted, and it will be interesting to see if history will repeat itself on the consequences of going to larger flat addresses versus adding segments.

David A. Patterson and John L. Hennessy
1990 [73, p. 483]

Segmented memory on the IBM PC has caused the industry, programmers, and customers untold grief, and the costs, could they ever be realistically estimated, would likely be in the billions of dollars. Different addressing modes are required to access data in a single 64 KB segment than are needed in a multi-segment space. Also, different procedure call and return instructions are required for single- and multi-segment code.

On the 80286 and earlier systems, loading an array element in a single segment takes one instruction, while loading an array element from a multi-segment array takes about 30 instructions. Using arrays larger than 64 KB is therefore very expensive in both time and space.

To make matters even worse, some compilers offer memory models in which incrementing a pointer (address variable) only increments the offset register, resulting in wrap-around in the same segment instead of correctly advancing into the next segment.

Compiler vendors must support multiple memory models, usually called *tiny*, *small*, *medium*, *compact*, *large*, and *huge*. Six versions of each library must be maintained and provided. Programmers must explicitly select the correct memory model at compilation time, and use *near* (16-bit) or *far* (20-bit, 24-bit, or 32-bit) pointers in their code. It is a common experience for an evolving program to grow from one memory model to another, requiring rewriting of code with new type size declarations. In assembly language, memory-referencing instructions must be modified. Slow processor speed, limited memory, and several years of unreliable compilers (traceable to the

complexities of the Intel architecture) caused many software vendors to program entirely in assembly language, instead of in a more easily maintained and ported high-level language.

Some compilers do not support the compile-time declaration of multi-segment arrays, so they must be allocated dynamically at run-time, and pointers, instead of arrays, must then be used.

In the 80286 and earlier models, the stack size is limited to one 64 KB segment, so programs with large arrays or structures inside procedures must be rewritten to allocate them dynamically.

Because of the different address models and call and return instructions, it is not sufficient for an IBM PC owner to install more physical memory in order to run bigger programs. The programs must be rewritten.

The result of all of this is that porting code developed on other architectures to the IBM PC is usually a very painful task, and sometimes, simply proves to be infeasible. Code developed explicitly for the IBM PC, especially if it uses *near* and *far* data types, is often difficult to port to other systems.

Computer historians will very likely conclude that the IBM/Microsoft choice of the segmented-memory Intel x86 processor family over the linear-address-space Motorola 680x0 family for the IBM PC is one of the greatest mistakes ever made in computer system design.

> Will the considerable protection-engineering effort, which must be borne by each generation of the 80x86 family, be put to good use, and will it prove any safer in practice than a paging system?
>
> David A. Patterson and John L. Hennessy
> 1990 [73, p. 449]

The Intel x86 architecture is not the only one with memory segmentation, but it has certainly been the one occasioning the most suffering.

The Gould, later models of the DEC PDP-10, the Hewlett-Packard PA-RISC, the IBM ESA/370 architecture, and the IBM RS/6000 systems all have segmented memory, but the segment sizes are sufficiently large that their limits are less frequently reached.

It is instructive to compare the approach to segmentation taken by Intel and that used by IBM's ESA/370. Each had the goal of producing upward compatibility with the vendors' previous architectures, the Intel 8008, and the IBM 370 and IBM 370-XA. In order to create a larger address than could be held in a single register, both vendors introduced segment registers.

In the Intel design, segment register names are explicitly encoded into every instruction that references memory, and are thus highly visible.

In the IBM ESA/370 design, a new access register set was added. For each of the 16 32-bit general registers, there is a corresponding access register.

Each access register contains flag bits, and a 13-bit index field. When a certain flag bit is set, extended addressing is called into play, and the index field is used to locate an entry in an address table which is then added to the 31-bit address from the general register, effectively producing a 44-bit address capable of addressing 16 TB, just as with the Intel 80386 and later processors. However, the segmentation is *invisible* in all but the small number of new instructions added in ESA/370 to manipulate the access registers and segment address table.

In the normal single-segment case, including all IBM 370 code that existed before ESA/370, all that needs to be done is to zero all of the access registers before a process begins. All instructions then execute as they used to, since the access register extended addressing bit is never found to be set. In a multi-segment program, compilers can generate additional instructions to set the access registers and address table.

The IBM design is obviously much cleaner than Intel's. Since IBM had a decade of experience with the Intel processors, perhaps this should have been expected.

3.4.5 Memory alignment

For those architectures with word-addressable memory, such as the IBM processors up to the early 1960s, and the CDC and Cray systems since then, there is usually no constraint on the *alignment* of data in memory: all addressable items of necessity lie at word boundaries.

With byte addressing, the situation changes. Memory designs of the IBM 360 processors in the 1960s required that data be aligned at byte addresses that were integral multiples of their natural size. Thus, an 8-byte double-precision floating-point value would have to be stored at a byte address whose lower three bits were zero. On System/360, if a non-aligned value was accessed, then a software interrupt handler was invoked to handle the fixup. The Motorola 68000 processor family also has this alignment requirement.

The reason for memory alignment restrictions is that although memories are byte addressable, they are usually *word aligned*, so two bus accesses would be necessary to load a non-aligned word.

The IBM System/370 architecture removed the memory alignment restriction, and so do the DEC VAX and Intel 80x86 architectures, although they all have a run-time performance hit for non-aligned memory accesses.

RISC architectures restored the alignment restriction, and thus, programs that were written in the 1970s and early 1980s on other machines may fail, or perform unexpectedly poorly, when run on a RISC processor.

Although compilers for high-level languages will ensure that individual variables are always allocated on correct memory boundaries, they *cannot* do so where a statement in the language imposes a particular storage ordering. Examples of such statements are Fortran `COMMON` and `EQUIVALENCE`,

Pascal and PL/I `record`, and C `struct`. Data misalignment can also arise from the passing of arguments of a narrow data type to a routine that declares them to be of a wider type; this is common in Fortran for scratch arrays.

Fortunately, the solution is simple: in memory structuring statements, put the widest variables first. For example, in Fortran `COMMON`, put `COMPLEX` and `DOUBLE PRECISION` values, which take two memory words, before the one-word `INTEGER`, `REAL`, and `LOGICAL` values.

With arrays, you can also choose array sizes to preserve alignment, even if you do not align by type sizes. For example, the Fortran statements

```
integer           ix
double precision  x
common / mydata / ix(4), x(3)
```

produce correct alignment. This technique is less desirable, however, because subsequent changes to array sizes can destroy the alignment.

Porting a program from a wide-word machine like a CDC or Cray to a narrower-word one can also introduce alignment problems where none existed before, when floating-point precision is changed from single to double.

Some compilers have options to warn of alignment problems, or with `record` and `struct` statements, to insert suitable padding to ensure proper alignment. Use them, but be cautious about automatic insertion of padding: it may destroy alignment assumptions elsewhere.

3.5 Virtual memory

Virtual memory is the last level in the memory hierarchy. It is the use of external disk storage, or rarely, large slow RAM, to provide the illusion of a much larger physical memory than really exists. Virtual memory first appeared in 1962, on an English computer called the Atlas.

Virtual memory hardware is designed to map linear blocks of the address space, called *memory pages*, into corresponding blocks of physical memory. Memory pages typically range from 512 bytes (0.5 KB) to 16384 bytes (16 KB) in size; 1 KB and 4 KB are probably the most common sizes in 1994. This mapping usually requires a combination of hardware and software support. The software is called into action when the hardware discovers that the needed address is not yet in physical memory, and issues a *page fault* interrupt.

Virtual memory mapping must happen at *every* memory reference, and the addressing hardware is designed to do this with minimal, or even zero, overhead.

Only when a page fault occurs is significant overhead required: the operating system must then do a context switch to the page fault handler,

which in turn must arrange to read the necessary block from a special area of the disk called the *swap area*. If all of the physical memory is currently in use, before it can read in the required block, it must find a memory page to swap out to disk. Ideally, this should be the page that is least likely to be needed in the near future, but since that is impossible to predict, many virtual memory implementations employ a *least-recently-used paging policy* to select a page that has not been used for some time, and then cross their fingers and hope that it probably won't be needed again soon.

Once the memory page has been fetched, another context switch restores control to the faulting process, and execution continues.

Besides the disk I/O in the page fault handler, and the table searches needed to find candidate pages when swapping out is required, the context switches are also relatively expensive, since they may involve expensive transfers between user and kernel privilege modes, and saving and restoring of all registers and any other relevant state information.

Because of the significant operating system complication, and the need for virtual addressing hardware, virtual memory is not supported on most personal computer operating systems, or by the processors used in them.

In the IBM PC family, virtual memory is not supported by DOS, or by processors prior to the Intel 80286. Rudimentary virtual memory support is provided by the OS/2 and Windows 3.1 operating systems.

In the Apple Macintosh family, virtual memory was not supported until version 7 of the Macintosh operating system, released in 1993, nine years after the Macintosh was first announced. It requires the Motorola 68020 processor with a memory-management unit (MMU) chip, or the Motorola 68030 or later processor, and it must be explicitly enabled by the user.

By contrast, all mainframe computers introduced since the mid 1970s, and all UNIX workstations, provide virtual memory. Since all current workstation models are based on processors that support 32-bit (or larger) address sizes, there is no reason, other than cost, why large virtual memory spaces cannot be routinely provided.[5]

Cray supercomputers are an exception to this practice. For performance reasons, they have *never* offered virtual memory, and they use faster static RAMs instead of DRAMs for main memory, so they don't provide cache memory either.

[5] With the introduction of 4 GB disks by DEC in 1993, with a surprisingly low cost, under US$3000, it is now economically feasible to configure a UNIX workstation with a completely accessible 32-bit (4 GB) address space. One vendor, Seagate, has announced 9 GB disks for the fall of 1994.

4 Memory bottlenecks

We have now completed a survey of the computer memory hierarchy, and presented data that allows quantitative comparisons of the relative performance of each of the levels.

The main conclusion that we have to offer is this: **always use the fastest possible level in the memory hierarchy**. With the fastest processors running with 5 ns cycle times, SRAM speeds of 15 ns, DRAM speeds of 70 ns to 80 ns, disk latency times of 10,000,000 ns, and disk transfer times of 10 MB/sec, or 400 ns per 32-bit word, there is an enormous range of memory performance. If data has to be fetched from cache memory, rather than from a register, the computer will run 3 to 4 times slower. If data comes instead from regular memory, the computer will run 15 to 20 times slower. If the data has to be fetched from disk, millions of instruction cycles will be lost.

4.1 Cache conflicts

If data comes from cache memory, there is the possibility that a *cache miss* will require reloading the cache from main memory, causing further loss of cycles. In such a case, it is likely that a word in cache will have to be replaced, possibly writing it back to memory before doing so. If that word is the one that will be needed by the next instruction to be executed, we have a case of a *cache conflict*. When does this happen? It happens when the two words map to the same cache location, which means that with a *direct-mapped cache*, their addresses are the same, modulo the cache size.

For a specific example, consider the MIPS R2000 CPU used in the DECstation 3100 workstation. The R2000 supports cache sizes from 4 KB to 64 KB, so let us assume the minimum 4 KB cache. The R2000 loads only one 32-bit word from memory on a cache miss. Thus, on this processor, referring to array elements that are 4096 bytes apart will cause cache conflicts. This is surprisingly easy, as the following Fortran code demonstrates:

```
      integer k
      real x(1024,32), y(1024)
      ...
      do 20 i = 1,1024
         y(i) = 0.0
         do 10 k = 1,31
             y(i) = y(i) + x(i,k)*x(i,k+1)
10       continue
20    continue
```

In this code, *every* iteration of the inner loop will cause a cache conflict, because the dot product needs elements that are 4096 bytes apart.

Fortunately, the code can be easily modified to reduce the frequency of cache conflicts: just change the array dimensions from 1024 to 1025 or more. The wasted space is worth the gain in performance.

Although you might expect that cache conflicts of this severity might be rare, this is not the case. Mathematical models based on grid subdivisions, such as finite element methods for solving differential equations, generally result in array sizes that are powers of 2, and it is just those powers that are likely to coincide with cache sizes which are also powers of two.

Several months ago, we were using such a program to benchmark several different computer systems to find which would be most suitable for a planned large computational effort. This program had two-dimensional arrays with 256 rows, and unexpectedly poor performance on some systems suggested that we look for cache conflicts. It turned out that changing a *single* character in the 5700-line program, the 6 in 256 to a 7, had the dramatic effect shown in Figure 15.

Machine	**O/S**	**Speedup (257 vs. 256)**
DEC Alpha 3000/500X	OSF 1.2	1.029
HP 9000/735	HP-UX 9.0	1.381
IBM 3090/600S-VF	AIX 2.1 MP370	0.952
IBM RS/6000-560	AIX 3.3	1.913
IBM RS/6000-580	AIX 3.2	2.435
SGI Indigo R4000	IRIX 4.0.5f	**4.376**
SGI Indy SC R4000	IRIX 5.1	3.678
Sun SPARCstation 10/30	SunOS 4.1.3	0.959
Sun SPARCstation 10/41	Solaris 2.1	0.998

Figure 15: **Cache conflict performance impact.** The last column shows the speedup obtained by increasing array dimensions from 256 to 257; this required only a single-character change in a PARAMETER statement.
The IBM 3090 vector processor shows a slowdown because increasing the array dimension required one extra vector iteration for each loop; with 256 elements, 8 32-element vectors span the loop, but 257 elements require 9 vectors.
It is unclear why the Sun SPARCstation 10/30 ran 4% slower; the problem size changed by only 0.5%.

It is good programming practice to keep declared array sizes distinct from the problem size. In Fortran, this means that two-dimensional arrays should be used like this:

```
subroutine sample (a,maxrow,maxcol, nrow,ncol, ...)
integer i, j, maxrow, maxcol, nrow, ncol
real    a(maxrow,maxcol)
```

```
...
do 20 j = 1,ncol
   do 10 i = 1,nrow
...
```

where $1 \le$ `nrow` $\le$ `maxrow` and $1 \le$ `ncol` $\le$ `maxcol`, rather than like this:

```
subroutine sample (a,nrow,ncol, ...)
integer i, j, nrow, ncol
real    a(nrow,ncol)
...
do 20 j = 1,ncol
   do 10 i = 1,nrow
...
```

Although this saves two arguments, and about as many microseconds per call, it is much less flexible than the first style, which separates the current problem size from the declared size, and facilitates adjustment of array dimensions to minimize cache conflicts.

Some programmers choose a style halfway between these two. Since the last dimension of a Fortran array is not required for address computations, they omit `maxcol` from the argument list, and use a constant value of 1 (Fortran 66), or an asterisk (Fortran 77), for the last dimension in the type declaration. While this saves one argument, it has the significant disadvantage that run-time subscript checking is then not possible, making debugging difficult.

4.2 Memory bank conflicts

DRAMs have a peculiar property that their *access time*, the time between the issuance of a read request and the time the data is delivered, is noticeably shorter than their *cycle time*, the minimum time between requests. Figure 16 gives some typical performance characteristics.

The access and cycle time difference means that in order to avoid delays in accessing memory repeatedly, memory is often *interleaved* into separate *memory banks*. Thus, in a design with four memory banks, words n, $n+4$, $\ldots$, $n+4k$ come from the first bank, words $n+1$, $n+1+4$, $\ldots$, $n+1+4k$ from the second bank, and so on. Each bank is independent, so that each can deliver data simultaneously. Thus, typical program code that accesses array data, and loads of a cache line, will not encounter additional cycle-time delays.

What does this mean for programmers? Well, if the program accesses data in the same memory bank, instead of data in interleaved banks, there will be a slowdown. For ease of bank determination, the number of banks is generally a power of two, as illustrated in Figure 17, so just as in the Fortran

Year of Introduction	Chip Size	Row access Slowest DRAM	Row access Fastest DRAM	Column Access	Cycle Time
1980	64 Kb	180 ns	150 ns	75 ns	250 ns
1983	256 Kb	150 ns	120 ns	50 ms	230 ns
1986	1 Mb	120 ns	100 ns	25 ms	190 ns
1989	4 Mb	100 ns	80 ns	20 ms	165 ns
1992	16 Mb	≈85 ns	≈65 ns	≈15 ns	≈140 ns

Figure 16: **Generations of fast and slow DRAM time.** The 1992 figures are estimates. Memory is organized as a rectangular matrix, and a complete memory request takes both a row and a column access. The data is taken from [73, Fig. 8.17, p. 426].

code example above for cache conflicts, there will be memory bank delays when the addresses of successive array elements differ by a multiple of the bank size. The same programming technique of choosing array dimensions which are not powers of two removes the delays.

4.3 Virtual memory page thrashing

We described in Section 3.5 how virtual memory works. A program's *working set* is the collection of data it needs over a time interval measured in seconds.

In the ideal case, the available physical memory is sufficient to hold the working set, and the first reference to each memory page in the swap area on disk will result in its being mapped into physical memory, so that no further swap activity will be needed until the working set changes.

In the worst case, *every* memory reference will reference data which is not yet in physical memory, and as Figure 5 on page 6 showed, delays of many thousands, or even millions, of cycles can result. When a process initiates disk I/O, most operating systems will suspend it in order to run another process that has been waiting for the CPU to become available. This means that a page swap can also require four, or more context switches, as control passes from the first process, to the kernel, to the new process, and then back again. The context switch changes the demands on memory, and additional virtual memory paging can be required. This is the main reason why Cray supercomputers do not provide virtual memory: it is simply impossible to make performance guarantees when a process' memory space is not entirely resident in physical memory.

A simple Fortran code segment for matrix multiplication illustrates the potential problem:

Machine	Year of Introduction	CPUs	Clock Speed (MHz)	Cycle Time (ns)	Memory Size (bytes)	Memory Bus Width (bits)	Memory Banks
Convex C-1	1985	1	10	100	1 GB	72	64
Convex C-2	1988	1–4	25	40	4 GB	72	64
Convex C-3	1991	1–8	60	16.67	4 GB	72	256
Cray 1	1976	1	80	12.5	8 MB	64	16
Cray 1/S	1979	1	80	12.5	32 MB	64	16
Cray X-MP	1983	1–4	105	9.5	64 MB	??	64
Cray X-MP	1986	1–4	118	8.5	64 MB	??	64
Cray Y-MP	1988	1–8	167	6.0	256 MB	??	256
Cray 2	1985	1–4	244	4.1	2 GB	??	128
Cray C-90	1991	1–16	238.1	4.2	4 GB	??	??
Cray 3	1993	1–16	480	2.08	4 GB	??	256 or 512
Cray 4	1995	1–64	1000	1.0	??	??	??
DEC Alpha 3000-400	1993	1	133	7.5	16 EB	256	??
DEC Alpha 3000-500	1993	1	150	6.7	16 EB	256	??
DEC Alpha 7000-600	1993	1	182	5.5	16 EB	256	??
DEC Alpha 10000-600	1993	1	200	5.0	16 EB	256	??
Hitachi S-810	1983	1	35.7 (s) 71.4 (v)	28.0 14.0	??	??	??
Hitachi S-820/80	1987	1	125 (s) 250 (v)	8.0 4.0	512 MB	512	??
IBM 3090S-VF	1987	1–6	66.7	15.0	2 GB	64	16
IBM ES/3090S-VF	1989	1–6	66.7	15.0	16 TB	64	??
IBM RS/6000-320	1990	1	20	50.0	256 MB	64	4
IBM RS/6000-530	1990	1	25	40.0	512 MB	128	4
IBM RS/6000-540	1990	1	30	33.3	256 MB	128	4
IBM RS/6000-580	1992	1	62.5	16.0	1 GB	128	4
IBM RS/6000-590	1993	1	66.6	15.0	2 GB	256	4
IBM RS/6000-990	1993	1	71.5	14.0	2 GB	256	4
NEC SX-3/SX-X	1989	1–4	344.8	2.9	1 GB	??	1024

Figure 17: **Memory interleaving on various computers.** Notice the bank increase on Cray supercomputers of successively higher performance, as well as the memory size increase. The large numbers of memory banks on the Cray machines after 1979 ensure that 64-element vector register loads can proceed in parallel without memory bank delays.
As chip density increases, interleaving increases memory cost, because each bank must be completely filled, preventing extending memory by small amounts. For example, a 16 MB memory can be built with 512 256 Kb chips in 16 banks, but with 32 4 Mb chips, only one bank is possible.

```
      integer i, j, k, m, n, p
      real a(m,n), b(m,p), c(p,n)
      ...
      do 30 j = 1,n
         do 20 i = 1,m
            a(i,j) = 0.0
            do 10 k = 1,p
               a(i,j) = a(i,j) + b(i,k) * c(k,j)
10          continue
20       continue
30    continue
```

Recall that Fortran stores arrays in memory with the first subscript varying most rapidly. We have arranged the code so that the outer two loops step over the array `a(*,*)` in storage order, and in the inner loop, the array `c(*,*)` is also accessed in storage order. However, the array `b(*,*)` is accessed across the rows, so successively fetched elements are spaced far apart in memory. If too little physical memory is available, every iteration of the inner loop could cause a page fault, so the worst case is `n`$\times$`m`$\times$`p` page faults.

Because the innermost statement is independent of the outer loop order, provided we handle the zeroing of `a(*,*)` properly in advance, we can reorder the loops six different ways: `i-j-k`, `i-k-j`, `j-k-i`, `j-i-k`, `k-j-i`, or `k-i-j`. No matter which of these we choose, at least one of the arrays will be accessed sub-optimally in row order in the inner loop. The optimal ordering depends on the machine architecture, and also on the compiler.

What is the solution? Assume that `m`, `n`, and `p` are all of order N, so that the matrix multiplication requires N^3 multiplications and $N^2(N-1)$ additions, or approximately $2N^3$ flops in all. If we transpose the `b(*,*)` matrix in a separate pair of loops before the matrix multiplication triply-nested loop, we can accomplish the transposition in $\mathcal{O}(N^2)$ iterations, some of which will surely generate page faults. The transposition is trivial to program if the array is square, and more complex algorithms exist for rectangular array transposition in place [5, 14, 16, 26, 56]. However, the $\mathcal{O}(N^3)$ operations of the matrix multiplication will run with minimal, or no, page faulting in the inner loops, depending on the length of the array columns. On completion of the matrix multiplication, the transposed matrix can be transposed a second time to restore it to its original form.

Transposition has a side benefit: on systems with caches, loading of a cache line will move into the cache data that will be required on the subsequent iterations.

4.4 Registers and data

Since registers are the fastest memory technology, we want frequently-used values kept in registers, rather than in cache or main memory. Except for assembly languages, however, most programming languages provide no way to designate which variables are to reside in registers. Even in the C programming language, which *does* have register attributes in type declarations, there is no guarantee that the compiler will follow the programmer's recommendation; indeed, it can ignore register declarations completely.

Instead, we must rely on the compiler to do a good job of assigning local values to registers for the duration of long-running code sequences, such as deeply-nested inner loops.

However, if the architecture is register poor, such as in the Intel x86 family, there is little the compiler can do. RISC machines with their large register sets fare considerably better here, and on today's RISC workstations, good compilers will place most local variables in registers if high optimization levels are selected. They will usually *not* do so if no optimization, or debug compilation, is selected.

Since no optimization is usually the default, the programmer must take care to explicitly request optimization. Regrettably, with most UNIX compilers, this inhibits the output of a symbol table, making debugging difficult. Thus, once a program has been debugged, it is important to recompile it with the highest optimization level, and of course, to thoroughly test it again, since optimization is the area of compilers where the most bugs tend to lurk.

Register assignment is not possible for values whose address is required in the code, such as with the C language & (address-of) operator. If the variable is passed as a procedure-call argument, it will have to be stored into memory, and then retrieved into a register on return.

If a value is used only once, little good is done by keeping it in a register. In the matrix multiplication example above, the innermost statement

```
a(i,j) = a(i,j) + b(i,k) * c(k,j)
```

achieves three reuses of `i`, three of `j`, and two of `k`, but none of the array elements themselves. Since the 1960s, most optimizing compilers have been programmed to recognize loop indices as prime candidates for register residency, and consequently, under optimization, the loop indices probably would not be materialized in memory, and substantial parts of the address computations would be moved outside the innermost loop. Indeed, `a(i,j)` would very likely be assigned to a register for the duration of the innermost loop, since both subscripts are constant in the loop.

One technique for performance improvement of array algorithms that was popularized in the 1970s is *loop unrolling*. In the matrix multiplication

example, the inner loop would be rewritten something like this, assuming prior transposition of the b(*,*) array.

```
      do 10 k = 1,(p/5)*5,5
         a(i,j) = a(i,j) + b(k  ,i) * c(k  ,j)
     x                   + b(k+1,i) * c(k+1,j)
     x                   + b(k+2,i) * c(k+2,j)
     x                   + b(k+3,i) * c(k+3,j)
     x                   + b(k+4,i) * c(k+4,j)
10    continue
      do 15 k = (p/5)*5 + 1,p
         a(i,j) = a(i,j) + b(k,i) * c(k,j)
15    continue
```

The second loop is a cleanup loop to handle the cases where p is not a multiple of 5. The first loop iterates only one-fifth as many times as the original loop, reducing the ratio of *loop overhead* (incrementing counters, array addressing, and loop counter testing and branching) to real work (the floating-point arithmetic). It also meshes well with cache-line fetches. Obviously, unrolling complicates the coding, and makes it easy to introduce bugs. Unrolling often gives a performance improvement of several percent. By the late 1980s, many optimizing compilers had been developed that could unroll the innermost loop automatically, avoiding the need for a human programmer to do the work.

Unfortunately, linear loop unrolling does nothing for register data reuse. Each of the array elements is fetched from memory, and used only once.

The solution to this dilemma was apparently first observed by Ron Bell, a researcher at IBM Austin laboratories where the IBM RS/6000 workstation was developed, and published in a technical report in August 1990 [13].

Bell's key observation is that matrix multiplication does not decree any special order of computation; indeed, with $m \times n$ parallel processors, each element of a(*,*) can be computed independently of all the others.

Bell therefore suggested computing a $q \times q$ block of the product simultaneously in the inner loop. The code to do this is quite complicated because of the need to have cleanup loops for each loop index, but we can see the general pattern by examining the statements in the inner loop for the case $q = 3$:

```
a(i+0,j+0) = a(i+0,j+0) + b(i+0,k) * c(k,j+0)
a(i+0,j+1) = a(i+0,j+1) + b(i+0,k) * c(k,j+1)
a(i+0,j+2) = a(i+0,j+2) + b(i+0,k) * c(k,j+2)

a(i+1,j+0) = a(i+1,j+0) + b(i+1,k) * c(k,j+0)
a(i+1,j+1) = a(i+1,j+1) + b(i+1,k) * c(k,j+1)
a(i+1,j+2) = a(i+1,j+2) + b(i+1,k) * c(k,j+2)
```

```
a(i+2,j+0) = a(i+2,j+0) + b(i+2,k) * c(k,j+0)
a(i+2,j+1) = a(i+2,j+1) + b(i+2,k) * c(k,j+1)
a(i+2,j+2) = a(i+2,j+2) + b(i+2,k) * c(k,j+2)
```

Notice the data usage patterns: each array element from the `b(*,*)` and `c(*,*)` arrays is used q times in the computation of q^2 elements of `a(*,*)`. If a compiler is sufficiently clever, it will recognize this, and only load the element once. The total number of data values in the inner loop is q^2 from `a(*,*)`, and q from each of `b(*,*)` and `c(*,*)`, or $q^2 + 2q$ in all; Figure 18 tabulates several values of this simple expression.

q	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
q^2+2q	3	8	15	24	35	48	63	80	99	120	143	168	195	224	255

Figure 18: **Inner loop data value counts in $q \times q$ unrolling.** The second row is the number of registers needed to hold all data values. Each value will be re-used q times.

The numbers in the second column of Figure 18 should be compared with the register counts for various architectures in Figure 11 on page 17.

It appears that with its 32 floating-point registers, the IBM RS/6000 should be able to handle the case of $q = 4$; with $q = 5$, there is a shortage of 3 registers, so some registers would have to be reused, and some data values would have to be loaded more than once. Thus, one would predict that $q = 4$ should be the optimal size for $q \times q$ loop unrolling on this architecture.

This turns out to be the case, as shown in Figure 19. Indeed, the improvement is dramatic. Without optimization, the straightforward code with the normal inner loop and no optimization reaches only 1.24 Mflops, while 4×4 unrolling with optimization produces 44.22 Mflops, an improvement by a factor of 35.7.

For the RS/6000 POWER architecture, the peak speed in Mflops is exactly twice the clock rate in MHz. Thus, the 25 MHz clock of the model 530 could potentially produce 50 Mflops. The 4×4 unrolling technique has achieved 88.4% of that result, which is astonishingly good. For comparison, Figure 20 shows the performance of the well-known LINPACK linear-equation solver benchmark on several supercomputers.

The last column of Figure 19 demonstrates how important code optimization is on RISC architectures; similar speedups have been obtained on RISC machines from other vendors.

Since these results were obtained in the fall of 1990, it has since been demonstrated that similar block unrolling techniques can produce comparable speedups in Gaussian elimination as well.

IBM RS/6000-530 DOUBLE PRECISION 100×100 Matrix Multiplication			
Loop type	**no-opt Mflops**	**opt Mflops**	**opt/ no-opt**
	$A = BC$		
normal	1.24	18.09	14.6
unrolled	2.46	16.25	6.6
2×2	2.21	30.61	13.9
3×3	3.09	33.17	10.7
4×4	3.92	44.22	11.3
5×5	4.55	31.84	7.0
	$A = B^T C$		
normal	1.27	18.09	14.2
unrolled	2.75	15.92	5.8
2×2	2.15	28.43	13.2
3×3	3.09	33.17	10.7
4×4	3.90	39.80	10.2
5×5	4.60	33.17	7.2

Figure 19: **IBM RS/6000-530 matrix multiplication performance.** The *normal* loop has one multiplication and addition. The *unrolled* loop uses five-fold linear unrolling. The second half of the table shows results for transposed B. The data is taken from [9].

The measurements in Figure 19 are for 100×100 matrices. The three matrices, each 80 KB in size in double-precision arithmetic, require 240 KB of memory, but the IBM RS/6000-530 has only 64 KB of cache. However, computation of one column of $\boldsymbol{A}$ requires all of $\boldsymbol{B}$ and only one column of $\boldsymbol{C}$, so the main demand on cache is from only one of the matrices. This suggests that larger matrices might show the effect of cache overflow, and this is indeed the case, as can be seen in Figure 21.

Notice the drastic plunge in performance for the normal and unrolled cases when the matrix size increases beyond 100×100; this happens because the matrices can no longer fit in cache memory. One way to fix this performance problem would be to introduce *strip mining* of the outer loops, that is, to rewrite them as multiple loops over blocks of the matrices, with block sizes chosen to maximize cache usage. This is exceedingly tedious to program correctly, however. Regrettably, very few compilers are capable of doing it automatically.

Fortunately, the technique of $q \times q$ unrolling, combined with pre- and post- matrix transposition, is capable of maintaining high performance lev-

Machine (1 processor)	100 × 100 Fortran Mflops	% Peak	1000 × 1000 Assembly Mflops	% Peak	Peak Mflops
Convex C-120	6.5	32%	17	85%	20
Convex C-210	17	34%	44	88%	50
Convex C-3810	44	37%	113	94%	120
Cray 1/S	27	17%	110	69%	160
Cray 2/S	41	8%	384	79%	488
Cray 3	241	25%	n/a	n/a	962
Cray C-90	387	41%	902	95%	952
Cray X-MP/4	66	28%	218	93%	235
Cray Y-MP/8-32	84	25%	308	93%	333
ETA 10E	62	16%	334	88%	381
Fujitsu VP-2100/10	112	22%	445	89%	500
Fujitsu VP-2200/10	127	13%	842	84%	1000
Fujitsu VP-2600/10	249	5%	4009	80%	5000
Hitachi S-820/80	36	4%	n/a	n/a	840
IBM 3090/180VF	12	10%	71	64%	116
IBM ES/9000-320VF	22	24%	91	68%	133
IBM ES/9000-411VF	23	13%	99	54%	182
IBM ES/9000-520VF	60	13%	338	76%	444
IBM ES/9000-711VF	86	15%	422	75%	563
NEC SX-1	36	6%	422	65%	650
NEC SX-2	43	3%	885	68%	1300
NEC SX-3/12	313	11%	2283	83%	2750
Stardent 1510	6.9	43%	13	81%	16

Figure 20: **LINPACK linear-equation solver performance on selected supercomputers, for single processors only.** The Stardent was billed as a 'personal' supercomputer, costing around US$50K–$100K. It has since been substantially outclassed by newer RISC workstations, and the company no longer exists. The Convex is marketed as a 'departmental' supercomputer, with prices in the range of US$0.5M–$2M. The other machines are all in the US$5M–$32M class.
Notice how little of peak performance is attainable for the small matrix, particularly for those machines with the fastest peak performance.
The LINPACK 100×100 benchmark is coded in Fortran, and no source code modifications are permitted. In the Fortran code, the innermost loops are replaced by calls to special subroutines for vector operations. On most supercomputers, this reduces performance because those subroutines have too little work to do, compared to the overhead of invoking them.
The 1000×1000 case can be coded any way the vendor wishes, as long as it gets the correct answer. On some machines, all, or part, of the code is programmed in hand-optimized assembly code.
The data are taken from the LINPACK benchmark report, available as a PostScript file, `performance`, in the `xnetlib` benchmark directory, and via anonymous `ftp` to either of `netlib.ornl.gov` or `netlib.att.com`. The report is updated regularly with new results. The source code can be found in the same places.

IBM RS/6000-530 DOUBLE PRECISION Matrix Multiplication			
		$A = BC$	$A = B^T C$
Loop type	**Size**	**Mflops**	**Mflops**
normal	100	13.27	18.95
	200	2.17	17.35
	300	1.86	17.71
	400	1.69	18.29
	500	1.49	18.28
unrolled	100	14.21	16.58
	200	2.17	15.65
	300	1.87	16.66
	400	1.68	16.77
	500	1.49	17.15
2×2	100	26.53	28.43
	200	5.41	29.56
	300	5.45	30.29
	400	5.30	30.44
	500	4.87	30.51
3×3	100	28.43	33.13
	200	9.56	31.60
	300	10.43	32.77
	400	9.26	32.70
	500	8.40	33.17
4×4	100	39.81	44.18
	200	16.80	40.41
	300	17.25	42.62
	400	15.50	43.34
	500	14.56	43.40
5×5	100	36.19	30.61
	200	19.46	31.29
	300	18.92	31.90
	400	17.19	32.57
	500	16.66	33.06

Figure 21: **IBM RS/6000-530 large matrix multiplication performance.** The precipitous drop in performance for matrices larger than 100×100 is dramatic evidence of the importance of cache memory. The difference between the last two columns shows the benefit of cache line loads and unit stride, not virtual memory paging effects, since the machine had ample memory to hold all of the matrices. The data is taken from [9].

els, and once again, as predicted, $q = 4$ is the optimal choice for this architecture.

The sharp-eyed reader may have noticed that the performance for the 100×100 case in Figure 21 is lower than that given in Figure 19. The reason is that in the larger benchmark, all matrices were dimensioned for the 500×500 case; this leaves memory gaps between successive columns for the smaller matrices, and results in some virtual memory paging degradation.

More details of the matrix multiplication algorithm implementation with case studies for selected architectures can be found in a separate report [9], and in a very large compendium of frequently-updated publicly-accessible results [10]. A study of the effect of the memory hierarchy on matrix multiplication for the IBM 3090 vector supercomputer can be found in [58].

5 Conclusions

This document has provided a broad survey of the computer memory hierarchy and its impact on program performance. There are many lessons to be learned from the data presented here. The most important are:

- Align variables at their natural memory boundaries.
- Choose array dimensions to minimize cache and memory bank conflicts.
- Select the memory spacing of successive array elements to minimize cache and memory bank conflicts, and virtual memory thrashing.
- Pick algorithms that will give maximal data reuse, so that compilers can keep values in registers, avoiding cache and memory altogether.

The first three points were widely known in the scientific computing community by the early 1980s,

The last one is new, and is particularly relevant to RISC architectures because of the processor-memory performance gap illustrated in Figure 1 on page 2. It dates only from the summer of 1990, and it is not yet widely appreciated and understood.

6 Further reading

The alert reader will have noted numerous references to Hennessy and Patterson's book [73]. If you want to understand more about the aspects of computer architecture that affect performance, the best recommendation I

can give is to acquire that book, and study it thoroughly. The noted computer architect who designed the famous DEC PDP-11, PDP-10, and VAX computers said this in the preface to the book:

> The authors have gone beyond the contributions of Thomas to Calculus and Samuelson to Economics. They have provided the definitive text and reference for computer architecture *and design*. To advance computing, I urge publishers to withdraw the scores of books on this topic so a new breed of architect/engineer can quickly emerge. This book won't eliminate the complex and errorful microprocessors from semiconductor companies, but it will hasten the education of engineers who can design better ones.
>
> C. Gordon Bell
> 1990 [73, p. ix]

As this document was going through its final editing stages at the end of January, 1994, I stopped to browse in a bookstore, and discovered a new book [31] by the same authors. According to the book's preface, it is both an update of the earlier volume [73], as well as a revision to make the material more accessible to less advanced readers. It also incorporates additional historical material. I look forward to many pleasant hours of study of this new book.

Wilkes' autobiography [98] and the papers in the collection edited by Randall [76] cover the early days of computing, starting with the ancient Greeks, and continuing up to 1980.

The evolution of IBM computers is well described in several books and journals. The early history of computing machinery, and of the company that became IBM, is treated in a biography of Herman Hollerith [3], the father of the punched card. The decades of the Watson presidencies are covered in Tom Watson Jr.'s very readable autobiography [96]. Bashe et al in a book [7] and an article [6] cover the IBM machines of up to the early 1960s, and Pugh et al in a book [74], and Padegs in an article [70], continue the story with the IBM 360 and early 370 systems. The architecture of the System/370 is described by Case and Padegs [24]. The IBM 370-XA architecture is briefly summarized by Padegs [71]. The March 1986 *IBM Journal of Research and Development*, and the January 1986 *IBM Systems Journal* are special issues devoted to the IBM 3090 and its vector facility. The March 1991 and July 1992 issues of the *IBM Journal of Research and Development* cover the System/390 and ES/9000 architectures. The January 1990 issue of the *IBM Journal of Research and Development* covers the architecture of the IBM RS/6000. Finally, Allen [1] provides a history of IBM language processor technology.

Bell et al [11] describe the architecture of the PDP-10 processor.

The January 1978 issue of the *Communications of the ACM* is devoted to articles about various computer architectures.

The history of Intel Corporation and the development of semiconductor memory and processor technology is summarized in [43].

Organick [69] covers the Intel iAPX 432 from a high-level language programmer's point of view, and Hunter, Ready, and Farquhar provide low-level architectural details [34].

The Intel i860 is presented in two books from Intel [41, 42].

Myers and Budde describe the architecture of the Intel i960 [67], and Intel has produced one volume [48] about the i960.

Intel has published several books on the 80x86 family [35–40, 44–47, 49–52]. Two books [52, 92] cover the Intel Pentium, released in 1993. Pentium is the first superscalar processor in the Intel product line.

There are numerous books on assembly language programming for the Intel 80x86 family: [15, 17–22, 57, 59–62, 68, 72, 79, 80, 84, 99]. Of these, Palmer and Morse's book [72] concentrates exclusively on the floating-point processor. The authors are the chief architects of the 8087 and 8086 processors, respectively, and Palmer was also one of the designers of the IEEE 754 Floating-Point Standard.

The major RISC architectures are described in Slater's book [85]. The book also covers specific chip implementations of several systems, and the individual chapters are written by independent industry analysts who are not above registering criticisms of particular features.

The DEC Alpha architecture is described in two books [29, 83]; the second is by the chief architects of Alpha.

Holt et al [32] cover version 1.0 of the Hewlett-Packard architecture in the HP 9000/8xx machines, but not the more recent version 1.1 architecture of the HP 9000/7xx series.

Kane and Heinrich [53, 54], Brüß [23], and Chow [27] treat the MIPS processors.

The Motorola 88000 RISC architecture is described in [66]. The Motorola 680x0 CISC architecture family is treated in several books [63–65].

The SPARC version 8 architecture is covered in three books [25, 87, 89], and the new 64-bit SPARC version 9 architecture in one book [97].

Siewiorek and Koopman, Jr. offer a case study of the Stardent personal supercomputer architecture [81]. Russell describes the architecture of the Cray 1 supercomputer [78], and the Robbins offer a study of the Cray X-MP [77]. Stevens and Sykora provide a short description of the Cray Y-MP [88]. Brief descriptions of Japanese supercomputers are given in articles by Takahashi et al [90] and Uchida et al [91] on the Fujitsu VP-2000 series, by Eoyang et al [30] and Kawabe et al [55] on the Hitachi S-820/80, and by Watanabe et al [94, 95] on the NEC SX-3/SX-X. Simmons et al [82] compare the performance of current Cray, Fujitsu, Hitachi, and NEC supercomputers.

Performance measurements of high-performance computers are always in demand, and performance claims are often inflated. Bailey's humorous note [4] points out some of the ways in which data can be distorted to fool the customer. Bailey also gives some interesting observations of the relative performance of RISC processors and vector supercomputers [5]; he concludes that the two chief areas of concern on RISC machines are the slowness of division, and inadequate memory bandwidth. He also provides recommendations about matrix-transposition algorithms that maximize cache usage.

Baskett and Hennessy [8] provide a very interesting survey of recent progress toward obtaining supercomputer performance from parallel machines built from microprocessors, at a fraction of the cost.

If you want to learn more about the evolution and costs of high-performance computing, you should read C. Gorden Bell's *Ultracomputers: A Teraflop Before Its Time* article [12]. Bell gives arguments about why a teraflop computer should not be built before 1995 or 1996, even if it is technologically feasible to do so earlier.

Short biographies of Nobel Prize winners can be found in Wasson's book [93]. Slater [86] provides longer biographies of computer pioneers, and therein, histories of most of the major computer companies.

References

[1] F. E. Allen. The history of language processor technology in IBM. *IBM J. Res. Develop.*, 25(5):535–548, September 1981.

[2] Anonymous. The Datamation 100: The new IT industry takes shape. *Datamation*, 39(12):12–128, 15 June 1993.

[3] Geoffrey D. Austrian. *Herman Hollerith—Forgotten Giant of Information Processing*. Columbia University Press, New York, NY, USA, 1982. ISBN 0-231-05146-8. xvi + 418 pp. LCCN QA76.2.H64 .A97.

[4] David H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, 4(8):54–55, August 1991.

[5] David H. Bailey. RISC microprocessors and scientific computing. In *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 645–654 (of xxii + 935). IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. ISBN 0-8186-4342-0. LCCN QA76.5 .S894 1993.

[6] C. J. Bashe, W. Buchholz, G. V. Hawkins, J. J. Ingram, and N. Rochester. The architecture of IBM's early computers. *IBM J. Res. Develop.*, 25(5):363–375, September 1981.

[7] Charles J. Bashe, Lyle R. Johnson, John H. Palmer, and Emerson W. Pugh. *IBM's Early Computers*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-02225-7. xviii + 716 pp. LCCN QA76.8 .I1015 .I245 1986.

[8] Forest Baskett and John L. Hennessy. Microprocessors–from the desktop to supercomputers. *Science*, 261:846–871, 13 August 1993.

[9] Nelson H. F. Beebe. High-performance matrix multiplication. Technical Report 3, Utah Supercomputing Institute, University of Utah, Salt Lake City, UT 84112, USA, 29 November 1990.

[10] Nelson H. F. Beebe. Matrix multiply benchmarks. Technical report, Center for Scientific Computing, Department of Mathematics, University of Utah, Salt Lake City, UT 84112, USA, November 1990. This report is updated regularly. An Internet e-mail list is maintained for announcements of new releases; to have your name added, send a request to `beebe@math.utah.edu`. The most recent version is available for anonymous `ftp` from the directory `/pub/benchmarks`. It is also accessible from the `tuglib` e-mail server. To get it, send a request with the texts `help` and `send index from benchmarks` to `tuglib@math.utah.edu`.

[11] C. G. Bell, A. Kotok, T. N. Hastings, and R. Hill. The evolution of the DECsystem 10. *Communications of the ACM*, 21(1):44–63, January 1978.

[12] C. Gordon Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[13] Ron Bell. IBM RISC System/6000 performance tuning for numerically intensive Fortran and C programs. Technical Report GG24-3611-00, IBM Corporation, August 1990.

[14] J. Boothroyd. Transpose vector stored array. *Communications of the ACM*, 10 (5):292–293, May 1967.

[15] David J. Bradley. *Assembly Language Programming for the IBM Personal Computer*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1984. ISBN 0-13-049189-6. xii + 340 pp. LCCN QA76.8.I2594 B7 1984. The author is one of the designers of the IBM PC. The book covers the 8088 and 8087 instruction sets, DOS and MASM, the IBM PC hardware, and the ROM BIOS. Somewhat more technical than [57].

[16] N. Brenner. Matrix transposition in place. *Communications of the ACM*, 16(11): 692–694, November 1973.

[17] Barry B. Brey. *The Intel Microprocessors—8086/8088, 80186, 80286, 80386, and 80486—Architecture, Programming, and Interfacing*. Merrill, New York, NY, USA, 1991. ISBN 0675213096. xii + 636 pp. LCCN QA76.8.I292 B75 1991.

[18] Barry B. Brey. *The advanced Intel microprocessors—80286, 80386, and 80486*. Merrill, New York, NY, USA, 1993. ISBN 0023142456. xiv + 745 pp. LCCN QA76.8.I2927 B74 1993.

[19] Barry B. Brey. *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*. Merrill, New York, NY, USA, 1994. ISBN 0023142472. xiv + 457 pp. LCCN QA76.8.I2674B73 1994.

[20] Barry B. Brey. *The Intel Microprocessors—8086/8088, 80186, 80286, 80386, and 80486—Architecture, Programming, and Interfacing*. Merrill, New York, NY, USA, third edition, 1994. ISBN 0023142502. xvii + 813 pp. LCCN QA76.8.I292B75 1994.

[21] Penn Brumm and Don Brumm. *80386/80486 Assembly Language Programming*. Windcrest/McGraw-Hill, Blue Ridge Summit, PA, USA, 1993. ISBN 0-8306-4099-1 (hardcover), 0-8306-4100-9 (paperback). xiii + 589 pp. LCCN QA76.8.I2928 B77 1993. US$29.45 (paperback), US$39.95 (hardcover).

[22] Penn Brumm, Don Brumm, and Leo J. Scanlon. *80486 Programming*. Windcrest/McGraw-Hill, Blue Ridge Summit, PA, USA, 1991. ISBN 0-8306-7577-9 (hardcover), 0-8306-3577-7 (paperback). xii + 496 pp. LCCN QA76.8.I2693 B78 1991. US$36.95 (hardcover), US$26.45 (paperback).

[23] Rolf-Jürgen Brüß. *RISC—The MIPS-R3000 Family*. Siemens Aktiengesellschaft, Berlin and Munich, Germany, 1991. ISBN 3-8009-4103-1. 340 pp. LCCN QA76.5 R48 1991.

[24] Richard P. Case and Andris Padegs. Architecture of the IBM System/370. *Communications of the ACM*, 21(1):73–96, January 1978.

[25] Ben J. Catanzaro, ed. *The SPARC Technical Papers*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1991. ISBN 0-387-97634-5, 3-540-97634-5. xvi + 501 pp. LCCN QA76.9.A73 S65 1991.

[26] E. G. Cate and D. W. Twigg. Analysis of in-situ transposition. *ACM Transactions on Mathematical Software*, 3(1):104–110, March 1977.

[27] Paul Chow, editor. *The MIPS-X RISC Microprocessor*. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1989. ISBN 0-7923-9045-8. xxiv + 231 pp. LCCN QA76.8.M524 M57 1989.

[28] John Cocke and Victoria Markstein. The evolution of RISC technology at IBM. *IBM J. Res. Develop.*, 34(1):4–11, January 1990.

[29] Digital Equipment Corporation. *Alpha Architecture Handbook*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1992.

[30] Christopher Eoyang, Raul H. Mendez, and Olaf M. Lubeck. The birth of the second generation: The Hitachi S-820/80. In *Proceedings, Supercomputing '88: November 14–18, 1988, Orlando, Florida*, pages 296–303. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1988. ISBN 0-8186-0882-X. LCCN QA76.5 S87a 1988.

[31] John L. Hennessy and David A. Patterson. *Computer Organization and Design —The Hardware/Software Interface*. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1994. ISBN 1-55860-281-X. xxiv + 648 pp. LCCN QA76.9 .C643 P37 1994. US$74.75.

[32] Wayne E. Holt (Ed.), Steven M. Cooper, Jason M. Goertz, Scott E. Levine, Joanna L. Mosher, Stanley R. Sieler, Jr., and Jacques Van Damme. *Beyond RISC —An Essential Guide to Hewlett-Packard Precision Architecture*. Software Research Northwest, Inc., 17710 100th Avenue SW, Vashon Island, WA 98070, USA, 1988. ISBN 0-9618813-7-2. xvii + 342 pp. LCCN QA76.8.H66 B49 1988.

[33] Martin E. Hopkins. A perspective on the 801/Reduced Instruction Set Computer. *IBM Systems Journal*, 26(1):107–121, 1987.

[34] Colin B. Hunter, James F. Ready, and Erin Farquhar. *Introduction to the Intel iAPX 432 Architecture*. Reston Publishing Co. Inc., Reston, VA, USA, 1985. ISBN 0-8359-3222-2. vii + 181 pp. LCCN QA76.8.I267 H86 1984. US$16.95.

[35] Intel. *The iAPX 286 Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1983. LCCN QA76.8.I264 I14 1983. The definitive statement of the 80286 and 80287 hardware at a strongly technical level. Not an instruction set reference, but does contain instruction timing tables. See also [37].

[36] Intel. *iAPX 286 Operating Systems Writer's Guide*. Intel Corporation, Santa Clara, CA, USA, 1983.

[37] Intel. *The iAPX 286 Programmer's Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1985. The definitive statement of what the 80286 and 80287 are. A valuable reference for instruction definitions. See also [35].

[38] Intel. *Introduction to the iAPX 286*. Intel Corporation, Santa Clara, CA, USA, 1985.

[39] Intel. *80386 Programmer's Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1986. ISBN 1-55512-022-9. LCCN QA76.8.I2928 E5 1986. The definitive statement of what the 80386 and 80387 are. A valuable reference for instruction definitions. See also [40].

[40] Intel. *80386 Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1987. ISBN 1-55512-069-5. LCCN TK7895.M5 E33 1986. The definitive statement of the 80386 and 80387 hardware at a strongly technical level. Not an instruction set reference, but does contain instruction timing tables. See also [39].

[41] Intel. *i860 64-bit Microprocessor Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1990. ISBN 1-55512-106-3. LCCN TK7895.M5 I57662 1990.

[42] Intel. *i860 64-bit Microprocessor Family Programmer's Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1991. ISBN 1-55512-135-7. LCCN QA76.8.I57 I44 1991.

[43] Intel Corporate Communications. Intel Corporation. In Allen Kent, James G. Williams, and Rosalind Kent, editors, *Encyclopedia of Microcomputers*, volume 9, pages 129–141. Marcel Dekker, Inc., New York, NY, USA and Basel, Switzerland, 1992. ISBN 0-8247-2708-8. LCCN QA76.15 .E52 1988.

[44] Intel Corporation. *iAPX 86/88, 186/188 User's Manual Hardware Reference*. Intel Corporation, Santa Clara, CA, USA, 1985. LCCN TK7889.I6 I52 1985.

[45] Intel Corporation. *80386 Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1986. ISBN 1-55512-069-5. LCCN TK7895.M5 E33 1986.

[46] Intel Corporation. *80286 Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, second edition, 1987. ISBN 1-55512-061-X. LCCN QA76.8.I2927 A18 1987.

[47] Intel Corporation. *386 Microprocessor Hardware Reference Manual*. Intel Corporation, Intel Corporation, 1988. ISBN 1-55512-035-0. 224 pp. LCCN QA76.8.I2927 I67 1988.

[48] Intel Corporation. *80960KB Hardware Designer's Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1988. ISBN 1-55512-033-4. LCCN TK7895.M5 I59 1988.

[49] Intel Corporation. *386 SX Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1990. ISBN 1-55512-105-5. LCCN QA76.8.I2686 I56 1990.

[50] Intel Corporation. *i486 Processor Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1990. ISBN 1-55512-112-8. LCCN TK7836.I57 1990.

[51] Intel Corporation. *Intel386 DX Microprocessor Hardware Reference Manual*. Intel Corporation, Santa Clara, CA, USA, 1991. LCCN QA76.8.I292815I57 1991.

[52] Intel Corporation. *Pentium Processor User's Manual*. Intel Corporation, Santa Clara, CA, USA, 1993.

[53] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1987. ISBN 0-13-584749-4. LCCN QA76.8.M52 K36 1987.

[54] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992. ISBN 0-13-590472-2. LCCN QA76.8.M52 K37 1992.

[55] S. Kawabe, M. Hirai, and S. Goto. Hitachi S-820 supercomputer system. In Raul Mendez, editor, *High-Performance Computing: Research and Practice in Japan*, pages 35–53 (of xvi + 274). John Wiley, New York, NY, USA, 1992. ISBN 0-471-92867-4. LCCN QA76.88.H54 1991. US$68.50.

[56] S. Laflin and M. A. Brebner. In-situ transposition of a rectangular matrix. *Communications of the ACM*, 13(5):324–326, May 1970.

[57] Robert Lafore. *Assembly Language Primer for the IBM PC and XT*. Plume/Waite, New York, NY, USA, 1984. ISBN 0-452-25497-3. viii + 501 pp. LCCN QA76.8.I2594 L34 1984. A companion book for [60], written for novice assembly-language programmers, with considerable emphasis on the IBM PC. More elementary than [15].

[58] Bowen Liu and Nelson Strother. Programming in VS Fortran on the IBM 3090 for maximum vector performance. *Computer*, 21:65–76, 1988.

[59] Yu-Cheng Liu and Glenn A. Gibson. *Microcomputer Systems: The 8086/8088 Family. Architecture, Programming, and Design*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1984. ISBN 0-13-580944-4. ix + 550 pp. LCCN QA76.8.I292 L58 1984. A broad treatment of the Intel 8086 and 8088, with shorter surveys of the 8087, 80186, and 80286. Nothing specific to the IBM PC.

[60] Christopher L. Morgan. *Bluebook of Assembly Routines for the IBM PC and XT*. Plume/Waite, New York, NY, USA, 1984. ISBN 0-452-25497-3. x + 244 pp. LCCN QA76.8.I2594 M64 1984. A handbook collection of many assembly language routines for the IBM PC. The graphics algorithms, particular line-drawing, could be substantially speeded up.

[61] Christopher L. Morgan and Mitchell Waite. *8086/8088 16-Bit Microprocessor Primer*. McGraw-Hill, New York, NY, USA, 1982. ISBN 0-07-043109-4. ix + 355 pp. LCCN QA76.8.I292 M66 1982. A general book on the 8086 and 8088 with shorter treatments of the 8087, 8089, 80186, and 80286, and support chips. Nothing specific to the IBM PC.

[62] Stephen P. Morse and Douglas J. Albert. *The 80286 Architecture*. John Wiley, New York, NY, USA, 1986. ISBN 0-471-83185-9. xiii + 279 pp. LCCN QA76.8.I2927 M67 1986. Morse is the chief architect of the Intel 8086. See also [37].

[63] Motorola. *MC68000 16/32-Bit Microprocessor Programmer's Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, fourth edition, 1984. ISBN 0-13-541400-8. xiii + 218 pp.

[64] Motorola. *MC68020 32-Bit Microprocessor User's Manual*. Motorola Corporation, Phoenix, AZ, USA, second edition, 1985. ISBN 0-13-566878-6.

[65] Motorola. *MC68881 Floating-Point Coprocessor User's Manual*. Motorola Corporation, Phoenix, AZ, USA, second edition, 1985.

[66] Motorola. *MC88100 RISC Microprocessor User's Manual*. Motorola Corporation, Phoenix, AZ, USA, second edition, 1989. ISBN 0-13-567090-X.

[67] Glenford J. Myers and David L. Budde. *The 80960 Microprocessor Architecture*. Wiley-Interscience, New York, NY, USA, 1988. ISBN 0-471-61857-8. xiii + 255 pp. LCCN QA76.8.I29284 M941 1988.

[68] Ross P. Nelson. *Microsoft's 80386/80486 Programming Guide*. Microsoft Press, Bellevue, WA, USA, second edition, 1991. ISBN 1556153430. xiv + 476 pp. LCCN QA76.8.I2684 N45 1991. US$24.95.

[69] Elliott I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, New York, NY, USA, 1983. ISBN 0-07-047719-1. xiii + 418 pp. LCCN QA76.8.I267 O73 1983. US$29.95.

[70] A. Padegs. System/360 and beyond. *IBM J. Res. Develop.*, 25(5):377–390, September 1981.

[71] A. Padegs. System/370 extended architecture: Design considerations. *IBM J. Res. Develop.*, 27(3):198–205, May 1983.

[72] John F. Palmer and Stephen P. Morse. *The 8087 Primer*. Wiley, New York, NY, USA, 1984. ISBN 0-471-87569-4. viii + 182 pp. LCCN QA76.8.I2923 P34 1984. Excellent coverage of the 8087 numeric coprocessor by the chief architects of the Intel 8087 (Palmer) and 8086 (Morse). Contains many candid statements about design decisions in these processors. A must for serious assembly language coding of the 8087 and 80287 chips. See also [37].

[73] David A. Patterson and John L. Hennessy. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1990. ISBN 1-55860-069-8. xxviii + 594 pp. LCCN QA76.9.A73 P377 1990.

[74] Emerson W. Pugh, Lyle R. Johnson, and John H. Palmer. *IBM's 360 and Early 370 Systems*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-16123-0. xx + 819 pp. LCCN QA76.8 .I12 P84 1991.

[75] George Radin. The 801 minicomputer. *IBM J. Res. Develop.*, 27(3):237–246, May 1983.

[76] Brian Randell, editor. *The Origins of Digital Computers—Selected Papers*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1982. ISBN 0-387-06169-X. xvi + 464 pp. LCCN TK7888.3.R36.

[77] Kay A. Robbins and Steven Robbins. *The Cray X-MP/Model 24*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1989. ISBN 0-387-97089-4, 3-540-97089-4. vi + 165 pp. LCCN QA76.8 C72 R63 1989.

[78] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63, January 1978.

[79] Leo J. Scanlon. *8086/88 Assembly Language Programming*. Robert J. Brady Co., Bowie, MD 20715, USA, 1984. ISBN 0-89303-424-X. ix + 213 pp. LCCN QA76.8.I292 S29 1984. A rather short treatment of assembly language programming for the 8088 and 8086, with a short chapter on the 8087. Nothing specific to the IBM PC, and not detailed enough for a beginner. [15] and [57] contain much more detail.

[80] Gary A. Shade. *Engineer's Complete Guide to PC-based Workstations: 80386/80486*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992. ISBN 0-13-249434-5. xviii + 286 pp. LCCN QA76.5.S4428 1992. US$27.00.

[81] Daniel P. Siewiorek and Philip John Koopman, Jr. *The Architecture of Supercomputers—Titan, A Case Study*. Academic Press, New York, NY, USA, 1991. ISBN 0-12-643060-8. xvii + 202 pp. LCCN QA76.5 S536 1991.

[82] Margaret H. Simmons, Harvey J. Wasserman, Olaf M. Lubeck, Christopher Eoyang, Raul Mendez, Hiroo Harada, and Misako Ishiguro. A performance comparison of four supercomputers. *Communications of the ACM*, 35(8):116–124, August 1992.

[83] Richard L. Sites and Richard Witek. *Alpha Architecture Reference Manual*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1992. ISBN 1-55558-098-X. LCCN QA76.9.A73 A46 1992.

[84] Thomas P. Skinner. *An Introduction to 8086/8088 Assembly Language Programming*. Wiley, New York, NY, USA, 1985. ISBN 0-471-80825-3. xiii + 222 pp. LCCN QA76.73.A8 S57 1985. US$17.95.

[85] Michael Slater. *A Guide to RISC microprocessors*. Academic Press, New York, NY, USA, 1992. ISBN 0-12-649140-2. x + 328 pp. LCCN TK7895.M5 G85 1992.

[86] Robert Slater. *Portraits in Silicon*. MIT Press, Cambridge, MA, USA, 1987. ISBN 0-262-19262-4. xiv + 374 pp. LCCN TK7885.2 .S57 1987.

[87] SPARC International, Inc. *The SPARC Architecture Manual—Version 8*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992. ISBN 0-13-825001-4. xxix + 316 pp. LCCN QA76.9.A73 S647 1992.

[88] K. G. Stevens, Jr. and Ron Sykora. The Cray Y-MP: A user's viewpoint. In *Digest of Papers: Intellectual Leverage/Compcon Spring 90, February 26–March 2, 1990, Thirty-third IEEE Computer International Conference, San Francisco. Los Alamitos, CA*, pages 12–15 (of 644). IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990. ISBN 0-8186-2028-5. LCCN QA75.5 C58 1990, TK7885.A1C53 1990.

[89] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA. *The SPARC Architecture Manual*, part no: 800-1399-07 edition, August 8 1987.

[90] M. Takahashi, Y. Oinaga, and K. Uchida. Fujitsu VP2000 series. In Raul Mendez, editor, *High-Performance Computing: Research and Practice in Japan*, pages 7–20 (of xvi + 274). John Wiley, New York, NY, USA, 1992. ISBN 0-471-92867-4. LCCN QA76.88.H54 1991. US$68.50.

[91] N. Uchida, M. Hirai, M. Yoshida, and K. Hotta. Fujitsu VP2000 series. In *Digest of Papers: Intellectual Leverage/Compcon Spring 90, February 26–March 2, 1990, Thirty-third IEEE Computer International Conference, San Francisco. Los Alamitos, CA*, pages 4–11 (of 644). IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990. ISBN 0-8186-2028-5. LCCN QA75.5 C58 1990, TK7885.A1C53 1990.

[92] Peter Varhol. *Pentium, Intel's 64-bit Superscalar Architecture*. Computer Technology Research Corp., 6 N. Atlantic Wharf, Charleston, SC 29401-2150, USA, 1993. ISBN 1-56607-016-3. iv + 170 pp. LCCN QA76.8.P46 V37 1993. US$210.00.

[93] Tyler Wasson. *Nobel Prize Winners*. The H. W. Wilson Company, New York, NY, USA, 1987. ISBN 0-8242-0756-4. xxxiv + 1165 pp. LCCN AS911.N9 N59 1987.

[94] T. Watanabe and A. Iwaga. The NEC SX-3 supercomputer series. In Raul Mendez, editor, *High-Performance Computing: Research and Practice in Japan*, pages 21–33 (of xvi + 274). John Wiley, New York, NY, USA, 1992. ISBN 0-471-92867-4. LCCN QA76.88.H54 1991. US$68.50.

[95] T. Watanabe, H. Matsumoto, and P. D. Tannenbaum. Hardware technology and architecture of the NEC SX-3/SX-X supercomputer system. In *Proceedings, Supercomputing '89: November 13–17, 1989, Reno, Nevada*, pages 842–846 (of xviii + 849). IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1989. ISBN 0-89791-341-8. LCCN QA76.5 S87a 1989.

[96] Thomas J. Watson Jr. *Father Son & Co.—My Life at IBM and Beyond*. Bantam Books, New York, NY, USA, 1990. ISBN 0-553-07011-8. xi + 468 pp. LCCN HD9696.C64 I4887 1990. Memoirs of IBM President Watson, the son of the founder of IBM.

[97] David L. Weaver and Tom Germond. *The SPARC Architecture Manual—Version 9*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994. ISBN 0-13-099227-5. xxii + 357 pp. US$33.00.

[98] Maurice V. Wilkes. *Memoirs of a Computer Pioneer*. MIT Press, Cambridge, MA, USA, 1985. ISBN 0-262-23122-0. viii + 240 pp. LCCN QA76.17 .W55 1985.

[99] Bik Chung Yeung. *8086/8088 Assembly Language Programming*. Wiley, New York, NY, USA, 1984. ISBN 0-471-90463-5. xi + 265 pp. LCCN QA 76.8 I292 Y48 1984. US$23.88.

Index