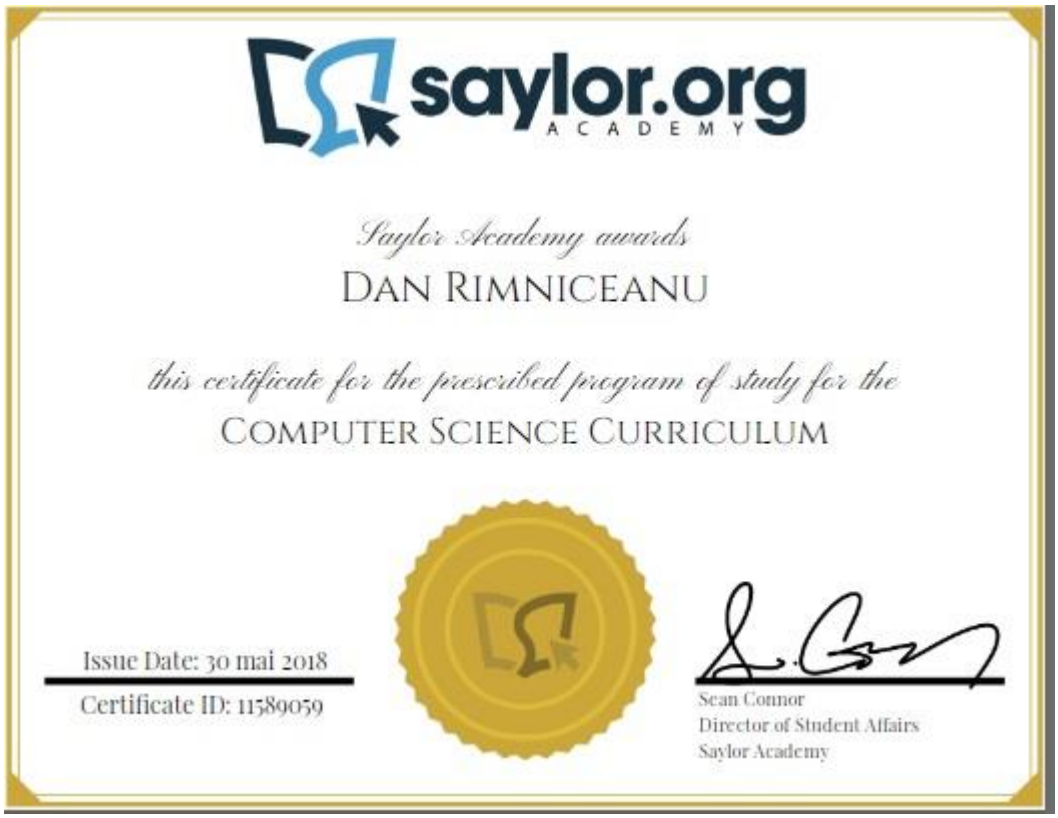


saylor.org
ACADEMY

Saylor Academy awards

DAN RIMNICEANU

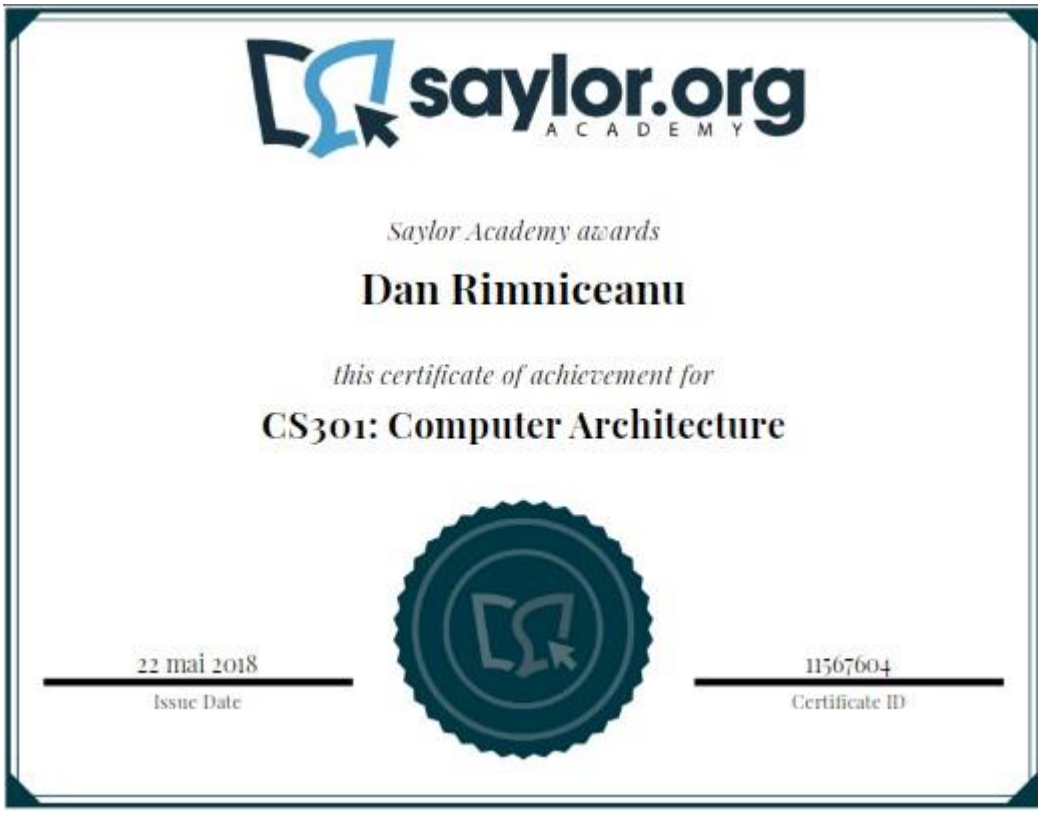
this certificate for the prescribed program of study for the

COMPUTER SCIENCE CURRICULUM

Issue Date: 30 mai 2018

Certificate ID: 11589059

Sean Connor
Director of Student Affairs
Saylor Academy



saylor.org
ACADEMY

Saylor Academy awards

Dan Rimniceanu

this certificate of achievement for

CS301: Computer Architecture

22 mai 2018
Issue Date

11567604
Certificate ID

Modern computer technology requires an understanding of both hardware and software, since the interaction between the two offers a framework for mastering the fundamentals of computing. The purpose of this course is to cultivate an understanding of modern computing technology through an in-depth study of the interface between hardware and software. In this course, you will study the history of modern computing technology before learning about modern computer architecture and a number of its essential features, including instruction sets, processor arithmetic and control, the Von Neumann architecture, pipelining, memory management, storage, and other input/output topics. The course will conclude with a look at the recent switch from sequential processing to parallel processing by looking at the parallel computing models and their programming implications.

Unit 1: Introduction to Computer Technology

In this unit, we will discuss some of the advances in technology that led to the development of modern computers. We will begin our study with a look at the different components of a computer. We will then discuss the ways in which we measure hardware and software performance before discussing the importance of computing power and how it motivated the switch from a single-core to a multi-core processor.

Upon successful completion of this unit, you will be able to:

- list the major advances that have taken place in the history of computer technology;
- list the basic elements in a computer block diagram; and
- identify and discuss some of the latest trends in computing industry.

1.1: Introduction to Computer Processors

Explore each of these articles, which focus on the early history of computers and give some insight into the early history of computers. They will introduce you to the powerful ideas that enabled computer architecture of today and that will influence computer architecture of tomorrow.

Charles Babbage and Howard Aiken

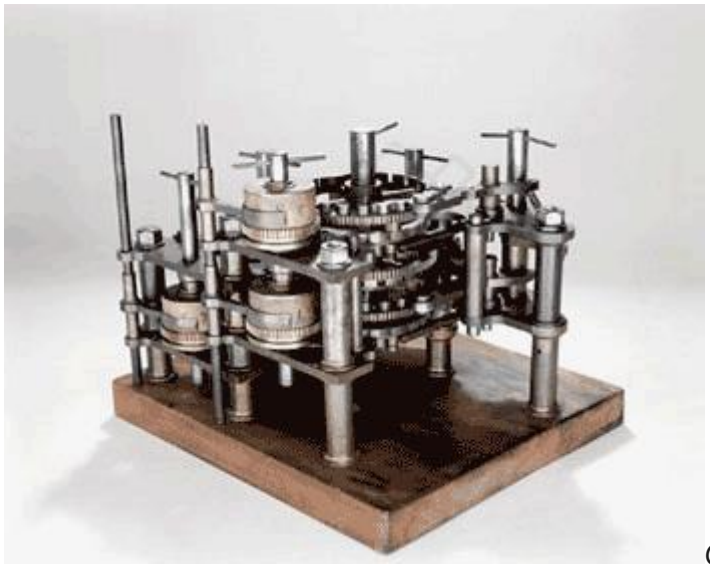
"In 1936, [Howard] Aiken had proposed his idea [to build a giant calculating machine] to the [Harvard University] Physics Department, … He was told by the chairman, Frederick Saunders, that a lab technician, Carmelo Lanza, had told him about a similar contraption already stored up in the Science Center attic.

Intrigued, Aiken had Lanza lead him to the machine, which turned out to be a set of brass wheels from English mathematician and philosopher Charles Babbage's unfinished "analytical engine" from nearly 100 years earlier.

Aiken immediately recognized that he and Babbage had the same mechanism in mind. Fortunately for Aiken, where lack of money and poor materials had left Babbage's dream incomplete, he would have much more success

Later, those brass wheels, along with a set of books that had been given to him by the grandson of Babbage, would occupy a prominent spot in Aiken's office. In an interview with I. Bernard Cohen '37, PhD '47, Victor S. Thomas Professor of the History of Science Emeritus, Aiken pointed to Babbage's books and said, "There's my education in computers, right there; this is the whole thing, everything I took out of a book."

-- The Harvard University Gazette. *Howard Aiken: Makin' a Computer Wonder* By Cassie Furguson

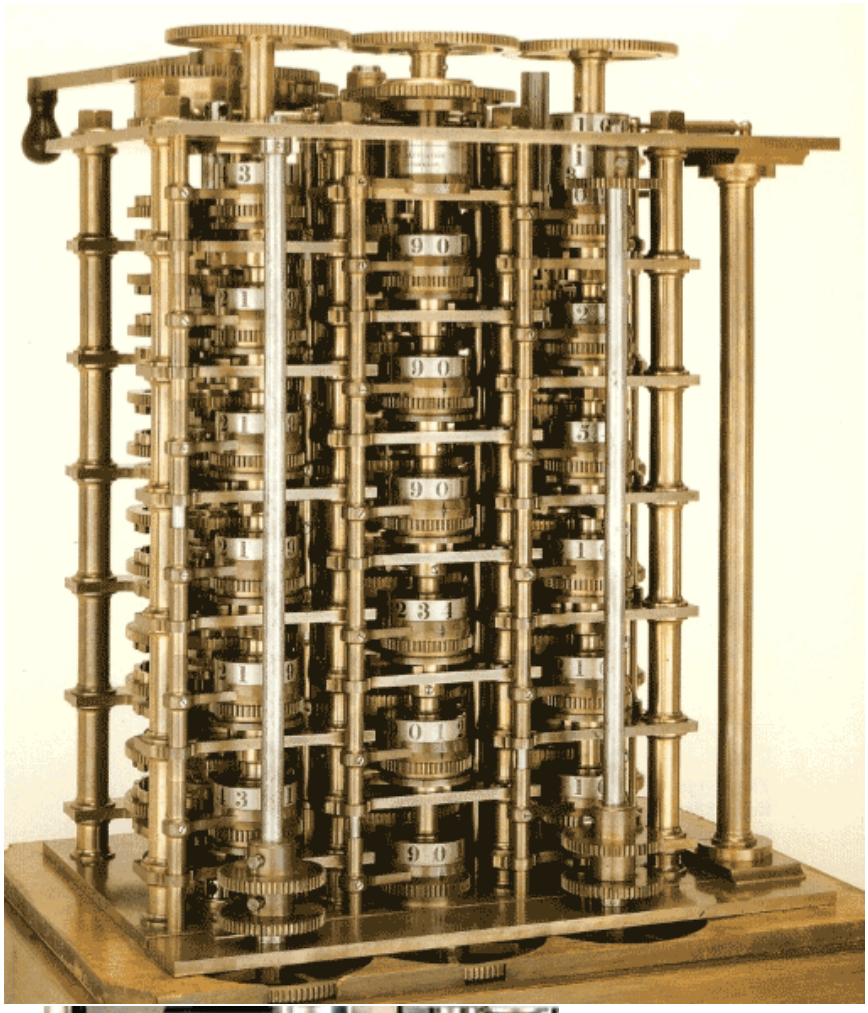


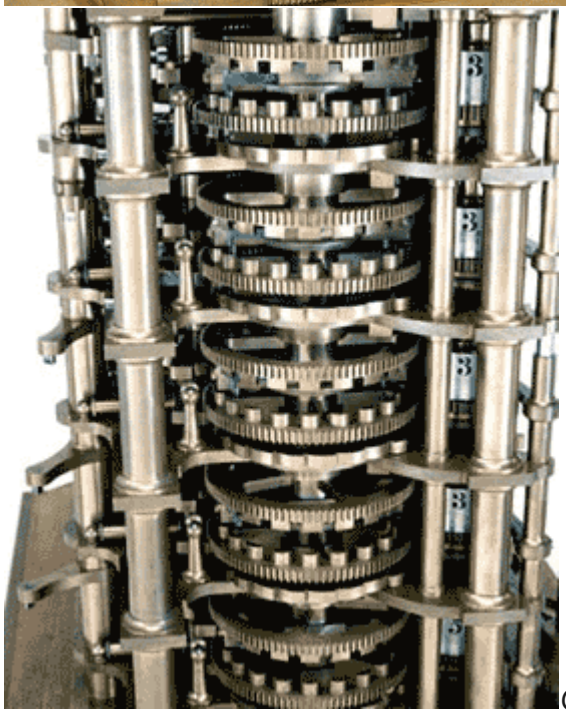
Charles Babbage's Difference Engine I Demo (front)

The quote is incorrect. The "brass wheels" were a small demonstration piece for the Difference Engine I not the Analytical Engine. They were one of six such pieces constructed by Babbage's son Henry after his fathers death. These demonstration pieces were distributed among various universities including Harvard. Aiken must have been sufficiently intrigued by the mechanism to investigate Babbage. In the course of this investigation he would have discovered Babbage's Analytical Engine and the similarities it bore to his own machine. It is not clear when Aiken was given Babbage's "books" or indeed what they contained. They did not contain plans of the Analytical Engine since the only plans have always been stored at the Science Museum at Kensington in London. Aiken may have been able to obtain some documents which together comprise the complete published account of the Analytical Engine. These documents along with Babbage's "books" would have given Aiken a high level description of Babbage's planned machine.

The pictures below show front and rear views of one of six demonstration pieces for the Difference Engine I created by Henry Babbage after his Fathers death. This piece is similar to the one shown to Howard Aiken in 1936.

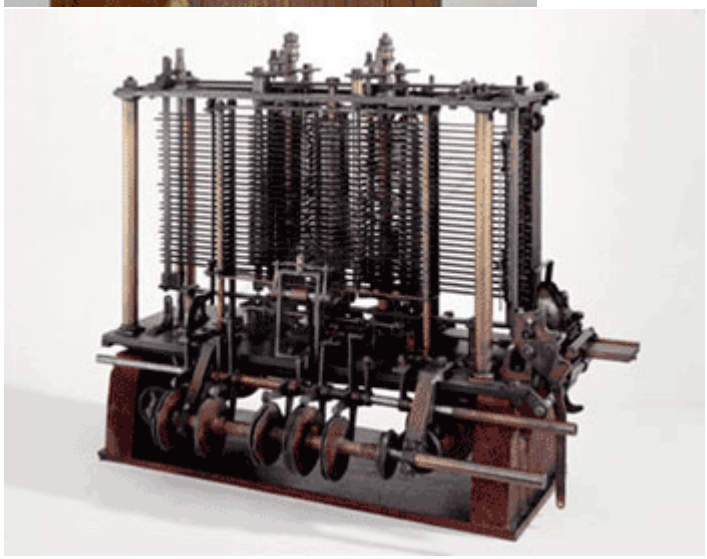
Aiken may also have seen photographs of the largest test piece of the Difference Engine I held by the Science Museum, Kensington, London, UK.

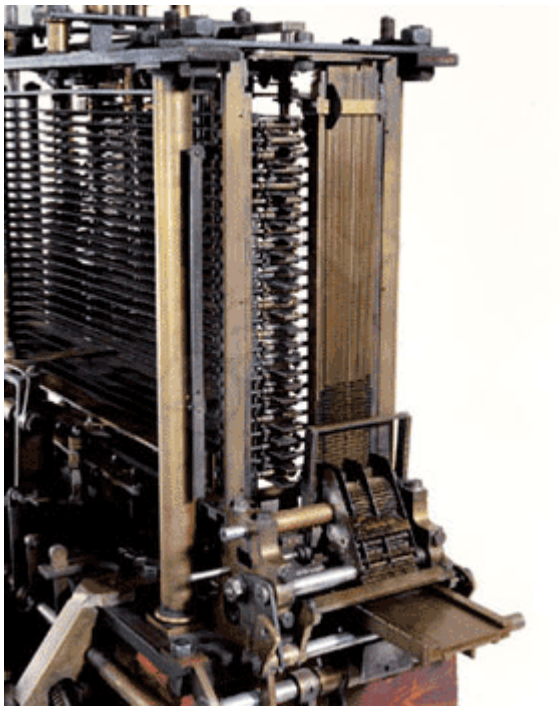




Charles Babbage's Difference Engine Demo pieces held at the Science museum in London.

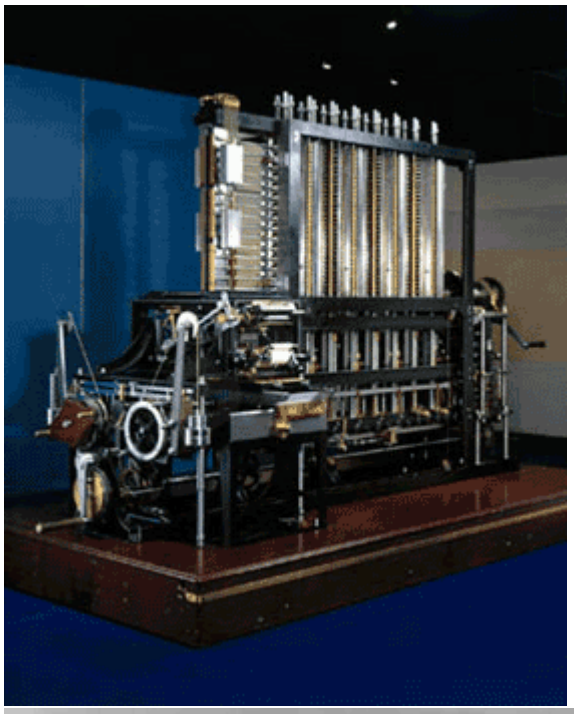
Two large fragments of the Analytical Engine were constructed by Babbage's son and Aiken may have seen photographs or otherwise become aware of their existence.

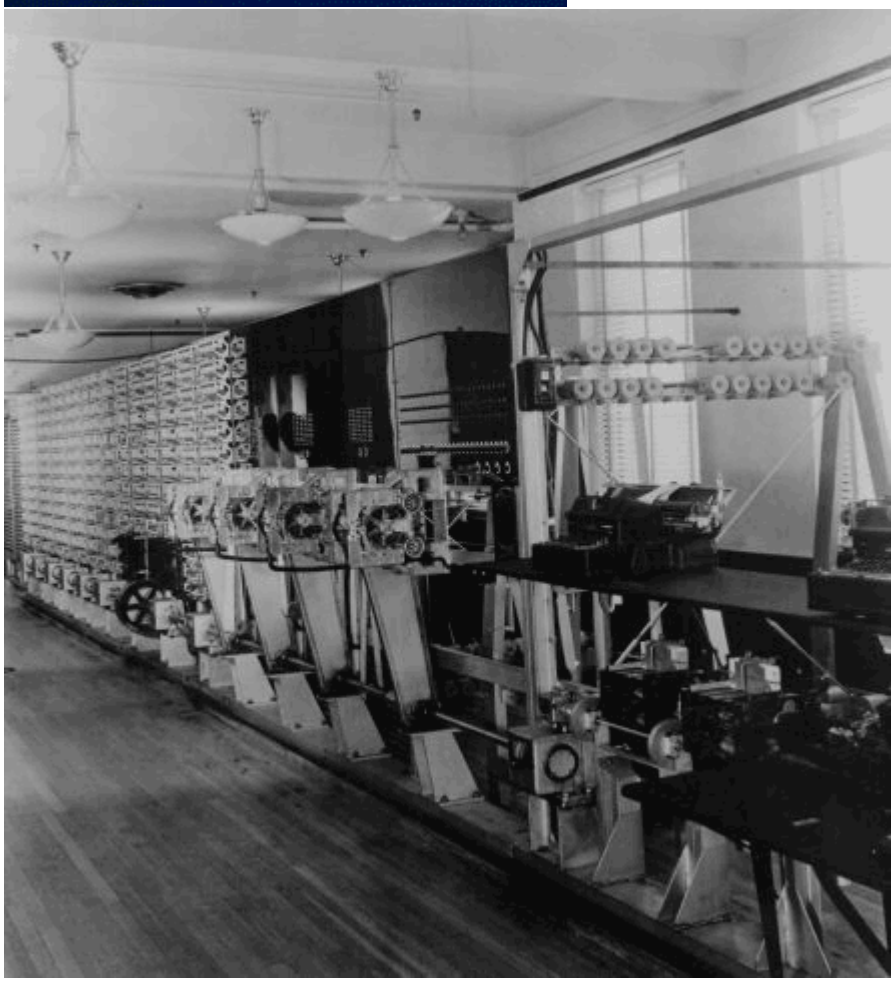




Charles Babbage's Analytical Engine pieces, constructed by his son.

In 1991 the Science Museum in London constructed the Difference Engine II, the printer was added in 2001. These pieces are on display in the Museum, which is well worth a visit. The construction of the Difference Engine II is documented by Doron Swade in his book *The Difference Engine*. The Difference Engine II was the last machine Babbage designed and employs lessons he learned from both the Difference Engine I and the Analytical Engine. For example the printer was designed for use by the Analytical Engine and Babbage reused it for the Difference Engine II. The similarity between the Difference Engine II and the machine that Aiken built is striking. Note the drive shaft running along the bottom of both machines and the general arrangement with printers at one end of a long tall frame. This may be the result of convergent evolution rather than direct influence but the similarity is still striking.





 *The Difference Engine II at the Science museum in London and Howard Aiken's Harvard Mk I, IBM Automatic Sequence Controlled Calculator under construction at Endicott*

In the foreword to the manual for the operation of the Automatic Sequence Controlled Calculator (ASCC) Howard Aiken states that "The appendices were prepared by Lieutenant [Grace] Hopper" with the assistance of others and that "[She] acted as general editor, and more than any other person is responsible for the book." It seems safe to conclude that Howard Aiken and Grace Hopper were not only influenced by Charles Babbage but they and their team held him in high regard and considered themselves guardians of his reputation and inheritors of his quest.

"If, unwarned by my example, any man shall undertake and shall succeed in really constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results."

-- Charles Babbage

The staff of the Computation Laboratory went on to have a considerable influence on the development of the modern computer. Not least of which was Grace Hooper, who developed the first compiler and several popular languages.

The influence of Howard Aiken and the IBM ASCC – Harvard Mk I machine on the later development of computers should not be over stated. The published notes from The Moore School Lectures (held in 1946) are rather scathing with respect to Aiken and his understanding of the direction in which the new electronic computing machines would lead.

"Hartree was very forward looking and was excited by the mathematical potential of the stored program computer. On the other hand, Aiken was absorbed in his own way of

doing things and does not appear to have been aware of the significance of the new electronic machines."

-- The Moore School Lectures (Charles Babbage Institute Reprint)

Unlike Aiken and his machine, Grace Hopper and some of her colleagues went on to have a significant influence in the early development of compilers and language design. One wonders what if any influence Babbage and Ada Lovelace had on Grace Hopper's ideas. Unfortunately I can find no comments by Hopper regarding either Babbage or Lovelace.

Source: John R. Harris, http://virtualtravelog.net/

History of Computing Hardware (1960–Present)

The main purpose of this article is to take a look at the history of computers from the third generation computers of the 1960s to today's technology of microcomputers, which have allowed for a computer presence in people's homes.

The **history of computing hardware** starting at 1960 is marked by the conversion from vacuum tube to solid-state devices such as transistors and then integrated circuit (IC) chips. By 1959, discrete transistors were considered sufficiently reliable and economical that they made further vacuum tube computers uncompetitive. Metal-oxide-semiconductor (MOS) large-scale integration (LSI) technology subsequently led to the development of semiconductor memory in the mid-to-late 1960s and then the microprocessor in the early 1970s. This led to primary computer memory moving away from magnetic-core memory devices to solid-state static and dynamic semiconductor memory, which greatly reduced the cost, size and power consumption of computers. These advances led to the miniaturized personal computer (PC) in the 1970s, starting with home computers and desktop computers, followed by laptops and then mobile computers over the next several decades.

Third generation

The mass increase in the use of computers accelerated with 'Third Generation' computers. These generally relied on integrated circuit (microchip) technology, starting around 1966 in the commercial market.

In 1958, Jack Kilby at Texas Instruments invented the hybrid integrated circuit (hybrid IC), which had external wire connections, making it difficult to mass-produce. In 1959, Robert Noyce at Fairchild Semiconductor invented the monolithic integrated circuit (IC) chip. It was made of silicon, whereas Kilby's chip was made of germanium. This basis for Noyce's monolithic IC was Fairchild's planar process, which allowed integrated circuits to be laid out using the same principles as those of printed circuits. The planar process was developed by Noyce's colleague Jean Hoerni in early 1959, based on the

silicon surface passivation and thermal oxidation processes developed by Mohamed M. Atalla at Bell Labs in the late 1950s.

Computers using IC chips began to appear in the early 1960s. For example, the 1961 Semiconductor Network Computer (Molecular Electronic Computer, Mol-E-Com), first monolithic integrated circuit general purpose computer (built for demonstration purposes, programmed to simulate a desk calculator) was built by Texas Instruments for the US Air Force.

Some of their early uses were in embedded systems, notably used by NASA for the Apollo Guidance Computer, by the military in the LGM-30 Minuteman intercontinental ballistic missile, the Honeywell ALERT airborne computer, and in the Central Air Data Computer used for flight control in the US Navy's F-14A Tomcat fighter jet.

An early commercial use was the 1965 SDS 92. IBM first used ICs in computers for the logic of the System/360 Model 85 shipped in 1969 and then made extensive use of ICs in its System/370 which began shipment in 1971.

The integrated circuit enabled the development of much smaller computers.
The minicomputer was a significant innovation in the 1960s and 1970s. It brought computing power to more people, not only through more convenient physical size but also through broadening the computer vendor field. Digital Equipment Corporation became the number two computer company behind IBM with their popular PDP and VAX computer systems. Smaller, affordable hardware also brought about the development of important new operating systems such as Unix.

1969 Data General Nova

In November 1966, Hewlett-Packard introduced the 2116A minicomputer, one of the first commercial 16-bit computers. It used CTμL (Complementary Transistor MicroLogic) in integrated circuits from Fairchild Semiconductor. Hewlett-Packard followed this with similar 16-bit computers, such as the 2115A in 1967, the 2114A in 1968, and others.

In 1969, Data General introduced the Nova and shipped a total of 50,000 at $8,000 each. The popularity of 16-bit computers, such as the Hewlett-Packard 21xx series and the Data General Nova, led the way toward word lengths that were multiples of the 8-bit byte. The Nova was first to employ medium-scale integration (MSI) circuits from Fairchild Semiconductor, with subsequent models using large-scale integrated (LSI) circuits. Also notable was that the entire central processor was contained on one 15-inch printed circuit board.

Large mainframe computers used ICs to increase storage and processing abilities. The 1965 IBM System/360 mainframe computer family are sometimes called third-generation computers; however their logic consisted primaritly of SLT hybrid circuits, which contained discrete transistors and diodes interconnected on a substrate with printed wires and printed passive components; the S/360 M85 and M91 did use ICs for some of their circuits. IBM's 1971 System/370 used ICs for their logic.

By 1971, the Illiac IV supercomputer was the fastest computer in the world, using about a quarter-million small-scale ECL logic gate integrated circuits to make up sixty-four parallel data processors.

Third-generation computers were offered well into the 1990s; for example the IBM ES9000 9X2 announced April 1994 used 5,960 ECL chips to make a 10-way processor. Other third-generation computers offered in the 1990s included the DEC VAX 9000 (1989), built from ECL gate arrays and custom chips, and the Cray T90 (1995).

Fourth generation

Third generation minicomputers were essentially scaled-down versions of mainframe computers, whereas the fourth generation's origins are fundamentally different. The basis of the fourth generation is the microprocessor, a computer processor contained on a single large-scale integration (LSI) MOS integrated circuit chip.

Microprocessor-based computers were originally very limited in their computational ability and speed, and were in no way an attempt to downsize the minicomputer. They were addressing an entirely different market.

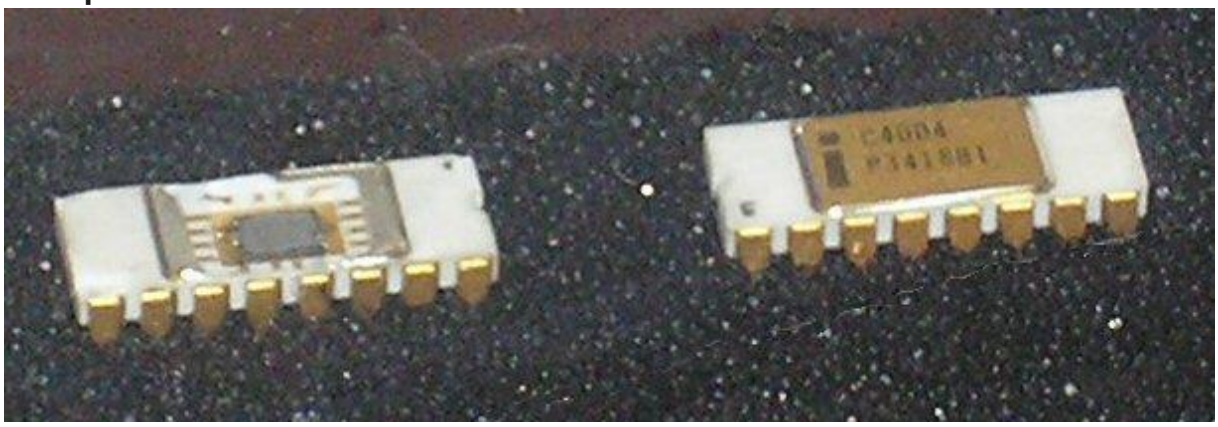
Processing power and storage capacities have grown beyond all recognition since the 1970s, but the underlying technology has remained basically the same of large-scale integration (LSI) or very-large-scale integration (VLSI) microchips, so it is widely regarded that most of today's computers still belong to the fourth generation.

Semiconductor memory

The MOSFET (metal-oxide-semiconductor field-effect transistor, or MOS transistor) was invented by Mohamed M. Atalla and Dawon Kahng at Bell Labs in 1959. In addition to

data processing, the MOSFET enabled the practical use of MOS transistors as memory cell storage elements, a function previously served by magnetic cores. Semiconductor memory, also known as MOS memory, was cheaper and consumed less power than magnetic-core memory. MOS random-access memory (RAM), in the form of static RAM (SRAM), was developed by John Schmidt at Fairchild Semiconductor in 1964. In 1966, Robert Dennard at the IBM Thomas J. Watson Research Center developed MOS dynamic RAM (DRAM). In 1967, Dawon Kahng and Simon Sze at Bell Labs developed the floating-gate MOSFET, the basis for MOS non-volatile memory such as EPROM, EEPROM and flash memory.

Microprocessors



1971: Intel 4004

The basic building block of every microprocessor is the metal-oxide-semiconductor field-effect transistor (MOSFET, or MOS transistor). The microprocessor has origins in the MOS integrated circuit (MOS IC) chip. The MOS IC was first proposed by Mohamed M. Atalla at Bell Labs in 1960, and then fabricated by Fred Heiman and Steven Hofstein at RCA in 1962. Due to rapid MOSFET scaling, MOS IC chips rapidly increased in complexity at a rate predicted by Moore's law, leading to large-scale integration (LSI) with hundreds of transistors on a single MOS chip by the late 1960s. The application of MOS LSI chips to computing was the basis for the first microprocessors, as engineers began recognizing that a complete computer processor could be contained on a single MOS LSI chip.

The earliest multi-chip microprocessors were the Four-Phase Systems AL-1 in 1969 and Garrett AiResearch MP944 in 1970, each using several MOS LSI chips. On November 15, 1971, Intel released the world's first single-chip microprocessor, the 4004, on a single MOS LSI chip. Its development was led by Federico Faggin, using silicon-gate MOS technology, along with Ted Hoff, Stanley Mazor and Masatoshi Shima. It was developed for a Japanese calculator company called Busicom as an alternative to hardwired circuitry, but computers were developed around it, with much of their processing abilities provided by one small microprocessor chip. The dynamic RAM (DRAM) chip was based on the MOS DRAM memory cell developed by Robert Dennard of IBM, offering kilobits of memory on one chip. Intel coupled the RAM chip with the microprocessor, allowing

fourth generation computers to be smaller and faster than prior computers. The 4004 was only capable of 60,000 instructions per second, but its successors brought ever-growing speed and power to computers, including the Intel 8008, 8080 (used in many computers using the CP/M operating system), and the 8086/8088 family. (The IBM personal computer (PC) and compatibles use processors that are still backwards-compatible with the 8086.) Other producers also made microprocessors which were widely used in microcomputers.

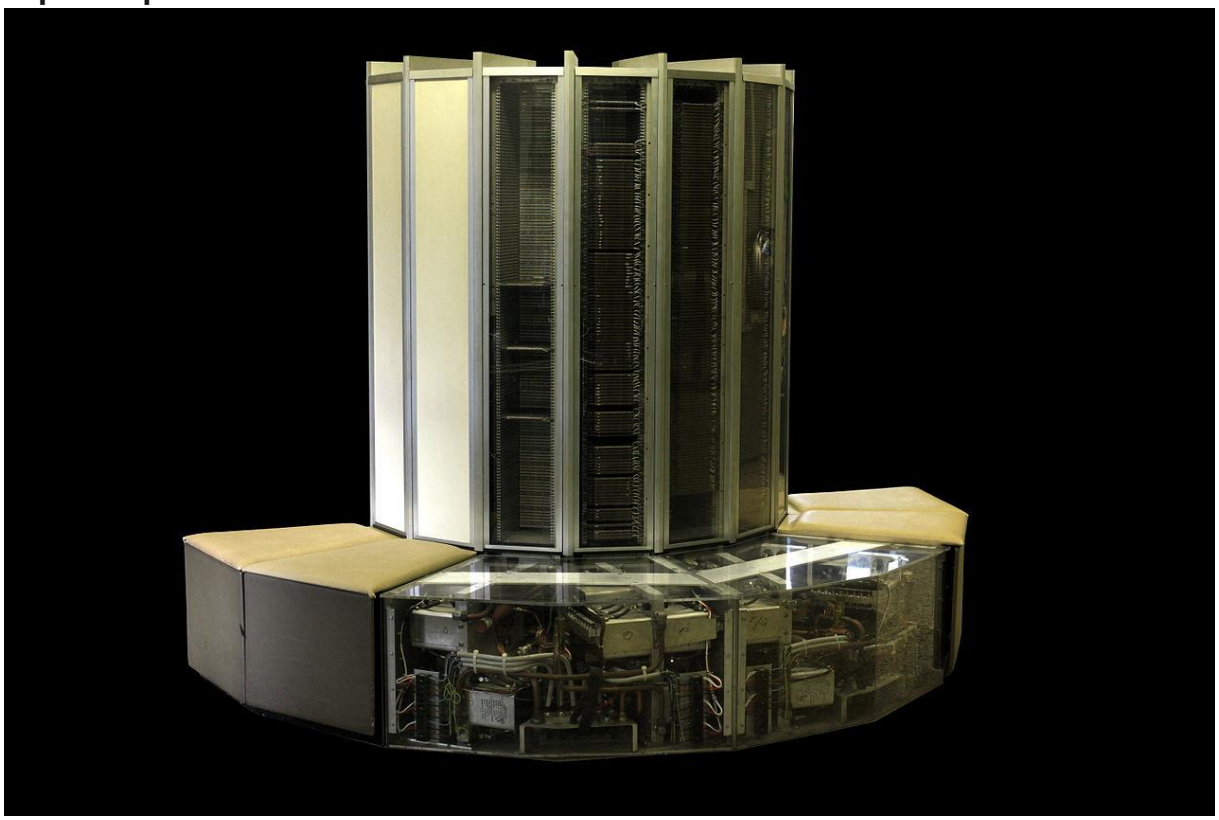
The following table shows a timeline of significant microprocessor development.

Year	Microprocessors
1971	Intel 4004
1972	Fairchild PPS-25; Intel 8008; Rockwell PPS-4
1973	Burroughs Mini-D; National IMP-16; NEC μCOM
1974	General Instrument CP1600; Intel 4040, 8080; Mostek 5065; Motorola 6800; National IMP-4, IMP-8, ISP-8A/500, PACE; Texas Instruments TMS 1000; Toshiba TLCS-12
1975	Fairchild F8; Hewlett Packard BPC; Intersil 6100; MOS Technology 6502; RCA CDP 1801; Rockwell PPS-8; Signetics 2650; Western Digital MCP-1600
1976	RCA CDP 1802; Signetics 8X300; Texas Instruments TMS9900; Zilog Z-80
1977	Intel 8085
1978	Intel 8086; Motorola 6801, 6809
1979	Intel 8088; Motorola 68000; Zilog Z8000
1980	National Semi 16032; Intel 8087
1981	DEC T-11; Harris 6120; IBM ROMP
1982	Hewlett Packard FOCUS; Intel 80186, 80188, 80286; DEC J-11; Berkeley RISC-I
1983	Stanford MIPS; Berkeley RISC-II
1984	Motorola 68020; National Semi 32032; NEC V20
1985	DEC MicroVAX 78032/78132; Harris Novix; Intel 80386; MIPS R2000
1986	NEC V60; Sun SPARC MB86900/86910; Zilog Z80000
1987	Acorn ARM2; DEC CVAX 78034; Hitachi Gmicro/200; Motorola 68030; NEC V70
1988	Intel 80386SX, i960; MIPS R3000

1989	DEC VAX DC520 Rigel; Intel 80486, i860
1990	IBM POWER1; Motorola 68040
1991	DEC NVAX; IBM RSC; MIPS R4000
1992	DEC Alpha 21064; Hewlett Packard PA-7100; Sun microSPARC I
1993	IBM POWER2, PowerPC 601; Intel Pentium; Hitachi SuperH
1994	DEC Alpha 21064A; Hewlett Packard PA-7100LC, PA-7200; IBM PowerPC 603, PowerPC 604, ESA/390 G1; Motorola 68060; QED R4600; NEC V850
1995	DEC Alpha 21164; HAL Computer SPARC64; Intel Pentium Pro; Sun UltraSPARC; IBM ESA/390 G2
1996	AMD K5; DEC Alpha 21164A; HAL Computer SPARC64 II; Hewlett Packard PA-8000; IBM P2SC, ESA/390 G3; MTI R10000; QED R5000
1997	AMD K6; IBM PowerPC 620, PowerPC 750, RS64, ESA/390 G4; Intel Pentium II; Sun UltraSPARC IIs
1998	DEC Alpha 21264; HAL Computer SPARC64 III; Hewlett Packard PA-8500; IBM POWER3, RS64-II, ESA/390 G5; QED RM7000; SGI MIPS R12000
1999	AMD Athlon; IBM RS64-III; Intel Pentium III; Motorola PowerPC 7400
2000	AMD Athlon XP, Duron; Fujitsu SPARC64 IV; IBM RS64-IV, z900; Intel Pentium 4
2001	IBM POWER4; Intel Itanium; Motorola PowerPC 7450; SGI MIPS R14000; Sun UltraSPARC III
2002	Fujitsu SPARC64 V; Intel Itanium 2
2003	AMD Opteron, Athlon 64; IBM PowerPC 970; Intel Pentium M
2004	IBM POWER5, PowerPC BGL
2005	AMD Athlon 64 X2, Opteron Athens; IBM PowerPC 970MP, Xenon; Intel Pentium D; Sun UltraSPARC IV, UltraSPARC T1
2006	IBM Cell/B.E., z9; Intel Core 2, Core Duo, Itanium Montecito
2007	AMD Opteron Barcelona; Fujitsu SPARC64 VI; IBM POWER6, PowerPC BGP; Sun UltraSPARC T2; Tilera TILE64
2008	AMD Opteron Shanghai, Phenom; Fujitsu SPARC64 VII; IBM PowerXCell 8i, z10; Intel Atom, Core i7; Tilera TILEPro64
2009	AMD Opteron Istanbul, Phenom II
2010	AMD Opteron Magny-cours; Fujitsu SPARC64 VII+; IBM POWER7, z196; Intel Itanium Tukwila, Westmere, Nehalem-EX; Sun SPARC T3

2011	AMD FX Bulldozer, Interlagos, Llano; Fujitsu SPARC64 VIIIfx; Freescale PowerPC e6500; Intel Sandy Bridge, Xeon E7; Oracle SPARC T4
2012	Fujitsu SPARC64 IXfx; IBM POWER7+, zEC12; Intel Itanium Poulson
2013	Fujitsu SPARC64 X; Intel Haswell; Oracle SPARC T5
2014	IBM POWER8
2015	IBM z13
2017	IBM POWER9, z14; AMD Ryzen

Supercomputers



1976: Cray-1 supercomputer

The powerful supercomputers of the era were at the other end of the computing spectrum from the microcomputers, and they also used integrated circuit technology. In 1976, the Cray-1 was developed by Seymour Cray, who had left Control Data in 1972 to form his own company. This machine was the first supercomputer to make vector processing practical. It had a characteristic horseshoe shape to speed processing by shortening circuit paths. Vector processing uses one instruction to perform the same operation on many arguments; it has been a fundamental supercomputer processing method ever since. The Cray-1 could calculate 150 million floating point operations per

second (150 megaflops). 85 were shipped at a price of $5 million each. The Cray-1 had a CPU that was mostly constructed of SSI and MSI ECL ICs.

Mainframes and minicomputers

Time-sharing computer terminals connected to central computers, such as the TeleVideo ASCII character mode smart terminal pictured here, were sometimes used before the advent of the PC.

Computers were generally large, costly systems owned by large institutions before the introduction of the microprocessor in the early 1970s — corporations, universities, government agencies, and the like. Users were experienced specialists who did not usually interact with the machine itself, but instead prepared tasks for the computer on off-line equipment, such as card punches. A number of assignments for the computer would be gathered up and processed in batch mode. After the jobs had completed, users

could collect the output printouts and punched cards. In some organizations, it could take hours or days between submitting a job to the computing center and receiving the output.

A more interactive form of computer use developed commercially by the middle 1960s. In a time-sharing system, multiple teleprinter terminals let many people share the use of one mainframe computer processor. This was common in business applications and in science and engineering.

A different model of computer use was foreshadowed by the way in which early, pre-commercial, experimental computers were used, where one user had exclusive use of a processor. Some of the first computers that might be called "personal" were early minicomputers such as the LINC and PDP-8, and later on VAX and larger minicomputers from Digital Equipment Corporation (DEC), Data General, Prime Computer, and others. They originated as peripheral processors for mainframe computers, taking on some routine tasks and freeing the processor for computation. By today's standards, they were physically large (about the size of a refrigerator) and costly (typically tens of thousands of US dollars), and thus were rarely purchased by individuals. However, they were much smaller, less expensive, and generally simpler to operate than the mainframe computers of the time, and thus affordable by individual laboratories and research projects. Minicomputers largely freed these organizations from the batch processing and bureaucracy of a commercial or university computing center.

In addition, minicomputers were more interactive than mainframes, and soon had their own operating systems. The minicomputer Xerox Alto (1973) was a landmark step in the development of personal computers, because of its graphical user interface, bit-mapped high resolution screen, large internal and external memory storage, mouse, and special software.

Microprocessor and cost reduction

In the minicomputer ancestors of the modern personal computer, processing was carried out by circuits with large numbers of components arranged on multiple large printed circuit boards. Minicomputers were consequently physically large and expensive to produce compared with later microprocessor systems. After the "computer-on-a-chip" was commercialized, the cost to produce a computer system dropped dramatically. The arithmetic, logic, and control functions that previously occupied several costly circuit boards were now available in one integrated circuit which was very expensive to design but cheap to produce in large quantities. Concurrently, advances in developing solid state memory eliminated the bulky, costly, and power-hungry magnetic core memory used in prior generations of computers.

Micral N

Micral N

In France, the company R2E (Réalisations et Etudes Electroniques) formed by five former engineers of the Intertechnique company, André Truong Trong Thi and François Gernelle introduced in February 1975 a microcomputer, the Micral N based on the Intel 8008. Originally, the computer had been designed by Gernelle, Lacombe, Beckmann and Benchitrite for the Institut National de la Recherche Agronomique to automate hygrometric measurements. The Micral N cost a fifth of the price of a PDP-8, about 8500FF ($1300). The clock of the Intel 8008 was set at 500 kHz, the memory was 16 kilobytes. A bus, called Pluribus was introduced and allowed connection of up to 14 boards. Different boards for digital I/O, analog I/O, memory, floppy disk were available from R2E.

Altair 8800 and IMSAI 8080

Development of the single-chip microprocessor was an enormous catalyst to the popularization of cheap, easy to use, and truly personal computers. The Altair 8800, introduced in a *Popular Electronics* magazine article in the January 1975 issue, at the time set a new low price point for a computer, bringing computer ownership to an admittedly select market in the 1970s. This was followed by the IMSAI 8080 computer, with similar abilities and limitations. The Altair and IMSAI were essentially scaled-down minicomputers and were incomplete: to connect a keyboard or teleprinter to them required heavy, expensive "peripherals". These machines both featured a front panel with

switches and lights, which communicated with the operator in binary. To program the machine after switching it on the bootstrap loader program had to be entered, without error, in binary, then a paper tape containing a BASIC interpreter loaded from a paper-tape reader. Keying the loader required setting a bank of eight switches up or down and pressing the "load" button, once for each byte of the program, which was typically hundreds of bytes long. The computer could run BASIC programs once the interpreter had been loaded.

1975: Altair 8800

The MITS Altair, the first commercially successful microprocessor kit, was featured on the cover of *Popular Electronics* magazine in January 1975. It was the world's first mass-produced personal computer kit, as well as the first computer to use an Intel 8080 processor. It was a commercial success with 10,000 Altairs being shipped. The Altair also inspired the software development efforts of Paul Allen and his high school friend Bill Gates who developed a BASIC interpreter for the Altair, and then formed Microsoft.

The MITS Altair 8800 effectively created a new industry of microcomputers and computer kits, with many others following, such as a wave of small business computers in the late 1970s based on the Intel 8080, Zilog Z80 and Intel 8085 microprocessor chips. Most ran the CP/M-80 operating system developed by Gary Kildall at Digital Research. CP/M-80 was the first popular microcomputer operating system to be used by many different hardware vendors, and many software packages were written for it, such as WordStar and dBase II.

Many hobbyists during the mid-1970s designed their own systems, with various degrees of success, and sometimes banded together to ease the job. Out of these house meetings the Homebrew Computer Club developed, where hobbyists met to talk about what they had done, exchange schematics and software, and demonstrate their systems. Many people built or assembled their own computers as per published designs. For example, many thousands of people built the Galaksija home computer later in the early 1980s.

It was arguably the Altair computer that spawned the development of Apple, as well as Microsoft which produced and sold the Altair BASIC programming language interpreter, Microsoft's first product. The second generation of microcomputers, those that appeared in the late 1970s, sparked by the unexpected demand for the kit computers at the electronic hobbyist clubs, were usually known as home computers. For business use these systems were less capable and in some ways less versatile than the large business computers of the day. They were designed for fun and educational purposes, not so much for practical use. And although you could use some simple office/productivity applications on them, they were generally used by computer enthusiasts for learning to program and for running computer games, for which the personal computers of the period were less suitable and much too expensive. For the more technical hobbyists home computers were also used for electronics interfacing, such as controlling model railroads, and other general hobbyist pursuits.

Microcomputer emerges

The "Big Three" computers of 1977: from left to right, the Commodore PET (PET 2001 model shown), the standard Apple II (with two Disk II drives) and the TRS-80 Model I.

The advent of the microprocessor and solid-state memory made home computing affordable. Early hobby microcomputer systems such as the Altair 8800 and Apple I introduced around 1975 marked the release of low-cost 8-bit processor chips, which had sufficient computing power to be of interest to hobby and experimental users. By 1977 pre-assembled systems such as the Apple II, Commodore PET, and TRS-80 (later dubbed the "1977 Trinity" by *Byte* Magazine) began the era of mass-market home computers; much less effort was required to obtain an operating computer, and applications such as games, word processing, and spreadsheets began to proliferate. Distinct from computers used in homes, small business systems were typically based on CP/M, until IBM introduced the IBM-PC, which was quickly adopted. The PC was heavily cloned, leading to mass production and consequent cost reduction throughout the 1980s. This expanded the PCs presence in homes, replacing the home computer category during the 1990s and leading to the currentmonoculture of architecturally identical personal computers.

Timeline of computer systems and important hardware

Year	Hardware
1959	Transistors: IBM 7090; IBM 1401
1960	DEC PDP 1
1961	Fairchild resistor transistor logic
1962	NPN transistor

1963	Mouse; CMOS patented
1964	CDC 6600; IBM Data Cell Drive
1965	DEC PDP 8; IBM 1130
1966	Integrated circuits: HP 2116A; Apollo Guidance Computer
1967	Fairchild built first MOS; Englebart applies for mouse patent
1969	Data General Nova
1969	Honeywell 316
1970	DEC PDP 11
1971	8" floppy disk; ILLIAC IV
1972	Atari founded; Cray Research founded
1973	Micral first microprocessor PC
1974	Altair 8800; Data General Eclipse
1975	Olivetti P6060; Cray-1
1976	Tandem Computers
1977	Apple II; TRS-80 Model 1; Commodore PET; 5.25" floppy
1978	DEC VAX 11
1979	Atari 400, 800
1980	Sinclair ZX80, Seagate hard disk drive
1981	IBM PC, Acorn BBC Micro
1982	Commodore 64, Sinclair ZX Spectrum
1983	Apple Lisa; 3.5" floppy
1984	Apple Mac; Apple Lisa 2
1985	PC's Limited (renamed Dell Computer Corporation in 1988); Amiga 1000
1986	Tandem Nonstop VLX
1987	Thinking Machine CM2; Tera Computer Founded
1988	Dell
1989	NeXT
1990	ETA10; CD-R

1991	Apple Switches to PowerPC
1992	HP 95LX; Palmtop PC
1993	Intel PPGA
1994	VESA Local Bus
1995	IBM Deep Blue chess computer
1996	USB 1.0
1997	Compaq buys Tandem; CD-RW
1998	iMac
1999	First BlackBerry device (850)
2000	USB 2
2003	Arduino
2005	Mac Mini; World's first desktop dual-core CPU Athlon 64 X2
2006	Apple transition to Intel
2007	iPhone 1
2008	USB 3.0
2010	Apple iPad
2012	IBM zEnterprise System; Raspberry Pi
2015	HoloLens

Source:
Wikipedia, https://en.wikipedia.org/wiki/History_of_computing_hardware_(1960s%E2%80%93present)

1.2: Components of a Computer

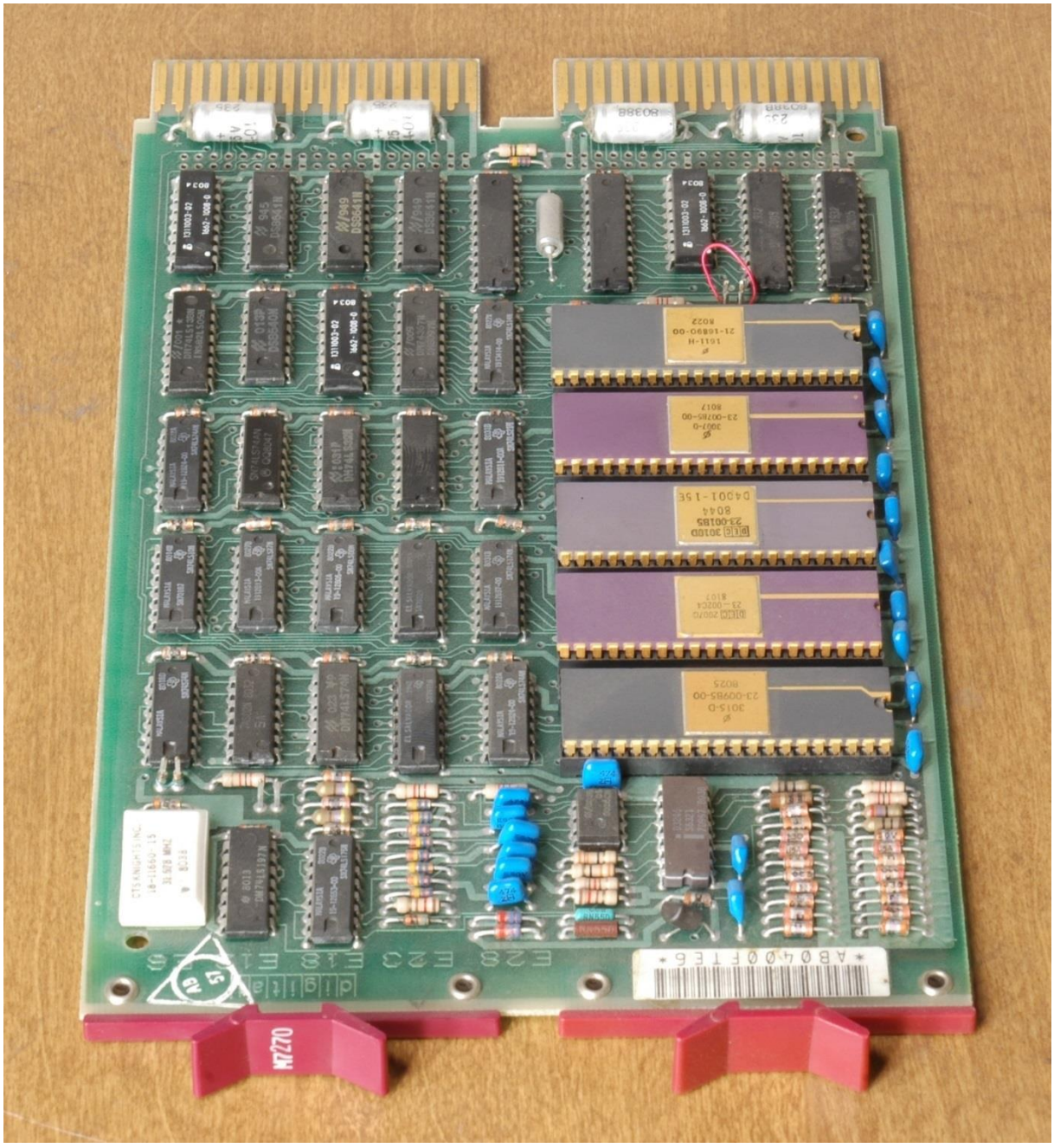
Personal Computer Hardware

Read this article for a solid overview of various components of a computer, including the motherboard, power supply, removable media devices, secondary storage, sound cards, and input and output peripherals.

Computer hardware includes the physical parts of a computer, such as the case, central processing unit (CPU), monitor, mouse, keyboard, computer data storage, graphics card, sound card, speakers and motherboard.

By contrast, software is the set of instructions that can be stored and run by hardware. Hardware is so-termed because it is " hard" or rigid with respect to changes, whereas software is "soft" because it is easy to change.

Hardware is typically directed by the software to execute any command or instruction . A combination of hardware and software forms a usable computing system, although other systems exist with only hardware.



PDP-11 CPU board

Von Neumann architecture

Von Neumann architecture scheme

The template for all modern computers is the Von Neumann architecture, detailed in a 1945 paper by Hungarian mathematician John von Neumann. This describes a design architecture for an electronic digital computer with subdivisions of a processing unit consisting of an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms. The meaning of the term has evolved to mean a stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the Von Neumann bottleneck and often limits the performance of the system.

Types of computer systems

Personal computer

Basic hardware components of a modern personal computer, including a monitor, a motherboard, a CPU, a RAM, two expansion cards, a power supply, an optical disc drive, a hard disk drive, a keyboard and a mouse

Inside a custom-built computer: power supply at the bottom has its own cooling fan

The personal computer is one of the most common types of computer due to its versatility and relatively low price. Desktop personal computers have a monitor , a keyboard, a mouse , and a computer case . The computer case holds the motherboard, fixed or removable disk drives for data storage, the power supply, and may contain other peripheral devices such as modems or network interfaces. Some models of desktop computers integrated the monitor and keyboard into the same case as the processor and power supply. Separating the elements allows the user to arrange the components in a pleasing, comfortable array, at the cost of managing power and data cables between them.

Laptops are designed for portability but operate similarly to desktop PCs. They may use lower-power or reduced size components, with lower performance than a similarly priced desktop computer. Laptops contain the keyboard, display, and processor in one case. The monitor in the folding upper cover of the case can be closed for transportation, to

protect the screen and keyboard. Instead of a mouse, laptops may have a trackpad or pointing stick.

Tablets are portable computer that uses a *touchscreen* as the primary input device. Tablets generally weigh less and are smaller than laptops. Some tablets include fold out keyboards, or offer connections to separate external keyboards.

Case

The computer case encloses most of the components of the system. It provides mechanical support and protection for internal elements such as the motherboard, disk drives, and power supplies, and controls and directs the flow of cooling air over internal components. The case is also part of the system to control electromagnetic interference radiated by the computer and protects internal parts from electrostatic discharge. Large tower cases provide space for multiple disk drives or other peripherals and usually stand on the floor, while desktop cases provide less expansion room. All-in-one style designs include a video display built into the same case. Portable and laptop computers require cases that provide impact protection for the unit. A current development in laptop computers is a detachable keyboard, which allows the system to be configured as a touch-screen tablet. Hobbyists may decorate the cases with colored lights, paint, or other features, in an activity called case modding.

Power supply

A power supply unit (PSU) converts alternating current (AC) electric power to low-voltage direct current (DC) power for the computer. Laptops can run on built-in rechargeable battery. The PSU typically uses a switched-mode power supply (SMPS), with power MOSFETs (power metal–oxide–semiconductor field-effect transistors) used in the converters and regulator circuits of the SMPS.

Motherboard

The motherboard is the main component of a computer. It is a board with integrated circuitry that connects the other parts of the computer including the CPU, the RAM, the disk drives (CD, DVD, hard disk, or any others) as well as any peripherals connected via the ports or the expansion slots. The integrated circuit (IC) chips in a computer typically contain billions of tiny metal–oxide–semiconductor field-effect transistors (MOSFETs).

Components directly attached to or to part of the motherboard include:

- The CPU (central processing unit), which performs most of the calculations which enable a computer to function, and is referred to as the brain of the computer which get a hold of program instruction from random-access memory (RAM),

interprets and processes it and then send it backs to computer result so that the relevant components can carry out the instructions. The CPU is a microprocessor, which is fabricated on a metal–oxide–semiconductor (MOS) integrated circuit (IC) chip. It is usually cooled by a heat sink and fan, or water-cooling system. Most newer CPU includes an on-die graphics processing unit (GPU). The clock speed of CPU governs how fast it executes instructions and is measured in GHz; typical values lie between 1 GHz and 5 GHz. Many modern computers have the option to overclock the CPU which enhances performance at the expense of greater thermal output and thus a need for improved cooling.

- The chipset, which includes the north bridge, mediates communication between the CPU and the other components of the system, including main memory; as well as south bridge, which is connected to the north bridge, and supports auxiliary interfaces and buses; and, finally, a Super I/O chip, connected through the south bridge, which supports the slowest and most legacy components like serial ports, hardware monitoring and fan control.
- Random-access memory (RAM), which stores the code and data that are being actively accessed by the CPU. For example, when a web browser is opened on the computer it takes up memory; this is stored in the RAM until the web browser is closed. It is typically a type of dynamic RAM (DRAM), such as synchronous DRAM (SDRAM), where MOS memory chips store data on memory cells consisting of MOSFETs and MOS capacitors. RAM usually comes on dual in-line memory modules (DIMMs) in the sizes of 2GB, 4GB, and 8GB, but can be much larger.
- Read-only memory (ROM), which stores the BIOS that runs when the computer is powered on or otherwise begins execution, a process known as Bootstrapping, or "booting" or "booting up". The ROM is typically a nonvolatile BIOS memory chip, which stores data on floating-gate MOSFET memory cells.
    - The BIOS (Basic Input Output System) includes boot firmware and power management firmware. Newer motherboards use Unified Extensible Firmware Interface (UEFI) instead of BIOS.
- Buses that connect the CPU to various internal components and to expand cards for graphics and sound.
- The CMOS (complementary MOS) battery, which powers the CMOS memory for date and time in the BIOS chip. This battery is generally a watch battery.
- The video card (also known as the graphics card), which processes computer graphics. More powerful graphics cards are better suited to handle strenuous tasks, such as playing intensive video games or running computer graphics software. A video card contains a graphics processing unit (GPU) and video memory (typically a type of SDRAM), both fabricated on MOS integrated circuit (MOS IC) chips.
- Power MOSFETs make up the voltage regulator module (VRM), which controls how much voltage other hardware components receive.

Expansion cards

An expansion card in computing is a printed circuit board that can be inserted into an expansion slot of a computer motherboard or backplane to add functionality to a computer system via the expansion bus. Expansion cards can be used to obtain or expand on features not offered by the motherboard.

Storage devices

A storage device is any computing hardware and digital media that is used for storing, porting and extracting data files and objects. It can hold and store information both temporarily and permanently and can be internal or external to a computer, server or any similar computing device. Data storage is a core function and fundamental component of computers.

Fixed media

Data is stored by a computer using a variety of media. Hard disk drives (HDDs) are found in virtually all older computers, due to their high capacity and low cost, but solid-state drives (SSDs) are faster and more power efficient, although currently more expensive than hard drives in terms of dollar per gigabyte, so are often found in personal computers built post-2007. SSDs use flash memory, which stores data on MOS memory chips consisting of floating-gate MOSFET memory cells. Some systems may use a disk array controller for greater performance or reliability.

Removable media

To transfer data between computers, an external flash memory device (such as a memory card or USB flash drive) or optical disc (such as a CD-ROM, DVD-ROM or BD-ROM) may be used. Their usefulness depends on being readable by other systems; the majority of machines have an optical disk drive (ODD), and virtually all have at least one Universal Serial Bus (USB) port.

Input and output peripherals

Input and output devices are typically housed externally to the main computer chassis. The following are either standard or very common to many computer systems.

Input device

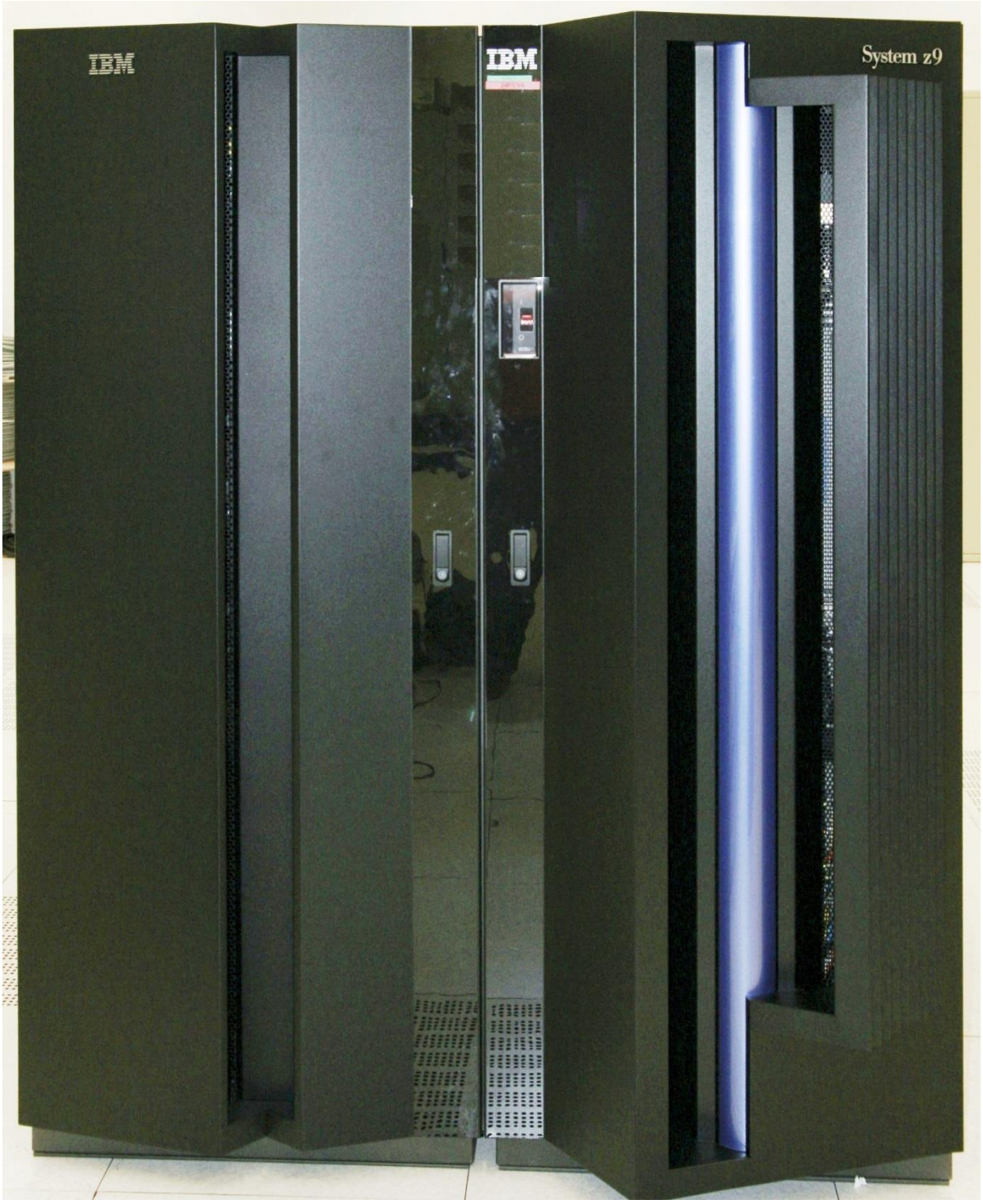
Input devices allow the user to enter information into the system, or control its operation. Most personal computers have a mouse and keyboard , but laptop systems typically use a touchpad instead of a mouse. Other input devices
include webcams, microphones, joysticks, and image scanners.

Output device

Output devices are designed around the senses of human beings. For
example, monitors display text that can be read, speakers produce sound that can be heard. Such devices also could include printers or a Braille embosser.

Mainframe computer

A mainframe computer is a much larger computer that typically fills a room and may cost many hundreds or thousands of times as much as a personal computer. They are designed to perform large numbers of calculations for governments and large enterprises.



An IBM System z9 mainframe

Departmental computing

In the 1960s and 1970s, more and more departments started to use cheaper and dedicated systems for specific purposes like process control and laboratory automation. A **minicomputer**, or colloquially **mini**, is a class of smaller computers that was developed

in the mid-1960's and sold for much less than mainframe and mid-size computers from IBM and its direct competitors.

Supercomputer

A supercomputer is superficially similar to a mainframe but is instead intended for extremely demanding computational tasks. As of June 2018, the fastest supercomputer on the TOP500 supercomputer list is the Summit, in the United States, with a LINPACK benchmark score of 122.3 PFLOPS Light, by around 29 PFLOPS.

The term supercomputer does not refer to a specific technology. Rather it indicates the fastest computations available at any given time. In mid-2011, the fastest supercomputers boasted speeds exceeding one petaflop, or 1 quadrillion (10^15 or 1,000 trillion) floating-point operations per second. Supercomputers are fast but extremely costly, so they are generally used by large organizations to execute computationally demanding tasks involving large data sets. Supercomputers typically run military and scientific applications. Although costly, they are also being used for commercial applications where huge amounts of data must be analyzed. For example, large banks employ supercomputers to calculate the risks and returns of various investment strategies, and healthcare organizations use them to analyze giant databases of patient data to determine optimal treatments for various diseases and problems incurring to the country.

Hardware upgrade

When using computer hardware, an upgrade means adding new or additional hardware to a computer that improves its performance, increases its capacity, or adds new features. For example, a user could perform a hardware upgrade to replace the hard drive with a faster one or a Solid State Drive (SSD) to get a boost in performance. The user may also install more Random Access Memory (RAM) so the computer can store additional temporary data, or retrieve such data at a faster rate. The user may add a USB 3.0 expansion card to fully use USB 3.0 devices, or could upgrade the Graphics Processing Unit (GPU) for cleaner, more advanced graphics, or more monitors. Performing such hardware upgrades may be necessary for aged computers to meet a new, or updated program's system requirements.

Sales

Global revenue from computer hardware in 2016 reached 408 billion Euros.

Recycling

Because computer parts contain hazardous materials, there is a growing movement to recycle old and outdated parts. Computer hardware contain dangerous chemicals such as: lead, mercury, nickel, and cadmium. According to the EPA these e-wastes have a harmful effect on the environment unless they are disposed of properly. Making hardware requires energy, and recycling parts will reduce air pollution, water pollution, as well as greenhouse gas emissions. Disposing unauthorized computer equipment is in fact illegal. Legislation makes it mandatory to recycle computers through the government approved facilities. Recycling a computer can be made easier by taking out certain reusable parts. For example, the RAM, DVD drive, the graphics card , hard drive or SSD, and other similar removable parts can be reused.

Many materials used in computer hardware can be recovered by recycling for use in future production. Reuse of tin, silicon, iron, aluminium, and a variety of plastics that are present in bulk in computers or other electronics can reduce the costs of constructing new systems. Components frequently contain copper, gold, tantalum, silver, platinum, palladium, and lead as well as other valuable materials suitable for reclamation.

Toxic computer components

The central processing unit contains many toxic materials. It contains lead and chromium in the metal plates. Resistors, semi-conductors, infrared detectors, stabilizers, cables, and wires contain cadmium. The circuit boards in a computer contain mercury, and chromium. When these types of materials, and chemicals are disposed improperly will become hazardous for the environment.

Environmental effects

According to the United States Environmental Protection Agency only around 15% of the e-waste actually is recycled. When e-waste byproducts leach into groundwater, are burned, or get mishandled during recycling, it causes harm. Health problems associated with such toxins include impaired mental development, cancer, and damage to the lungs, liver, and kidneys. That's why even wires have to be recycled. Different companies have different techniques to recycle a wire. The most popular one is the grinder that separates the copper wires from the plastic/rubber casing. When the processes are done there are two different piles left; one containing the copper powder, and the other containing plastic/rubber pieces. Computer monitors, mice, and keyboards all have a similar way of being recycled. For example, first, each of the parts are taken apart then all of the inner parts get separated and placed into its own bin.

Computer components contain many toxic substances, like dioxins, polychlorinated biphenyls (PCBs), cadmium, chromium, radioactive isotopes and mercury. A typical computer monitor may contain more than 6% lead by weight, much of which is in the lead glass of the cathode ray tube (CRT). A typical 15 inch (38 cm) computer monitor

may contain 1.5 pounds (1 kg) of lead but other monitors have been estimated to have up to 8 pounds (4 kg) of lead. Circuit boards contain considerable quantities of lead-tin solders that are more likely to leach into groundwater or create air pollution due to incineration. In US landfills, about 40% of the lead content levels are from e-waste. The processing (e.g. incineration and acid treatments) required to reclaim these precious substances may release, generate, or synthesize toxic byproducts.

Recycling of computer hardware is considered environmentally friendly because it prevents hazardous waste, including heavy metals and carcinogens, from entering the atmosphere, landfill or waterways. While electronics consist a small fraction of total waste generated, they are far more dangerous. There is stringent legislation designed to enforce and encourage the sustainable disposal of appliances, the most notable being the Waste Electrical and Electronic Equipment Directive of the European Union and the United States National Computer Recycling Act.

Efforts for minimizing computer hardware waste

As computer hardware contain a wide number of metals inside, the United States Environmental Protection Agency (EPA) encourages the collection and recycling of computer hardware. "E-cycling", the recycling of computer hardware, refers to the donation, reuse, shredding and general collection of used electronics. Generically, the term refers to the process of collecting, brokering, disassembling, repairing and recycling the components or metals contained in used or discarded electronic equipment, otherwise known as electronic waste (e-waste). "E-cyclable" items include, but are not limited to: televisions, computers, microwave ovens, vacuum cleaners, telephones and cellular phones, stereos, and VCRs and DVDs just about anything that has a cord, light or takes some kind of battery.

Recycling a computer is made easier by a few of the national services, such as Dell and Apple. Both companies will take back the computer of their make or any other make. Otherwise a computer can be donated to Computer Aid International which is an organization that recycles and refurbishes old computers for hospitals, schools, universities, etc.

Source: Wikipedia, https://en.wikipedia.org/wiki/Computer_hardware

1.3: The Role of Processor Performance

CPU and Processor Time Counter

Read this introduction to computer performance and information about computing processor time. In most programming languages, there is a process to measure elapsed

time, such as time() in C. By subtracting the time at the beginning of a process from the time at the end you get the total time for a particular operation. Usually short operations are put in a loop that repeats the operation a sufficient number of times to get an accurate measurement.

About

CPU time: The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute.

Definition

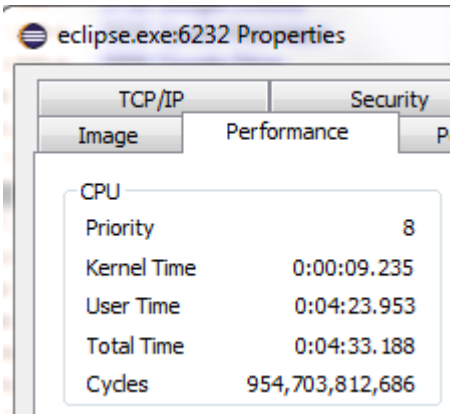
How do you measure Execution Time?

+KernelUserTotalTimeTimeTimeKernelTime+UserTimeTotalTime

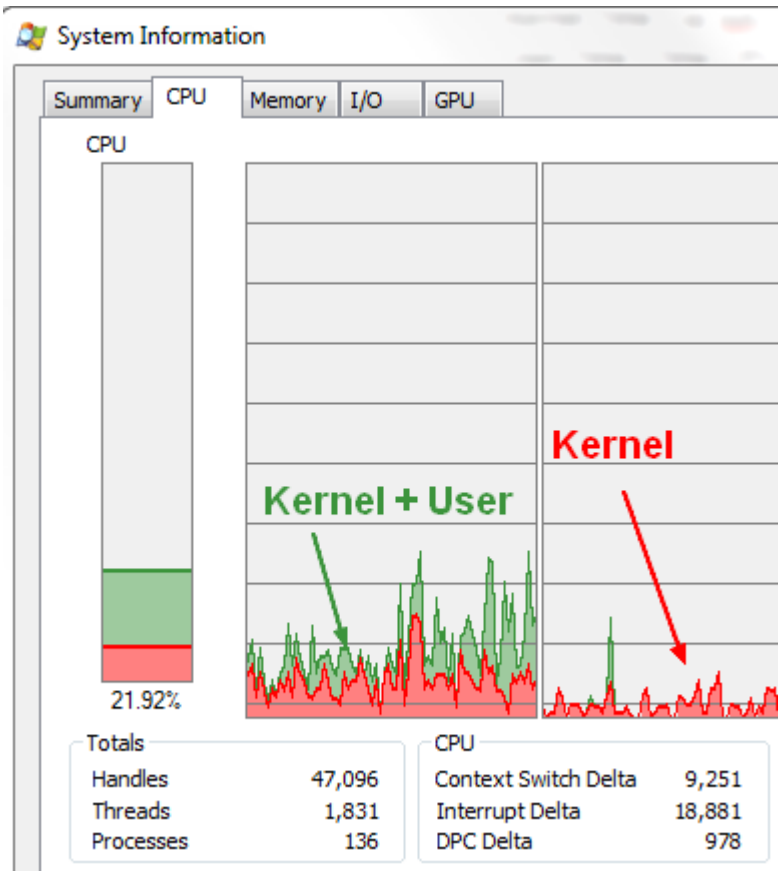
CPU Time	Running which code
Kernel (Sys) Time	Operating system code - the time of CPU spent on the kernel (system) code only within the given process - other processes and the time when our process is blocked are not included.
User Time	Time CPU spends running the program code (ie CPU time spent on non-kernel (user-mode) code only within the given process – so other processes and the time when our process is blocked are not included.).
Total time	User + Kernel
Real Time	Wall clock (total elapsed time) (can be lower than total time because of parallelism or longer because of wait) - user-perceived time it took to execute the command – from the start to the end of the call, including time slices used by other processes and the time when our process is blocked (e.g. I/O waiting)

Process Explorer

To see CPU time, you can look at the performance tab of a process in Process Explorer:



Red in the CPU usage graph indicates CPU usage in kernel-mode whereas green is the sum of kernel-mode and user-mode execution.



A data collector set can be configured via logman.exe to log the "% Processor Time" counter in the "Processor Information" object for this purpose.

Formula

Every conventional processor has a clock with a fixed cycle time (or clock rate). At every CPU cycle, an instruction is executed.

$$\text{CPU Time} = \text{CPU cycles executed} * \text{Cycle times}$$CPU Time=CPU cycles executed∗Cycle times

$$\text{CPU cycles} = \text{Instructions executed} * \text{Average Clock Cycles per Instruction (CPI)}$$CPU cycles=Instructions executed∗Average Clock Cycles per Instruction (CPI)

Putting it all together:

$$\text{CPU Execution Time} = \text{Instructions count} * \text{CPI} * \text{Clock Cycle Time}$$CPU Execution Time=Instructions count∗CPI∗Clock Cycle Time

where:

$$\text{Instructions Count} \quad \text{CPI} \quad \text{Clock Cycle Time} = = = \frac{\text{Instructions}}{\text{Programs}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Seconds}}{\text{Cycle}}$$Instructions Count=InstructionsProgramsCPI=CyclesInstructionClock Cycle Time=SecondsCycle

Note:

- CPI is somewhat artificial (since it is computed from the other numbers) but it seems to be intuitive and useful.
- Use dynamic instruction count (#instructions executed), not static (#instructions in compiled code)

Performance

Performance is the response time distributed through the request lifetime.

Busy vs Wait

The busy vs. wait percentage shown in monitoring tool is generally:

$$\text{CPU Busy vs Wait Event} = \text{User} + \text{Kernel}\text{Wallclock Time}$$CPU Busy vs Wait Event=User+KernelWallclock Time

CPU Performance

$$\text{CPU Performance} = 1\text{Total CPU Time}$$CPU Performance=1Total CPU Time

System Performance

$$\text{System Performance} = 1\text{Wallclock Time}$$System Performance=1Wallclock Time

Units (MHz to ns)

- Rate is often measured in MHz (millions of cycles per second)
- Time is often measured in ns (nanoseconds)

$$\text{X MHz } 500 \text{ MHz} = \approx 1000\text{X ns}2 \text{ ns clock}$$X MHz =1000X ns500 MHz≈2 ns clock

---

Source: DataCademia, https://datacadamia.com/counter/resource/system/cpu/time

Microprocessor Design and Performance

This article gives more information on computer performance, including some helpful definitions.

Clock Cycles

The clock signal is a 1-bit signal that oscillates between a "1" and a "0" with a certain frequency. When the clock transitions from a "0" to a "1" it is called the **positive edge**, and when the clock transitions from a "1" to a "0" it is called the **negative edge**.

The time it takes to go from one positive edge to the next positive edge is known as the **clock period**, and represents one **clock cycle**.

The number of clock cycles that can fit in 1 second is called the **clock frequency**. To get the clock frequency, we can use the following formula:

$$\text{Clock Frequency} = 1\text{Clock Period}$$Clock Frequency=1Clock Period

Clock frequency is measured in units of *cycles per second*.

Cycles per Instruction

In many microprocessor designs, it is common for multiple clock cycles to transpire while performing a single instruction. For this reason, it is frequently useful to keep a count of how many cycles are required to perform a single instruction. This number is known as the **cycles per instruction**, or CPI of the processor.

Because all processors may operate using a different CPI, it is not possible to accurately compare multiple processors simply by comparing the clock frequencies. It is more useful to compare the number of **instructions per second**, which can be calculated as such:

$$\text{Instructions per Second} = \frac{\text{Clock Frequency}}{\text{CPI}}$$

Instructions per Second=Clock FrequencyCPI

One of the most common units of measure in modern processors is the "MIPS", which stands for *millions of instructions per second*. A processor with 5 MIPS can perform 5 million instructions every second. Another common metric is "FLOPS", which stands for *floating point operations per second*. MFLOPS is a million FLOPS, GFLOPS is a billion FLOPS, and TFLOPS is a trillion FLOPS.

Instruction count

The "instruction count" in microprocessor performance measurement is the number of instructions executed during the run of a program. Typical benchmark programs have instruction counts in the millions or billions -- even though the program itself may be very short, those benchmarks have inner loops that are repeated millions of times.

Some microprocessor designers have the freedom to add instructions to or remove instructions from the instruction set. Typically the only way to reduce the instruction count is to add instructions such that those inner loops can be re-written in a way that does the necessary work using fewer instructions -- those instructions do "more work" per instruction.

Sometimes, counter-intuitively, we can improve overall CPU performance (i.e., reduce CPU time) in a way that increases the instruction count, by using instructions in that inner loop that may do "less work" per instruction, but those instructions finish in less time.

CPU Time

CPU Time is the amount of time it takes the CPU to complete a particular program. CPU time is a function of the amount of time it takes to complete instructions, and the number of instructions in the program:

$$\text{CPU time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$CPU time=Instruction Count×CPI×Clock Cycle Time

Sometimes we can improve one of the 3 components alone, reducing CPU time. But quite often we find a tradeoff -- say, a technique that increases instruction count, but reduces the clock cycle time -- and we have to measure the total CPU time to see if that technique makes the overall performance better or worse.

Performance

Amdahls Law

Amdahl's Law is a law concerned with computer performance and optimization. Amdahl's law states that an improvement in the speed of a single processor component will have a comparatively small effect on the performance of the overall processor unit.

In the most general sense, Amdahl's Law can be stated mathematically as follows:

$$\Delta = 1 \sum_{k=0}^{n} \left(P_k S_k \right)$$Δ=1∑nk=0(PkSk)

where:

- Δ is the factor by which the program is sped up or slowed down,
- P_k is a percentage of the instructions that can be improved (or slowed),
- S_k is the speed-up multiplier (where 1 is no speed-up and no slowing),
- k represents a label for each different percentage and speed-up, and
- n is the number of different speed-up/slow-downs resulting from the system change.

For instance, if we make a speed improvement in the memory module, only the instructions that deal directly with the memory module will experience a speedup. In this case, the percentage of load and store instructions in our program will be P_0, and the factor by which those instructions are sped up will be S_0. All other instructions, which are not affected by the memory unit will be P_1, and the speed up will be S_1 Where:

$$P_1 = 1 - P_0$$P1=1−P0

$$S_1 = 1$$S1=1

We set S_1 to 1 because those instructions are not sped up or slowed down by the change to the memory unit.

Source: Wikibooks, https://en.wikibooks.org/wiki/Microprocessor_Design/Performance

Computing Benchmarks

This article discusses benchmarks, which help measure computer performance.

In computing, a **benchmark** is the act of running a computer program, a set of programs, or other operations, in order to assess the relative **performance** of an object, normally by running a number of standard tests and trials against it. The term *benchmark* is also commonly utilized for the purposes of elaborately designed benchmarking programs themselves.

Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or database management systems (DBMS).

Benchmarks provide a method of comparing the performance of various subsystems across different chip/system architectures.

Test suites are a type of system intended to assess the **correctness** of software.

Purpose

As computer architecture advanced, it became more difficult to compare the performance of various computer systems simply by looking at their specifications. Therefore, tests were developed that allowed comparison of different architectures. For example, Pentium 4 processors generally operated at a higher clock frequency than Athlon XP or PowerPC processors, which did not necessarily translate to more computational power; a processor with a slower clock frequency might perform as well as or even better than a processor operating at a higher frequency. See BogoMips and the megahertz myth.

Benchmarks are designed to mimic a particular type of workload on a component or system. Synthetic benchmarks do this by specially created programs that impose the workload on the component. Application benchmarks run real-world programs on the system. While application benchmarks usually give a much better measure of real-world performance on a given system, synthetic benchmarks are useful for testing individual components, like a hard disk or networking device.

Benchmarks are particularly important in CPU design, giving processor architects the ability to measure and make tradeoffs in microarchitectural decisions. For example, if a benchmark extracts the key algorithms of an application, it will contain the performance-sensitive aspects of that application. Running this much smaller snippet on a cycle-accurate simulator can give clues on how to improve performance.

Prior to 2000, computer and microprocessor architects used SPEC to do this, although SPEC's Unix-based benchmarks were quite lengthy and thus unwieldy to use intact.

Computer manufacturers are known to configure their systems to give unrealistically high performance on benchmark tests that are not replicated in real usage. For instance, during the 1980s some compilers could detect a specific mathematical operation used in a well-known floating-point benchmark and replace the operation with a faster mathematically equivalent operation. However, such a transformation was rarely useful outside the benchmark until the mid-1990s, when RISC and VLIW architectures emphasized the importance of compiler technology as it related to performance. Benchmarks are now regularly used by compiler companies to improve not only their own benchmark scores, but real application performance.

CPUs that have many execution units – such as a superscalar CPU, a VLIW CPU, or a reconfigurable computing CPU – typically have slower clock rates than a sequential CPU with one or two execution units when built from transistors that are just as fast. Nevertheless, CPUs with many execution units often complete real-world and benchmark tasks in less time than the supposedly faster high-clock-rate CPU.

Given the large number of benchmarks available, a manufacturer can usually find at least one benchmark that shows its system will outperform another system; the other systems can be shown to excel with a different benchmark.

Manufacturers commonly report only those benchmarks (or aspects of benchmarks) that show their products in the best light. They also have been known to mis-represent the significance of benchmarks, again to show their products in the best possible light. Taken together, these practices are called *bench-marketing*.

Ideally benchmarks should only substitute for real applications if the application is unavailable, or too difficult or costly to port to a specific processor or computer system. If performance is critical, the only benchmark that matters is the target environment's application suite.

Challenges

Benchmarking is not easy and often involves several iterative rounds in order to arrive at predictable, useful conclusions. Interpretation of benchmarking data is also extraordinarily difficult. Here is a partial list of common challenges:

- Vendors tend to tune their products specifically for industry-standard benchmarks. Norton SysInfo (SI) is particularly easy to tune for, since it mainly biased toward the speed of multiple operations. Use extreme caution in interpreting such results.

- Some vendors have been accused of "cheating" at benchmarks – doing things that give much higher benchmark numbers, but make things worse on the actual likely workload.
- Many benchmarks focus entirely on the speed of computational performance, neglecting other important features of a computer system, such as:
    - Qualities of service, aside from raw performance. Examples of unmeasured qualities of service include security, availability, reliability, execution integrity, serviceability, scalability (especially the ability to quickly and nondisruptively add or reallocate capacity), etc. There are often real trade-offs between and among these qualities of service, and all are important in business computing. Transaction Processing Performance Council Benchmark specifications partially address these concerns by specifying ACID property tests, database scalability rules, and service level requirements.
    - In general, benchmarks do not measure Total cost of ownership. Transaction Processing Performance Council Benchmark specifications partially address this concern by specifying that a price/performance metric must be reported in addition to a raw performance metric, using a simplified TCO formula. However, the costs are necessarily only partial, and vendors have been known to price specifically (and only) for the benchmark, designing a highly specific "benchmark special" configuration with an artificially low price. Even a tiny deviation from the benchmark package results in a much higher price in real world experience.
    - Facilities burden (space, power, and cooling). When more power is used, a portable system will have a shorter battery life and require recharging more often. A server that consumes more power and/or space may not be able to fit within existing data center resource constraints, including cooling limitations. There are real trade-offs as most semiconductors require more power to switch faster. See also performance per watt.
    - In some embedded systems, where memory is a significant cost, better code density can significantly reduce costs.
- Vendor benchmarks tend to ignore requirements for development, test, and disaster recovery computing capacity. Vendors only like to report what might be narrowly required for production capacity in order to make their initial acquisition price seem as low as possible.
- Benchmarks are having trouble adapting to widely distributed servers, particularly those with extra sensitivity to network topologies. The emergence of grid computing, in particular, complicates benchmarking since some workloads are "grid friendly", while others are not.
- Users can have very different perceptions of performance than benchmarks may suggest. In particular, users appreciate predictability – servers that always meet or exceed service level agreements. Benchmarks tend to emphasize mean scores (IT perspective), rather than maximum worst-case response times (real-time computing perspective), or low standard deviations (user perspective).

- Many server architectures degrade dramatically at high (near 100%) levels of usage – "fall off a cliff" – and benchmarks should (but often do not) take that factor into account. Vendors, in particular, tend to publish server benchmarks at continuous at about 80% usage – an unrealistic situation – and do not document what happens to the overall system when demand spikes beyond that level.
- Many benchmarks focus on one application, or even one application tier, to the exclusion of other applications. Most data centers are now implementing virtualization extensively for a variety of reasons, and benchmarking is still catching up to that reality where multiple applications and application tiers are concurrently running on consolidated servers.
- There are few (if any) high quality benchmarks that help measure the performance of batch computing, especially high volume concurrent batch and online computing. Batch computing tends to be much more focused on the predictability of completing long-running tasks correctly before deadlines, such as end of month or end of fiscal year. Many important core business processes are batch-oriented and probably always will be, such as billing.
- Benchmarking institutions often disregard or do not follow basic scientific method. This includes, but is not limited to: small sample size, lack of variable control, and the limited repeatability of results.

Benchmarking Principles

There are seven vital characteristics for benchmarks. These key properties are:

1. Relevance: Benchmarks should measure relatively vital features.
2. Representativeness: Benchmark performance metrics should be broadly accepted by industry and academia.
3. Equity: All systems should be fairly compared.
4. Repeatability: Benchmark results can be verified.
5. Cost-effectiveness: Benchmark tests are economical.
6. Scalability: Benchmark tests should work across systems possessing a range of resources from low to high.
7. Transparency: Benchmark metrics should be easy to understand.

Types of benchmark

1. Real program
    - word processing software
    - tool software of CAD
    - user's application software (i.e.: MIS)
2. Component Benchmark / Microbenchmark
    - core routine consists of a relatively small and specific piece of code.
    - measure performance of a computer's basic components

    - may be used for automatic detection of computer's hardware parameters like number of registers, cache size, memory latency, etc.
3. Kernel
    - contains key codes
    - normally abstracted from actual program
    - popular kernel: Livermore loop
    - linpack benchmark (contains basic linear algebra subroutine written in FORTRAN language)
    - results are represented in Mflop/s.
4. Synthetic Benchmark
    - Procedure for programming synthetic benchmark:
        - take statistics of all types of operations from many application programs
        - get proportion of each operation
        - write program based on the proportion above
    - Types of Synthetic Benchmark are:
        - Whetstone
        - Dhrystone
    - These were the first general purpose industry standard computer benchmarks. They do not necessarily obtain high scores on modern pipelined computers.
5. I/O benchmarks
6. Database benchmarks
    - measure the throughput and response times of database management systems (DBMS)
7. Parallel benchmarks
    - used on machines with multiple cores and/or processors, or systems consisting of multiple machines

Common benchmarks

Industry standard (audited and verifiable)

- Business Applications Performance Corporation (BAPCo)
- Embedded Microprocessor Benchmark Consortium (EEMBC)
- Linked Data Benchmark Council (LDBC)
    - Semantic Publishing Benchmark (SPB): an LDBC benchmark inspired by the Media/Publishing industry for testing the performance of RDF engines
    - Social Network Benchmark (SNB): an LDBC benchmark for testing the performance of RDF engines consisting of three distinct benchmarks (Interactive Workload, Business Intelligence Workload, Graph Analytics Workload) on a common dataset
- Standard Performance Evaluation Corporation (SPEC), in particular their SPECint and SPECfp

- Transaction Processing Performance Council (TPC): DBMS benchmarks
    - TPC-A: measures performance in update-intensive database environments typical in on-line transaction processing (OLTP) applications
    - TPC-C: an on-line transaction processing (OLTP) benchmark
    - TPC-H: a decision support benchmark

Open source benchmarks

- AIM Multiuser Benchmark – composed of a list of tests that could be mixed to create a 'load mix' that would simulate a specific computer function on any UNIX-type OS.
- Bonnie++ – filesystem and hard drive benchmark
- BRL-CAD – cross-platform architecture-agnostic benchmark suite based on multithreaded ray tracing performance; baselined against a VAX-11/780; and used since 1984 for evaluating relative CPU performance, compiler differences, optimization levels, coherency, architecture differences, and operating system differences.
- Collective Knowledge – customizable, cross-platform framework to crowdsource benchmarking and optimization of user workloads (such as deep learning) across hardware provided by volunteers
- Coremark – Embedded computing benchmark
- Data Storage Benchmark – an RDF continuation of the LDBC Social Network Benchmark, from the Hobbit Project
- DEISA Benchmark Suite – scientific HPC applications benchmark
- Dhrystone – integer arithmetic performance, often reported in DMIPS (Dhrystone millions of instructions per second)
- DiskSpd – Command-line tool for storage benchmarking that generates a variety of requests against computer files, partitions or storage devices
- Embench™ - portable, open-source benchmarks, for benchmarking deeply embedded systems; they assume the presence of no OS, minimal C library support and, in particular, no output stream. Embench is a project of the Free and Open Source Silicon Foundation.
- Faceted Browsing Benchmark – benchmarks systems that support browsing through linked data by iterative transitions performed by an intelligent user, from the Hobbit Project
- Fhourstones – an integer benchmark
- HINT – designed to measure overall CPU and memory performance
- Iometer – I/O subsystem measurement and characterization tool for single and clustered systems.
- IOzone – Filesystem benchmark
- Kubestone – Benchmarking Operator for Kubernetes and OpenShift
- LINPACK benchmarks – traditionally used to measure FLOPS
- Livermore loops

- NAS parallel benchmarks
- NBench – synthetic benchmark suite measuring performance of integer arithmetic, memory operations, and floating-point arithmetic
- PAL – a benchmark for realtime physics engines
- PerfKitBenchmarker – A set of benchmarks to measure and compare cloud offerings.
- Phoronix Test Suite – open-source cross-platform benchmarking suite for Linux, OpenSolaris, FreeBSD, OSX and Windows. It includes a number of other benchmarks included on this page to simplify execution.
- POV-Ray – 3D render
- Tak (function) – a simple benchmark used to test recursion performance
- TATP Benchmark – Telecommunication Application Transaction Processing Benchmark
- TPoX – An XML transaction processing benchmark for XML databases
- VUP (VAX unit of performance) – also called VAX MIPS
- Whetstone – floating-point arithmetic performance, often reported in millions of Whetstone instructions per second (MWIPS)

Microsoft Windows benchmarks

- BAPCo: MobileMark, SYSmark, WebMark
- CrystalDiskMark
- Futuremark: 3DMark, PCMark
- PassMark Software Pty Ltd
- PiFast
- SuperPrime
- Super PI
- UserBenchmark
- Whetstone
- Windows System Assessment Tool, included with Windows Vista and later releases, providing an index for consumers to rate their systems easily
- Worldbench (discontinued)

Others

- AnTuTu – commonly used on phones and ARM-based devices.
- Berlin SPARQL Benchmark (BSBM) – defines a suite of benchmarks for comparing the performance of storage systems that expose SPARQL endpoints via the SPARQL protocol across architectures
- Geekbench – A cross-platform benchmark for Windows, Linux, macOS, iOS and Android.
- iCOMP – the Intel comparative microprocessor performance, published by Intel

- Khornerstone
- Lehigh University Benchmark (LUBM) – facilitates the evaluation of Semantic Web repositories via extensional queries over a large data set that commits to a single realistic ontology
- Performance Rating – modeling scheme used by AMD and Cyrix to reflect the relative performance usually compared to competing products.
- SunSpider – a browser speed test
- VMmark – a virtualization benchmark suite.
- RenderStats – a 3D rendering benchmark database.

Source: Wikipedia, https://en.wikipedia.org/wiki/Benchmark_(computing)

Amdahl's Law

This article introduces Amdahl's law, which can be used to measure how certain improvements to performance can be measured. A general form of Amdahl's law is: (that part of the time that is improved/the improvement factor) plus that part of the time that is not improved equals the new improved time. Speedup factor is the old time divided by the new time. Be sure to study the examples.

In computer architecture, **Amdahl's law** (or **Amdahl's argument**) is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example, if a program needs 20 hours to complete using a single thread, but a one hour portion of the program cannot be parallelized, therefore only the remaining 19 hours (p = 0.95) of execution time can be parallelized, then regardless of how many threads are devoted to a parallelized execution of this program, the minimum execution time cannot be less than one hour. Hence, the theoretical speedup is limited to at most 20 times the single thread performance, $(11-p=20)$(11−p=20).

The theoretical speedup of the latency of the execution of a program as a function of the number of processors executing it, according to Amdahl's law. The speedup is limited by the serial part of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

Definition

Amdahl's law can be formulated in the following way:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where

- S_{latency} is the theoretical speedup of the execution of the whole task;
- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

Furthermore,

$$\left\{\begin{array}{l}\end{array}\right.S_{\text{latency}}(s)\leq 1-p\lim_{s\to\infty}S_{\text{latency}}(s)=1-p.$$

{Slatency(s)≤1−plims→∞Slatency(s)=1−p.

shows that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless of the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

Amdahl's law applies only to the cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, Gustafson's law gives a less pessimistic and more realistic assessment of the parallel performance.

Derivation

A task executed by a system whose resources are improved compared to an initial similar system can be split up into two parts:

- a part that does not benefit from the improvement of the resources of the system;
- a part that benefits from the improvement of the resources of the system.

An example is a computer program that processes files from disk. A part of that program may scan the directory of the disk and create a list of files internally in memory. After that, another part of the program passes each file to a separate thread for processing. The part that scans the directory and creates the file list cannot be sped up on a parallel computer, but the part that processes the files can.

The execution time of the whole task before the improvement of the resources of the system is denoted as TT. It includes the execution time of the part that would not benefit from the improvement of the resources and the execution time of the one that would benefit from it. The fraction of the execution time of the task that would benefit from the improvement of the resources is denoted by pp. The one concerning the part that would not benefit from it is therefore $1-p$1−p. Then:

$$T=(1-p)T+pT$$

T=(1−p)T+pT

It is the execution of the part that benefits from the improvement of the resources that is accelerated by the factor ss after the improvement of the resources. Consequently, the execution time of the part that does not benefit from it remains the same, while the part that benefits from it becomes:

$$_{ps}T$$

psT

The theoretical execution time $T(s)$T(s) of the whole task after the improvement of the resources is then:

$$T(s)=(1-p)T+psTT(s)=(1-p)T+psT$$

Amdahl's law gives the theoretical speedup in latency of the execution of the whole task *at fixed workload* WW, which yields

$$S_{latency}(s)=TWT(s)W=TT(s)=11-p+psSlatency(s)=TWT(s)W=TT(s)=11-p+ps$$

Parallel programs

If 30% of the execution time may be the subject of a speedup, *p* will be 0.3; if the improvement makes the affected part twice as fast, *s* will be 2. Amdahl's law states that the overall speedup of applying the improvement will be:

$$S_{latency}=11-p+ps=11-0.3+0.32=1.18Slatency=11-p+ps=11-0.3+0.32=1.18$$

For example, assume that we are given a serial task which is split into four consecutive parts, whose percentages of execution time are p_1 = 0.11, p_2 = 0.18, p_3 = 0.23, and p_4 = 0.48 respectively. Then we are told that the 1st part is not sped up, so s_1 = 1, while the 2nd part is sped up 5 times, so s_2 = 5, the 3rd part is sped up 20 times, so s_3 = 20, and the 4th part is sped up 1.6 times, so s_4 = 1.6. By using Amdahl's law, the overall speedup is

$$S_{latency}=1p1s1+p2s2+p3s3+p4s4=10.111+0.185+0.2320+0.481.6=2.19Slatency=1p1s1+p2s2+p3s3+p4s4=10.111+0.185+0.2320+0.481.6=2.19$$

Notice how the 5 times and 20 times speedup on the 2nd and 3rd parts respectively don't have much effect on the overall speedup when the 4th part (48% of the execution time) is accelerated by only 1.6 times.

Serial programs

Assume that a task has two independent parts, A and B. Part B takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this reduces the time of the whole computation only slightly. In contrast, one may need to perform less work to make part A perform twice as fast. This will make the computation much faster than by optimizing part B, even though part B's speedup is greater in terms of the ratio, (5 times versus 2 times).

For example, with a serial program in two parts *A* and *B* for which T_A = 3 s and T_B = 1 s,

- if part *B* is made to run 5 times faster, that is s = 5 and $p = T_B/(T_A + T_B)$ = 0.25, then

  $$S_{\text{latency}} = \frac{1}{1 - 0.25 + \frac{0.25}{5}} = 1.25$$

- if part *A* is made to run 2 times faster, that is s = 2 and $p = T_A/(T_A + T_B)$ = 0.75, then

  $$S_{\text{latency}} = \frac{1}{1 - 0.75 + \frac{0.75}{2}} = 1.60$$

Therefore, making part *A* to run 2 times faster is better than making part *B* to run 5 times faster. The percentage improvement in speed can be calculated as

$$\text{percentage improvement} = 100\left(1 - \frac{1}{S_{\text{latency}}}\right)$$

- Improving part *A* by a factor of 2 will increase overall program speed by a factor of 1.60, which makes it 37.5% faster than the original computation.
- However, improving part *B* by a factor of 5, which presumably requires more effort, will achieve an overall speedup factor of 1.25 only, which makes it 20% faster.

Optimizing the sequential part of parallel programs

If the non-parallelizable part is optimized by a factor of O, then

$$T(O,s)=(1-p)TO+psT.T(O,s)=(1-p)TO+psT.$$

It follows from Amdahl's law that the speedup due to parallelism is given by

$$S_{latency}(O,s)=T(O)T(O,s)=(1-p)_{1O}+p_{1-pO+ps}Slatency(O,s)=T(O)T(O,s)=(1-p)1O+p1-pO+ps$$

When $s=1$s=1, we have $S_{latency}(O,s)=1$Slatency(O,s)=1, meaning that the speedup is measured with respect to the execution time after the non-parallelizable part is optimized.

When $s=\infty$s=∞,

$$S_{latency}(O,\infty)=T(O)T(O,s)=(1-p)_{1O}+p_{1-pO+ps}=1+p1-pOSlatency(O,\infty)=T(O)T(O,s)=(1-p)1O+p1-pO+ps=1+p1-pO$$

If $1-p=0.4$1−p=0.4, $O=2$O=2, and $s=5$s=5, then:

$$S_{latency}(O,s)=T(O)T(O,s)=0.4_{12}+0.6_{0.42+0.65}=2.5Slatency(O,s)=T(O)T(O,s)=0.412+0.60.42+0.65=2.5$$

Transforming sequential parts of parallel programs into parallelizable

Next, we consider the case wherein the non-parallelizable part is reduced by a factor of O'O′, and the parallelizable part is correspondingly increased. Then

$$T'(O',s)=1-pO'T+(1-1-pO')TsT′(O′,s)=1−pO′T+(1−1−pO′)Ts$$

It follows from Amdahl's law that the speedup due to parallelism is given by

$$S'_{latency}(O',s)=T'(O')T'(O',s)=1_{1-pO'}+(1-1-pO')_{1s}Slatency′(O′,s)=T′(O′)T′(O′,s)=11−pO′+(1−1−pO′)1s$$

The derivation above is in agreement with Jakob Jenkov's analysis of the execution time vs. speedup tradeoff.

Relation to the law of diminishing returns

Amdahl's law is often conflated with the law of diminishing returns, whereas only a special case of applying Amdahl's law demonstrates law of diminishing returns. If one picks optimally (in terms of the achieved speedup) what to improve, then one will see monotonically decreasing improvements as one improves. If, however, one picks non-

optimally, after improving a sub-optimal component and moving on to improve a more optimal component, one can see an increase in the return. Note that it is often rational to improve a system in an order that is "non-optimal" in this sense, given that some improvements are more difficult or require larger development time than others.

Amdahl's law does represent the law of diminishing returns if on considering what sort of return one gets by adding more processors to a machine, if one is running a fixed-size computation that will use all available processors to their capacity. Each new processor added to the system will add less usable power than the previous one. Each time one doubles the number of processors the speedup ratio will diminish, as the total throughput heads toward the limit of $1/(1 - p)$.

This analysis neglects other potential bottlenecks such as memory bandwidth and I/O bandwidth. If these resources do not scale with the number of processors, then merely adding processors provides even lower returns.

An implication of Amdahl's law is that to speedup real applications which have both serial and parallel portions, heterogeneous computing techniques are required. For example, a CPU-GPU heterogeneous processor may provide higher performance and energy efficiency than a CPU-only or GPU-only processor.

Source: Wikipedia, https://en.wikipedia.org/wiki/Amdahl%27s_law

1.4: The Power Problem

The Need For A Radical New Type Of Computer Architecture

This article is about the challenges facing computer architecture in building more powerful computers for high performance applications and for faster, cheaper, more efficient computers for IT applications. The author is responding to another article on extreme-scale computing, which you may read by following the link if you desire.

Irving Wladawsky-Berger is a former chief strategist at IBM and he has also worked as GM of IBM's supercomputer group.

In the post *Extreme Scale Computing* he explains the challenges that supercomputers face in reaching the next stage: 1,000 times more powerful than current petaflop (1 followed by 15 zeros) supercomputers.

The challenge in getting to this next stage of 'exascale computing,' is power consumption and heat generation. And that means we will need a new type of architecture.

Massively parallel architectures, using tens to hundreds of thousands of processors from the PC and Unix markets have dominated supercomputing over the past twenty years. They got us into the terascale and petascale ranges. But, they will not get us to exascale.

Mr Wladawsky-Berger says:

"Another massive technology and architectural transition now looms for supercomputing and the IT industry in general."

The reason that this is not just a supercomputer problem, but also one for the IT industry, is because of the rise in importance of cloud computing.

The technology requirements are quite similar, especially the need for low power, low cost components. They also share similar requirements for highly efficient, autonomic system management. One can actually view cloud-based systems as a kind of exascale class supercomputers designed to support embarrassingly parallel workloads, such as massive information analysis or huge numbers of sensors and mobile devices.

If we can build exascale computing supercomputers, we will also be able to build smaller systems that use the same basic technologies and that will have many business applications, not just in cloud computing.

But there is a big problem: the power consumed, and the heat generated by computing components -- tremendous numbers of computing components.

...the only way to now increase performance toward an exascale system is massive parallelism, an exaflop supercomputer might have 100s of millions of processing elements or cores. Such massive parallelism will require major innovations in the architecture, software and applications for exascale systems.

A future exascale computer will likely make use of low power consuming processors used in consumer mobile devices. (Possibly chips such as Apple's A4 chip, which offers ten hours of battery life in the iPad.)

If we can crack the formidable challenges of processors, software architectures, and how to develop applications designed for extreme parallel processing, we will be very close to cracking some very large computational problems that currently are out of reach, from "economics and medicine to business and government."

Fortunately, the Obama administration has recognized the need for exascale computing and has listed it in its Strategy for American Innovation.

You can read the full article here.

Source: Tom Foremski, https://www.siliconvalleywatcher.com/the-need-for-a-radical-new-type-of-computer-architecture/

1.5: The Switch to Parallel Processing

Parallel Computing Landscape

Watch this lecture for an understanding of the reasons behind the switch to parallel computing. This lecture provides motivation, insight into thinking about computer architecture, and an explanation computing trends.

Source: Stanford University, https://www.youtube.com/watch?v=On-k-E5HpcQ

1.6: Case Study: A Recent Intel Processor

Data Types, Operators, and Variables

The beginning of the lecture is administrative, so you may skip to around 16:14. It introduces the concept of computational thinking. While the course is an introduction to programming, computational thinking applies to both software (via programming) and to hardware (via computer architecture). Pay particular attention to the computer block diagram, the program counter and the way instructions relate to the computer hardware.

Source: Eric Grimson and John Guttag, Massachusetts Institute of Technology, https://www.youtube.com/watch?v=k6U-i4gXkLM

Unit 2: Instructions: Hardware Language

In order to understand computer architecture, you need to understand the components that comprise a computer and their interconnections. Sets of instructions, called programs, describe the computations that computers carry out. The instructions are strings of binary digits. When symbols are used for the binary strings, the instructions are called assembly language instructions. Components interpret the instructions and send signals to other components that cause the instruction to be carried out.

In this unit, you will build on your knowledge of programming from *CS102: Introduction to Computer Science II* to learn how to program with an assembly language. You will use the instructions of a real processor, MIPS, to understand the basics of hardware language. We will also discuss the different classes of instructions typically found in computers and compare the MIPS instructions to those found in other popular processors made by Intel and ARM.

- Upon successful completion of this unit, you will be able to:
    - draw the block diagram for the interface between hardware and software, and show how software instructs hardware to accomplish a simple program statement;
    - convert an integer into its binary and other representations;
    - convert a decimal number into its floating point representation;

- explain how programs written in high-level programming languages, such as C or Java, can be translated into the language of the hardware; and show the process for a simple C program statement; and
- write a simple MIPS assembly language program.

- 2.1: Computer Hardware Operations

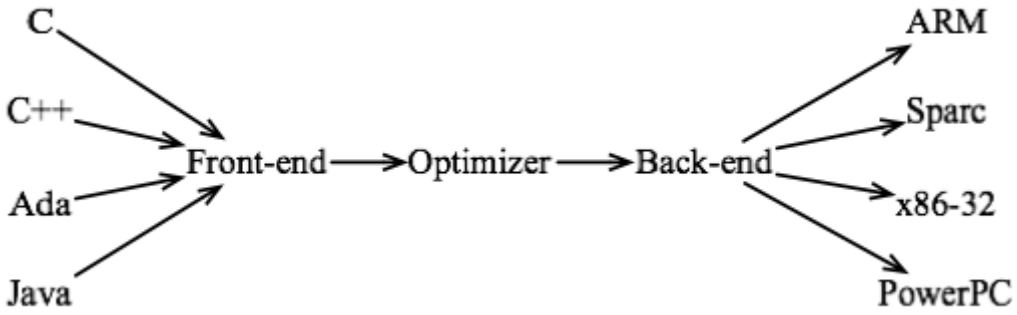
Introduction to Programming Languages

Read this article to see how a program is edited, compiled (or assembled), linked, and executed in the computer.

There are different ways in which we can execute
programs. Compilers, interpreters and virtual machines are some tools that we can use to accomplish this task. All these tools provide a way to simulate in hardware the semantics of a program. Although these different technologies exist with the same core purpose - to execute programs - they do it in very different ways. They all have advantages and disadvantages, and in this chapter we will look more carefully into these trade-offs. Before we continue, one important point must be made: in principle any programming language can be compiled or interpreted. However, some execution strategies are more natural in some languages than in others.

Compiled Programs

Compilers are computer programs that translate a high-level programming language to a low-level programming language. The product of a compiler is an executable file, which is made of instructions encoded in a specific machine code. Hence, an executable program is specific to a type of computer architecture. Compilers designed for distinct programming languages might be quite different; nevertheless, they all tend to have the overall macro-architecture described in the figure below:



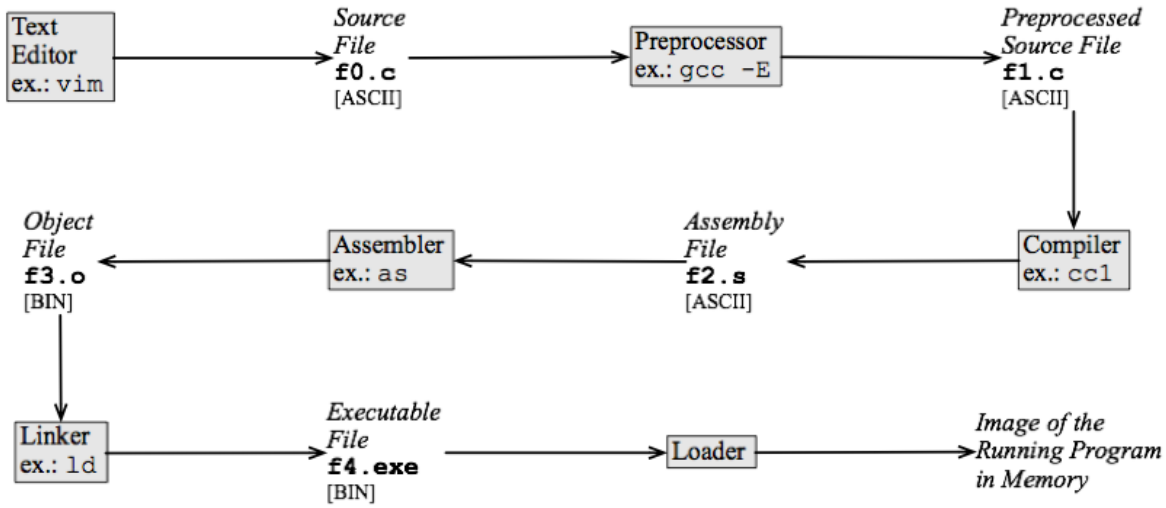
A compiler has a front end, which is the module in charge of transforming a program, written in a high-level source language into an intermediate representation that the compiler will process in the next phases. It is in the front end that we have the parsing of the input program, as we have seen in the last two chapters. Some compilers, such

as gcc can parse several different input languages. In this case, the compiler has a different front end for each language that it can handle. A compiler also has a back end, which does code generation. If the compiler can target many different computer architectures, then it will have a different back-end for each of them. Finally, compilers generally do some code optimization. In other words, they try to improve the program, given a particular criterion of efficiency, such as speed, space or energy consumption. In general the optimizer is not allowed to change the semantics of the input program.

The main advantage of execution via compilation is speed. Because the source program is translated directly to machine code, this program will most likely be faster than if it were interpreted. Nevertheless, as we will see in the next section, it is still possible, although unlikely, that an interpreted program run faster than its machine code equivalent. The main disadvantage of execution by compilation is portability. A compiled program targets a specific computer architecture, and will not be able to run in a different hardware.

The life cycle of a compiled program

A typical C program, compiled by gcc, for instance, will go through many transformations before being executed in hardware. This process is similar to a production line in which the output of a stage becomes the input to the next stage. In the end, the final product, an executable program, is generated. This long chain is usually invisible to the programmer. Nowadays, integrated development environments (IDE) combine the several tools that are part of the compilation process into a single execution environment. However, to demonstrate how a compiler works, we will show the phases present in the execution of a standard C file compiled with gcc. These phases, their products and some examples of tools are illustrated in the figure below.



The aim of the steps seen above is to translate a source file to a code that a computer can run. First of all, the programmer uses a text editor to create a source file, which

contains a program written in a high-level programming language. In this example, we are assuming C. There exist every sort of text editor that can be used here. Some of them provide supporting in the form of syntax highlighting or an integrated debugger, for instance. Lets assume that we have just edited the following file, which we want to compile:

```
#define CUBE(x) (x)*(x)*(x)
intmain(){
 inti=0;
 intx=2;
 intsum=0;
 while(i++<100){
  sum+=CUBE(x);
 }
 printf("The sum is %d\n",sum);
}
```

After editing the C file, a preprocessor is used to expand the macros present in the source code. Macro expansion is a relatively simple task in C, but it can be quite complicated in languages such as lisp, for instance, which take care of avoiding typical problems of macro expansion such as variable capture. During the expansion phase, the body of the macro replaces every occurrence of its name in the program's source code. We can invoke gcc's preprocessor via a command such as

`gcc -E f0.c -o f1.c`. The result of preprocessing our example program is the code below. Notice that the call `CUBE(x)` has been replaced by the expression `(x)*(x)*(x)`.

```
intmain(){
 inti=0;
 intx=2;
 intsum=0;
 while(i++<100){
  sum+=(x)*(x)*(x);
 }
 printf("The sum is %d\n",sum);
}
```

In the next phase we convert the source program into assembly code. This phase is what we normally call compilation: a text written in the C grammar will be converted into a program written in the x86 assembly grammar. It is during this step that we perform the parsing of the C program. In Linux we can translate the source file, e.g., `f1.c` to assembly via the command `cc1 f1.c -o f2.s`, assuming that `cc1` is the system's compiler. This command is equivalent to the call `gcc -S f1.c -o f2.s`. The assembly program can be seen in the left side of the figure below. This program is written in the assembly language used in the x86 architecture. There are many different computer architectures, such as ARM, PowerPC and Alpha. The assembly language produced for any of them would be rather different than the program below. For comparison purposes, we have printed the ARM version of the same program at the right side of the figure. These two assembly languages follow very different design philosophies: x86 uses a CISC instruction set, while ARM follows more closely the RISC approach. Nevertheless, both files, the x86's and the ARM's have a similar syntactic skeleton. The assembly language has a linear structure: a program is a list-like sequence of instructions. On the other hand, the C language has a

syntactic structure that looks more like a tree. Because of this syntactic gap, this phase contains the most complex translation step that the program will experiment during its life cycle.

```
# Assembly of x86 # Assembly of ARM
 .cstring _main:
LC0: @ BB#0:
 .ascii "The sum is %d\12\0" push{r7, lr}
 .text movr7, sp
.globl _main subsp, sp, #16
_main: movr1, #2
 pushl %ebp movr0, #0
 movl %esp, %ebp strr0, [r7, #-4]
 subl $40, %esp strr0, [sp, #8]
 movl $0, -20(%ebp) stmsp, {r0, r1}
 movl $2, -16(%ebp) bLBB0_2
 movl $0, -12(%ebp) LBB0_1:
 jmp L2 ldrr0, [sp, #4]
L3: ldrr3, [sp]
 movl -16(%ebp), %eax mulr1, r0, r0
 imull -16(%ebp), %eax mlar2, r1, r0, r3
 imull -16(%ebp), %eax strr2, [sp]
 addl %eax, -12(%ebp) LBB0_2:
L2: ldrr0, [sp, #8]
 cmpl $99, -20(%ebp) addr1, r0, #1
 setle %al cmpr0, #99
 addl $1, -20(%ebp) strr1, [sp, #8]
 testb %al, %al bleLBB0_1
 jne L3 @ BB#3:
 movl -12(%ebp), %eax ldrr0, LCPI0_0
 movl %eax, 4(%esp) ldrr1, [sp]
 movl $LC0, (%esp) LPC0_0:
 call _printf addr0, pc, r0
 leave bl_printf
 ret ldrr0, [r7, #-4]
 movsp, r7
 pop{r7, lr}
 movpc, lr
```

It is during the translation from the high-level language to the assembly language that the compiler might apply code optimizations. These optimizations must obey the semantics of the source program. An optimized program should do the same thing as its original version. Nowadays compilers are very good at changing the program in such a way that it becomes more efficient. For instance, a combination of two well-known optimizations, loop unwinding and constant propagation can optimize our example program to the point that the loop is completely removed. As an example, we can run the optimizer using the following command, assuming again that `cc1` is the default compiler that gcc uses: `cc1 -O1 f1.c -o f2.opt.s`. The final program that we produce this time, `f2.opt.s` is surprisingly concise:

```
 .cstring
LC0:
 .ascii "The sum is %d\12\0"
 .text
.globl _main
_main:
```

```
pushl%ebp
movl%esp, %ebp
subl$24, %esp
movl$800, 4(%esp)
movl$LC0, (%esp)
call_printf
leave
ret
```

The next step in the compilation chain consists in the translation of the assembly language to binary code. The assembly program is still readable by people. The binary program, also called an object file can, of course, be read by human beings, but there are not many human beings who are up to this task these days. Translating from assembly to binary code is a rather simple task, because both these languages have the same syntactic structure. Only their lexical structure differs. Whereas the assembly file is written with ASCII [w:Assembly_language#Opcode_mnemonics_and_extended_mnemonics|[mnemonics]], the binary file contains sequences of zeros and ones that the hardware processor recognizes. A typical tool used in this phase is the `as` assembler. We can produce an object file with the command below `as f2.s -o f3.o`.

The object file is not executable yet. It does not contain enough information to specify where to find the implementation of the printf function, for example. In the next step of the compilation process we change this file so that the address of functions defined in an external libraries be visible. Each operating system provides programmers with a number of libraries that can be used together with code that they create. A special software, the linker can find the address of functions in these libraries, thus fixing the blank addresses in the object file. Different operating systems use different linkers. A typical tool, in this case, is `ld` or `collect2`. For instance, in order to produce the executable program in a Mac OS running Leopard, we can use the command `collect2 -o f4.exe -lcrt1.10.5.o f3.o -lSystem`.

At this point we almost have an executable file, but our linked binary program is bound to suffer a last transformation before we can see its output. All the addresses in the binary code are relative. We must replace these addresses by absolute values, which point correctly to the targets of the function calls and other program objects. This last step is the responsibility of a program called loader. The loader dumps an image of the program into memory and runs it.

Source:
Wikibooks, https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Compiled_Programs

Machine Code

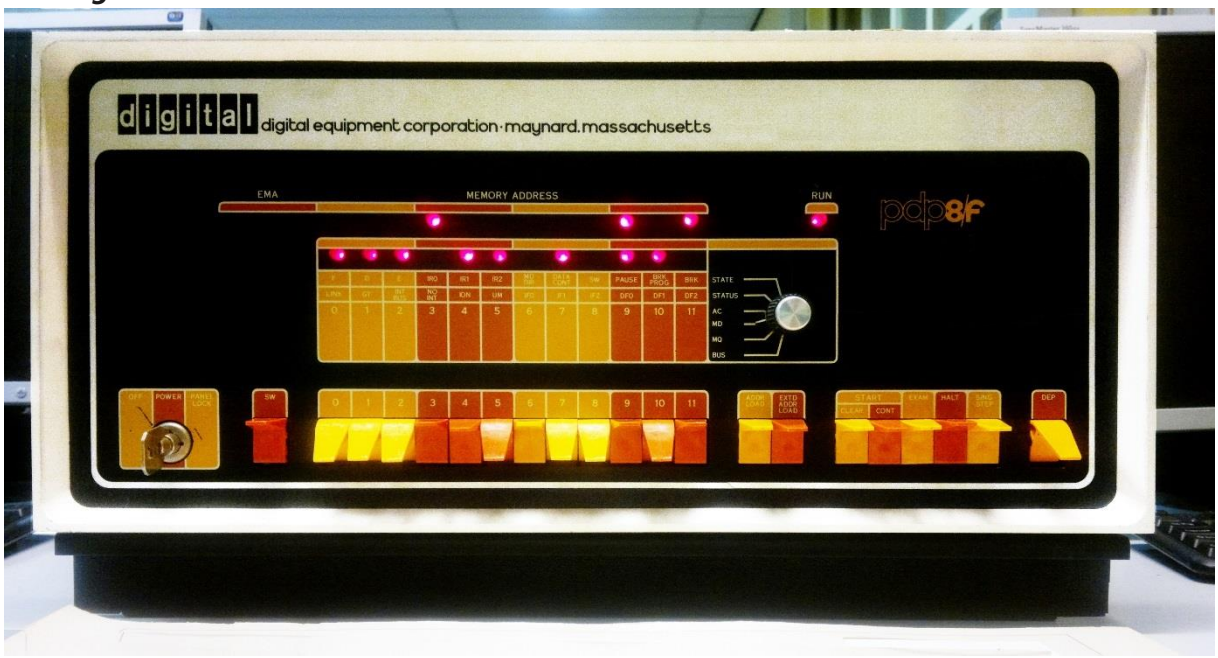
Read this article, which discusses how a MIPS assembly language statement is assembled into machine code. Carefully study the example in the article.

Machine code is a computer program written in **machine language.** It uses the instruction set of a particular computer architecture. It is usually written in binary. Machine code is the lowest level of software. Other programming languages are translated into machine code so the computer can execute them.

An instruction tells the process what operation to perform. Each instruction is made up of an opcode (operation code) and operand(s). The operands are usually memory addresses or data. An instruction set is a list of the opcodes available for a computer. Machine code is what assembly code and other programming languages are compiled to or interpreted as.

Program builders turn code into another language or machine code. Machine code is sometimes called **native code**. This is used when talking about things that work on only some computers.

Writing machine code



Front panel of an early minicomputer, with switches for entering machine code

Machine code can be written in different forms:

- Using a number of switches. This generates a sequence of *1* and *0*. This was used in the early days of computing. Since the 1970s, it is no longer used.
- Using a Hex editor. This allows the use of opcodes instead of the number of the command.
- Using an Assembler. Assembly languages are simpler than opcodes. Their syntax is easier to understand than machine language but harder than high level

languages. The assembler will translate the source code into machine code on its own.
- Using a High-level programming language allows programs that use code that is easier to read and write. These programs are translated into machine code. The translation can happen in many steps. Java programs are first optimized into bytecode. Then it is translated into machine language when it is used.

Typical instructions of machine code

There are many kinds of instructions found usually found in an instruction set:

- Arithmetical operations: Addition, subtraction, multiplication, division.
- Logical operations: Conjunction, disjunction, negation.
- Operations acting on single bits: Shifting bits to the left or right.
- Operations acting on memory: copying a value from one register to another.
- Operations that compare two values: bigger than, smaller than, equal.
- Operations that combine other operations: add, compare, and copy if equal to some value(as one operation), jump to some point in the program if a register is zero.
- Operations that act on program flow: jump to some address.
- Operations that convert data types: e.g. convert a 32-bit integer to a 64-bit integer, convert a floating point value to an integer (by truncating).

Many modern processors use microcode for some of the commands. More complex commands tend to use it. This is often done with CISC architectures.

Instructions

Every processor or processor family has its own instruction set. Instructions are patterns of bits that correspond to different commands that can be given to the machine. Thus, the instruction set is specific to a class of processors using (mostly) the same architecture.

Newer processor designs often include all the instructions of a predecessor and may add additional instructions. Sometimes, a newer design will discontinue or alter the meaning of an instruction code (typically because it is needed for new purposes), affecting code compatibility; even nearly completely compatible processors may show slightly different behavior for some instructions, but this is rarely a problem.

Systems may also differ in other details, such as memory arrangement, operating systems, or peripheral devices. Because a program normally relies on such factors, different systems will typically not run the same machine code, even when the same type of processor is used.

Most instructions have one or more opcode fields. They specify the basic instruction type. Other fields may give the type of the operands, the addressing mode, and so on. There may also be special instructions that are contained in the opcode itself. These instructions are called *immediates*.

Processor designs can be different in other ways. Different instructions can have different lengths. Also, they can have the same length. Having all instructions have the same length can simplify the design.

Example

The MIPS architecture has instructions which are 32 bits long. This section has examples of code. The general type of instruction is in the *op* (operation) field. It is the highest 6 bits. J-type (jump) and I-type (immediate) instructions are fully given by *op*. R-type (register) instructions include the field *funct*. It determines the exact operation of the code.

The fields used in these types are:

```
  6    5    5    5      5       6 bits
[ op | rs | rt | rd | shamt | funct ] R-type
[ op | rs | rt |   address/immediate] I-type
[ op |        target address        ] J-type
```

rs, *rt*, and *rd* indicate register operands. *shamt* gives a shift amount.
The *address* or *immediate* fields contain an operand directly.

Example: add the registers 1 and 2. Place the result in register 6. It is encoded:

```
[   op   |   rs   |   rt   |   rd  |shamt| funct]
    0        1        2        6      0      32    decimal
 000000    00001    00010    00110  00000  100000  binary
```

Load a value into register 8. Take it from the memory cell 68 cells after the location listed in register 3:

```
[   op   |   rs   |   rt   | address/immediate]
    35       3        8             68          decimal
  100011   00011    01000    00000 00001 000100 binary
```

Jump to the address 1024:

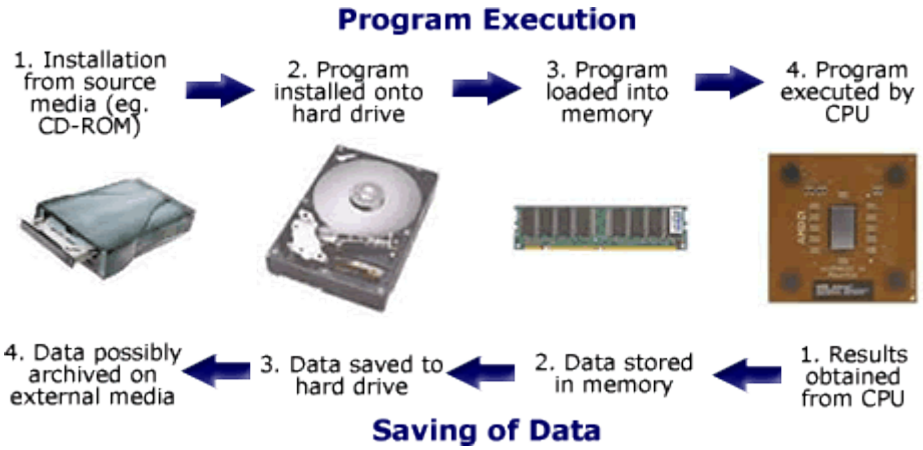
```
[   op  |         target address      ]
    2                 1024                decimal
  000010 00000 00000 00000 10000 000000   binary
```

Source: Wikipedia, https://simple.wikipedia.org/wiki/Machine_code

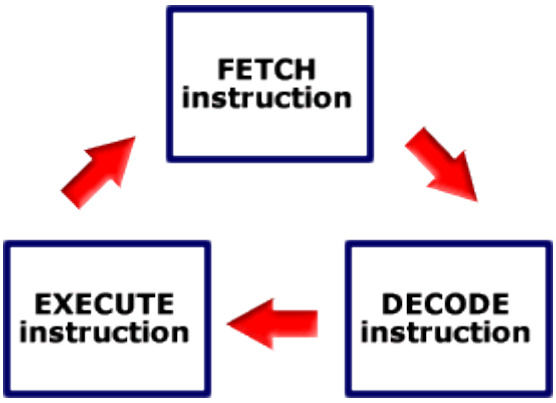
The Machine Cycle

Read this article about how a computer actually executes the machine code produced by the compiler (assembler). Registers are simply small memory units that are usually 8, 16, 32, or more bits long. The program counter and instruction register are used in the machine cycle.

When software is installed onto a computer (by downloading it from the Internet or installing off a disk), the program and any associated files are stored in secondary memory. The program's code is stored as a series of bits that represent machine instructions. The code remains there until the user chooses to execute the program in question, on which point sections of the code are loaded into the computer's memory.



To actually run the code, the processor then needs to retrieve instructions one by one from memory so it can run them. This process consists of three stages: fetching the instruction, decoding the instruction, and executing the instruction - these three steps are known as the **machine cycle**. A processor spends all of its time in this cycle, endlessly retrieving the next instruction, decoding it, and running it.



- **Fetch**
  In the fetch cycle, the control unit looks at the program counter register (PC) to get the memory address of the next instruction. It then requests this instruction from main memory and places it in the instruction register (IR).

- **Decode**
  Here, the control unit checks the instruction that is now stored within the instruction register (IR). It looks at the instruction - which is just a sequence of 0s and 1s and decides what needs to be done. Does the instruction say to add two numbers? Does it say to load a value from memory? Where in memory? The control unit interprets the binary instruction to answer questions like these.

- **Execute**
  Now the control unit sends the signals that tell the ALU, memory, and other components signals to cause them to perform the correct work.

The video below demonstrates a simple program running on a computer. This imaginary computer shown uses special registers to hold memory address (MAR) and data that just came in from memory (MBR). It also uses registers called AL and BL to hold values temporarily. Don't worry about the details of those other registers, focus on the fetch/decode/execute cycle and how the PC and IR are used.

Materials on this page adapted with permission from *Microprocessor Tutorial* by Matthew Eastaugh

---

Source: Andrew
Scholer, http://computerscience.chemeketa.edu/cs160Reader/ComputerArchitecture/MachineCycle.html

2.2: Number Representation in Computers

Introduction to Number Systems

Read this introduction to number systems.

Number Systems

Our number system is a western adaptation of the Hindu-Arabic numeral system developed somewhere between the first and fourth centuries AD. However, numbers have been recorded with tally marks throughout history. The Ishango Bone from Africa is about 25,000 years old. It's the lower leg bone from a baboon, and contains tally marks. We know the marks were used for counting because they appear in distinct groups.

This reindeer antler from France is about 15,000 years old, and also shows clearly grouped tally marks.

Of course, we still use tally marks today!

Base ten numbers (the ones you have probably been using your whole life), and base *b* numbers (the ones you've been learning about in this chapter) are both positional number systems.

Definition

A **positional number system** is one way of writing numbers. It has unique symbols for 1 through $b - 1$, where b is the base of the system. Modern positional number systems also include a symbol for 0.

The **positional value** of each symbol depends on its position in the number:

- The positional value of a symbol in the first position is just its face value.
- The positional value of a symbol in the second position is b times its value.
- The positional value of a symbol in the third position is times its value.
- And so on.

The value of a number is the sum of the positional values of its digits.

Definition

In an **additive number system,** the value of a written number is the sum of the face values of the symbols that make up the number. The only symbol necessary for an additive number system is a symbol for 1, however many additive number systems contain other symbols.

History: Roman numerals

The ancient Romans used a version of an additive number systems. The Romans represented numbers this way:

number	Roman Numeral
1	I
5	V
10	X
50	L
100	C
500	D
1,000	M

So the number 2013 would be represented as MMXIII. This is read as 2,000 (two M's), one ten (one X), and three ones (three I's).

For any additive number system very large numbers become impractical to write. To represent the number one million in Roman numerals it would take one thousand M's!

However, the Roman numerals did have one efficiency advantage: The order of the symbols mattered. If a symbol to the left was smaller than the symbol to the right, it would be subtracted instead of added. So for example nine is represented as IX rather than VIIII.

Think / Pair / Share
If you don't already know how to use Roman numerals, research it a little bit. Then answer these questions.

- Write the numbers 1–20 in Roman numerals.
- What is the maximum number of symbols needed to write any number between 1 and 1,000 in Roman Numerals? Justify your answer.

The earliest positional number systems are attributed to the Babylonians (base 60) and the Mayans (base 20). These positional systems were both developed before they had a symbol or a clear concept for zero. Instead of using 0, a blank space was used to indicate skipping a particular place value. This could lead to ambiguity.

Suppose we didn't have a symbol for 0, and someone wrote the number

2 3

It would be impossible to tell if they mean 23, 203, 2003, or maybe two separate numbers (two and three).

Leonardo Pisano Bigollo, more commonly known as **Fibonacci**, played a pivotal role in guiding Europe out of a long period in which the importance and development of math was in marked decline. He was born in Italy around 1170 CE to Guglielmo Bonacci, a successful merchant. Guglielmo brought his son with him to what is now Algeria, and Leonardo was educated in mathematics mathematics there.

Fibonacci

At the time, Roman Numerals dominated Europe, and the official means of calculations was the abacus. Muḥammad ibn Mūsā al-Khwārizmī described the use of Hindu-Arabic system in his book *On the Calculation with Hindu Numerals* in 825 CE, but it was not well-known in Europe.

Statue of al-Khwarizmi at Amirkabir University of Technology

Fibonacci's book *Liber Abaci* described the Hindu-Arabic system and its business applications for a European readership. His book was well-received throughout Europe, and it marked the beginning of a reawakening of European mathematics.

History: Hawaiian numbers

The Hindu-Arabic number system is now used nearly exclusively throughout the globe. But many cultures had their own number systems before contact and trade with other countries spread the work of al-Khwārizmī throughout the world.

There is evidence that pre-contact Hawaiians actually used two different number systems. Depending on what they were counting, they might use base 4 instead (or a mixed base-10 and base-4 system). One theory is that certain objects (fish, taro, etc.) were often put

in bundles of 4, so were more natural to count by 4's than by 10's. The number four also had spiritual significance in Hawaiian culture.

Humans have 5 fingers on each hand, making base ten a natural choice for counting. But there are 4 gaps between the fingers, meaning that a hand can carry four fish or taro plants or similar objects, making base four a natural choice for some cultures.

In the mixed base system, instead of powers of 10, numbers are broken down into sums of numbers that look like 4 times a power of 10 (40, 400, 4000, etc.).

1	ʻekahi
2	ʻelua
3	ʻekolu
4	ʻehā (or kauna)
5	ʻelima
6	ʻeono
7	ʻehiku
8	ʻewalu
9	ʻeiwa
10	ʻumi
11–19	ʻumi kumamā {kahi, lua, kolu, hā, etc.}
20	iwakālua
21–29	Iwakālua kumamā {kahi, lua, kolu, hā, etc.}
30	kanakolu

31–39	kanakolu kumamā {kahi, lua, etc.}
40	kanahā
400	lau
4,000	mano
40,000	kini
400,000	lehu

Here are a few examples (refer to the table above for the Hawaiian names of the numbers):

Example
ʻekolu kini, ʻewalu lau me ʻekahi

translates to three 40,000ʻs, eight 400ʻs, and one;

$3 \cdot 40000 + 8 \cdot 400 + 1 = 123201$

Example
$5207 = 1 \cdot 4000 + 3 \cdot 400 + 7$

would be ʻekahi mano, ʻekolu lau me ʻehiku

On Your Own
Work on the following exercises on your own or with a partner.

1. Translate this Hawaiian number to English and then write it in base ten.

ʻekahi kanahā me kanakolu kumamāiwa

2. Translate this base-ten number to Hawaiian.

1,573

Source: Michelle Manes, https://pressbooks-dev.oer.hawaii.edu/math111/chapter/number-systems/

More on Number Systems

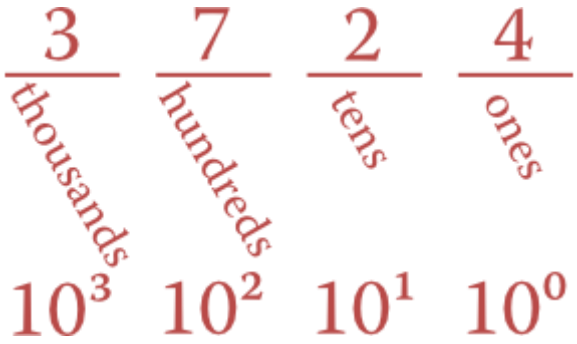
Read this article on how numbers can be represented in 1's and 0's that the computer can understand. Study the way numbers can be converted into other representations, like binary, 2's complement, and so on. Also be sure to watch the videos, including the one on floating point representation. Study the examples of binary addition to make sure you understand how it works.

Number systems and binary

Here are some informal notes on number systems and binary numbers.

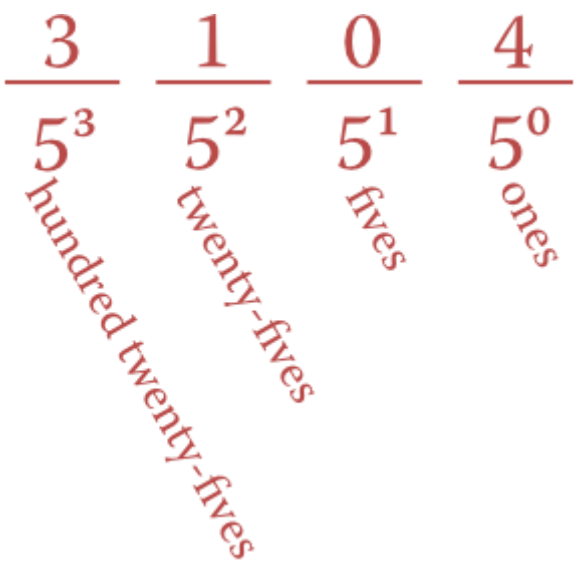
Positional numbering system

Our normal number system is a positional system, where the position (column) of a digit represents its value. Starting from the right, we have the ones column, tens column, hundreds, thousands, and so on. Thus the number 3724 stands for three THOUSAND, seven HUNDRED, two TENS (called twenty), and four ONES.



The values of those columns derive from the powers of ten, which is then called the **base** of the number system. The base ten number system is also called **decimal.**

There is nothing special about base ten, except that it's what you learned from a young age. A positional numbering system can use any quantity as its base. Let's take, for example, base five. In base five, the columns represent the quantities (from right to left) one, five, twenty-five, and a hundred twenty-five. We need to use five symbols to indicate quantities from zero up to four. For simplicity, let's keep the same numerals we know: 0, 1, 2, 3, and 4.



The number shown in this figure, 3104 in base five, represents the same quantity that we usually write as 404 in base ten. That's because it is three × one hundred twenty-five (= 375), plus one × twenty-five (= 25) plus four ones (= 4), so 375 + 25 + 4 = 404.

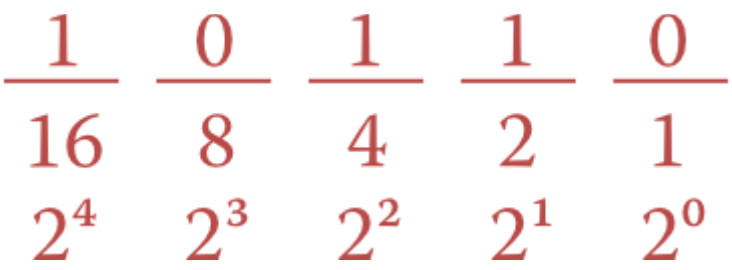
You can count directly in base five; it looks like this: 0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31, 32, 33, 34, 40, 41, 42, 43, 44, 100. (Those correspond to quantities from zero to twenty-five.)

You should try counting and converting between other bases. Below is an interesting video overview of the **dozenal** (base twelve) number system.

Binary numbers

Computer systems use **binary** numbers – that just means they are in base **two.** Using two as the base is really convenient and flexible, because we need only two 'symbols' and there are so many ways we can represent them: zero/one, on/off, up/down, high/low, positive/negative, etc.

In binary, the columns are (from right to left) 1, 2, 4, 8, 16, 32, and so on. Using a zero means we exclude that column's quantity, and a one means we include it.



So the binary number 10110 is the quantity 16 + 4 + 2 = 22. Each binary digit (a one or a zero) is called a **bit.** The largest five-bit binary number, then is 11111 = 16 + 8 + 4 + 2 + 1 = 31.

It's worthwhile to learn to count in binary, at least from zero to fifteen:

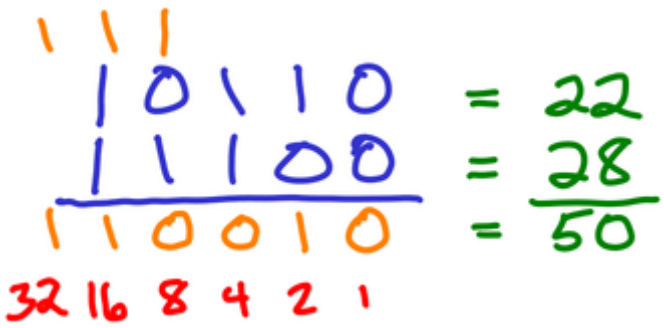
```
0000 = 0    0100 = 4    1000 = 8    1100 = 12
0001 = 1    0101 = 5    1001 = 9    1101 = 13
0010 = 2    0110 = 6    1010 = 10   1110 = 14
0011 = 3    0111 = 7    1011 = 11   1111 = 15
```

Binary arithmetic

It's relatively easy to add numbers directly in binary. Line up the columns and then proceed from right to left, as usual. There are only four possible cases:

- If a column has no ones, write a zero below.
- If a column has one one, write a one below.
- If a column has two ones, write a zero and carry a one to the next column (to the left).
- Finally, if a column has three ones (possible due to an incoming carry), write a one and carry a one to the next column.

Below is an example of adding 10110 plus 11100. The result is 110010, and you can see the carry bits above the original numbers, in orange.



When adding this way, it's always a good idea to check your work by converting the numbers to decimal and checking the addition. In this case, we're adding 22 (10110) to 28 (11100) to get 50 (110010).

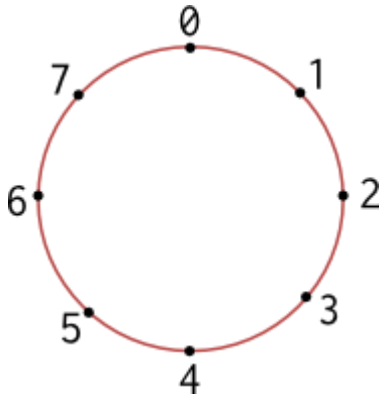
Fixed-size binary numbers

We generally arrange numbers along a line, that goes off to infinity. Indeed, in binary we can always continue counting by adding more and more columns that are powers of two.

However, in most computer systems and programs we use **fixed-size** numbers. That is, we decide in advance how many bits will be used to represent the number. For example,

a **32-bit computer** represents most of its numbers and addresses using 32 bits. The largest such number is $2^{32}-1$ = 4,294,967,295.

When your numbers have a fixed number of bits, then there is no number line heading off into infinity. Instead, we arrange the numbers around a circle, like a clock. Below is the **number wheel** for 3-bit integers. The smallest 3-bit integer is zero, and the largest is seven. Then, if you attempt to keep counting, it just wraps around to zero again.



When you perform arithmetic with fixed-size numbers, you throw away any extra carry bit; the result cannot exceed the designated size. For example, see what happens if we try to add 110 + 011 using 3-bit integers:

In 3-bit arithmetic, 6 plus 3 is 1. You can make sense of this on the number wheel. Addition corresponds to walking clock-wise around the wheel. So start at 6, and go clockwise by 3. That lands on 1, which is 6+3.

Below is a video about fixed-size binary numbers in old video games.

Signed magnitude

Now we'll look at **signed** numbers – that is, numbers that can be positive or negative. There are two techniques for encoding signed numbers. The first one is called **signed magnitude.** It appears simple at first, but that simplicity hides some awkward properties.

Here's how it works. We use a fixed width, and then the left-most bit represents the sign. So 4-bit signed magnitude looks like this:

```
 ___ ___ ___ ___
sign  4   2   1
```

where having the sign bit set to '1' means the magnitude is interpreted as negative. Thus, `0110` is `+6` whereas `1110` is `-6`. In this system, the largest positive number is `0111 = +7` and the most negative number is `1111 = -7`.

One of the unfortunate effects of this representation is there are two ways to write zero: `0000` and also `1000`. There is no such thing as negative zero, so this doesn't really make sense.

Two's complement

The second way to represent signed quantities is called **two's complement.** Although this looks trickier at first, it actually works really well. Below is the interpretation of 4-bit two's complement. All we need to do compared to normal unsigned numbers is negate the value of the left-most bit.

```
___ ___ ___ ___
-8   4   2   1
```

So `+6` is `0110` as before, but what about `-6`? We need to turn on the negative 8, and then add two: `1010`. To represent `-1`, you turn on all the bits: `1111`, because that produces `-8+4+2+1 = -8+7 = -1`.

The nice thing about two's complement is that you can add these numbers and everything just works out. Let's try adding `7` and `-3`:

```
0 1 1 1 =  7
1 1 0 1 = -3
--------- ----
0 1 0 0 =  4
```

It's also relatively easy to **negate** a number – that is, to go from `+6` to `-6` or from `-3` to `+3`. Here are the steps:

1. First, flip all the bits. That is, all the zeroes become ones and all the ones become zeroes.
2. Next, add one.

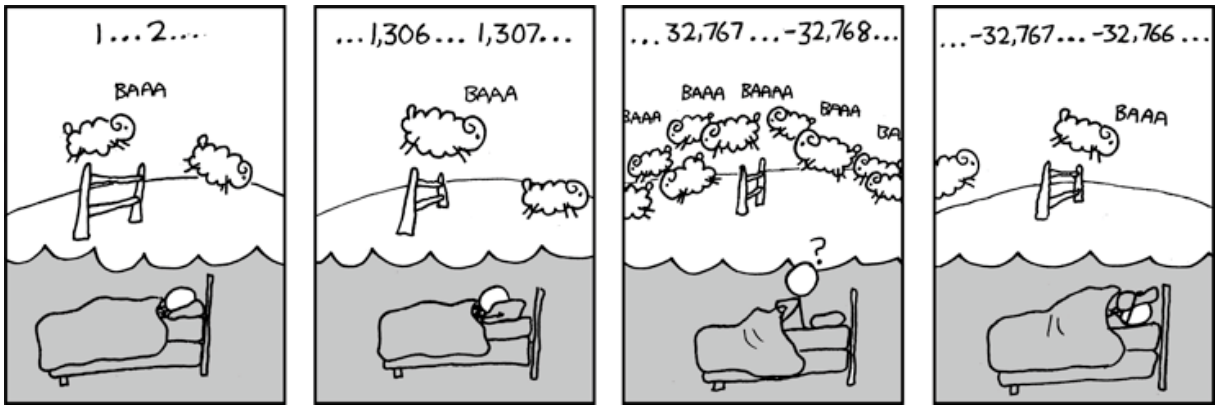
For example here is how we produce `-6` from `+6`:

```
0 1 1 0 = + 6
1 0 0 1   (flip all the bits)
+ 1       (add one)
---------
1 0 1 0 = - 6
```

You don't even have to reverse these steps in order to convert back:

```
1 0 1 0 = - 6
0 1 0 1   (flip all the bits)
+ 1       (add one)
---------
0 1 1 0 = + 6
```

Here's a cartoon from XKCD about counting sheep using two's complement!



How many bits is this person using?

Octal and hexadecimal

Finally, I want to introduce two number systems that are very useful as **abbreviations** for binary. They work so well because their bases are powers of two.

Octal is base eight, so we use the symbols 0–7 and the values of the columns are:

```
___ ___ ___ ___
512  64  8   1
8³   8²  8¹  8⁰
```

The real value of octal, however, is that **each octal digit maps to exactly three binary digits.** So, on octal number like `3714` maps as shown:

```
  3       7       1       4      octal number
0 1 1   1 1 1   0 0 1   1 0 0    binary number
(4 2 1  4 2 1   4 2 1   4 2 1)
```

Hexadecimal is base sixteen, so we use the symbols 0–9 and then A to represent ten, B for eleven, and so on up to F for fifteen. The values of the columns are:

```
____ ____ ____ ____
4096 256   16   1
 16³ 16²   16¹  16⁰
```

So a hexadecimal number like `2A5C` has the value `2×4096 + 10×256 + 5×16 + 12×1 = 10844` in base ten.

In hexadecimal, **each digit maps to exactly four bits.** So here is that same number in binary:

```
   2         A         5         C
 0 0 1 0   1 0 1 0   0 1 0 1   1 1 0 0
(8 4 2 1   8 4 2 1   8 4 2 1   8 4 2 1)
```

Below are two great video overviews of hexadecimal. (**Note** when you watch these – the Brits often pronounce zero as 'naught'.)

Practice problems

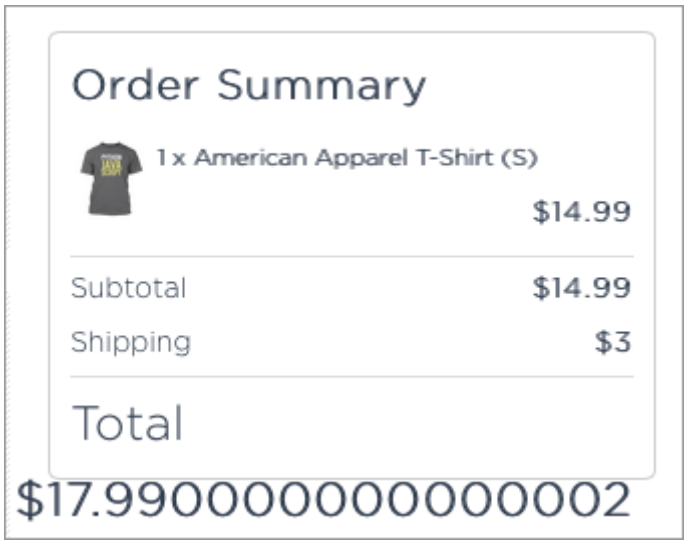
1. Convert the following base ten (decimal) numbers into binary.
    1. 6
    2. 18
    3. 51
    4. 63
2. Convert the following unsigned binary numbers into base ten.
    1. 1010
    2. 1101
    3. 1000
    4. 10001
3. What do all **odd** numbers have in common, when written in binary? (Hint: try writing the quantities 3, 5, 7, 9, 11 in binary.)
4. Using 7-bit signed (two's complement) binary numbers, what is the largest positive number? What is the most negative number?
5. Convert the following 5-bit **signed** (two's complement) binary numbers into base ten.
    1. 01101
    2. 01111
    3. 10011
    4. 11111
6. Convert the following 16-bit binary number into hexadecimal, and then into octal.

    `0 1 1 1 1 1 1 1 0 0 1 1 1 0 1 0`
7. Convert the following hexadecimal numbers into binary:
    1. 9D
    2. C4
    3. A17E
8. Add and verify the following **unsigned** binary numbers.

```
9.   1 0 1 1 1 1        1 1 0 1 1 1
10.+   1 1 1 0 1      + 1 0 0 1 0 0
     ---------------    ---------------
```

Solutions here

Extra: floating-point



@jaffathecake on Twitter

Source: Christopher League, https://liucs.net/cs101s14/n1-binary.html

Floating Points

Read this article on the representation of real numbers. Read up to the section labeled "Addition and subtraction."

In computing, **floating-point arithmetic** (**FP**) is arithmetic using formulaic representation of real numbers as an approximation to support a trade-off between range and precision. For this reason, floating-point computation is often found in systems which include very small and very large real numbers, which require fast processing times. A number is, in general, represented approximately to a fixed number of significant digits (the significand) and scaled using an exponent in some fixed base; the base for the scaling is normally two, ten, or sixteen. A number that can be represented exactly is of the following form:

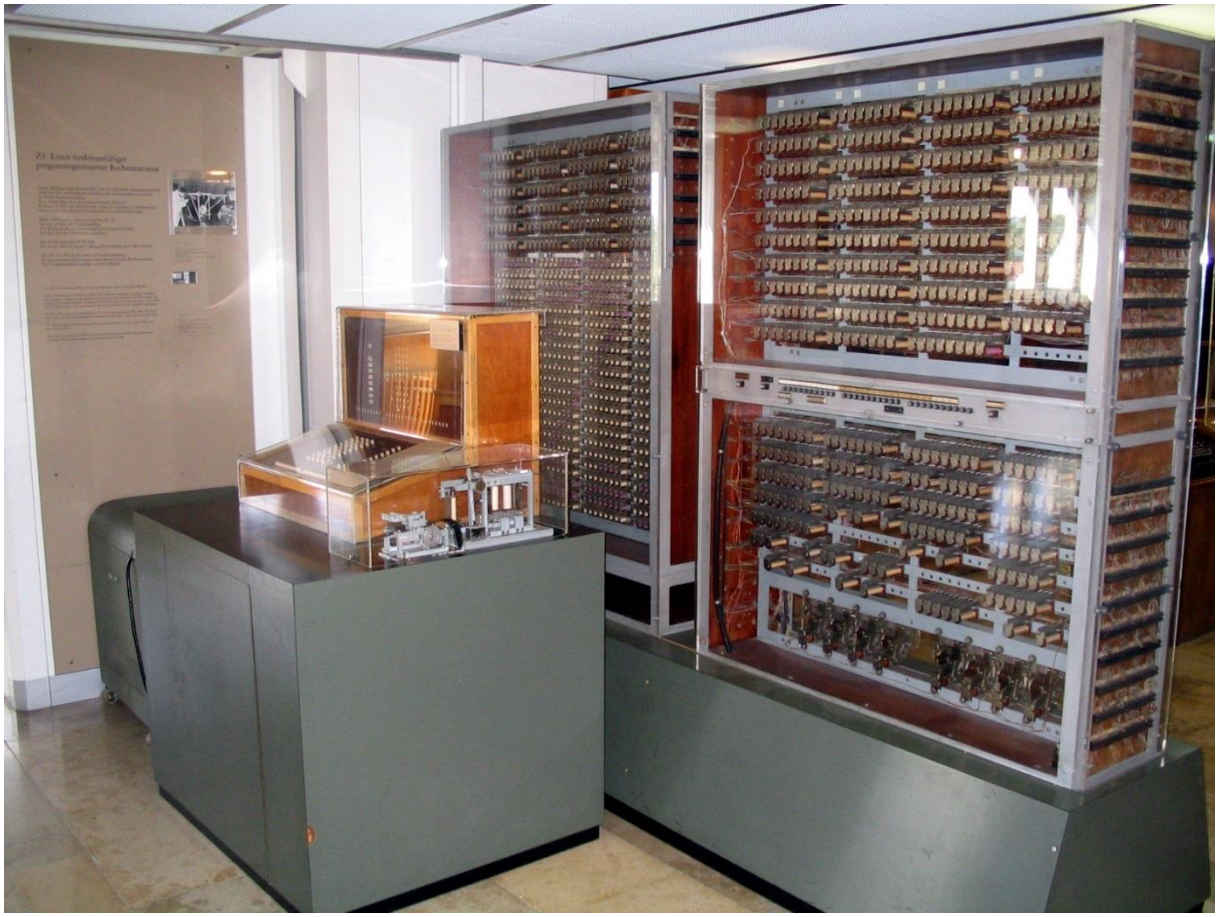
$$\text{significand} \times \text{base}^{\text{exponent}}, \text{significand} \times \text{base}^{\text{exponent}},$$

where significand is an integer, base is an integer greater than or equal to two, and exponent is also an integer. For example:

$$1.2345 = 12345 \underbrace{}_{\text{significand}} \times 10 \underbrace{}_{\text{base}} ^{-4 \underbrace{}_{\text{exponent}}}. 1.2345 = 12345 \\ \text{significand} \times 10 \\ \text{base} ^{-4} \\ \text{exponent}.$$

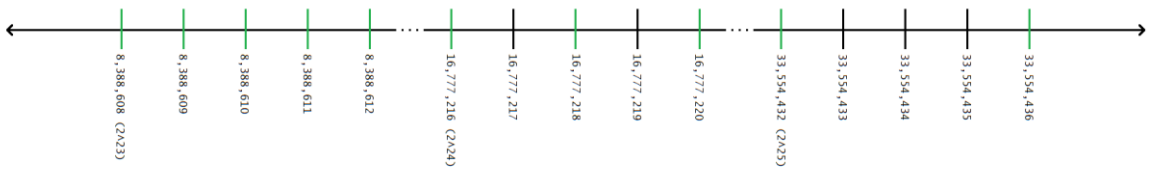
The term *floating point* refers to the fact that a number's radix point (*decimal point*, or, more commonly in computers, *binary point*) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent

component, and thus the floating-point representation can be thought of as a kind of scientific notation.



An early electromechanical programmable computer, the Z3, included floating-point arithmetic (replica on display at Deutsches Museum in Munich).

A floating-point system can be used to represent, with a fixed number of digits, numbers of different orders of magnitude: e.g. the distance between galaxies or the diameter of an atomic nucleus can be expressed with the same unit of length. The result of this dynamic range is that the numbers that can be represented are not uniformly spaced; the difference between two consecutive representable numbers varies with the chosen scale.



Single-precision floating point numbers: the green lines mark representable values.

Over the years, a variety of floating-point representations have been used in computers. In 1985, the IEEE 754 Standard for Floating-Point Arithmetic was established, and since the 1990s, the most commonly encountered representations are those defined by the IEEE.

The speed of floating-point operations, commonly measured in terms of FLOPS, is an important characteristic of a computer system, especially for applications that involve intensive mathematical calculations.

A floating-point unit (FPU, colloquially a math coprocessor) is a part of a computer system specially designed to carry out operations on floating-point numbers.

Overview

Floating-point numbers

A number representation specifies some way of encoding a number, usually as a string of digits.

There are several mechanisms by which strings of digits can represent numbers. In common mathematical notation, the digit string can be of any length, and the location of the radix point is indicated by placing an explicit "point" character (dot or comma) there. If the radix point is not specified, then the string implicitly represents an integer and the unstated radix point would be off the right-hand end of the string, next to the least significant digit. In fixed-point systems, a position in the string is specified for the radix point. So a fixed-point scheme might be to use a string of 8 decimal digits with the decimal point in the middle, whereby "00012345" would represent 0001.2345.

In scientific notation, the given number is scaled by a power of 10, so that it lies within a certain range—typically between 1 and 10, with the radix point appearing immediately after the first digit. The scaling factor, as a power of ten, is then indicated separately at the end of the number. For example, the orbital period of Jupiter's moon Io is 152,853.5047 seconds, a value that would be represented in standard-form scientific notation as 1.528535047×10^{5} seconds.

Floating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of:

- A signed (meaning positive or negative) digit string of a given length in a given base (or radix). This digit string is referred to as the *significand*, *mantissa*, or *coefficient*. The length of the significand determines the *precision* to which numbers can be represented. The radix point position is assumed always to be somewhere within the significand—often just after or just before the most significant digit, or to the right of the rightmost (least significant) digit. This article generally follows the convention that the radix point is set just after the most significant (leftmost) digit.

- A signed integer exponent (also referred to as the *characteristic*, or *scale*), which modifies the magnitude of the number.

To derive the value of the floating-point number, the *significand* is multiplied by the *base* raised to the power of the *exponent*, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent—to the right if the exponent is positive or to the left if the exponent is negative.

Using base-10 (the familiar decimal notation) as an example, the number 152,853.5047, which has ten decimal digits of precision, is represented as the significand 1,528,535,047 together with 5 as the exponent. To determine the actual value, a decimal point is placed after the first digit of the significand and the result is multiplied by 10^5 to give 1.528535047×10^5, or 152,853.5047. In storing such a number, the base (10) need not be stored, since it will be the same for the entire range of supported numbers, and can thus be inferred.

Symbolically, this final value is:

$$sb_{p-1} \times b^e, sbp-1 \times be,$$

where s is the significand (ignoring any implied decimal point), p is the precision (the number of digits in the significand), b is the base (in our example, this is the number *ten*), and e is the exponent.

Historically, several number bases have been used for representing floating-point numbers, with base two (binary) being the most common, followed by base ten (decimal floating point), and other less common varieties, such as base sixteen (hexadecimal floating point[nb 1]), base eight (octal floating point), base four (quaternary floating point), base three (balanced ternary floating point) and even base 256 and base 65,536.

A floating-point number is a rational number, because it can be represented as one integer divided by another; for example 1.45×10^3 is $(145/100) \times 1000$ or 145,000/100. The base determines the fractions that can be represented; for instance, 1/5 cannot be represented exactly as a floating-point number using a binary base, but 1/5 can be represented exactly using a decimal base (0.2, or 2×10^{-1}). However, 1/3 cannot be represented exactly by either binary (0.010101...) or decimal (0.333...), but in base 3, it is trivial (0.1 or 1×3^{-1}) . The occasions on which infinite expansions occur depend on the base and its prime factors.

The way in which the significand (including its sign) and exponent are stored in a computer is implementation-dependent. The common IEEE formats are described in detail later and elsewhere, but as an example, in the binary single-precision (32-bit) floating-point representation, $p=24$p=24, and so the significand is a string of 24 bits. For instance, the number π's first 33 bits are:

$$11001001\ 00001111\ 11011010\_\ 10100010\ 0.11001001\ 00001111\ 11011010\_\ 10100010\ 0.$$

In this binary expansion, let us denote the positions from 0 (leftmost bit, or most significant bit) to 32 (rightmost bit). The 24-bit significand will stop at position 23, shown as the underlined bit 0 above. The next bit, at position 24, is called the *round bit* or *rounding bit*. It is used to round the 33-bit approximation to the nearest 24-bit number (there are specific rules for halfway values, which is not the case here). This bit, which is 1 in this example, is added to the integer formed by the leftmost 24 bits, yielding:

$$11001001\ 00001111\ 11011011\_.11001001\ 00001111\ 11011011\_.$$

When this is stored in memory using the IEEE 754 encoding, this becomes the significand s. The significand is assumed to have a binary point to the right of the leftmost bit. So, the binary representation of π is calculated from left-to-right as follows:

$$=\approx\approx(\sum_{n=0}^{p-1}bit_n\times2^{-n})\times2^e(1\times2^{-0}+1\times2^{-1}+0\times2^{-2}+0\times2^{-3}+1\times2^{-4}+\cdots+1\times2^{-23})\times2^1 1.5707964\times2 3.1415928(\sum_{n=0}^{p-1}bit_n\times2^{-n})\times2^e=(1\times2^{-0}+1\times2^{-1}+0\times2^{-2}+0\times2^{-3}+1\times2^{-4}+\cdots+1\times2^{-23})\times2^1\approx1.5707964\times2\approx3.1415928$$

where p is the precision (24 in this example), n is the position of the bit of the significand from the left (starting at 0 and finishing at 23 here) and e is the exponent (1 in this example).

It can be required that the most significant digit of the significand of a non-zero number be non-zero (except when the corresponding exponent would be smaller than the minimum one). This process is called *normalization*. For binary formats (which uses only the digits 0 and 1), this non-zero digit is necessarily 1. Therefore, it does not need to be represented in memory; allowing the format to have one more bit of precision. This rule is variously called the *leading bit convention*, the *implicit bit convention*, the *hidden bit convention*, or the *assumed bit convention*.

Alternatives to floating-point numbers

The floating-point representation is by far the most common way of representing in computers an approximation to real numbers. However, there are alternatives:

- Fixed-point representation uses integer hardware operations controlled by a software implementation of a specific convention about the location of the binary or decimal point, for example, 6 bits or digits from the right. The hardware to manipulate these representations is less costly than floating point, and it can be used to perform normal integer operations, too. Binary fixed point is usually used in special-purpose applications on embedded processors that can only do integer arithmetic, but decimal fixed point is common in commercial applications.

- Logarithmic number systems (LNSs) represent a real number by the logarithm of its absolute value and a sign bit. The value distribution is similar to floating point, but the value-to-representation curve (*i.e.*, the graph of the logarithm function) is smooth (except at 0). Conversely to floating-point arithmetic, in a logarithmic number system multiplication, division and exponentiation are simple to implement, but addition and subtraction are complex. The (symmetric) level-index arithmetic (LI and SLI) of Charles Clenshaw, Frank Olver and Peter Turner is a scheme based on a generalized logarithm representation.
- Tapered floating-point representation, which does not appear to be used in practice.
- Where greater precision is desired, floating-point arithmetic can be implemented (typically in software) with variable-length significands (and sometimes exponents) that are sized depending on actual need and depending on how the calculation proceeds. This is called arbitrary-precision floating-point arithmetic.
- Floating-point expansions are another way to get a greater precision, benefiting from the floating-point hardware: a number is represented as an unevaluated sum of several floating-point numbers. An example is double-double arithmetic, sometimes used for the C type `long double`.
- Some simple rational numbers (*e.g.*, 1/3 and 1/10) cannot be represented exactly in binary floating point, no matter what the precision is. Using a different radix allows one to represent some of them (*e.g.*, 1/10 in decimal floating point), but the possibilities remain limited. Software packages that perform rational arithmetic represent numbers as fractions with integral numerator and denominator, and can therefore represent any rational number exactly. Such packages generally need to use "bignum" arithmetic for the individual integers.
- Interval arithmetic allows one to represent numbers as intervals and obtain guaranteed bounds on results. It is generally based on other arithmetics, in particular floating point.
- Computer algebra systems such as Mathematica, Maxima, and Maple can often handle irrational numbers like $\pi\pi$ or $3-\sqrt{3}$ in a completely "formal" way, without dealing with a specific encoding of the significand. Such a program can evaluate expressions like "$\sin(3\pi)$sin(3π)" exactly, because it is programmed to process the underlying mathematics directly, instead of using approximate values for each intermediate calculation.

History

In 1914, Leonardo Torres y Quevedo designed an electro-mechanical version of Charles Babbage's Analytical Engine, and included floating-point arithmetic. In 1938, Konrad Zuse of Berlin completed the Z1, the first binary, programmable mechanical computer; it uses a 24-bit binary floating-point number representation with a 7-bit signed exponent, a 17-bit significand (including one implicit bit), and a sign bit. The more reliable relay-based Z3, completed in 1941, has representations for both positive and negative

infinities; in particular, it implements defined operations with infinity, such as $1/\infty=0$1/∞=0, and it stops on undefined operations, such as $0\times\infty$0×∞.

Konrad Zuse, architect of the Z3 computer, which uses a 22-bit binary floating-point representation.

Zuse also proposed, but did not complete, carefully rounded floating-point arithmetic that includes $\pm\infty$±∞ and NaN representations, anticipating features of the IEEE Standard by four decades. In contrast, von Neumann recommended against floating-point numbers for the 1951 IAS machine, arguing that fixed-point arithmetic is preferable.

The first *commercial* computer with floating-point hardware was Zuse's Z4 computer, designed in 1942–1945. In 1946, Bell Laboratories introduced the Mark V, which implemented decimal floating-point numbers.

The Pilot ACE has binary floating-point arithmetic, and it became operational in 1950 at National Physical Laboratory, UK. Thirty-three were later sold commercially as the English Electric DEUCE. The arithmetic is actually implemented in software, but with a one megahertz clock rate, the speed of floating-point and fixed-point operations in this machine were initially faster than those of many competing computers.

The mass-produced IBM 704 followed in 1954; it introduced the use of a biased exponent. For many decades after that, floating-point hardware was typically an optional feature, and computers that had it were said to be "scientific computers", or to have "scientific computation" (SC) capability (see also Extensions for Scientific Computation (XSC)). It was not until the launch of the Intel i486 in 1989 that *general-purpose* personal computers had floating-point capability in hardware as a standard feature.

The UNIVAC 1100/2200 series, introduced in 1962, supported two floating-point representations:

- *Single precision*: 36 bits, organized as a 1-bit sign, an 8-bit exponent, and a 27-bit significand.
- *Double precision*: 72 bits, organized as a 1-bit sign, an 11-bit exponent, and a 60-bit significand.

The IBM 7094, also introduced in 1962, supports single-precision and double-precision representations, but with no relation to the UNIVAC's representations. Indeed, in 1964, IBM introduced hexadecimal floating-point representations in its System/360 mainframes; these same representations are still available for use in modern z/Architecture systems. However, in 1998, IBM included IEEE-compatible binary floating-point arithmetic to its mainframes; in 2005, IBM also added IEEE-compatible decimal floating-point arithmetic.

Initially, computers used many different representations for floating-point numbers. The lack of standardization at the mainframe level was an ongoing problem by the early 1970s for those writing and maintaining higher-level source code; these manufacturer floating-point standards differed in the word sizes, the representations, and the rounding behavior and general accuracy of operations. Floating-point compatibility across multiple computing systems was in desperate need of standardization by the early 1980s, leading to the creation of the IEEE 754 standard once the 32-bit (or 64-bit) word had become commonplace. This standard was significantly based on a proposal from Intel, which was designing the i8087 numerical coprocessor; Motorola, which was designing the 68000 around the same time, gave significant input as well.

In 1989, mathematician and computer scientist William Kahan was honored with the Turing Award for being the primary architect behind this proposal; he was aided by his student (Jerome Coonen) and a visiting professor (Harold Stone).

Among the x86 innovations are these:

- A precisely specified floating-point representation at the bit-string level, so that all compliant computers interpret bit patterns the same way. This makes it possible to accurately and efficiently transfer floating-point numbers from one computer to another (after accounting for endianness).

- A precisely specified behavior for the arithmetic operations: A result is required to be produced as if infinitely precise arithmetic were used to yield a value that is then rounded according to specific rules. This means that a compliant computer program would always produce the same result when given a particular input, thus mitigating the almost mystical reputation that floating-point computation had developed for its hitherto seemingly non-deterministic behavior.
- The ability of exceptional conditions (overflow, divide by zero, etc.) to propagate through a computation in a benign manner and then be handled by the software in a controlled fashion.

Range of floating-point numbers

A floating-point number consists of two fixed-point components, whose range depends exclusively on the number of bits or digits in their representation. Whereas components linearly depend on their range, the floating-point range linearly depends on the significand range and exponentially on the range of exponent component, which attaches outstandingly wider range to the number.

On a typical computer system, a *double-precision* (64-bit) binary floating-point number has a coefficient of 53 bits (including 1 implied bit), an exponent of 11 bits, and 1 sign bit. Since $2^{10} = 1024$, the complete range of the positive normal floating-point numbers in this format is from $2^{-1022} \approx 2 \times 10^{-308}$ to approximately $2^{1024} \approx 2 \times 10^{308}$.

The number of normalized floating-point numbers in a system (*B*, *P*, *L*, *U*) where

- *B* is the base of the system,
- *P* is the precision of the system to *P* numbers,
- *L* is the smallest exponent representable in the system,
- and *U* is the largest exponent used in the system)

is $2(B-1)(B^{P-1})(U-L+1)+1$ $2(B-1)(B^P-1)(U-L+1)+1$.

There is a smallest positive normalized floating-point number,

Underflow level = UFL = B^L BL,

which has a 1 as the leading digit and 0 for the remaining digits of the significand, and the smallest possible value for the exponent.

There is a largest floating-point number,

Overflow level = OFL = $(1-B^{-P})(B^{U+1})$ $(1-B-P)(BU+1)$,

which has $B - 1$ as the value for each digit of the significand and the largest possible value for the exponent.

In addition, there are representable values strictly between −UFL and UFL. Namely, positive and negative zeros, as well as denormalized numbers.

IEEE 754: floating point in modern computers

Main article: IEEE 754

Floating-point formats

IEEE 754

- 16-bit: Half (binary16)
- 32-bit: Single (binary32), decimal32
- 64-bit: Double (binary64), decimal64
- 128-bit: Quadruple (binary128), decimal128
- 256-bit: Octuple (binary256)
- Extended-precision formats (40-bit or 80-bit)

Other

- Minifloat
- bfloat16
- Microsoft Binary Format
- IBM floating-point architecture
- Posit
- G.711 8-bit floats
- Arbitrary precision

The IEEE standardized the computer representation for binary floating-point numbers in IEEE 754 (a.k.a. IEC 60559) in 1985. This first standard is followed by almost all modern machines. It was revised in 2008. IBM mainframes support IBM's own hexadecimal floating point format and IEEE 754-2008 decimal floating point in addition to the IEEE 754 binary format. The Cray T90 series had an IEEE version, but the SV1 still uses Cray floating-point format.

The standard provides for many closely related formats, differing in only a few details. Five of these formats are called *basic formats* and others are termed *extended formats*; three of these are especially widely used in computer hardware and languages:

- Single precision, usually used to represent the "float" type in the C language family (though this is not guaranteed). This is a binary format that occupies 32 bits (4 bytes) and its significand has a precision of 24 bits (about 7 decimal digits).
- Double precision, usually used to represent the "double" type in the C language family (though this is not guaranteed). This is a binary format that occupies 64 bits (8 bytes) and its significand has a precision of 53 bits (about 16 decimal digits).
- Double extended, also called "extended precision" format. This is a binary format that occupies at least 79 bits (80 if the hidden/implicit bit rule is not used) and its significand has a precision of at least 64 bits (about 19 decimal digits). A format

satisfying the minimal requirements (64-bit significand precision, 15-bit exponent, thus fitting on 80 bits) is provided by the x86 architecture. Often on such processors, this format can be used with "long double" in the C language family (the C99 and C11 standards "IEC 60559 floating-point arithmetic extension-Annex F" recommend the 80-bit extended format to be provided as "long double" when available), though extended precision is not available with MSVC.
For alignment purposes, many tools store this 80-bit value in a 96-bit or 128-bit space. On other processors, "long double" may stand for a larger format, such as quadruple precision, or just double precision, if any form of extended precision is not available.

Increasing the precision of the floating point representation generally reduces the amount of accumulated round-off error caused by intermediate calculations. Less common IEEE formats include:

- Quadruple precision (binary128). This is a binary format that occupies 128 bits (16 bytes) and its significand has a precision of 113 bits (about 34 decimal digits).
- Decimal double precision (decimal64) and decimal quadruple precision (decimal128) decimal floating-point formats. These formats, along with the decimal single precision (decimal32) format, are intended for performing decimal rounding correctly.
- Half, also called binary16, a 16-bit floating-point value. It is being used in the NVIDIA Cg graphics language, and in the openEXR standard.

Any integer with absolute value less than 2^{24} can be exactly represented in the single precision format, and any integer with absolute value less than 2^{53} can be exactly represented in the double precision format. Furthermore, a wide range of powers of 2 times such a number can be represented. These properties are sometimes used for purely integer data, to get 53-bit integers on platforms that have double precision floats but only 32-bit integers.

The standard specifies some special values, and their representation:
positive infinity (+∞), negative infinity (−∞), a negative zero (−0) distinct from ordinary ("positive") zero, and "not a number" values (NaNs).

Comparison of floating-point numbers, as defined by the IEEE standard, is a bit different from usual integer comparison. Negative and positive zero compare equal, and every NaN compares unequal to every value, including itself. All values except NaN are strictly smaller than +∞ and strictly greater than −∞. Finite floating-point numbers are ordered in the same way as their values (in the set of real numbers).

Internal representation

Floating-point numbers are typically packed into a computer datum as the sign bit, the exponent field, and the significand or mantissa, from left to right. For the IEEE 754 binary

formats (basic and extended) which have extant hardware implementations, they are apportioned as follows:

Type	Sign	Exponent	Significand field	Total bits	Exponent bias	Bits precision	Number of decimal digits
Half (IEEE 754-2008)	1	5	10	16	15	11	~3.3
Single	1	8	23	32	127	24	~7.2
Double	1	11	52	64	1023	53	~15.9
x86 extended precision	1	15	64	80	16383	64	~19.2
Quad	1	15	112	128	16383	113	~34.0

While the exponent can be positive or negative, in binary formats it is stored as an unsigned number that has a fixed "bias" added to it. Values of all 0s in this field are reserved for the zeros and subnormal numbers; values of all 1s are reserved for the infinities and NaNs. The exponent range for normalized numbers is [−126, 127] for single precision, [−1022, 1023] for double, or [−16382, 16383] for quad. Normalized numbers exclude subnormal values, zeros, infinities, and NaNs.

In the IEEE binary interchange formats the leading 1 bit of a normalized significand is not actually stored in the computer datum. It is called the "hidden" or "implicit" bit. Because of this, single precision format actually has a significand with 24 bits of precision, double precision format has 53, and quad has 113.

For example, it was shown above that π, rounded to 24 bits of precision, has:

- sign = 0 ; e = 1 ; s = 110010010000111111011011 (including the hidden bit)

The sum of the exponent bias (127) and the exponent (1) is 128, so this is represented in single precision format as

- 0 10000000 10010010000111111011011 (excluding the hidden bit) = 40490FDB as a hexadecimal number.

An example of a layout for 32-bit floating point is

and the 64 bit layout is similar.

Special values

Signed zero

Main article: Signed zero

In the IEEE 754 standard, zero is signed, meaning that there exist both a "positive zero" (+0) and a "negative zero" (−0). In most run-time environments , positive zero is usually printed as "0" and the negative zero as "-0". The two values behave as equal in numerical comparisons, but some operations return different results for +0 and −0. For instance, 1/(−0) returns negative infinity, while 1/+0 returns positive infinity (so that the identity $1/(1/\pm\infty) = \pm\infty$ is maintained). Other common functions with a discontinuity at $x=0$ which might treat +0 and −0 differently include $\log(x)$, $\operatorname{signum}(x)$, and the principal square root of $y + xi$ for any negative number y. As with any approximation scheme, operations involving "negative zero" can occasionally cause confusion. For example, in IEEE 754, $x = y$ does not always imply $1/x = 1/y$, as $0 = -0$ but $1/0 \neq 1/-0$.

Subnormal numbers

Main article: Subnormal numbers

Subnormal values fill the underflow gap with values where the absolute distance between them is the same as for adjacent values just outside the underflow gap. This is an improvement over the older practice to just have zero in the underflow gap, and where underflowing results were replaced by zero (flush to zero).

Modern floating-point hardware usually handles subnormal values (as well as normal values), and does not require software emulation for subnormals.

Infinities

Further information on the concept of infinite: Infinity

The infinities of the extended real number line can be represented in IEEE floating-point datatypes, just like ordinary floating-point values like 1, 1.5, etc. They are not error values in any way, though they are often (but not always, as it depends on the rounding) used as replacement values when there is an overflow. Upon a divide-by-zero exception, a positive or negative infinity is returned as an exact result. An infinity can also be introduced as a numeral (like C's "INFINITY" macro, or "∞" if the programming language allows that syntax).

IEEE 754 requires infinities to be handled in a reasonable way, such as

- $(+\infty) + (+7) = (+\infty)$
- $(+\infty) \times (-2) = (-\infty)$
- $(+\infty) \times 0 = \text{NaN}$ – there is no meaningful thing to do

NaNs

Main article: NaN

IEEE 754 specifies a special value called "Not a Number" (NaN) to be returned as the result of certain "invalid" operations, such as 0/0, ∞×0, or sqrt(−1). In general, NaNs will be propagated i.e. most operations involving a NaN will result in a NaN, although functions that would give some defined result for any given floating-point value will do so for NaNs as well, e.g. NaN ^ 0 = 1. There are two kinds of NaNs: the default *quiet* NaNs and, optionally, *signaling* NaNs. A signaling NaN in any arithmetic operation (including numerical comparisons) will cause an "invalid operation" exception to be signaled.

The representation of NaNs specified by the standard has some unspecified bits that could be used to encode the type or source of error; but there is no standard for that encoding. In theory, signaling NaNs could be used by a runtime system to flag uninitialized variables, or extend the floating-point numbers with other special values without slowing down the computations with ordinary values, although such extensions are not common.

IEEE 754 design rationale

William Kahan. A primary architect of the Intel 80x87 floating-point coprocessor and IEEE 754 floating-point standard.

It is a common misconception that the more esoteric features of the IEEE 754 standard discussed here, such as extended formats, NaN, infinities, subnormals etc., are only of interest to numerical analysts, or for advanced numerical applications; in fact the opposite is true: these features are designed to give safe robust defaults for numerically unsophisticated programmers, in addition to supporting sophisticated numerical libraries by experts. The key designer of IEEE 754, William Kahan notes that it is incorrect to "... [deem] features of IEEE Standard 754 for Binary Floating-Point Arithmetic that ...[are] not appreciated to be features usable by none but numerical experts. The facts are quite the

opposite. In 1977 those features were designed into the Intel 8087 to serve the widest possible market... Error-analysis tells us how to design floating-point arithmetic, like IEEE Standard 754, moderately tolerant of well-meaning ignorance among programmers".

- The special values such as infinity and NaN ensure that the floating-point arithmetic is algebraically completed, such that every floating-point operation produces a well-defined result and will not—by default—throw a machine interrupt or trap. Moreover, the choices of special values returned in exceptional cases were designed to give the correct answer in many cases, e.g. continued fractions such as R(z) := 7 − 3/[z − 2 − 1/(z − 7 + 10/[z − 2 − 2/(z − 3)])] will give the correct answer in all inputs under IEEE 754 arithmetic as the potential divide by zero in e.g. R(3) = 4.6 is correctly handled as +infinity and so can be safely ignored. As noted by Kahan, the unhandled trap consecutive to a floating-point to 16-bit integer conversion overflow that caused the loss of an Ariane 5 rocket would not have happened under the default IEEE 754 floating-point policy.
- Subnormal numbers ensure that for *finite* floating-point numbers x and y, x − y = 0 if and only if x = y, as expected, but which did not hold under earlier floating-point representations.
- On the design rationale of the x87 80-bit format, Kahan notes: "This Extended format is designed to be used, with negligible loss of speed, for all but the simplest arithmetic with float and double operands. For example, it should be used for scratch variables in loops that implement recurrences like polynomial evaluation, scalar products, partial and continued fractions. It often averts premature Over/Underflow or severe local cancellation that can spoil simple algorithms". Computing intermediate results in an extended format with high precision and extended exponent has precedents in the historical practice of scientific calculation and in the design of scientific calculators e.g. Hewlett-Packard's financial calculators performed arithmetic and financial functions to three more significant decimals than they stored or displayed. The implementation of extended precision enabled standard elementary function libraries to be readily developed that normally gave double precision results within one unit in the last place (ULP) at high speed.
- Correct rounding of values to the nearest representable value avoids systematic biases in calculations and slows the growth of errors. Rounding ties to even removes the statistical bias that can occur in adding similar figures.
- Directed rounding was intended as an aid with checking error bounds, for instance in interval arithmetic. It is also used in the implementation of some functions.
- The mathematical basis of the operations enabled high precision multiword arithmetic subroutines to be built relatively easily.
- The single and double precision formats were designed to be easy to sort without using floating-point hardware. Their bits as a two's-complement integer already sort the positives correctly, and the negatives reversed. If that integer is negative, xor with its maximum positive, and the floats are sorted as integers.

Other notable floating-point formats

In addition to the widely used IEEE 754 standard formats, other floating-point formats are used, or have been used, in certain domain-specific areas.

- The Bfloat16 format requires the same amount of memory (16 bits) as the IEEE 754 half-precision format, but allocates 8 bits to the exponent instead of 5, thus providing the same range as a single-precision IEEE 754 number. The tradeoff is a reduced precision, as the significand field is reduced from 10 to 7 bits. This format is mainly used in the training of machine learning models, where range is more valuable than precision. Many machine learning accelerators provide hardware support for this format.
- The TensorFloat-32 format provides the best of the Bfloat16 and half-precision formats, having 8 bits of exponent as the former and 10 bits of significand field as the latter. This format was introduced by Nvidia, which provides hardware support for it in the Tensor Cores of its GPUs based on the Nvidia Ampere architecture. The drawback of this format is its total size of 19 bits, which is not a power of 2. However, according to Nvidia, this format should only be used internally by hardware to speed up computations, while inputs and outputs should be stored in the 32-bit single-precision IEEE 754 format.

Bfloat16 and TensorFloat-32 formats specifications, compared with IEEE 754 half-precision and single-precision standard formats

Type	Sign	Exponent	Significand field	Total bits
Half-precision	1	5	10	16
Bfloat16	1	8	7	16
TensorFloat-32	1	8	10	19
Single-precision	1			
1				

Representable numbers, conversion and rounding

By their nature, all numbers expressed in floating-point format are rational numbers with a terminating expansion in the relevant base (for example, a terminating decimal expansion in base-10, or a terminating binary expansion in base-2). Irrational numbers, such as π or √2, or non-terminating rational numbers, must be approximated. The number of digits (or bits) of precision also limits the set of rational numbers that can be

represented exactly. For example, the decimal number 123456789 cannot be exactly represented if only eight decimal digits of precision are available (would be rounded to 123456790 or 123456780 where the rightmost digit 0 is not explicitly stored).

When a number is represented in some format (such as a character string) which is not a native floating-point representation supported in a computer implementation, then it will require a conversion before it can be used in that implementation. If the number can be represented exactly in the floating-point format then the conversion is exact. If there is not an exact representation then the conversion requires a choice of which floating-point number to use to represent the original value. The representation chosen will have a different value from the original, and the value thus adjusted is called the *rounded value*.

Whether or not a rational number has a terminating expansion depends on the base. For example, in base-10 the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...). In base-2 only rationals with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating. Any rational with a denominator that has a prime factor other than 2 will have an infinite binary expansion. This means that numbers which appear to be short and exact when written in decimal format may need to be approximated when converted to binary floating-point. For example, the decimal number 0.1 is not representable in binary floating-point of any finite precision; the exact binary representation would have a "1100" sequence continuing endlessly:

$e = -4$; $s = 1100110011001100110011001100110011...$,

where, as previously, s is the significand and e is the exponent.

When rounded to 24 bits this becomes

$e = -4$; $s = 110011001100110011001101$,

which is actually 0.100000001490116119384765625 in decimal.

As a further example, the real number π, represented in binary as an infinite sequence of bits is

11.0010010000111111011010101000100010000101101000110000100011010011...

but is

11.00100100000111111011011

when approximated by rounding to a precision of 24 bits.

In binary single-precision floating-point, this is represented as $s = 1.00100100000111111011011$ with $e = 1$. This has a decimal value of

3.1415927410125732421875,

whereas a more accurate approximation of the true value of π is

3.14159265358979323846264338327950...

The result of rounding differs from the true value by about 0.03 parts per million, and matches the decimal representation of π in the first 7 digits. The difference is the discretization error and is limited by the machine epsilon.

The arithmetical difference between two consecutive representable floating-point numbers which have the same exponent is called a unit in the last place (ULP). For example, if there is no representable number lying between the representable numbers $1.45a70c22_{hex}$ and $1.45a70c24_{hex}$, the ULP is 2×16^{-8}, or 2^{-31}. For numbers with a base-2 exponent part of 0, i.e. numbers with an absolute value higher than or equal to 1 but lower than 2, an ULP is exactly 2^{-23} or about 10^{-7} in single precision, and exactly 2^{-53} or about 10^{-16} in double precision. The mandated behavior of IEEE-compliant hardware is that the result be within one-half of a ULP.

Rounding modes

Rounding is used when the exact result of a floating-point operation (or a conversion to floating-point format) would need more digits than there are digits in the significand. IEEE 754 requires *correct rounding*: that is, the rounded result is as if infinitely precise arithmetic was used to compute the value and then rounded (although in implementation only three extra bits are needed to ensure this). There are several different rounding schemes (or *rounding modes*). Historically, truncation was the typical approach. Since the introduction of IEEE 754, the default method (*round to nearest, ties to even*, sometimes called Banker's Rounding) is more commonly used. This method rounds the ideal (infinitely precise) result of an arithmetic operation to the nearest representable value, and gives that representation as the result. In the case of a tie, the value that would make the significand end in an even digit is chosen. The IEEE 754 standard requires the same rounding to be applied to all fundamental algebraic operations, including square root and conversions, when there is a numeric (non-NaN) result. It means that the results of IEEE 754 operations are completely determined in all bits of the result, except for the representation of NaNs. ("Library" functions such as cosine and log are not mandated.)

Alternative rounding options are also available. IEEE 754 specifies the following rounding modes:

- round to nearest, where ties round to the nearest even digit in the required position (the default and by far the most common mode)
- round to nearest, where ties round away from zero (optional for binary floating-point and commonly used in decimal)
- round up (toward +∞; negative results thus round toward zero)
- round down (toward −∞; negative results thus round away from zero)

- round toward zero (truncation; it is similar to the common behavior of float-to-integer conversions, which convert −3.9 to −3 and 3.9 to 3)

Alternative modes are useful when the amount of error being introduced must be bounded. Applications that require a bounded error are multi-precision floating-point, and interval arithmetic. The alternative rounding modes are also useful in diagnosing numerical instability: if the results of a subroutine vary substantially between rounding to + and − infinity then it is likely numerically unstable and affected by round-off error.

Floating-point arithmetic operations

For ease of presentation and understanding, decimal radix with 7 digit precision will be used in the examples, as in the IEEE 754 *decimal32* format. The fundamental principles are the same in any radix or precision, except that normalization is optional (it does not affect the numerical value of the result). Here, *s* denotes the significand and *e* denotes the exponent.

Addition and subtraction

A simple method to add floating-point numbers is to first represent them with the same exponent. In the example below, the second number is shifted right by three digits, and one then proceeds with the usual addition method:

```
 123456.7 = 1.234567 × 10^5
 101.7654 = 1.017654 × 10^2 = 0.001017654 × 10^5
 Hence:
 123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
 = (1.234567 × 10^5) + (0.001017654 × 10^5)
 = (1.234567 + 0.001017654) × 10^5
 = 1.235584654 × 10^5
```

In detail:

```
  e=5;  s=1.234567 (123456.7)
+ e=2; s=1.017654 (101.7654)
  e=5; s=1.234567
+ e=5; s=0.001017654 (after shifting)
--------------------
  e=5; s=1.235584654 (true sum: 123558.4654)
```

This is the true result, the exact sum of the operands. It will be rounded to seven digits and then normalized if necessary. The final result is

```
  e=5; s=1.235585 (final sum: 123558.5)
```

The lowest three digits of the second operand (654) are essentially lost. This is round-off error. In extreme cases, the sum of two non-zero numbers may be equal to one of them:

```
  e=5;  s=1.234567
+ e=-3; s=9.876543
  e=5; s=1.234567
+ e=5; s=0.00000009876543 (after shifting)
----------------------
  e=5; s=1.23456709876543 (true sum)
```

```
  e=5; s=1.234567 (after rounding and normalization)
```

In the above conceptual examples it would appear that a large number of extra digits would need to be provided by the adder to ensure correct rounding; however, for binary addition or subtraction using careful implementation techniques only two extra *guard* bits and one extra *sticky* bit need to be carried beyond the precision of the operands.

Another problem of loss of significance occurs when two nearly equal numbers are subtracted. In the following example $e = 5$; $s = 1.234571$ and $e = 5$; $s = 1.234567$ are representations of the rationals 123457.1467 and 123456.659.

```
  e=5; s=1.234571
- e=5; s=1.234567
----------------
 e=5;  s=0.000004
 e=-1; s=4.000000 (after rounding and normalization)
```

The best representation of this difference is $e = -1$; $s = 4.877000$, which differs more than 20% from $e = -1$; $s = 4.000000$. In extreme cases, all significant digits of precision can be lost (although gradual underflow ensures that the result will not be zero unless the two operands were equal). This *cancellation* illustrates the danger in assuming that all of the digits of a computed result are meaningful. Dealing with the consequences of these errors is a topic in numerical analysis; see also Accuracy problems.

Multiplication and division

To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized.

```
  e=3; s=4.734612
× e=5; s=5.417242
-----------------------
  e=8; s=25.648538980104  (true product)
  e=8; s=25.64854         (after rounding)
  e=9; s=2.564854         (after normalization)
```

Similarly, division is accomplished by subtracting the divisor's exponent from the dividend's exponent, and dividing the dividend's significand by the divisor's significand.

There are no cancellation or absorption problems with multiplication or division, though small errors may accumulate as operations are performed in succession. In practice, the way these operations are carried out in digital logic can be quite complex (see Booth's multiplication algorithm and Division algorithm). For a fast, simple method, see the Horner method.

Dealing with exceptional cases

Floating-point computation in a computer can run into three kinds of problems:

- An operation can be mathematically undefined, such as ∞/∞, or division by zero.

- An operation can be legal in principle, but not supported by the specific format, for example, calculating the square root of −1 or the inverse sine of 2 (both of which result in complex numbers).
- An operation can be legal in principle, but the result can be impossible to represent in the specified format, because the exponent is too large or too small to encode in the exponent field. Such an event is called an overflow (exponent too large), underflow (exponent too small) or denormalization (precision loss).

Prior to the IEEE standard, such conditions usually caused the program to terminate, or triggered some kind of trap that the programmer might be able to catch. How this worked was system-dependent, meaning that floating-point programs were not portable. (The term "exception" as used in IEEE 754 is a general term meaning an exceptional condition, which is not necessarily an error, and is a different usage to that typically defined in programming languages such as a C++ or Java, in which an "exception" is an alternative flow of control, closer to what is termed a "trap" in IEEE 754 terminology).

Here, the required default method of handling exceptions according to IEEE 754 is discussed (the IEEE 754 optional trapping and other "alternate exception handling" modes are not discussed). Arithmetic exceptions are (by default) required to be recorded in "sticky" status flag bits. That they are "sticky" means that they are not reset by the next (arithmetic) operation, but stay set until explicitly reset. The use of "sticky" flags thus allows for testing of exceptional conditions to be delayed until after a full floating-point expression or subroutine: without them exceptional conditions that could not be otherwise ignored would require explicit testing immediately after every floating-point operation. By default, an operation always returns a result according to specification without interrupting computation. For instance, 1/0 returns +∞, while also setting the divide-by-zero flag bit (this default of ∞ is designed to often return a finite result when used in subsequent operations and so be safely ignored).

The original IEEE 754 standard, however, failed to recommend operations to handle such sets of arithmetic exception flag bits. So while these were implemented in hardware, initially programming language implementations typically did not provide a means to access them (apart from assembler). Over time some programming language standards (e.g., C99/C11 and Fortran) have been updated to specify methods to access and change status flag bits. The 2008 version of the IEEE 754 standard now specifies a few operations for accessing and handling the arithmetic flag bits. The programming model is based on a single thread of execution and use of them by multiple threads has to be handled by a means outside of the standard (e.g. C11 specifies that the flags have thread-local storage).

IEEE 754 specifies five arithmetic exceptions that are to be recorded in the status flags ("sticky bits"):

- **inexact**, set if the rounded (and returned) value is different from the mathematically exact result of the operation.

- **underflow**, set if the rounded value is tiny (as specified in IEEE 754) *and* inexact (or maybe limited to if it has denormalization loss, as per the 1984 version of IEEE 754), returning a subnormal value including the zeros.
- **overflow**, set if the absolute value of the rounded value is too large to be represented. An infinity or maximal finite value is returned, depending on which rounding is used.
- **divide-by-zero**, set if the result is infinite given finite operands, returning an infinity, either +∞ or −∞.
- **invalid**, set if a real-valued result cannot be returned e.g. sqrt(−1) or 0/0, returning a quiet NaN.

Fig. 1: resistances in parallel, with total resistance R_{tot}Rtot

The default return value for each of the exceptions is designed to give the correct result in the majority of cases such that the exceptions can be ignored in the majority of codes. *inexact* returns a correctly rounded result, and *underflow* returns a denormalized small value and so can almost always be ignored. *divide-by-zero* returns infinity exactly, which will typically then divide a finite number and so give zero, or else will give an *invalid* exception subsequently if not, and so can also typically be ignored. For example, the effective resistance of n resistors in parallel (see fig. 1) is given by $R_{tot}=1/(1/R_1+1/R_2+\cdots+1/R_n)$Rtot=1/(1/R1+1/R2+⋯+1/Rn). If a short-circuit develops with R_1R1 set to 0, $1/R_1$1/R1 will return +infinity which will give a final R_{tot}Rtot of 0, as expected (see the continued fraction example of IEEE 754 design rationale for another example).

Overflow and *invalid* exceptions can typically not be ignored, but do not necessarily represent errors: for example, a root-finding routine, as part of its normal operation, may evaluate a passed-in function at values outside of its domain, returning NaN and an *invalid* exception flag to be ignored until finding a useful start point.

Accuracy problems

The fact that floating-point numbers cannot precisely represent all real numbers, and that floating-point operations cannot precisely represent true arithmetic operations, leads to many surprising situations. This is related to the finite precision with which computers generally represent numbers.

For example, the non-representability of 0.1 and 0.01 (in binary) means that the result of attempting to square 0.1 is neither 0.01 nor the representable number closest to it. In 24-bit (single precision) representation, 0.1 (decimal) was given previously as $e = -4$; $s = 11001100110011001100110$, which is

0.100000001490116119384765625 exactly.

Squaring this number gives

0.010000000298023226097399174250313080847263336181640625 exactly.

Squaring it with single-precision floating-point hardware (with rounding) gives

0.010000000707805156707763671875 exactly.

But the representable number closest to 0.01 is

0.009999999776482582092285156250 exactly.

Also, the non-representability of π (and π/2) means that an attempted computation of tan(π/2) will not yield a result of infinity, nor will it even overflow. It is simply not possible for standard floating-point hardware to attempt to compute tan(π/2), because π/2 cannot be represented exactly. This computation in C:

```
/* Enough digits to be sure we get the correct approximation. */
double pi = 3.14159265358979323846264338327795;
double z = tan(pi/2.0);
```

will give a result of 16331239353195370.0. In single precision (using the tanf function), the result will be −22877332.0.

By the same token, an attempted computation of sin(π) will not yield zero. The result will be (approximately) 0.1225×10^{-15} in double precision, or -0.8742×10^{-7} in single precision.

While floating-point addition and multiplication are both commutative ($a + b = b + a$ and $a \times b = b \times a$), they are not necessarily associative. That is, $(a + b) + c$ is not necessarily equal to $a + (b + c)$. Using 7-digit significand decimal arithmetic:

```
a = 1234.567, b = 45.67834, c = 0.0004
(a + b) + c:
 1234.567   (a)
+  45.67834 (b)
____________
 1280.24534 rounds to 1280.245
 1280.245   (a + b)
+   0.0004  (c)
____________
1280.2454 rounds to 1280.245 ← (a + b) + c
a + (b + c):
  45.67834  (b)
+  0.0004   (c)
____________
```

```
 45.67874
 1234.567   (a)
 + 45.67874 (b + c)
 ____________
 1280.24574 rounds to 1280.246 ← a + (b + c)
```

They are also not necessarily distributive. That is, $(a + b) \times c$ may not be the same as $a \times c + b \times c$:

```
 1234.567 × 3.333333 = 4115.223
 1.234567 × 3.333333 = 4.115223
 4115.223 + 4.115223 = 4119.338
 but
 1234.567 + 1.234567 = 1235.802
 1235.802 × 3.333333 = 4119.340
```

In addition to loss of significance, inability to represent numbers such as π and 0.1 exactly, and other slight inaccuracies, the following phenomena may occur:

- Cancellation: subtraction of nearly equal operands may cause extreme loss of accuracy. When we subtract two almost equal numbers we set the most significant digits to zero, leaving ourselves with just the insignificant, and most erroneous, digits.[124] For example, when determining a derivative of a function the following formula is used:

  $$Q(h)=f(a+h)-f(a)h.$$ Q(h)=f(a+h)−f(a)h.

  Intuitively one would want an *h* very close to zero, however when using floating-point operations, the smallest number won't give the best approximation of a derivative. As *h* grows smaller the difference between f (a + h) and f(a) grows smaller, cancelling out the most significant and least erroneous digits and making the most erroneous digits more important. As a result the smallest number of *h* possible will give a more erroneous approximation of a derivative than a somewhat larger number. This is perhaps the most common and serious accuracy problem.

- Conversions to integer are not intuitive: converting (63.0/9.0) to integer yields 7, but converting (0.63/0.09) may yield 6. This is because conversions generally truncate rather than round. Floor and ceiling functions may produce answers which are off by one from the intuitively expected value.
- Limited exponent range: results might overflow yielding infinity, or underflow yielding a subnormal number or zero. In these cases precision will be lost.
- Testing for safe division is problematic: Checking that the divisor is not zero does not guarantee that a division will not overflow.
- Testing for equality is problematic. Two computational sequences that are mathematically equal may well produce different floating-point values.

Incidents

- On February 25, 1991, a loss of significance in a MIM-104 Patriot missile battery prevented it from intercepting an incoming Scud missile in Dhahran, Saudi Arabia, contributing to the death of 28 soldiers from the U.S. Army's 14th Quartermaster Detachment.

Machine precision and backward error analysis

Machine precision is a quantity that characterizes the accuracy of a floating-point system, and is used in backward error analysis of floating-point algorithms. It is also known as unit roundoff or *machine epsilon*. Usually denoted E_{mach}, its value depends on the particular rounding being used.

With rounding to zero,

$$E_{mach}=B^{1-P}, Emach=B1-P,$$

whereas rounding to nearest,

$$E_{mach}=\frac{1}{2}B^{1-P}. Emach=12B1-P.$$

This is important since it bounds the *relative error* in representing any non-zero real number x within the normalized range of a floating-point system:

$$|||fl(x)-xx|||\leq E_{mach}. |fl(x)-xx|\leq Emach.$$

Backward error analysis, the theory of which was developed and popularized by James H. Wilkinson, can be used to establish that an algorithm implementing a numerical function is numerically stable. The basic approach is to show that although the calculated result, due to roundoff errors, will not be exactly correct, it is the exact solution to a nearby problem with slightly perturbed input data. If the perturbation required is small, on the order of the uncertainty in the input data, then the results are in some sense as accurate as the data "deserves". The algorithm is then defined as *backward stable*. Stability is a measure of the sensitivity to rounding errors of a given numerical procedure; by contrast, the condition number of a function for a given problem indicates the inherent sensitivity of the function to small perturbations in its input and is independent of the implementation used to solve the problem.

As a trivial example, consider a simple expression giving the inner product of (length two) vectors xx and yy , then

$$fl(x\cdot y)=fl(fl(x_1\cdot y_1)+fl(x_2\cdot y_2)), \text{ where } fl() \text{ indicates correctly rounded floating-point}$$
$$\text{arithmetic}=fl((x_1\cdot y_1)(1+\delta_1)+(x_2\cdot y_2)(1+\delta_2)), \text{ where } \delta_n\leq E_{mach}, \text{ from}$$

$$above=((x_1\cdot y_1)(1+\delta_1)+(x_2\cdot y_2)(1+\delta_2))(1+\delta_3)=(x_1\cdot y_1)(1+\delta_1)(1+\delta_3)+(x_2\cdot y_2)(1+\delta_2)(1+\delta_3),$$ fl(x·y)=fl(fl(x1·y1)+fl(x2·y2)), where fl() indicates correctly rounded floating-pointarithmetic=fl((x1·y1)(1+δ1)+(x2·y2)(1+δ2)), where δn≤Emach, from above=((x1·y1)(1+δ1)+(x2·y2)(1+δ2))(1+δ3)=(x1·y1)(1+δ1)(1+δ3)+(x2·y2)(1+δ2)(1+δ3),

and so

$$fl(x\cdot y)=x\hat{}\cdot y\hat{},$$ fl(x·y)=x^·y^,

where

$$x\hat{}_1=x_1(1+\delta_1);x\hat{}_2=x_2(1+\delta_2);$$ x^1=x1(1+δ1);x^2=x2(1+δ2);

$$y\hat{}_1=y_1(1+\delta_3);y\hat{}_2=y_2(1+\delta_3),$$ y^1=y1(1+δ3);y^2=y2(1+δ3),

where

$$\delta_n\leq E_{mach}$$ δn≤Emach

by definition, which is the sum of two slightly perturbed (on the order of E_{mach}) input data, and so is backward stable. For more realistic examples in numerical linear algebra, see Higham 2002 and other references below.

Minimizing the effect of accuracy problems

Although, as noted previously, individual arithmetic operations of IEEE 754 are guaranteed accurate to within half a ULP, more complicated formulae can suffer from larger errors due to round-off. The loss of accuracy can be substantial if a problem or its data are ill-conditioned, meaning that the correct result is hypersensitive to tiny perturbations in its data. However, even functions that are well-conditioned can suffer from large loss of accuracy if an algorithm numerically unstable for that data is used: apparently equivalent formulations of expressions in a programming language can differ markedly in their numerical stability. One approach to remove the risk of such loss of accuracy is the design and analysis of numerically stable algorithms, which is an aim of the branch of mathematics known as numerical analysis. Another approach that can protect against the risk of numerical instabilities is the computation of intermediate (scratch) values in an algorithm at a higher precision than the final result requires, which can remove, or reduce by orders of magnitude, such risk: IEEE 754 quadruple precision and extended precision are designed for this purpose when computing at double precision.

For example, the following algorithm is a direct implementation to compute the function A(x) = (x−1) / (exp(x−1) − 1) which is well-conditioned at 1.0, however it can be shown to

be numerically unstable and lose up to half the significant digits carried by the arithmetic when computed near 1.0.

```
1 doubleA(doubleX)
2 {
3    doubleY,Z;// [1]
4    Y=X-1.0;
5    Z=exp(Y);
6    if(Z!=1.0)Z=Y/(Z-1.0);// [2]
7    return(Z);
8 }
```

If, however, intermediate computations are all performed in extended precision (e.g. by setting line [1] to C99 long double), then up to full precision in the final double result can be maintained. Alternatively, a numerical analysis of the algorithm reveals that if the following non-obvious change to line [2] is made:

```
if(Z!=1.0)Z=log(Z)/(Z-1.0);
```

then the algorithm becomes numerically stable and can compute to full double precision.

To maintain the properties of such carefully constructed numerically stable programs, careful handling by the compiler is required. Certain "optimizations" that compilers might make (for example, reordering operations) can work against the goals of well-behaved software. There is some controversy about the failings of compilers and language designs in this area: C99 is an example of a language where such optimizations are carefully specified to maintain numerical precision. See the external references at the bottom of this article.

A detailed treatment of the techniques for writing high-quality floating-point software is beyond the scope of this article, and the reader is referred to, and the other references at the bottom of this article. Kahan suggests several rules of thumb that can substantially decrease by orders of magnitude the risk of numerical anomalies, in addition to, or in lieu of, a more careful numerical analysis. These include: as noted above, computing all expressions and intermediate results in the highest precision supported in hardware (a common rule of thumb is to carry twice the precision of the desired result i.e. compute in double precision for a final single precision result, or in double extended or quad precision for up to double precision results); and rounding input data and results to only the precision required and supported by the input data (carrying excess precision in the final result beyond that required and supported by the input data can be misleading, increases storage cost and decreases speed, and the excess bits can affect convergence of numerical procedures: notably, the first form of the iterative example given below converges correctly when using this rule of thumb). Brief descriptions of several additional issues and techniques follow.

As decimal fractions can often not be exactly represented in binary floating-point, such arithmetic is at its best when it is simply being used to measure real-world quantities over a wide range of scales (such as the orbital period of a moon around Saturn or the mass of a proton), and at its worst when it is expected to model the interactions of quantities expressed as decimal strings that are expected to be exact. An example of the

latter case is financial calculations. For this reason, financial software tends not to use a binary floating-point number representation. The "decimal" data type of the C# and Python programming languages, and the decimal formats of the IEEE 754-2008 standard, are designed to avoid the problems of binary floating-point representations when applied to human-entered exact decimal values, and make the arithmetic always behave as expected when numbers are printed in decimal.

Expectations from mathematics may not be realized in the field of floating-point computation. For example, it is known that $(x+y)(x-y)=x^2-y^2$(x+y)(x−y)=x2−y2, and that $\sin^2\theta+\cos^2\theta=1$sin2θ+cos2θ=1, however these facts cannot be relied on when the quantities involved are the result of floating-point computation.

The use of the equality test (`if (x==y) ...`) requires care when dealing with floating-point numbers. Even simple expressions like `0.6/0.2-3==0` will, on most computers, fail to be true (in IEEE 754 double precision, for example, `0.6/0.2-3` is approximately equal to -4.44089209850063e-16). Consequently, such tests are sometimes replaced with "fuzzy" comparisons (`if (abs(x-y) < epsilon) ...`, where epsilon is sufficiently small and tailored to the application, such as 1.0E−13). The wisdom of doing this varies greatly, and can require numerical analysis to bound epsilon. Values derived from the primary data representation and their comparisons should be performed in a wider, extended, precision to minimize the risk of such inconsistencies due to round-off errors. It is often better to organize the code in such a way that such tests are unnecessary. For example, in computational geometry, exact tests of whether a point lies off or on a line or plane defined by other points can be performed using adaptive precision or exact arithmetic methods.

Small errors in floating-point arithmetic can grow when mathematical algorithms perform operations an enormous number of times. A few examples are matrix inversion, eigenvector computation, and differential equation solving. These algorithms must be very carefully designed, using numerical approaches such as Iterative refinement, if they are to work well.

Summation of a vector of floating-point values is a basic algorithm in scientific computing, and so an awareness of when loss of significance can occur is essential. For example, if one is adding a very large number of numbers, the individual addends are very small compared with the sum. This can lead to loss of significance. A typical addition would then be something like

```
 3253.671
+   3.141276
-----------
 3256.812
```

The low 3 digits of the addends are effectively lost. Suppose, for example, that one needs to add many numbers, all approximately equal to 3. After 1000 of them have been added, the running sum is about 3000; the lost digits are not regained. The Kahan summation algorithm may be used to reduce the errors.

Round-off error can affect the convergence and accuracy of iterative numerical procedures. As an example, Archimedes approximated π by calculating the perimeters of polygons inscribing and circumscribing a circle, starting with hexagons, and successively doubling the number of sides. As noted above, computations may be rearranged in a way that is mathematically equivalent but less prone to error (numerical analysis). Two forms of the recurrence formula for the circumscribed polygon are:

$t_0 = 13 - \sqrt{t_0 = 13}$

- First form: $t_{i+1} = t_{2i} + 1 - - - - - \sqrt{-1 t_i t_i + 1 = t_i 2 + 1 - 1 t_i}$
- second form: $t_{i+1} = t_i t_{2i} + 1 - - - - - \sqrt{+1 t_i + 1 = t_i t_i 2 + 1 + 1}$

$\pi \sim 6 \times 2_i \times t_i \pi \sim 6 \times 2i \times t_i$, converging as $i \to \infty i \to \infty$

Here is a computation using IEEE "double" (a significand with 53 bits of precision) arithmetic:

```
 i 6 × 2ⁱ × tᵢ, first form 6 × 2ⁱ × tᵢ, second form
-----------------------------------------------------------
 0 3.4641016151377543863 3.4641016151377543863
 1 3.2153903091734710173 3.2153903091734723496
 2 3.1596599420974940120 3.1596599420975006733
 3 3.1460862151314012979 3.1460862151314352708
 4 3.1427145996453136334 3.1427145996453689225
 5 3.1418730499801259536 3.1418730499798241950
 6 3.1416627470548084133 3.1416627470568494473
 7 3.1416101765997805905 3.1416101766046906629
 8 3.1415970343230776862 3.1415970343215275928
 9 3.1415937488171150615 3.1415937487713536668
10 3.1415929278733740748 3.1415929273850979885
11 3.1415927256228504127 3.1415927220386148377
12 3.1415926717412858693 3.1415926707019992125
13 3.1415926189011456060 3.1415926578678454728
14 3.1415926717412858693 3.1415926546593073709
15 3.1415919358822321783 3.1415926538571730119
16 3.1415926717412858693 3.1415926536566394222
17 3.1415810075796233302 3.1415926536065061913
18 3.1415926717412858693 3.1415926535939728836
19 3.1414061547378810956 3.1415926535908393901
20 3.1405434924008406305 3.1415926535900560168
21 3.1400068646912273617 3.1415926535898608396
22 3.1349453756585929919 3.1415926535898122118
23 3.1400068646912273617 3.1415926535897995552
24 3.2245152435345525443 3.1415926535897968907
25 3.1415926535897962246
26 3.1415926535897962246
27 3.1415926535897962246
28 3.1415926535897962246
 The true value is 3.14159265358979323846264338327...
```

While the two forms of the recurrence formula are clearly mathematically equivalent, the first subtracts 1 from a number extremely close to 1, leading to an increasingly problematic loss of significant digits. As the recurrence is applied repeatedly, the accuracy improves at first, but then it deteriorates. It never gets better than about 8

digits, even though 53-bit arithmetic should be capable of about 16 digits of precision. When the second form of the recurrence is used, the value converges to 15 digits of precision.

Source: Wikipedia, https://en.wikipedia.org/wiki/Floating-point_arithmetic

Practice with Number Systems

Study this article if you need more practice at understanding number systems.

Objective

Familiarize the learner with method for expressing numbers and convert one method to another.

Introduction

A writing method for expressing numbers is called a "numeral system". In the most common numeral system, we write numbers with combinations of 10 symbols {0,1,2,3,4,5,6,7,8,9}. These symbols are called digits, and numbers that are expressed using 10 digits are called "decimal" or "base-10" numbers. The other most common numeral systems are binary, hexadecimal, and octal. The binary numeral system, or base-2 number system, represents numeric values using two symbols: 0 and 1. More specifically, the usual base-2 system is a positional notation with a radix of 2. Because of its straightforward implementation in digital electronic circuitry using logic gates, the binary system is used internally by almost all modern computers.

Decimal Numeral System

In the first method discussed we write numbers with combinations of 10 symbols {0,1,2,3,4,5,6,7,8,9} called digits. Numbers that are expressed with 10 digits are called "base-10" numbers or "Decimal Numeral System". **For example:**

2 (one digit)

45 (two digit)

643 (three digit)

8785 (four digit)

etc.

In Decimal Numeral Systems, the value of a digit is multiplied according to its placement in a numerical sequence: (base-number ^ 0,1,2,3,...), from right to left.

First digit = (base-number ^ 0): 10^0 = 1

Second digit =(base-number ^ 1): 10^1 = 10

Third digit =(base-number ^ 2): 10^2 = 100

Fourth digit =(base-number ^ 3): 10^3 = 1000

etc.

For example:

20= (2*10)+(0*1)=20+0=20

456=(4*100)+(5*10)+(6*1)=400+50+6

84568=(8*10000)+(4*1000)+(5*100)+(6*10)+(8*1)=80000+4000+500+60+8

Binary Numeral System

Numbers expressed with 2 symbols (0, 1) are called binary, or "base-2" numbers.

For example:

1 (one-digit-read: 1)

10 (two-digit-read: 1, 0)

100 (three-digit-read: 1,0,0)

1101 (four-digit-read: 1, 1, 0, 1)

etc.

In the Binary Numeral System, digits have a value specified, this value being equal with (base-number ^ 0,1,2,3,...): (right to left)

First digit (base-number^0): 2^0 = 1

Second digit (base-number^1): 2^1 = 2

Third digit (base-number^2): 2^2 = 4

Fourth digit (base-number^3): 2^3 = 8

etc.

Converting Binary to Decimal
To convert binary to decimal, each digit is multiplied by the value of its position, and the results are added.

For example:

10 = (1*2^1) + (0*2^0) = 1*2 + 0*1 = 2 + 0 = 2 → 10 (binary) =2 (decimal)

101 = (1*2^2) + (0*2^1) + (1*2^0) = 1*4 + 0*2 + 1*1 = 4 + 0 + 1 = 5 → 101 (binary) =5 (decimal)

11001 = (1*2^4) + (1*2^3) + (0*2^2) + (0*2^1) + (1*2^0) = 1*16 + 1*8 + 0*4 + 0*2 + 1*1 = 16 + 8 + 0 + 0 + 1 = 25 → 11001 (binary) =25 (decimal)

111011 = (1*2^5) + (1*2^4) + (1*2^3) + (0*2^2) + (1*2^1) + (1*2^0) = 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1*1 = 32 + 16 + 8 + 0 + 2 + 1 = 59 → 111011 (binary) =59 (decimal)

Converting Decimal to Binary
To convert decimal to binary

Divide the decimal number by 2

- If there IS a remainder the rightmost column will be a 1
- If there is NO remainder, the rightmost column will be a 0.

Then repeat the process, moving one column to the left each time until you have divided down to 1.

Example 1

15/2 = 7 remainder 1 (Binary number = ???1)
7/2 = 3 remainder 1 (Binary number = ??11)
3/2 = 1 remainder 1 (Binary number = ?111)
The final result will always be 1 in the leftmost column (Binary number = 1111)

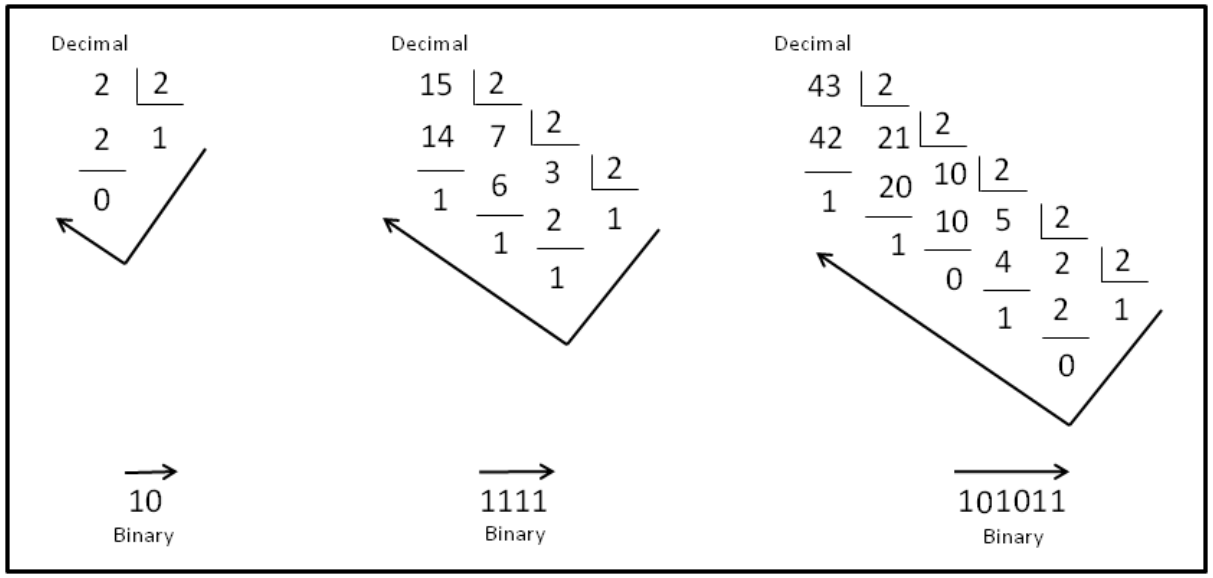
Example 2

74/2 = 37 remainder 0 (Binary number = ??????0)
37/2 = 18 remainder 1 (Binary number = ?????10)
18/2 = 9 remainder 0 (Binary number = ????010)
9/2 = 4 remainder 1 (Binary number = ???1010)
4/2 = 2 remainder 0 (Binary number = ??01010)
2/2 = 1 remainder 0 (Binary number = ?001010)
The final result will always be 1 in the leftmost column (Binary number = 1001010)

NB - Although I've put ? in at each stage, you won't know how many columns are needed until you complete the process.

For a shortcut to see how many columns are needed, find the largest factor of 2 that is smaller than the decimal number you started with, e.g.

Example 1: The largest factor less than 74 is 64, which is 2 to the power 6. As the furthest right column is 2 to the power 0, this means we need 7 columns.



Hexadecimal Numeral System

Numbers written with 16 symbols {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} are called "base-16" numbers. For example:

A (one digit)

B5 (two digit)

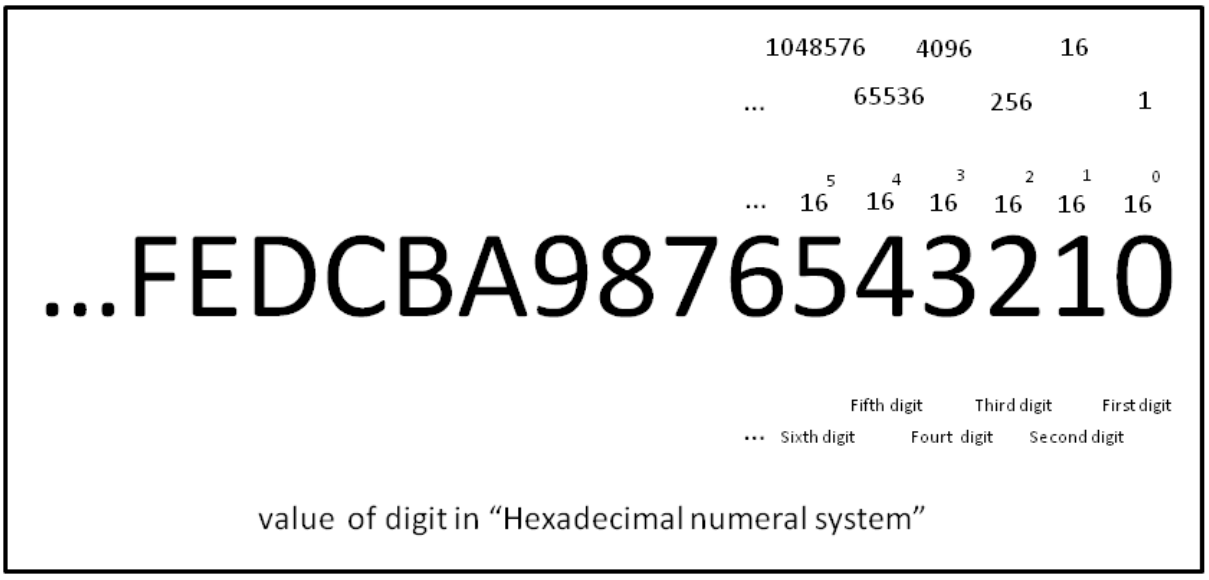
6C3 (three digit)

AF85 (four digit)

etc.

so:
A(hexadecimal)=10(decimal).
B(hexadecimal)=11(decimal).
C(hexadecimal)=12(decimal).
D(hexadecimal)=13(decimal).
E(hexadecimal)=14(decimal).
F(hexadecimal)=15(decimal)

In the "Hexadecimal Numeral System", digits have a value specified, this value of digits is equal with (base-number^ 0,1,2,3,...):(right to left)



First digit (base-number ^ 0): 16^0 = 1

Second digit (base-number ^ 1): 16^1 = 16

Third digit (base-number ^ 2): 16^2 = 256

fourth digit (base-number ^ 3): 16^3 = 4096

etc.

Converting Hexadecimal to Decimal
To convert hexadecimal to decimal, each digit is multiplied by the value of its position, and the results are added.

For example:

A = (10*16^0) = 10*1 = 10 → A(hexadecimal) =10(decimal)

B5 = (11*16^1) + (5*16^0) = 11*16 + 5*1 = 181 → B5(hexadecimal) =181(decimal)

6C3 = (6*16^2) + (12*16^1) + (3*16^0) = 6*256 + 12*16 + 3*1 = 1536 + 192 + 3 = 1731 → 6C3(hexadecimal) =1731(decimal)

AF85 = (10*16^3) + (15*16^2) + (8*16^1) + (5*16^0) = 10*4096 + 15*256 + 8*16 + 5*1 = 40960 + 3840 + 128 + 5 = 44933 → AF85(hexadecimal) =44933(decimal)

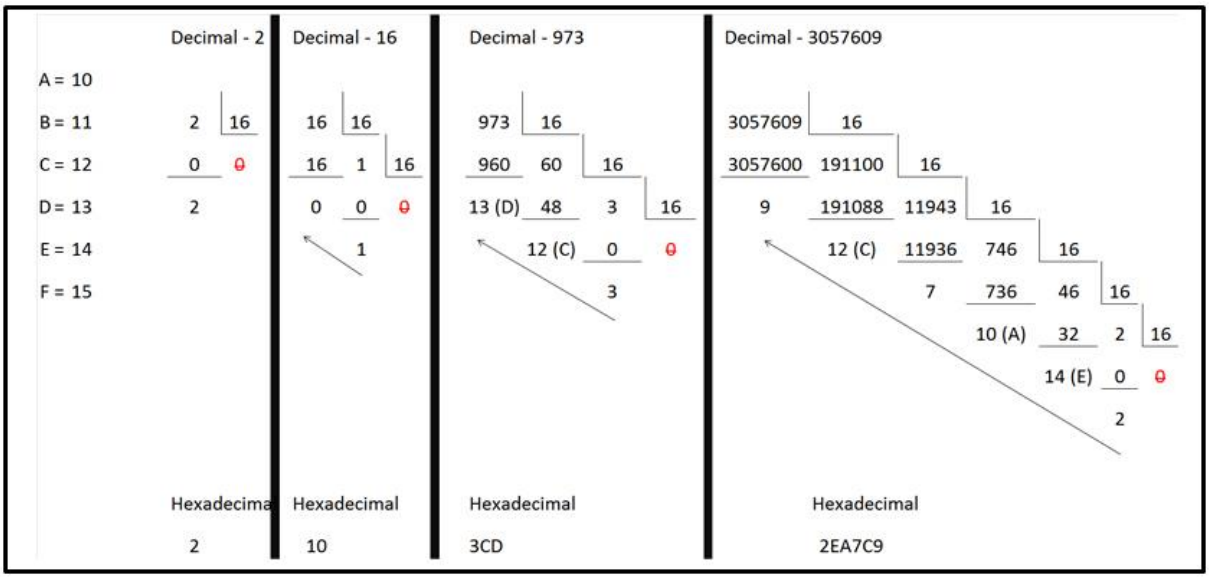
Converting Decimal to Hexadecimal

To convert decimal to hexadecimal

- Divide the decimal number by 16 - the remainder given is the last hexadecimal value.
- The quotient is then divided by 16 to get another remainder. Like the binary calculation the values are read right to left (first remainder value is the last in the hexadecimal number, then next to last, third to last, etc.)
- The process is terminated once a dividend (numerator) of less than 16 is reached. Continuing to divide by 16 would give a quotient of 0 which is indivisible.
- Keep in mind that 10-15 are represented as single character "numbers" in the hexadecimal system. A=10 , B=11 , C = 12, D = 13 , E= 14 , F =15 - remainders must reflect their appropriate hexadecimal value.

Examples

- Decimal 15
    - 15/16 remainder is 15 (>16 so process terminates) so the "number" value is F
- Decimal 16
    - 16/16 remainder is 0 [hex ?0]
    - The quotient of 1 is then divided - 1/16 which leaves a remainder of 1 (giving a quotient of 0 so process terminates) [hex 10]
- Decimal 45
    - 45/16 – remainder 13 [hex ?D]
    - Quotient 2 | 2/16 – remainder 2 [hex 2D]
- Decimal 47825
    - 47825/16 - remainder 1 [hex ???1]
    - Quotient 2989 | 2989/16 – remainder 13 [hex ??D1]
    - Quotient 186 | 186/16 – remainder 10 [hex ?AD1]
    - Quotient 11 | 11/16 – remainder 11 [hex BAD1]



Source: Wikiversity, https://en.wikiversity.org/wiki/Numeral_systems

Converting Decimal Numbers to Binary

This article demonstrates one method for converting from decimal to binary. This conversion is often used as a first step in converting from decimal to any other representation.

This is a quick method to convert a decimal number into a binary number.

The binary number is constructed from right to left.

1. If the decimal number is even, write a 0. If it's odd, write a 1.
2. Divide the number in two and round down.
3. Again, if the decimal number is even, write a 0. If it's odd, write a 1.
4. Keep dividing the number until you get to 1.

Example

Convert 99 to binary:

- 99 is odd, so start with: 1
- Divide 99 in 2 (49.5) and round down. 49 is odd, so add a 1 on the left: 11.
- Divide 49 in 2 (24.5) and round down. 24 is even, so add a 0 on the left: 011.
- Divide 24 in 2 (12). 12 is even, so add another 0 on the left: 0011.
- Divide 12 in 2 (6). 6 is even, so add another 0 on the left: 00011.
- Divide 6 in 2 (3). 3 is odd, so add a 1 on the left: 100011.
- Divide 3 in 2 (1.5) and round down. 1 is odd, so add a 1 on the left: 1100011.

99 is 1100011 in binary.

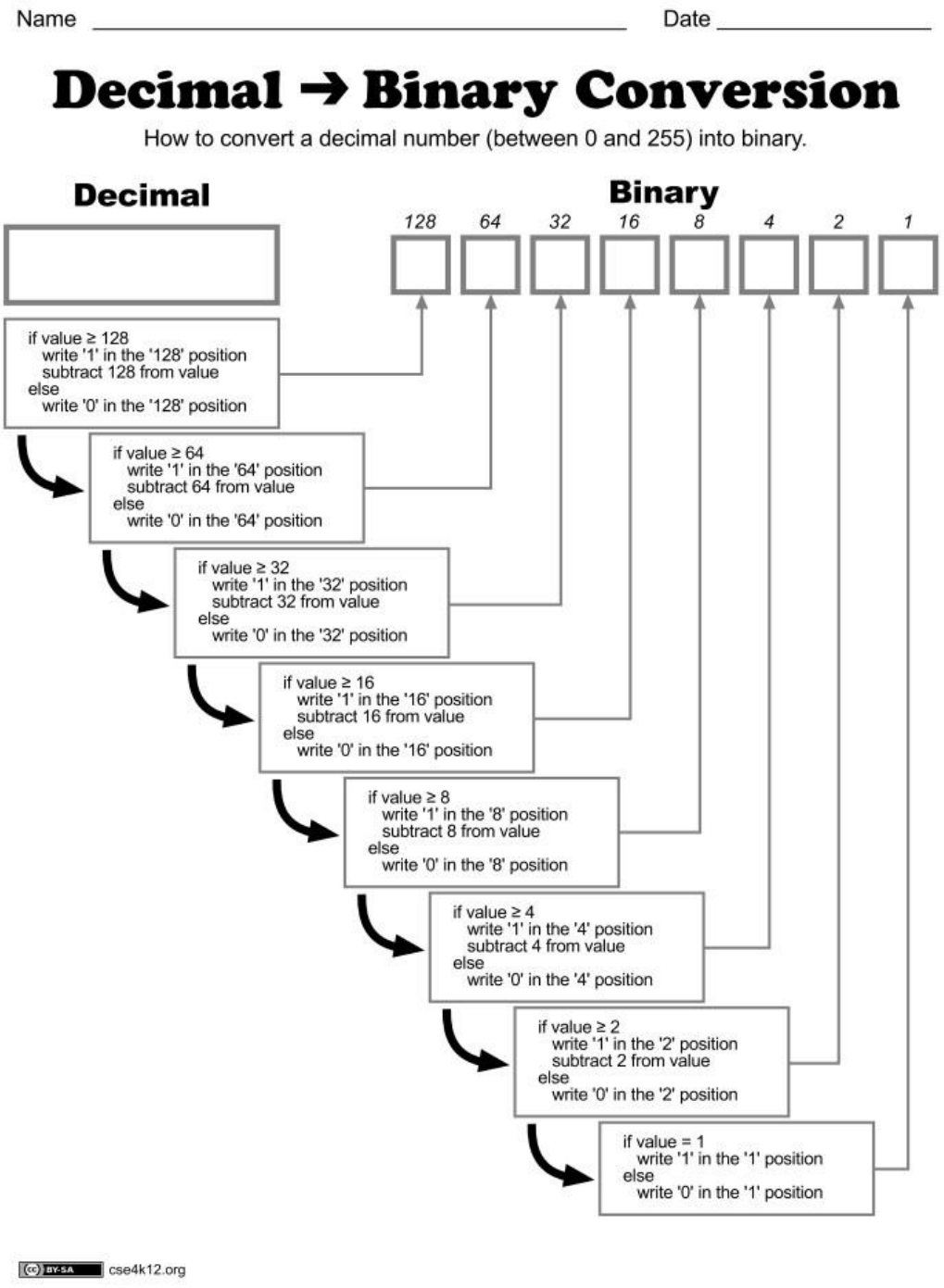
Memorization

Since this is meant to be done in one's head, maybe the ones and zeros could be kept track of with finger positions. Keep fingers by sides or on the table in front of you, slightly off the surface. Since it's calculated from right to left, the first binary digit would be represented by your right pinky and each additional digit would move to the left. If it's a 1, put your fingertip down. If it's a 0, put your finger down, but curl the finger.

Source: Art of
Memory, https://artofmemory.com/wiki/How_to_Convert_Decimal_Numbers_to_Binary

Another Way to Convert from Decimal to Binary

This article illustrates a second method for converting from decimal to binary. Use whichever method you prefer.



Source: cse4k12.org, https://cse4k12.org/binary/dec2bin.pdf

Fractions

This page contains an interactive decimal to binary conversion when there are fractions involved. Click on the 0's to change them into 1's. You can also see the 2's complement representation for negative numbers.

You might have looked at the interactive binary page and wondered what happens if we want to store numbers that aren't positive integers? After all, you've seen decimals and negative numbers on your computer. This page will show you how we can represent decimals/fractions and negative decimal numbers using binary. If you want to take this idea further, you can also look at normalised floating-point binary.

On the binary page, we said that in a number system based on **ten**s, each column heading (units, tens, hundreds, etc.) is **ten** times the value of the column heading to its right, and in a number system based on **two**s, each column heading is **two** times the one to its right. This relationship continues to the right of the decimal point, so in *denary* the column headings are 1/10, 1/100, 1/1000, etc., and in binary they are 1/2, 1/4, 1/8, etc.

Negative numbers are most often represented using something called *two's complement* (although there are other methods), with the left-most bit indicating whether the number is positive (0) or negative (1). To create a *one's complement* binary number, simply take a binary number and swap the 0s and 1s - e.g. 01100101 becomes 10011010. If we then add 1 to the one's complement, we get the two's complement - e.g. 01100101 becomes 10011011.

Click here to interact with binary fractions.

While on the page linked above, see if you can you work out how to represent 0.25. What about 0.75? Or -0.5? You will see that some numbers - e.g. 0.6 - can't be represented as an eight-bit binary number. This is the main problem that we encounter when trying to store *floating point numbers* on a computer. For an example of how this can affect us in practice, have a look at my first version of the standard-form page (and follow the instructions at the top).

At first sight, negative fractions in binary can look a bit confusing. Remember, though, that shifting binary numbers one place to the left (or right) multiplies (or divides) their value by *two*. For example, the binary representation of three (using three bits) is 011. The one's complement of this is 100, and the two's complement is 101, so minus three is 101. Shifting that two places to the right to give 1.01 is the same as dividing by four (diving by two and then by two again), and -3/4 is -0.75.

For a more in-depth discussion of number bases, look at the *Number Bases* page in the *Mathematics* section. You can also watch an introduction to binary and examples of how computers use binary on the *AdvancedICT* YouTube channel.

Why not practice your programming skills by creating a program that uses these techniques? Try to create a program that will convert a decimal number to binary - I can think of at least two ways to do it (one of which uses bitwise Boolean logic). Click here to download some curriculum programming examples.

Source: Advanced ICT, https://www.advanced-ict.info/interactive/fractions.html

2.3: Instruction Representation

MIPS Instructions

Read this article for an introduction to the three different instruction formats for the MIPS processor: the R-Format, the I-Format, and the J-Format instructions. MIPS is an acronym that stands for Microprocessor Instructions without Interlocked Pipeline Stages. MIPS is a RISC (Reduced Instruction Set Computer) introduced by MIPS technologies. Also, ISA, if you encounter it, stands for Instruction Set Architecture.

This page describes the implementation details of the MIPS instruction formats.

R Instructions

R instructions are used when all the data values used by the instruction are located in registers.

All R-type instructions have the following format:

```
OP rd, rs, rt
```

Where "OP" is the mnemonic for the particular instruction. *rs*, and *rt* are the source registers, and *rd* is the destination register. As an example, the **add** mnemonic can be used as:

```
add $s1, $s2, $s3
```

Where the values in *$s2* and *$s3* are added together, and the result is stored in *$s1*. In the main narrative of this book, the operands will be denoted by these names.

R Format

Converting an R mnemonic into the equivalent binary machine code is performed in the following way:

opcode	rs	rt	rd	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

opcode

The opcode is the machinecode representation of the instruction mnemonic. Several related instructions can have the same opcode. The opcode field is 6 bits long (bit 26 to bit 31).

rs, rt, rd

The numeric representations of the source registers and the destination register. These numbers correspond to the $X representation of a register, such as $0 or $31. Each of these fields is 5 bits long. (25 to 21, 20 to 16, and 15 to 11, respectively). Interestingly, rather than rs and rt being named r1 and r2 (for source register 1 and 2), the registers were named "rs" and "rt" for register source, register target and register destination.

Shift (shamt)

Used with the shift and rotate instructions, this is the amount by which the source operand *rs* is rotated/shifted. This field is 5 bits long (6 to 10).

Funct

For instructions that share an opcode, the **funct** parameter contains the necessary control codes to differentiate the different instructions. 6 bits long (0 to 5). Example: Opcode 0x00 accesses the ALU, and the funct selects which ALU function to use.

Function Codes

Because several functions can have the same opcode, R-Type instructions need a function (Func) code to identify what exactly is being done - for example, 0x00 refers to an ALU operation and 0x20 refers to ADDing specifically.

I Instructions

I instructions are used when the instruction must operate on an immediate value and a register value. Immediate values may be a maximum of 16 bits long. Larger numbers may not be manipulated by immediate instructions.

I instructions are called in the following way:

```
OP rt, IMM(rs)
```

However, **beq** and **bne** instructions are called in the following way:

```
OP rs, rt, IMM
```

Where *rt* is the target register, *rs* is the source register, and *IMM* is the immediate value. The immediate value can be up to 16 bits long. For instance, the **addi** instruction can be called as:

```
addi $s1, $s2, 100
```

Where the value of $s2 plus 100 is stored in $s1.

I Format

I instructions are converted into machine code words in the following format:

opcode	rs	rt	IMM
6 bits	5 bits	5 bits	16 bits

Opcode

The 6-bit opcode of the instruction. In I instructions, all mnemonics have a one-to-one correspondence with the underlying opcodes. This is because there is no **funct** parameter to differentiate instructions with an identical opcode. 6 bits (26 to 31)

rs, rt

The source and target register operands, respectively. 5 bits each (21 to 25 and 16 to 20, respectively).

IMM

The 16 bit immediate value. 16 bits (0 to 15). This value is usually used as the offset value in various instructions, and depending on the instruction, may be expressed in two's complement.

J Instructions

J instructions are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers.

J instructions are called in the following way:

```
OP LABEL
```

Where *OP* is the mnemonic for the particular jump instruction, and *LABEL* is the target address to jump to.

J Format

J instructions have the following machine-code format:

Opcode Pseudo-Address

Opcode

The 6 bit opcode corresponding to the particular jump command. (26 to 31).

Address

A 26-bit shortened address of the destination. (0 to 25). The two least significant bits are removed, and the 4 most significant bits are removed, and assumed to be the same as the current instruction's address.

FR Instructions

FR instructions are similar to the R instructions described above, except they are reserved for use with floating-point numbers:

Opcode fmt ft fs fd funct

FI Instructions

FI instructions are similar to the I instructions described above, except they are reserved for use with floating-point numbers:

Opcode fmt ft Imm

Opcodes

The following table contains a listing of MIPS instructions and the corresponding opcodes. Opcode and funct numbers are all listed in hexadecimal.

Mnemonic	Meaning	Type	Opcode	Funct
`add`	Add	R	0x00	0x20
`addi`	Add Immediate	I	0x08	NA
`addiu`	Add Unsigned Immediate	I	0x09	NA
`addu`	Add Unsigned	R	0x00	0x21

Mnemonic	Meaning	Type	Opcode	Funct
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
blez	Branch if Less Than or Equal to Zero	I	0x06	NA
bne	Branch if Not Equal	I	0x05	NA
bgtz	Branch on Greater Than Zero	I	0x07	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
jr	Jump to Address in Register	R	0x00	0x08
lb	Load Byte	I	0x20	NA
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
mfhi	Move from HI Register	R	0x00	0x10
mthi	Move to HI Register	R	0x00	0x11
mflo	Move from LO Register	R	0x00	0x12
mtlo	Move to LO Register	R	0x00	0x13
mfc0	Move from Coprocessor 0	R	0x10	NA
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA

Mnemonic	Meaning	Type	Opcode	Funct
sh	Store Halfword	I	0x29	NA
slt	Set to 1 if Less Than	R	0x00	0x2A
slti	Set to 1 if Less Than Immediate	I	0x0A	NA
sltiu	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
sltu	Set to 1 if Less Than Unsigned	R	0x00	0x2B
sll	Logical Shift Left	R	0x00	0x00
srl	Logical Shift Right (0-extended)	R	0x00	0x02
sra	Arithmetic Shift Right (sign-extended)	R	0x00	0x03
sub	Subtract	R	0x00	0x22
subu	Unsigned Subtract	R	0x00	0x23
sw	Store Word	I	0x2B	NA

Source: Wikibooks, https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats

2.4: Logical and Arithmetic Instructions

MIPS Arithmetic Instructions

Read this article to learn about arithmetic and logical instructions for the MIPS processor.

Register Arithmetic Instructions

Instruction: **add** type: **R Type**

This instruction adds the two operands together, and stores the result in the destination register. Negative numbers are handled automatically using two's complement notation, so different instructions do not need to be used for signed and unsigned numbers.

Instruction: **sub** type: **R Type**

The sub instruction subtracts the second source operand from the first source operand, and stores the result in the destination operand. In pseudo-code, the operation performs the following:

```
rd := rs - rt
```

Both **add** and **sub** trap if overflow occurs. However, some programming systems, like C, ignore integer overflow, so to improve performance "unsigned" versions of the instructions don't trap on overflow.

Instruction: **addu** type: **R Type**

Instruction: **subu** type: **R Type**

Multiplication and Division

The multiply and divide operations are slightly different from other operations. Even if they are R-type operations, they only take 2 operands. The result is stored in a special 64-bit result register. We will talk about the result register after this section.

Instruction: **mult** type: **R Type**

This operation multiplies the two operands together, and stores the result in rd. Multiplication operations must differentiate between signed and unsigned quantities, because the simplicity of Two's Complement Notation does not carry over to multiplication. The **mult** instruction multiplies and sign extends signed numbers.

The result of multiplying 2 32-bit numbers is a 64-bit result. We will discuss the 64-bit results below.

Instruction: **multu** type: **R Type**

The **multu** instruction multiplies the two operands together, and stores the result in rd. This instruction is for unsigned numbers only, and does not sign extend a negative result. This operation also creates a 64-bit result.

Instruction: **div** type: **R Type**

The div instruction divides the first argument by the second argument. The quotient is stored in the lowest 32-bits of the result register. The remainder is stored in the highest 32-bits of the result register. Like multiplication, division requires a differentiation between signed and unsigned numbers. This operation uses signed numbers.

Instruction: **divu** type: **R Type**

Like the div instruction, this operation divides the first operand by the second operand. The quotient is stored in the lowest 32-bits of the result, and the remainder is stored in the highest 32-bits of the result. This operand divides unsigned numbers, and will not sign-extend the result.

64-Bit Results

The 64-bit result register is broken into two 32-bit segments: **HI** and **LO**. We can interface with these registers using the mfhi and mflo operations, respectively.

Instruction: **mfhi** type: **R Type**

Takes only 1 operand. This instruction moves the high-32 bits of the result register into the target register.

Instruction: **mflo** type: **R Type**

Also takes only 1 operand. Moves the value from the LO part of the result register into the specified register.

If the upper (most significant) 32 bits of a product are unimportant to computation, programmers may save a step by using instructions that discard the upper 32 bits.

Instruction: **mul** type: **R Type**

There is no unsigned version of the **mul** instruction. The **mul** instruction may also clobber the existing values in HI and LO.

Register Logic Instructions

These operations perform bit-wise logical operations on their operands.

Instruction: **and** type: **R Type**

Performs a bitwise AND operation on the two operands, and stores the result in rd.

Instruction: **or** type: **R Type**

Performs a bitwise OR operation on the two operands, and stores the result in rd.

Instruction: **nor** type: **R Type**

Performs a bitwise NOR operation on the two operands, and stores the result in rd.

Instruction: **xor** type: **R Type**

Performs a bitwise XOR operation on the two operands, and stores the result in rd.

Immediate Arithmetic Instructions

These instructions sign-extend the 16-bit immediate value to 32-bits and performs the same operation as the instruction without the trailing "i".

Instruction: **addi** type: **I Type**

Instruction: **addiu** type: **I Type**

To subtract, use a negative immediate.

Immediate Logic Instructions

All logical functions zero-extend the immediate.

Instruction: **andi** type: **I Type**

Takes the bitwise AND of rs with the immediate and stores the result in rt.

Instruction: **ori** type: **I Type**

Takes the bitwise OR of rs with the immediate and stores the result in rt.

Instruction: **xori** type: **I Type**

Takes the bitwise XOR of rs with a the immediate and stores the result in rt.

Shift instructions

Instruction: **sll** type: **R Type**

Logical shift left: rd ← rt << shamt. Fills bits from right with zeros.

Instruction: **srl** type: **R Type**

Logical shift right: rd ← rt >> shamt. Fills bits from left with zeros.

Instruction: **sra** type: **R Type**

Arithmetic shift right. If rt is negative, the leading bits are filled in with ones instead of zeros: rd ← rt >> shamt.

Because not all shift amounts are known in advance, MIPS defines versions of these instructions that shift by the amount in the rs register. The behavior is otherwise identical.

Instruction: **sllv** type: **R Type**

Instruction: **srlv** type: **R Type**

Instruction: **srav** type: **R Type**

Source: Wikibooks, https://en.wikibooks.org/wiki/MIPS_Assembly/Arithmetic_Instructions

2.5: Control Instructions

MIPS Control Flow Instructions

Read this article to learn about the control flow instructions for the MIPS processor, including the basic ones: jump and branch instructions.

Jump Instruction

The jump instructions load a new value into the PC register, which stores the value of the instruction being executed. This causes the next instruction read from memory to be retrieved from a new location.

Instruction: **j** type: **J Type**

The **j** instruction loads an immediate value into the PC register. This immediate value is either a numeric offset or a label (and the assembler converts the label into an offset).

Instruction: **jr** type: **R Type**

The **jr** instruction loads the PC register with a value stored in a register. As such, the jr instruction can be called as such:

```
jr $t0
```

assuming the target jump location is located in $t0.

Jump and Link

Jump and Link instructions are similar to the jump instructions, except that they store the address of the next instruction (the one immediately after the jump) in the return address ($ra; $31) register. This allows a subroutine to return to the main body routine after completion.

Instruction: **jal** type: **J Type**

Like the **j** instruction, except that the return address is loaded into the $ra register.

Instruction: **jalr** type: **R Type**

The same as the **jr** instruction, except that the return address is loaded into a specified register (or $ra if not specified)

Example
Let's say that we have a subroutine that starts with the label MySub. We can call the subroutine using the following line:

```
jal MySub
...
```

And we can define MySub as follows to return to the main body of the parent routine:

```
jr $ra
```

Branch Instructions
Instead of using rt as a destination operand, rs and rt are both used as source operands and the immediate is sign extended and added to the PC to calculate the address of the instruction to jump to if the branch is taken.

Instruction: **beq** type: **I Type**

Branch if rs and rt are equal. If rs = rt, PC ← PC + 4 + imm.

Instruction: **bne** type: **I Type**

Branch if rs and rt are not equal. If rs ≠ rt, PC ← PC + 4 + imm.

Instruction: **bgez** type: **I Type**

Branch if rs is greater than or equal to zero. If rs ≥ 0, PC ← PC + 4 + imm.

Instruction: **blez** type: **I Type**

Branch if rs is less than or equal to zero. If rs ≤ 0, PC ← PC + 4 + imm.

Instruction: **bgtz** type: **I Type**

Branch if rs is greater than zero. If rs > 0, PC ← PC + 4 + imm.

Instruction: **bltz** type: **I Type**

Branch if rs is less than zero. If rs < 0, PC ← PC + 4 + imm.

Set Instructions
These instructions set rd to 1 if their condition is true. They can be used in combination with **beq** and **bne** and $zero to branch based on the comparison of two registers.

Instruction: **slt** type: **R Type**

If rs < rt, rd ← 1, else 0.

Instruction: **slti** type: **I Type**

If rs < imm, rd ← 1, else 0. The immediate is sign extended.

Instruction: **sltu** type: **R Type**

If rs < rt, rd ← 1, else 0. Treat rs and rt as unsigned integers.

Instruction: **sltiu** type: **I Type**

If rs < imm, rd ← 1, else 0. The immediate is sign extended, but both rs and the extended immediate are treated as *unsigned* integers. The sign extension allows the immediate to represent both very large and very small unsigned integers.

Source:
Wikibooks, https://en.wikibooks.org/wiki/MIPS_Assembly/Control_Flow_Instructions

2.6: Instructions for Memory Operations

MIPS Memory Instructions

Read this article to learn about memory instructions for the MIPS processor.

Load and store instructions use a special syntax:

```
instr rt, imm(rs)
```

The memory address used for the load or store is rs + imm. The immediate is sign-extended.

Load Instructions

Instruction: **lbu** type: **I Type**

Loads a byte and does not sign-extend the value.

Instruction: **lhu** type: **I Type**

Loads a halfword, or two bytes, and does not sign-extend the value. The halfword must be aligned (i.e., it must start at an even address).

Instruction: **lw** type: **I Type**

Loads a word (four-bytes) from memory. The word must be aligned (i.e., the last two bits of the address must be zero).

Store Instructions

Instruction: **sb** type: **I Type**

Stores the least significant (rightmost) byte of rt to memory.

Instruction: **sh** type: **I Type**

Stores the least significant (rightmost) halfword of rt to memory.

Instruction: **sw** type: **I Type**

Stores the contents of rt in memory.

Source: Wikibooks, https://en.wikibooks.org/wiki/MIPS_Assembly/Memory_Instructions

2.7: Different Modes for Addressing Memory

Addressing Memory

Read this article to study the various formats for addressing memory.

Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs. The various addressing modes that are defined in a given instruction set architecture define how the machine language instructions in that architecture identify the operand(s) of each instruction. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

In computer programming, addressing modes are primarily of interest to those who write in assembly languages and to compiler writers. For a related concept see orthogonal instruction set which deals with the ability of any instruction to use any addressing mode.

Caveats

Note that there is no generally accepted way of naming the various addressing modes. In particular, different authors and computer manufacturers may give different names to the

same addressing mode, or the same names to different addressing modes. Furthermore, an addressing mode which, in one given architecture, is treated as a single addressing mode may represent functionality that, in another architecture, is covered by two or more addressing modes. For example, some complex instruction set computer (CISC) architectures, such as the Digital Equipment Corporation (DEC) VAX, treat registers and literal or immediate constants as just another addressing mode. Others, such as the IBM System/360 and its successors, and most reduced instruction set computer (RISC) designs, encode this information within the instruction. Thus, the latter machines have three distinct instruction codes for copying one register to another, copying a literal constant into a register, and copying the contents of a memory location into a register, while the VAX has only a single "MOV" instruction.

The term "addressing mode" is itself subject to different interpretations: either "memory address calculation mode" or "operand accessing mode". Under the first interpretation, instructions that do not read from memory or write to memory (such as "add literal to register") are considered not to have an "addressing mode". The second interpretation allows for machines such as VAX which use operand mode bits to allow for a register or for a literal operand. Only the first interpretation applies to instructions such as "load effective address".

The addressing modes listed below are divided into code addressing and data addressing. Most computer architectures maintain this distinction, but there are (or have been) some architectures which allow (almost) all addressing modes to be used in any context.

The instructions shown below are purely representative in order to illustrate the addressing modes, and do not necessarily reflect the mnemonics used by any particular computer.

Number of addressing modes

Different computer architectures vary greatly as to the number of addressing modes they provide in hardware. There are some benefits to eliminating complex addressing modes and using only one or a few simpler addressing modes, even though it requires a few extra instructions, and perhaps an extra register. It has proven much easier to design pipelined CPUs if the only addressing modes available are simple ones.

Most RISC architectures have only about five simple addressing modes, while CISC architectures such as the DEC VAX have over a dozen addressing modes, some of which are quite complicated. The IBM System/360 architecture had only three addressing modes; a few more have been added for the System/390.

When there are only a few addressing modes, the particular addressing mode required is usually encoded within the instruction code (e.g. IBM System/360 and successors, most RISC). But when there are lots of addressing modes, a specific field is often set aside in

the instruction to specify the addressing mode. The DEC VAX allowed multiple memory operands for almost all instructions, and so reserved the first few bits of each operand specifier to indicate the addressing mode for that particular operand. Keeping the addressing mode specifier bits separate from the opcode operation bits produces an orthogonal instruction set.

Even on a computer with many addressing modes, measurements of actual programs indicate that the simple addressing modes listed below account for some 90% or more of all addressing modes used. Since most such measurements are based on code generated from high-level languages by compilers, this reflects to some extent the limitations of the compilers being used.

Useful side effect

Some instruction set architectures, such as Intel x86 and IBM/360 and its successors, have a **load effective address** instruction. This performs a calculation of the effective operand address, but instead of acting on that memory location, it loads the address that would have been accessed into a register. This can be useful when passing the address of an array element to a subroutine. It may also be a slightly sneaky way of doing more calculations than normal in one instruction; for example, using such an instruction with the addressing mode "base+index+offset" (detailed below) allows one to add two registers and a constant together in one instruction.

Simple addressing modes for code

Absolute or direct

```
 +----+------------------------------+
 |jump|              address         |
 +----+------------------------------+
 (Effective PC address = address)
```

The effective address for an absolute instruction address is the address parameter itself with no modifications.

PC-relative

```
 +----+------------------------------+
 |jump|              offset          |        jump relative
 +----+------------------------------+
 (Effective PC address = next instruction address + offset, offset may be
negative)
```

The effective address for a PC-relative instruction address is the offset parameter added to the address of the next instruction. This offset is usually signed to allow reference to code both before and after the instruction.

This is particularly useful in connection with jumps, because typical jumps are to nearby instructions (in a high-level language most **if** or **while** statements are reasonably short). Measurements of actual programs suggest that an 8 or 10 bit offset is large enough for some 90% of conditional jumps (roughly ±128 or ±512 bytes).

Another advantage of PC-relative addressing is that the code may be position-independent, i.e. it can be loaded anywhere in memory without the need to adjust any addresses.

Some versions of this addressing mode may be conditional referring to two registers ("jump if reg1=reg2"), one register ("jump unless reg1=0") or no registers, implicitly referring to some previously-set bit in the status register. See also conditional execution below.

Register indirect

```
+-------+-----+
|jumpVia| reg |
+-------+-----+
(Effective PC address = contents of register 'reg')
```

The effective address for a Register indirect instruction is the address in the specified register. For example, (A7) to access the content of address register A7.

The effect is to transfer control to the instruction whose address is in the specified register.

Many RISC machines, as well as the CISC IBM System/360 and successors, have subroutine call instructions that place the return address in an address register—the register-indirect addressing mode is used to return from that subroutine call.

Sequential addressing modes

Sequential execution

```
+------+
|  nop |           execute the following instruction
+------+
(Effective PC address = next instruction address)
```

The CPU, after executing a sequential instruction, immediately executes the following instruction.

Sequential execution is not considered to be an addressing mode on some computers.

Most instructions on most CPU architectures are sequential instructions. Because most instructions are sequential instructions, CPU designers often add features that deliberately sacrifice performance on the other instructions—branch instructions—in order to make these sequential instructions run faster.

Conditional branches load the PC with one of 2 possible results, depending on the condition—most CPU architectures use some other addressing mode for the "taken" branch, and sequential execution for the "not taken" branch.

Many features in modern CPUs -- instruction prefetch and more complex pipelineing, out-of-order execution, etc. -- maintain the illusion that each instruction finishes before the next one begins, giving the same final results, even though that's not exactly what happens internally.

Each "basic block" of such sequential instructions exhibits both temporal and spatial locality of reference.

CPUs that do not use sequential execution

CPUs that do not use sequential execution with a program counter are extremely rare. In some CPUs, each instruction always specifies the address of next instruction. Such CPUs have an instruction pointer that holds that specified address; it is not a program counter because there is no provision for incrementing it. Such CPUs include some drum memory computers such as the IBM 650, the SECD machine, and the RTX 32P.

Other computing architectures go much further, attempting to bypass the von Neumann bottleneck using a variety of alternatives to the program counter.

Conditional execution

Some computer architectures have conditional instructions (such as ARM, but no longer for all instructions in 64-bit mode) or conditional load instructions (such as x86) which can in some cases make conditional branches unnecessary and avoid flushing the instruction pipeline. An instruction such as a 'compare' is used to set a condition code, and subsequent instructions include a test on that condition code to see whether they are obeyed or ignored.

Skip

```
+------+-----+-----+
|skipEQ| reg1| reg2| skip the next instruction if reg1=reg2
+------+-----+-----+
(Effective PC address = next instruction address + 1)
```

Skip addressing may be considered a special kind of PC-relative addressing mode with a fixed "+1" offset. Like PC-relative addressing, some CPUs have versions of this addressing mode that only refer to one register ("skip if reg1=0") or no registers, implicitly referring to some previously-set bit in the status register. Other CPUs have a version that selects a specific bit in a specific byte to test ("skip if bit 7 of reg12 is 0").

Unlike all other conditional branches, a "skip" instruction never needs to flush the instruction pipeline, though it may need to cause the next instruction to be ignored.

Simple addressing modes for data

Register (or Register Direct)

```
+------+-----+-----+-----+
|  mul | reg1| reg2| reg3|     reg1 := reg2 * reg3;
+------+-----+-----+-----+
```

This "addressing mode" does not have an effective address and is not considered to be an addressing mode on some computers.

In this example, all the operands are in registers, and the result is placed in a register.

Base plus offset, and variations

This is sometimes referred to as 'base plus displacement'

```
+------+-----+-----+----------------+
| load | reg | base|      offset    |      reg := RAM[base + offset]
+------+-----+-----+----------------+
(Effective address = offset + contents of specified base register)
```

The offset is usually a signed 16-bit value (though the 80386 expanded it to 32 bits).

If the offset is zero, this becomes an example of *register indirect* addressing; the effective address is just the value in the base register.

On many RISC machines, register 0 is fixed at the value zero. If register 0 is used as the base register, this becomes an example of *absolute addressing*. However, only a small portion of memory can be accessed (64 kilobytes, if the offset is 16 bits).

The 16-bit offset may seem very small in relation to the size of current computer memories (which is why the 80386 expanded it to 32-bit). It could be worse: IBM System/360 mainframes only have an unsigned 12-bit offset. However, the principle of locality of reference applies: over a short time span, most of the data items a program wants to access are fairly close to each other.

This addressing mode is closely related to the indexed absolute addressing mode.

Example 1: Within a subroutine a programmer will mainly be interested in the parameters and the local variables, which will rarely exceed 64 KB, for which one base register (the frame pointer) suffices. If this routine is a class method in an object-oriented language, then a second base register is needed which points at the attributes for the current object (**this** or **self** in some high level languages).

Example 2: If the base register contains the address of a composite type (a record or structure), the offset can be used to select a field from that record (most records/structures are less than 32 kB in size).

Immediate/literal

```
+------+-----+-----+----------------+
|  add | reg1| reg2|     constant   | reg1 := reg2 + constant;
+------+-----+-----+----------------+
```

This "addressing mode" does not have an effective address, and is not considered to be an addressing mode on some computers.

The constant might be signed or unsigned. For example, `move.l #$FEEDABBA, D0` to move the immediate hex value of "FEEDABBA" into register D0.

Instead of using an operand from memory, the value of the operand is held within the instruction itself. On the DEC VAX machine, the literal operand sizes could be 6, 8, 16, or 32 bits long.

Andrew Tanenbaum showed that 98% of all the constants in a program would fit in 13 bits (see RISC design philosophy).

Implicit

```
+-----------------+
| clear carry bit |
+-----------------+
+-------------------+
| clear Accumulator |
+-------------------+
```

The implied addressing mode, also called the implicit addressing mode (X86 assembly language), does not explicitly specify an effective address for either the source or the destination (or sometimes both).

Either the source (if any) or destination effective address (or sometimes both) is implied by the opcode.

Implied addressing was quite common on older computers (up to mid-1970s). Such computers typically had only a single register in which arithmetic could be performed—the accumulator. Such accumulator machines implicitly reference that accumulator in almost every instruction. For example, the operation < a := b + c; > can be done using the sequence < load b; add c; store a; > -- the destination (the accumulator) is implied in every "load" and "add" instruction; the source (the accumulator) is implied in every "store" instruction.

Later computers generally had more than one general purpose register or RAM location which could be the source or destination or both for arithmetic—and so later computers need some other addressing mode to specify the source and destination of arithmetic.

Among the x86 instructions, some use implicit registers for one of the operands or results (multiplication, division, counting conditional jump).

Many computers (such as x86 and AVR) have one special-purpose register called the stack pointer which is implicitly incremented or decremented when pushing or popping data from the stack, and the source or destination effective address is (implicitly) the address stored in that stack pointer.

Many 32-bit computers (such as 68000, ARM, or PowerPC) have more than one register which could be used as a stack pointer—and so use the "register autoincrement indirect" addressing mode to specify which of those registers should be used when pushing or popping data from a stack.

Some current computer architectures (e.g. IBM/390 and Intel Pentium) contain some instructions with implicit operands in order to maintain backwards compatibility with earlier designs.

On many computers, instructions that flip the user/system mode bit, the interrupt-enable bit, etc. implicitly specify the special register that holds those bits. This simplifies the hardware necessary to trap those instructions in order to meet the Popek and Goldberg virtualization requirements—on such a system, the trap logic does not need to look at any operand (or at the final effective address), but only at the opcode.

A few CPUs have been designed where every operand is always implicitly specified in every instruction -- zero-operand CPUs.

Other addressing modes for code or data

Absolute/direct

```
+------+-----+---------------------------------------+
| load | reg |                 address               |
+------+-----+---------------------------------------+
(Effective address = address as given in instruction)
```

This requires space in an instruction for quite a large address. It is often available on CISC machines which have variable-length instructions, such as x86.

Some RISC machines have a special *Load Upper Literal* instruction which places a 16- or 20-bit constant in the top half of a register. That can then be used as the base register in a base-plus-offset addressing mode which supplies the low-order 16 or 12 bits. The combination allows a full 32-bit address.

Indexed absolute

```
 +------+-----+-----+-------------------------------+
 | load | reg |index|              address          |
 +------+-----+-----+-------------------------------+
 (Effective address = address + contents of specified index register)
```

This also requires space in an instruction for quite a large address. The address could be the start of an array or vector, and the index could select the particular array element required. The processor may scale the index register to allow for the size of each array element.

Note that this is more or less the same as base-plus-offset addressing mode, except that the offset in this case is large enough to address any memory location.

Example 1: Within a subroutine, a programmer may define a string as a local constant or a static variable. The address of the string is stored in the literal address in the instruction. The offset—which character of the string to use on this iteration of a loop—is stored in the index register.

Example 2: A programmer may define several large arrays as globals or as class variables. The start of the array is stored in the literal address (perhaps modified at program-load time by a relocating loader) of the instruction that references it. The offset—which item from the array to use on this iteration of a loop—is stored in the index register. Often the instructions in a loop re-use the same register for the loop counter and the offsets of several arrays.

Base plus index

```
 +------+-----+-----+-----+
 | load | reg | base|index|
 +------+-----+-----+-----+
 (Effective address = contents of specified base register + contents of specified
index register)
```

The base register could contain the start address of an array or vector, and the index could select the particular array element required. The processor may scale the index register to allow for the size of each array element. This could be used for accessing elements of an array passed as a parameter.

Base plus index plus offset

```
 +------+-----+-----+-----+----------------+
 | load | reg | base|index|        offset  |
 +------+-----+-----+-----+----------------+
 (Effective address = offset + contents of specified base register + contents of
specified index register)
```

The base register could contain the start address of an array or vector of records, the index could select the particular record required, and the offset could select a field within

that record. The processor may scale the index register to allow for the size of each array element.

Scaled

```
 +------+-----+-----+-----+
 | load | reg | base|index|
 +------+-----+-----+-----+
 (Effective address = contents of specified base register + scaled contents of
specified index register)
```

The base register could contain the start address of an array or vector data structure, and the index could contain the offset of the one particular array element required.

This addressing mode dynamically scales the value in the index register to allow for the size of each array element, e.g. if the array elements are double precision floating-point numbers occupying 8 bytes each then the value in the index register is multiplied by 8 before being used in the effective address calculation. The scale factor is normally restricted to being a power of two, so that shifting rather than multiplication can be used.

Register indirect

```
 +------+------+-----+
 | load | reg1 | base|
 +------+------+-----+
 (Effective address = contents of base register)
```

A few computers have this as a distinct addressing mode. Many computers just use *base plus offset* with an offset value of 0. For example, (A7)

Register autoincrement indirect

```
 +------+-----+-------+
 | load | reg | base  |
 +------+-----+-------+
 (Effective address = contents of base register)
```

After determining the effective address, the value in the base register is incremented by the size of the data item that is to be accessed. For example, (A7)+ would access the content of the address register A7, then increase the address pointer of A7 by 1 (usually 1 word). Within a loop, this addressing mode can be used to step through all the elements of an array or vector.

In high-level languages it is often thought to be a good idea that functions which return a result should not have side effects (lack of side effects makes program understanding and validation much easier). This addressing mode has a side effect in that the base register is altered. If the subsequent memory access causes an error (e.g. page fault, bus error, address error) leading to an interrupt, then restarting the instruction becomes

much more problematic since one or more registers may need to be set back to the state they were in before the instruction originally started.

There have been at least two computer architectures which have had implementation problems with regard to recovery from interrupts when this addressing mode is used:

- Motorola 68000 (address is represented in 24 bits). Could have one or two autoincrement register operands. The 68010+ resolved the problem by saving the processor's internal state on bus or address errors.
- DEC VAX. Could have up to 6 autoincrement register operands. Each operand access could cause two page faults (if operands happened to straddle a page boundary). Of course the instruction itself could be over 50 bytes long and might straddle a page boundary as well!

Register autodecrement indirect

```
+------+-----+-----+
| load | reg | base|
+------+-----+-----+
(Effective address = new contents of base register)
```

Before determining the effective address, the value in the base register is decremented by the size of the data item which is to be accessed.

Within a loop, this addressing mode can be used to step backwards through all the elements of an array or vector. A stack can be implemented by using this mode in conjunction with the previous addressing mode (autoincrement).

See the discussion of side-effects under the autoincrement addressing mode.

Memory indirect

Any of the addressing modes mentioned in this article could have an extra bit to indicate indirect addressing, i.e. the address calculated using some mode is in fact the address of a location (typically a complete word) which contains the actual effective address.

Indirect addressing may be used for code or data. It can make implementation of *pointers*, *references*, or *handles* much easier, and can also make it easier to call subroutines which are not otherwise addressable. Indirect addressing does carry a performance penalty due to the extra memory access involved.

Some early minicomputers (e.g. DEC PDP-8, Data General Nova) had only a few registers and only a limited addressing range (8 bits). Hence the use of memory indirect addressing was almost the only way of referring to any significant amount of memory.

PC-relative

```
+------+------+---------+----------------+
| load | reg1 | base=PC |     offset     |
+------+------+---------+----------------+
reg1 := RAM[PC + offset]
(Effective address = PC + offset)
```

The PC-relative addressing mode can be used to load a register with a value stored in program memory a short distance away from the current instruction. It can be seen as a special case of the "base plus offset" addressing mode, one that selects the program counter (PC) as the "base register".

There are a few CPUs that support PC-relative data references. Such CPUs include:

The MOS 6502 and its derivatives used relative addressing for all branch instructions. Only these instructions used this mode, jumps used a variety of other addressing modes.

The x86-64 architecture and the 64-bit ARMv8-A architecture have PC-relative addressing modes, called "RIP-relative" in x86-64 and "literal" in ARMv8-A. The Motorola 6809 also supports a PC-relative addressing mode.

The PDP-11 architecture, the VAX architecture, and the 32-bit ARM architectures support PC-relative addressing by having the PC in the register file.

When this addressing mode is used, the compiler typically places the constants in a literal pool immediately before or immediately after the subroutine that uses them, to prevent accidentally executing those constants as instructions.

This addressing mode, which always fetches data from memory or stores data to memory and then sequentially falls through to execute the next instruction (the effective address points to data), should not be confused with "PC-relative branch" which does not fetch data from or store data to memory, but instead branches to some other instruction at the given offset (the effective address points to an executable instruction).

Obsolete addressing modes

The addressing modes listed here were used in the 1950–1980 period, but are no longer available on most current computers. This list is by no means complete; there have been many other interesting and peculiar addressing modes used from time to time, e.g. absolute-minus-logical-OR of two or three index registers.

Multi-level memory indirect

If the word size is larger than the address, then the word referenced for memory-indirect addressing could itself have an indirect flag set to indicate another memory indirect cycle. This flag is referred to as an **indirection bit**, and the resulting pointer is a tagged

pointer, the indirection bit tagging whether it is a direct pointer or an indirect pointer. Care is needed to ensure that a chain of indirect addresses does not refer to itself; if it does, one can get an infinite loop while trying to resolve an address.

The IBM 1620, the Data General Nova, the HP 2100 series, and the NAR 2 each have such a multi-level memory indirect, and could enter such an infinite address calculation loop. The memory indirect addressing mode on the Nova influenced the invention of indirect threaded code.

The DEC PDP-10 computer with 18-bit addresses and 36-bit words allowed multi-level indirect addressing with the possibility of using an index register at each stage as well.

Memory-mapped registers

On some computers, the registers were regarded as occupying the first 8 or 16 words of memory (e.g. ICL 1900, DEC PDP-10). This meant that there was no need for a separate "add register to register" instruction – one could just use the "add memory to register" instruction.

In the case of early models of the PDP-10, which did not have any cache memory, a tight inner loop loaded into the first few words of memory (where the fast registers were addressable if installed) ran much faster than it would have in magnetic core memory.

Later models of the DEC PDP-11 series mapped the registers onto addresses in the input/output area, but this was primarily intended to allow remote diagnostics. Confusingly, the 16-bit registers were mapped onto consecutive 8-bit byte addresses.

Memory indirect and autoincrement

The DEC PDP-8 minicomputer had eight special locations (at addresses 8 through 15). When accessed via memory indirect addressing, these locations would automatically increment after use. This made it easy to step through memory in a loop without needing to use any registers to handle the steps.

The Data General Nova minicomputer had 16 special memory locations at addresses 16 through 31. When accessed via memory indirect addressing, 16 through 23 would automatically increment before use, and 24 through 31 would automatically decrement before use.

Zero page

The Data General Nova, Motorola 6800 family, and MOS Technology 6502 family of processors had very few internal registers. Arithmetic and logical instructions were mostly

performed against values in memory as opposed to internal registers. As a result, many instructions required a two-byte (16-bit) location to memory. Given that opcodes on these processors were only one byte (8 bits) in length, memory addresses could make up a significant part of code size.

Designers of these processors included a partial remedy known as "zero page" addressing. The initial 256 bytes of memory ($0000 – $00FF; a.k.a., page "0") could be accessed using a one-byte absolute or indexed memory address. This reduced instruction execution time by one clock cycle and instruction length by one byte. By storing often-used data in this region, programs could be made smaller and faster.

As a result, the zero page was used similarly to a register file. On many systems, however, this resulted in high utilization of the zero page memory area by the operating system and user programs, which limited its use since free space was limited.

Direct page

The zero page address mode was enhanced in several late model 8-bit processors, including the WDC 65816, the CSG 65CE02, and the Motorola 6809. The new mode, known as "direct page" addressing, added the ability to move the 256-byte zero page memory window from the start of memory (offset address $0000) to a new location within the first 64 KB of memory.

The CSG 65CE02 allowed the direct page to be moved to any 256-byte boundary within the first 64 KB of memory by storing an 8-bit offset value in the new base page (B) register. The Motorola 6809 could do the same with its direct page (DP) register. The WDC 65816 went a step further and allowed the direct page to be moved to any location within the first 64 KB of memory by storing a 16-bit offset value in the new direct (D) register.

As a result, a greater number of programs were able to utilize the enhanced direct page addressing mode versus legacy processors that only included the zero page addressing mode.

Scaled index with bounds checking

This is similar to scaled index addressing, except that the instruction has two extra operands (typically constants), and the hardware checks that the index value is between these bounds.

Another variation uses vector descriptors to hold the bounds; this makes it easy to implement dynamically allocated arrays and still have full bounds checking.

Indirect to bit field within word

Some computers had special indirect addressing modes for subfields within words.

The GE/Honeywell 600 series character addressing indirect word specified either 6-bit or 9-bit character fields within its 36-bit word.

The DEC PDP-10, also 36-bit, had special instructions which allowed memory to be treated as a sequence of fixed-size bit fields or bytes of any size from 1 bit to 36 bits. A one-word sequence descriptor in memory, called a "byte pointer", held the current word address within the sequence, a bit position within a word, and the size of each byte.

Instructions existed to load and store bytes via this descriptor, and to increment the descriptor to point at the next byte (bytes were not split across word boundaries). Much DEC software used five 7-bit bytes per word (plain ASCII characters), with one bit per word unused. Implementations of C had to use four 9-bit bytes per word, since the 'malloc' function in C assumes that the size of an *int* is some multiple of the size of a *char*; the actual multiple is determined by the system-dependent compile-time operator sizeof.

Index next instruction

The Elliott 503, the Elliott 803, and the Apollo Guidance Computer only used absolute addressing, and did not have any index registers. Thus, indirect jumps, or jumps through registers, were not supported in the instruction set. Instead, it could be instructed to *add the contents of the current memory word to the next instruction*. Adding a small value to the next instruction to be executed could, for example, change a `JUMP 0` into a `JUMP 20`, thus creating the effect of an indexed jump. Note that the instruction is modified on-the-fly and remains unchanged in memory, i.e. it is not self-modifying code. If the value being added to the next instruction was large enough, it could modify the opcode of that instruction as well as or instead of the address.

Glossary

Indirect

Data referred to through a pointer or address.

Immediate

Data embedded directly in an instruction or command list.

Index

A dynamic offset, typically held in an index register, possibly scaled by an object size.

Offset

An immediate value added to an address; e.g., corresponding to structure field access in the C programming language.

Relative

An address formed relative to another address.

Post increment

The stepping of an address past data used, similar to `*p++` in the C programming language, used for stack pop operations.

Pre decrement

The decrementing of an address prior to use, similar to `*--p` in the C programming language, used for stack push operations.

Source: Wikipedia, https://en.wikipedia.org/wiki/Addressing_mode

MIPS Instruction Format

MIPS assembly is an assembly language, which is a mnemonic or meaningful code for the machine language format in computer programming. Read this section to get a sense of how the addresses to memory are coded in a MIPS microprocessor.

Jump Instruction

The jump instructions load a new value into the PC register, which stores the value of the instruction being executed. This causes the next instruction read from memory to be retrieved from a new location.

Instruction: **j** type: **J Type**

The **j** instruction loads an immediate value into the PC register. This immediate value is either a numeric offset or a label (and the assembler converts the label into an offset).

Instruction: **jr** type: **R Type**

The **jr** instruction loads the PC register with a value stored in a register. As such, the jr instruction can be called as such:

```
jr $t0
```

assuming the target jump location is located in $t0.

Jump and Link

Jump and Link instructions are similar to the jump instructions, except that they store the address of the next instruction (the one immediately after the jump) in the return address ($ra; $31) register. This allows a subroutine to return to the main body routine after completion.

Instruction: **jal** type: **J Type**

Like the **j** instruction, except that the return address is loaded into the $ra register.

Instruction: **jalr** type: **R Type**

The same as the **jr** instruction, except that the return address is loaded into a specified register (or $ra if not specified)

Example

Let's say that we have a subroutine that starts with the label MySub. We can call the subroutine using the following line:

```
jal MySub
...
```

And we can define MySub as follows to return to the main body of the parent routine:

```
jr $ra
```

Branch Instructions

Instead of using rt as a destination operand, rs and rt are both used as source operands and the immediate is sign extended and added to the PC to calculate the address of the instruction to jump to if the branch is taken.

Instruction: **beq** type: **I Type**

Branch if rs and rt are equal. If rs = rt, PC ← PC + 4 + imm.

Instruction: **bne** type: **I Type**

Branch if rs and rt are not equal. If rs ≠ rt, PC ← PC + 4 + imm.

Instruction: **bgez** type: **I Type**

Branch if rs is greater than or equal to zero. If rs ≥ 0, PC ← PC + 4 + imm.

Instruction: **blez** type: **I Type**

Branch if rs is less than or equal to zero. If rs ≤ 0, PC ← PC + 4 + imm.

Instruction: **bgtz** type: **I Type**

Branch if rs is greater than zero. If rs > 0, PC ← PC + 4 + imm.

Instruction: **bltz** type: **I Type**

Branch if rs is less than zero. If rs < 0, PC ← PC + 4 + imm.

Set Instructions

These instructions set rd to 1 if their condition is true. They can be used in combination with **beq** and **bne** and $zero to branch based on the comparison of two registers.

Instruction: **slt** type: **R Type**

If rs < rt, rd ← 1, else 0.

Instruction: **slti** type: **I Type**

If rs < imm, rd ← 1, else 0. The immediate is sign extended.

Instruction: **sltu** type: **R Type**

If rs < rt, rd ← 1, else 0. Treat rs and rt as unsigned integers.

Instruction: **sltiu** type: **I Type**

If rs < imm, rd ← 1, else 0. The immediate is sign extended, but both rs and the extended immediate are treated as *unsigned* integers. The sign extension allows the immediate to represent both very large and very small unsigned integers.

Source:
Wikibooks, https://en.wikibooks.org/wiki/MIPS_Assembly/Control_Flow_Instructions

2.8: Case Study: Intel and ARM Instructions

X86 Instructions and ARM Architecture

Read this article, which gives two examples of instructions set architectures (ISAs). Look over how the different microprocessors address memory. Take note of similarities and differences of format, instructions and type of instructions, and addressing modes between these two as well as between these and the MIPS instructions of the previous sections.

x86 Instructions

Conventions

The following template will be used for instructions that take no operands:

Instr

The following template will be used for instructions that take 1 operand:

Instr arg

The following template will be used for instructions that take 2 operands. Notice how the format of the instruction is different for different assemblers.

Instr src, dest	GAS Syntax
Instr dest, src	Intel Syntax

The following template will be used for instructions that take 3 operands. Notice how the format of the instruction is different for different assemblers.

Instr aux, src, dest	**GAS Syntax**
Instr dest, src, aux	**Intel Syntax**

Suffixes
Some instructions, especially when built for non-Windows platforms (i.e. Unix, Linux, etc.), require the use of suffixes to specify the size of the data which will be the subject of the operation. Some possible suffixes are:

- `b` (byte) = 8 bits.
- `w` (word) = 16 bits.
- `l` (long) = 32 bits.
- `q` (quad) = 64 bits.

An example of the usage with the mov instruction on a 32-bit architecture, GAS syntax:

```
movl $0x000F, %eax   # Store the value F into the eax register
```

On Intel Syntax you don't have to use the suffix. Based on the register name and the used immediate value the compiler knows which data size to use.

```
MOV EAX, 0x000F
```

Source: Wikibooks, https://en.wikibooks.org/wiki/X86_Assembly/X86_Instructions

Unit 3: Fundamentals of Digital Logic Design

We will begin this unit with an overview of digital components, identifying the building blocks of digital logic. We will build on that foundation by writing truth tables and learning about more complicated sequential digital systems with memory. This unit serves as background information for the processor design techniques we learn in later units.

- Upon successful completion of this unit, you will be able to:
  - describe the evolution of physical components used to implement Boolean logic in the design of digital processors and computers;
  - design a simple digital circuit from a truth table or a K map;
  - design a 4 bit adder; and
  - design a simple sequential circuit using state diagrams and state transition tables.

- 3.1: Beginning Design: Logic Gates, Truth Table, and Logic Equations

Logic Design Principles

Study these important design principles, which are applicable to any type of design, particularly computer system design, software, and hardware. Consider these principles alongside the other design considerations we use as a guide to computer system design.

Throughout the text, the description of a design principle presents its name in a boldfaced display, and each place that the principle is used highlights it in underlined italics.

Design principles applicable to many areas of computer systems

- **Adopt sweeping simplifications.** So you can see what you are doing.
- **Avoid excessive generality.** If it is good for everything, it is good for nothing.
- **Avoid rarely used components.** Deterioration and corruption accumulate unnoticed—until the next use.
- **Be explicit.** Get all of the assumptions out on the table.
- **Decouple modules with indirection.** Indirection supports replaceability.
- **Design for iteration.** You won't get it right the first time, so make it easy to change.
- **End-to-end argument.** The application knows best.
- **Escalating complexity principle.** Adding a feature increases complexity out of proportion.
- **Incommensurate scaling rule.** Changing a parameter by a factor of ten requires a new design.
- **Keep digging principle.** Complex systems fail for complex reasons.

- **Law of diminishing returns.** The more one improves some measure of goodness, the more effort the next improvement will require.
- **Open design principle.** Let anyone comment on the design; you need all the help you can get.
- **Principle of least astonishment.** People are part of the system. Choose interfaces that match the user's experience, expectations, and mental models.
- **Robustness principle.** Be tolerant of inputs, strict on outputs.
- **Safety margin principle.** Keep track of the distance to the edge of the cliff or you may fall over the edge.
- **Unyielding foundations rule.** It is easier to change a module than to change the modularity.

Design principles applicable to specific areas of computer systems

- ***Atomicity:* Golden rule of atomicity.** Never modify the only copy!
- ***Coordination:* One-writer principle.** If each variable has only one writer, coordination is simpler.
- ***Durability:* The durability mantra.** Multiple copies, widely separated and independently administered.
- ***Security:* Minimize secrets.** Because they probably won't remain secret for long.
- ***Security:* Complete mediation.** Check every operation for authenticity, integrity, and authorization.
- ***Security:* Fail-safe defaults.** Most users won't change them, so set defaults to do something safe.
- ***Security:* Least privilege principle.** Don't store lunch in the safe with the jewels.
- ***Security:* Economy of mechanism.** The less there is, the more likely you will get it right.
- ***Security:* Minimize common mechanism.** Shared mechanisms provide unwanted communication paths.

Design Hints (useful but not as compelling as design principles)

- **Exploit brute force**
- **Instead of reducing latency, hide it**
- **Optimize for the common case**
- **Separate mechanism from policy**

Source: Jerome H. Saltzer and M. Frans Kaashoek, https://ocw.mit.edu/resources/res-6-004-principles-of-computer-system-design-an-introduction-spring-2009/online-textbook/principles_open_5_0.pdf

Logic Gates

Read the introduction, look at the different logic families under "Electronic Gates" and study carefully the sections "Symbols" and "Truth Tables." Scan the rest of the article.

Truth tables list the total possible input combinations of 1's and 0's and the corresponding outputs for each gate. Logic devices are physical implementations of Boolean logic and are built from components, which have gotten larger and more complex over time, for example: relays and transistors, gates, registers, multiplexors, adders, multipliers, ALUs (arithmetic logic units), data buses, memories, interfaces, and processors. These devices respond to control and data signals specified in machine instructions to perform the functions for which they were designed.

A **logic gate** is an idealized or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Depending on the context, the term may refer to an **ideal logic gate**, one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device (see Ideal and real op-amps for comparison).

Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

Logic circuits include such devices as multiplexers, registers, arithmetic logic units (ALUs), and computer memory, all the way up through complete microprocessors, which may contain more than 100 million gates. In modern practice, most gates are made from MOSFETs (metal–oxide–semiconductor field-effect transistors).

Compound logic gates AND-OR-Invert (AOI) and OR-AND-Invert (OAI) are often employed in circuit design because their construction using MOSFETs is simpler and more efficient than the sum of the individual gates.

In reversible logic, Toffoli gates are used.

Electronic gates

A functionally complete logic system may be composed of relays, valves (vacuum tubes), or transistors. The simplest family of logic gates uses bipolar transistors, and is called resistor–transistor logic (RTL). Unlike simple diode logic gates (which do not have a gain element), RTL gates can be cascaded indefinitely to produce more complex logic functions. RTL gates were used in early integrated circuits. For higher speed and better density, the resistors used in RTL were replaced by diodes resulting in diode–transistor logic (DTL). Transistor–transistor logic (TTL) then supplanted DTL. As integrated circuits became more complex, bipolar transistors were replaced with smaller field-effect transistors (MOSFETs); see PMOS and NMOS. To reduce power consumption still further,

most contemporary chip implementations of digital systems now use CMOS logic. CMOS uses complementary (both n-channel and p-channel) MOSFET devices to achieve a high speed with low power dissipation.

For small-scale logic, designers now use prefabricated logic gates from families of devices such as the TTL 7400 series by Texas Instruments, the CMOS 4000 series by RCA, and their more recent descendants. Increasingly, these fixed-function logic gates are being replaced by programmable logic devices, which allow designers to pack many mixed logic gates into a single integrated circuit. The field-programmable nature of programmable logic devices such as FPGAs has reduced the 'hard' property of hardware; it is now possible to change the logic design of a hardware system by reprogramming some of its components, thus allowing the features or function of a hardware implementation of a logic system to be changed. Other types of logic gates include, but are not limited to:

Logic family	Abbreviation	Description
Diode logic	DL	
Tunnel diode logic	TDL	Exactly the same as diode logic but can perform at a higher speed.
Neon logic	NL	Uses neon bulbs or 3 element neon trigger tubes to perform logic.
Core diode logic	CDL	Performed by semiconductor diodes and small ferrite toroidal cores for moderate speed and moderate power level.
4Layer Device Logic	4LDL	Uses thyristors and SCRs to perform logic operations where high current and or high voltages are required.
Direct-coupled transistor logic	DCTL	Uses transistors switching between saturated and cutoff states to perform logic. The transistors require carefully controlled parameters. Economical because few other components are needed, but tends to be susceptible to noise because of the lower voltage levels employed. Often considered to be the father to modern TTL logic.
Metal-oxide-semiconductor logic	MOS	Uses MOSFETs (metal-oxide-semiconductor field-effect transistors), the basis for most modern logic gates. The MOS logic family includes PMOS logic, NMOS logic, complementary MOS (CMOS), and BiCMOS (bipolar CMOS).
Current-mode logic	CML	Uses transistors to perform logic but biasing is from constant current sources to prevent saturation and allow extremely fast switching. Has high noise immunity despite fairly low logic levels.
Quantum-dot cellular automata	QCA	Uses tunnelable q-bits for synthesizing the binary logic bits. The electrostatic repulsive force in between two electrons in the quantum dots assigns the electron configurations (that defines

high-level logic state 1 or low-level logic state 0) under the suitably driven polarizations. This is a transistorless, currentless, junctionless binary logic synthesis technique allowing it to have very fast operation speeds.

Electronic logic gates differ significantly from their relay-and-switch equivalents. They are much faster, consume much less power, and are much smaller (all by a factor of a million or more in most cases). Also, there is a fundamental structural difference. The switch circuit creates a continuous metallic path for current to flow (in either direction) between its input and its output. The semiconductor logic gate, on the other hand, acts as a high-gain voltage amplifier, which sinks a tiny current at its input and produces a low-impedance voltage at its output. It is not possible for current to flow between the output and the input of a semiconductor logic gate.

Another important advantage of standardized integrated circuit logic families, such as the 7400 and 4000 families, is that they can be cascaded. This means that the output of one gate can be wired to the inputs of one or several other gates, and so on. Systems with varying degrees of complexity can be built without great concern of the designer for the internal workings of the gates, provided the limitations of each integrated circuit are considered.

The output of one gate can only drive a finite number of inputs to other gates, a number called the 'fan-out limit'. Also, there is always a delay, called the 'propagation delay', from a change in input of a gate to the corresponding change in its output. When gates are cascaded, the total propagation delay is approximately the sum of the individual delays, an effect which can become a problem in high-speed circuits. Additional delay can be caused when many inputs are connected to an output, due to the distributed capacitance of all the inputs and wiring and the finite amount of current that each output can provide.

History and development

The binary number system was refined by Gottfried Wilhelm Leibniz (published in 1705), influenced by the ancient *I Ching*'s binary system. Leibniz established that using the binary system combined the principles of arithmetic and logic.

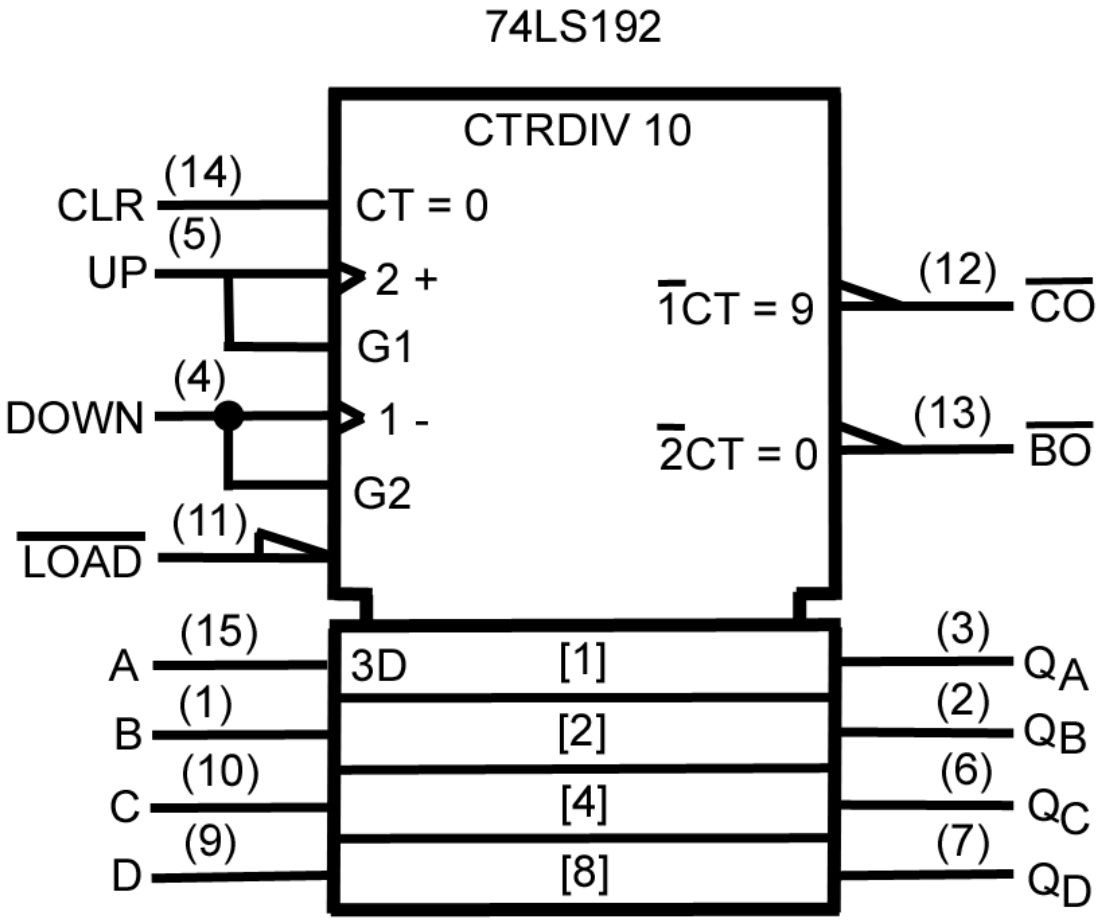
In an 1886 letter, Charles Sanders Peirce described how logical operations could be carried out by electrical switching circuits. Eventually, vacuum tubes replaced relays for logic operations. Lee De Forest's modification, in 1907, of the Fleming valve can be used as a logic gate. Ludwig Wittgenstein introduced a version of the 16-row truth table as proposition 5.101 of *Tractatus Logico-Philosophicus* (1921). Walther Bothe, inventor of the coincidence circuit, got part of the 1954 Nobel Prize in physics, for the first modern electronic AND gate in 1924. Konrad Zuse designed and built electromechanical logic gates for his computer Z1 (from 1935–38).

From 1934 to 1936, NEC engineer Akira Nakashima introduced switching circuit theory in a series of papers showing that two-valued Boolean algebra, which he discovered independently, can describe the operation of switching circuits. His work was later cited by Claude E. Shannon, who elaborated on the use of Boolean algebra in the analysis and design of switching circuits in 1937. Using this property of electrical switches to implement logic is the fundamental concept that underlies all electronic digital computers. Switching circuit theory became the foundation of digital circuit design, as it became widely known in the electrical engineering community during and after World War II, with theoretical rigor superseding the *ad hoc* methods that had prevailed previously.

Metal-oxide-semiconductor (MOS) logic originates from the MOSFET (metal-oxide-semiconductor field-effect transistor), invented by Mohamed M. Atalla and Dawon Kahng at Bell Labs in 1959. They first demonstrated both PMOS logic and NMOS logic in 1960. Both types were later combined and adapted into complementary MOS (CMOS) logic by Chih-Tang Sah and Frank Wanlass at Fairchild Semiconductor in 1963.

Active research is taking place in molecular logic gates.

Symbols



A synchronous 4-bit up/down decade counter symbol (74LS192) in accordance with ANSI/IEEE Std. 91-1984 and IEC Publication 60617-12.

There are two sets of symbols for elementary logic gates in common use, both defined in ANSI/IEEE Std 91-1984 and its supplement ANSI/IEEE Std 91a-1991. The "distinctive shape" set, based on traditional schematics, is used for simple drawings and derives from United States Military Standard MIL-STD-806 of the 1950s and 1960s. It is sometimes unofficially described as "military", reflecting its origin. The "rectangular shape" set, based on ANSI Y32.14 and other early industry standards as later refined by IEEE and IEC, has rectangular outlines for all types of gate and allows representation of a much wider range of devices than is possible with the traditional symbols. The IEC standard, IEC 60617-12, has been adopted by other standards, such as EN 60617-12:1999 in Europe, BS EN 60617-12:1999 in the United Kingdom, and DIN EN 60617-12:1998 in Germany.

The mutual goal of IEEE Std 91-1984 and IEC 60617-12 was to provide a uniform method of describing the complex logic functions of digital circuits with schematic symbols. These functions were more complex than simple AND and OR gates. They could be

medium scale circuits such as a 4-bit counter to a large scale circuit such as a microprocessor.

IEC 617-12 and its successor IEC 60617-12 do not explicitly show the "distinctive shape" symbols, but do not prohibit them. These are, however, shown in ANSI/IEEE 91 (and 91a) with this note: "The distinctive-shape symbol is, according to IEC Publication 617, Part 12, not preferred, but is not considered to be in contradiction to that standard". IEC 60617-12 correspondingly contains the note (Section 2.1) "Although non-preferred, the use of other symbols recognized by official national standards, that is distinctive shapes in place of symbols, shall not be considered to be in contradiction with this standard. Usage of these other symbols in combination to form complex symbols (for example, use as embedded symbols) is discouraged". This compromise was reached between the respective IEEE and IEC working groups to permit the IEEE and IEC standards to be in mutual compliance with one another.

A third style of symbols, DIN 40700 (1976), was in use in Europe and is still widely used in European academia.

In the 1980s, schematics were the predominant method to design both circuit boards and custom ICs known as gate arrays. Today custom ICs and the field-programmable gate array are typically designed with Hardware Description Languages (HDL) such as Verilog or VHDL.

Type	Distinctive shape (IEEE Std 91/91a-1991)	Rectangular shape (IEEE Std 91/91a-1991) (IEC 60617-12:1997)	Boolean algebra between A & B	Truth table
1-Input gates				
Buffer			A	A
			0	0
			1	1
NOT (inverter)			A	A
			0	1
			1	0

In electronics a NOT gate is more commonly called an inverter. The circle on the symbol is called a *bubble* and is used in logic diagrams to indicate a logic negation between the external logic state and the internal logic state (1 to 0 or vice versa). On a circuit diagram it must be accompanied by a statement asserting that the *positive logic convention* or *negative logic convention* is being used (high voltage level = 1 or low voltage level = 1, respectively). The *wedge* is used in circuit diagrams to directly indicate an active-low (low voltage level = 1) input or output without requiring a uniform convention throughout the circuit diagram. This is called *Direct Polarity Indication*. See IEEE Std 91/91A and IEC 60617-12. Both the *bubble* and the *wedge* can be used on distinctive-shape and rectangular-shape symbols on circuit diagrams, depending on the logic convention used. On pure logic diagrams, only the *bubble* is meaningful.

Conjunction and Disjunction

AND $A \cdot B$ or $A \wedge B$

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR $A + B$ or $A \vee B$

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

Alternative denial and Joint denial

NAND $A \cdot B\overline{}$ or $A \uparrow B$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

NOR $A+B\overline{\quad\quad\quad}A+B^{-}$ or $A\downarrow BA\downarrow B$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive or and Biconditional

XOR $A\oplus BA\oplus B$ or $A\veebar BA\veebar B$

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

The output of a two input exclusive-OR is true only when the two input values are *different*, and false if they are equal, regardless of the value. If there are more than two inputs, the output of the distinctive-shape symbol is undefined. The output of the rectangular-shaped symbol is true if the number of true inputs is exactly one or exactly the number following the "=" in the qualifying symbol.

XNOR $A\oplus B\overline{\quad\quad\quad}A\oplus B^{-}$ or $A\odot BA\odot B$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

Truth tables

Output comparison of 1-input logic gates.

INPUT OUTPUT

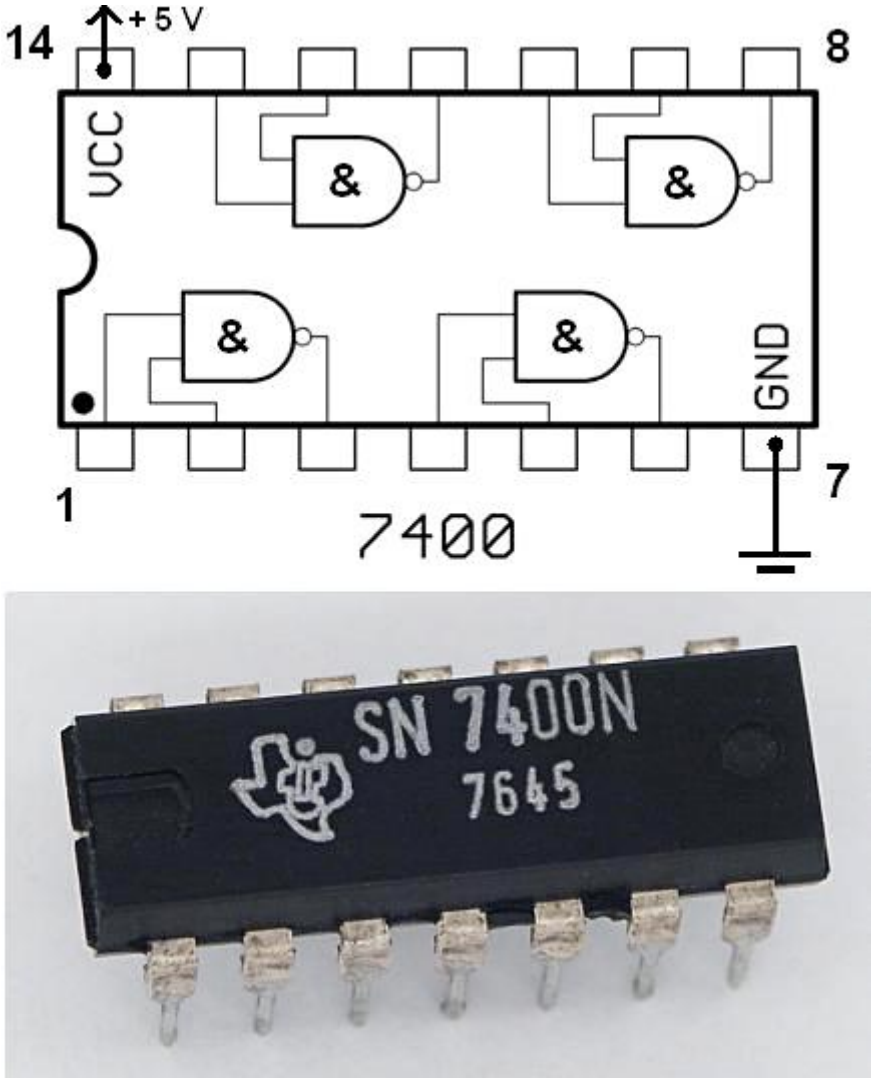
A	Buffer	Inverter
0	0	1
1	1	0

Output comparison of 2-input logic gates.

INPUT		**OUTPUT**					
A	B	AND	NAND	OR	NOR	XOR	XNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Universal logic gates

Further information on the theoretical basis: Functional completeness



The 7400 chip, containing four NANDs. The two additional pins supply power (+5 V) and connect the ground.

Charles Sanders Peirce (during 1880–81) showed that NOR gates alone (or alternatively NAND gates alone) can be used to reproduce the functions of all the other logic gates, but his work on it was unpublished until 1933. The first published proof was by Henry M. Sheffer in 1913, so the NAND logical operation is sometimes called Sheffer stroke; the logical NOR is sometimes called *Peirce's arrow*. Consequently, these gates are sometimes called *universal logic gates*.

De Morgan equivalent symbols

By use of De Morgan's laws, an *AND* function is identical to an *OR* function with negated inputs and outputs. Likewise, an *OR* function is identical to an *AND* function with negated inputs and outputs. A NAND gate is equivalent to an OR gate with negated inputs, and a NOR gate is equivalent to an AND gate with negated inputs.

This leads to an alternative set of symbols for basic gates that use the opposite core symbol (*AND* or *OR*) but with the inputs and outputs negated. Use of these alternative symbols can make logic circuit diagrams much clearer and help to show accidental connection of an active high output to an active low input or vice versa. Any connection that has logic negations at both ends can be replaced by a negationless connection and a suitable change of gate or vice versa. Any connection that has a negation at one end and no negation at the other can be made easier to interpret by instead using the De Morgan equivalent symbol at either of the two ends. When negation or polarity indicators on both ends of a connection match, there is no logic negation in that path (effectively, bubbles "cancel"), making it easier to follow logic states from one symbol to the next. This is commonly seen in real logic diagrams – thus the reader must not get into the habit of associating the shapes exclusively as OR or AND shapes, but also take into account the bubbles at both inputs and outputs in order to determine the "true" logic function indicated.

A De Morgan symbol can show more clearly a gate's primary logical purpose and the polarity of its nodes that are considered in the "signaled" (active, on) state. Consider the simplified case where a two-input NAND gate is used to drive a motor when either of its inputs are brought low by a switch. The "signaled" state (motor on) occurs when either one OR the other switch is on. Unlike a regular NAND symbol, which suggests AND logic, the De Morgan version, a two negative-input OR gate, correctly shows that OR is of interest. The regular NAND symbol has a bubble at the output and none at the inputs (the opposite of the states that will turn the motor on), but the De Morgan symbol shows both inputs and output in the polarity that will drive the motor.

De Morgan's theorem is most commonly used to implement logic gates as combinations of only NAND gates, or as combinations of only NOR gates, for economic reasons.

Data storage

Logic gates can also be used to store data. A storage element can be constructed by connecting several gates in a "latch " circuit. More complicated designs that use clock signals and that change only on a rising or falling edge of the clock are called edge-triggered " flip-flops". Formally, a flip-flop is called a bistable circuit, because it has two stable states which it can maintain indefinitely. The combination of multiple flip-flops in parallel, to store a multiple-bit value, is known as a register. When using any of these gate setups the overall system has memory; it is then called a sequential logic system since its output can be influenced by its previous state(s), i.e. by the *sequence* of input states. In contrast, the output from combinational logic is purely a combination of its present inputs, unaffected by the previous input and output states.

These logic circuits are known as computer memory. They vary in performance, based on factors of speed, complexity, and reliability of storage, and many different types of designs are used based on the application.

Three-state logic gates

A tristate buffer can be thought of as a switch. If B is on, the switch is closed. If B is off, the switch is open.

A three-state logic gate is a type of logic gate that can have three different outputs: high (H), low (L) and high-impedance (Z). The high-impedance state plays no role in the logic, which is strictly binary. These devices are used on buses of the CPU to allow multiple chips to send data. A group of three-states driving a line with a suitable control circuit is basically equivalent to a multiplexer , which may be physically distributed over separate devices or plug-in cards.

In electronics, a high output would mean the output is sourcing current from the positive power terminal (positive voltage). A low output would mean the output is sinking current to the negative power terminal (zero voltage). High impedance would mean that the output is effectively disconnected from the circuit.

Implementations

Since the 1990s, most logic gates are made in CMOS (complementary metal oxide semiconductor) technology that uses both NMOS and PMOS transistors. Often millions of logic gates are packaged in a single integrated circuit .

There are several logic families with different characteristics (power consumption, speed, cost, size) such as: RDL (resistor–diode logic), RTL (resistor-transistor logic), DTL (diode–transistor logic), TTL (transistor–transistor logic) and CMOS. There are also sub-variants, e.g. standard CMOS logic vs. advanced types using still CMOS technology, but with some optimizations for avoiding loss of speed due to slower PMOS transistors.

Non-electronic implementations are varied, though few of them are used in practical applications. Many early electromechanical digital computers, such as the Harvard Mark I, were built from relay logic gates, using electro-mechanical relays. Logic gates can be made using pneumatic devices, such as the Sorteberg relay or mechanical logic gates, including on a molecular scale. Logic gates have been made out of DNA (see DNA nanotechnology) and used to create a computer called MAYA (see MAYA-II). Logic gates can be made from quantum mechanical effects (though quantum computing usually diverges from boolean design; see quantum logic gate). Photonic logic gates use nonlinear optical effects.

In principle any method that leads to a gate that is functionally complete (for example, either a NOR or a NAND gate) can be used to make any kind of digital logic circuit. Note that the use of 3-state logic for bus systems is not needed, and can be replaced by digital multiplexers, which can be built using only simple logic gates (such as NAND gates, NOR gates, or AND and OR gates).

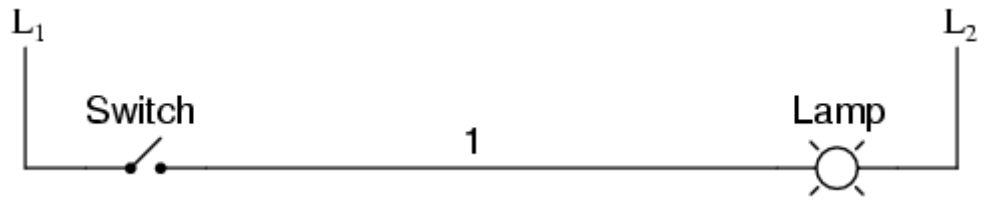
Source: Wikipedia, https://en.wikipedia.org/wiki/Logic_gate

Ladder Logic

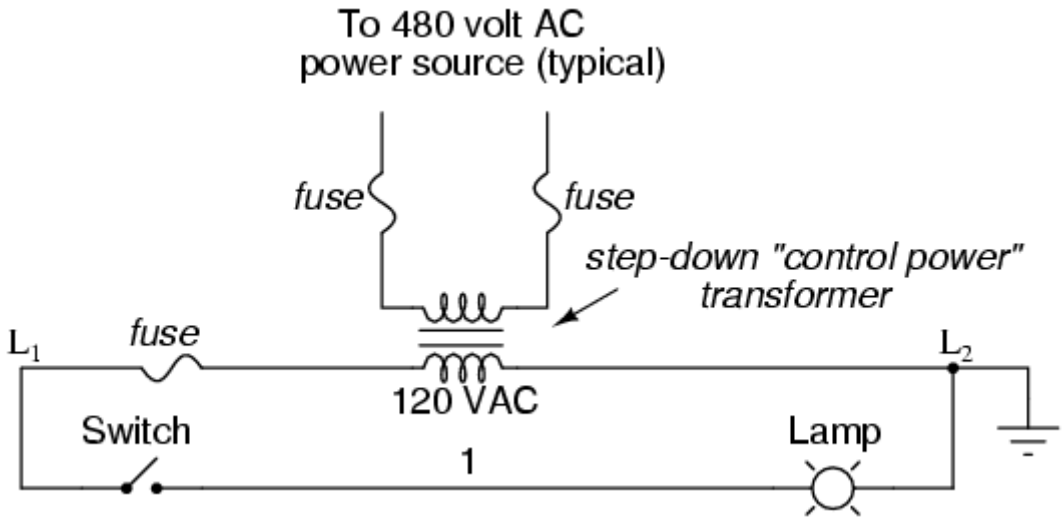
This section on ladder logic is optional. If you would like to know more, read these sections.

Ladder Diagrams

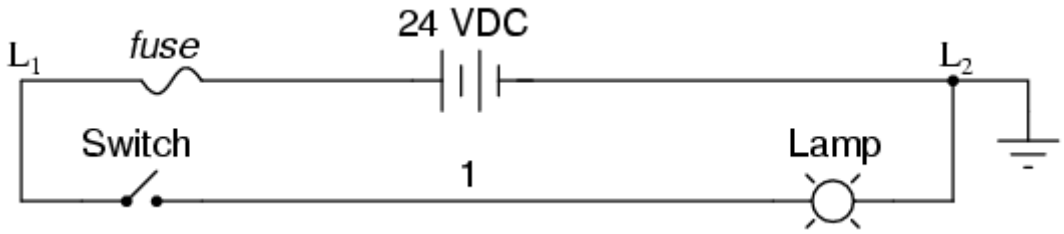
Ladder diagrams are specialized schematics commonly used to document industrial control logic systems. They are called "ladder" diagrams because they resemble a ladder, with two vertical rails (supply power) and as many "rungs" (horizontal lines) as there are control circuits to represent. If we wanted to draw a simple ladder diagram showing a lamp that is controlled by a hand switch, it would look like



The "L_1" and "L_2" designations refer to the two poles of a 120 VAC supply unless otherwise noted. L_1 is the "hot" conductor, and L_2 is the grounded ("neutral") conductor. These designations have nothing to do with inductors, just to make things confusing. The actual transformer or generator supplying power to this circuit is omitted for simplicity. In reality, the circuit looks something like this:

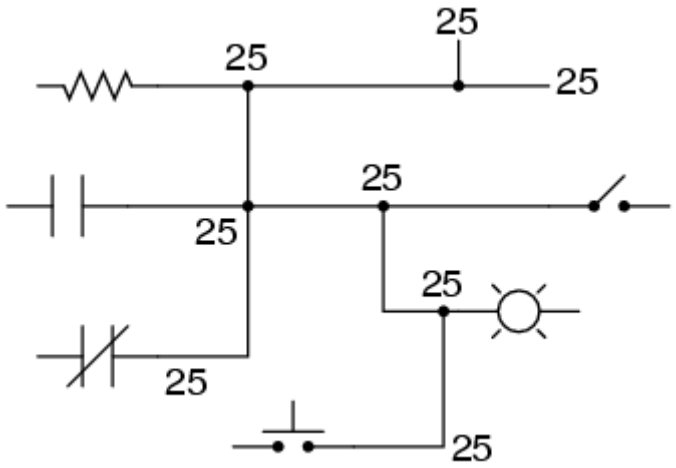


Typically in industrial relay logic circuits, but not always, the operating voltage for the switch contacts and relay coils will be 120 volts AC. Lower voltage AC and even DC systems are sometimes built and documented according to "ladder" diagrams:



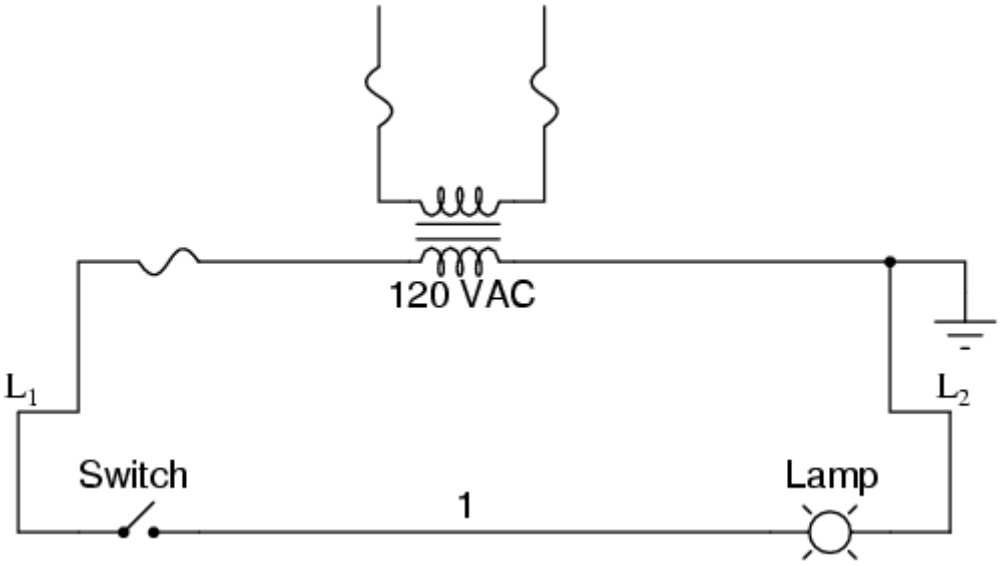
So long as the switch contacts and relay coils are all adequately rated, it really doesn't matter what level of voltage is chosen for the system to operate with.

Note the number "1" on the wire between the switch and the lamp. In the real world, that wire would be labeled with that number, using heat-shrink or adhesive tags, wherever it was convenient to identify. Wires leading to the switch would be labeled "L_1" and "1", respectively. Wires leading to the lamp would be labeled "1" and "L_2", respectively. These wire numbers make assembly and maintenance very easy. Each conductor has its own unique wire number for the control system that its used in. Wire numbers do not change at any junction or node, even if wire size, color, or length changes going into or out of a connection point. Of course, it is preferable to maintain consistent wire colors, but this is not always practical. What matters is that any one, electrically continuous point in a control circuit possesses the same wire number. Take this circuit section, for example, with wire #25 as a single, electrically continuous point threading to many different devices:

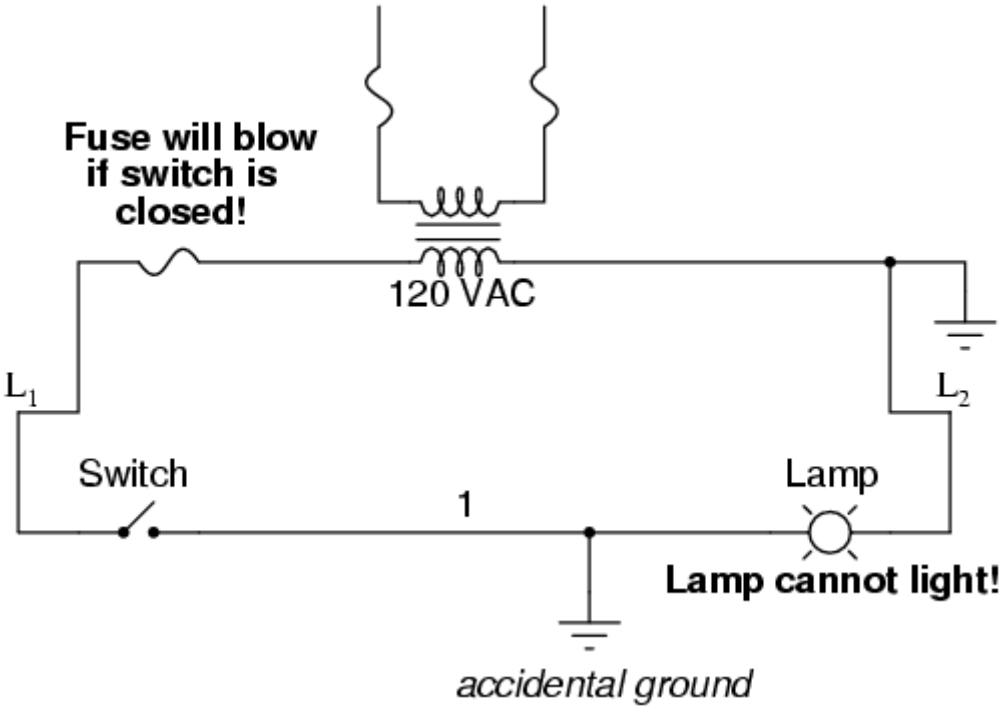


In ladder diagrams, the load device (lamp, relay coil, solenoid coil, etc.) is almost always drawn at the right-hand side of the rung. While it doesn't matter electrically where the relay coil is located within the rung, it *does* matter which end of the ladder's power supply is grounded, for reliable operation.

Take for instance this circuit:

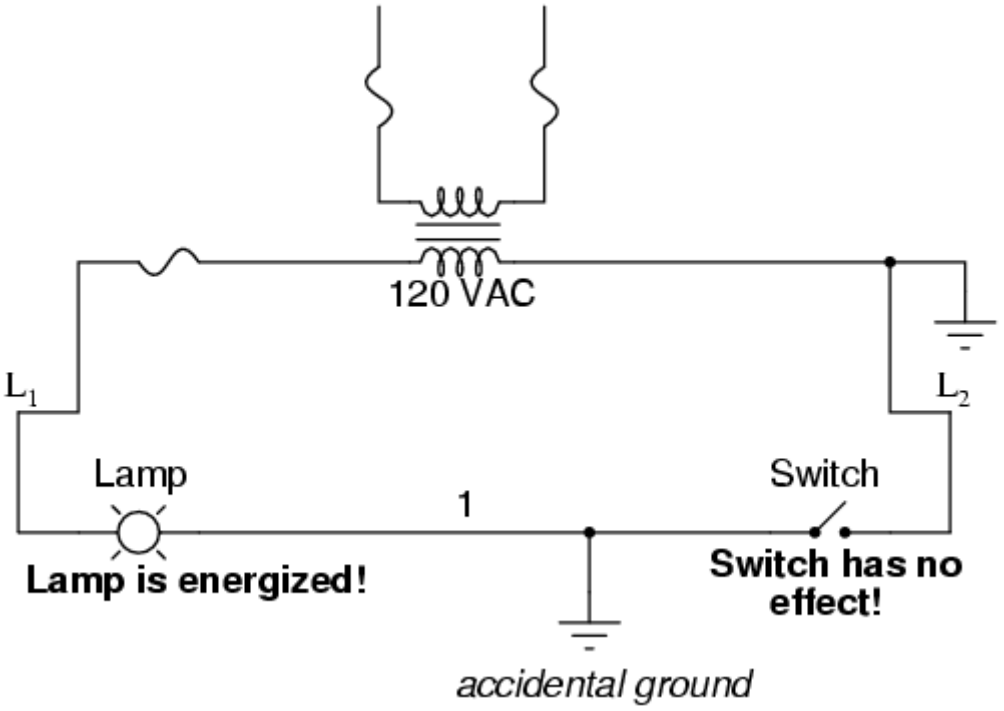


Here, the lamp (load) is located on the right-hand side of the rung, and so is the ground connection for the power source. This is no accident or coincidence; rather, it is a purposeful element of good design practice. Suppose that wire #1 were to accidentally come in contact with ground, the insulation of that wire having been rubbed off so that the bare conductor came in contact with grounded, metal conduit. Our circuit would now function like this:



With both sides of the lamp connected to ground, the lamp will be "shorted out" and unable to receive power to light up. If the switch were to close, there would be a short-circuit, immediately blowing the fuse.

However, consider what would happen to the circuit with the same fault (wire #1 coming in contact with ground), except this time we'll swap the positions of switch and fuse (L_2 is still grounded):



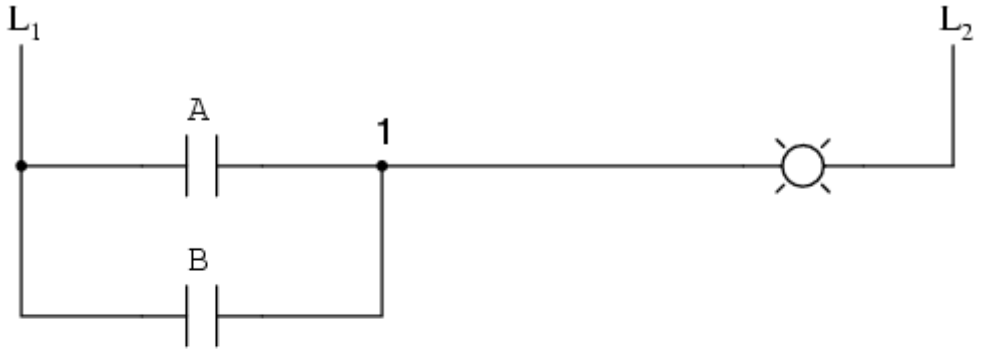
This time the accidental grounding of wire #1 will force power to the lamp while the switch will have no effect. It is much safer to have a system that blows a fuse in the event of a ground fault than to have a system that uncontrollably energizes lamps, relays, or solenoids in the event of the same fault. For this reason, the load(s) must always be located nearest the grounded power conductor in the ladder diagram.

REVIEW:

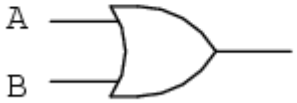
- Ladder diagrams (sometimes called "ladder logic") are a type of electrical notation and symbology frequently used to illustrate how electromechanical switches and relays are interconnected.
- The two vertical lines are called "rails" and attach to opposite poles of a power supply, usually 120 volts AC. L_1 designates the "hot" AC wire and L_2 the "neutral" (grounded) conductor.
- Horizontal lines in a ladder diagram are called "rungs", each one representing a unique parallel circuit branch between the poles of the power supply.
- Typically, wires in control systems are marked with numbers and/or letters for identification. The rule is, all permanently connected (electrically common) points must bear the same label.

Digital Logic Functions

We can construct simple logic functions for our hypothetical lamp circuit, using multiple contacts, and document these circuits quite easily and understandably with additional rungs to our original "ladder". If we use standard binary notation for the status of the switches and lamp (0 for unactuated or de-energized; 1 for actuated or energized), a truth table can be made to show how the logic works:

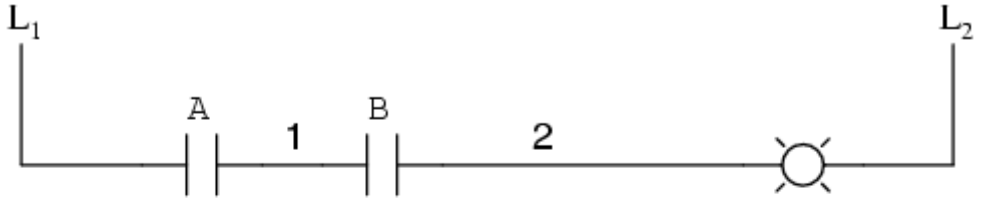


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

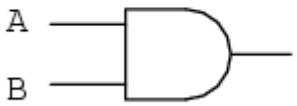


Now, the lamp will come on if either contact A or contact B is actuated, because all it takes for the lamp to be energized is to have at least one path for current from wire L_1 to wire 1. What we have is a simple OR logic function, implemented with nothing more than contacts and a lamp.

We can mimic the AND logic function by wiring the two contacts in series instead of parallel:

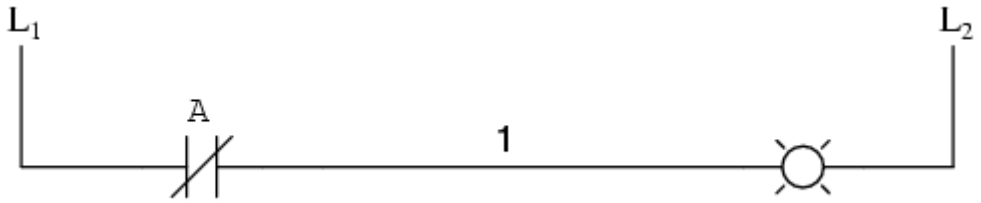


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

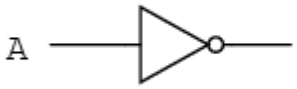


Now, the lamp energizes only if contact A *and* contact B are simultaneously actuated. A path exists for current from wire L_1 to the lamp (wire 2) if and only if *both* switch contacts are closed.

The logical inversion, or NOT, function can be performed on a contact input simply by using a normally-closed contact instead of a normally-open contact:

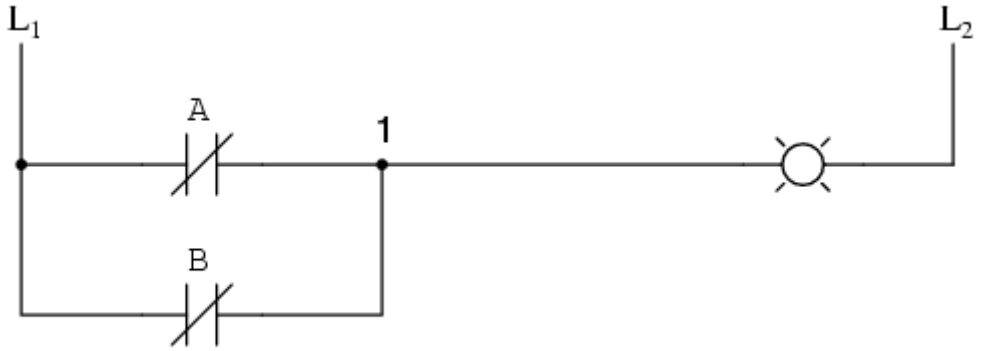


A	Output
0	1
1	0

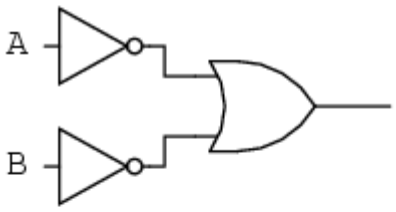


Now, the lamp energizes if the contact is *not* actuated, and de-energizes when the contact *is* actuated.

If we take our OR function and invert each "input" through the use of normally-closed contacts, we will end up with a NAND function. In a special branch of mathematics known as *Boolean algebra*, this effect of gate function identity changing with the inversion of input signals is described by *DeMorgan's Theorem*, a subject to be explored in more detail in a later chapter.



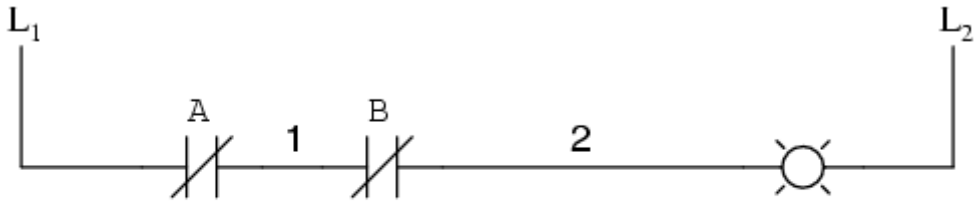
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0



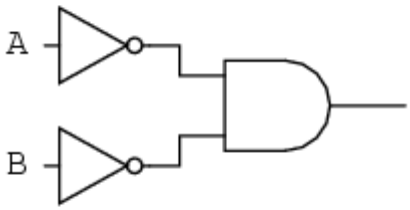
or

The lamp will be energized if *either* contact is unactuated. It will go out only if *both* contacts are actuated simultaneously.

Likewise, if we take our AND function and invert each "input" through the use of normally-closed contacts, we will end up with a NOR function:



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

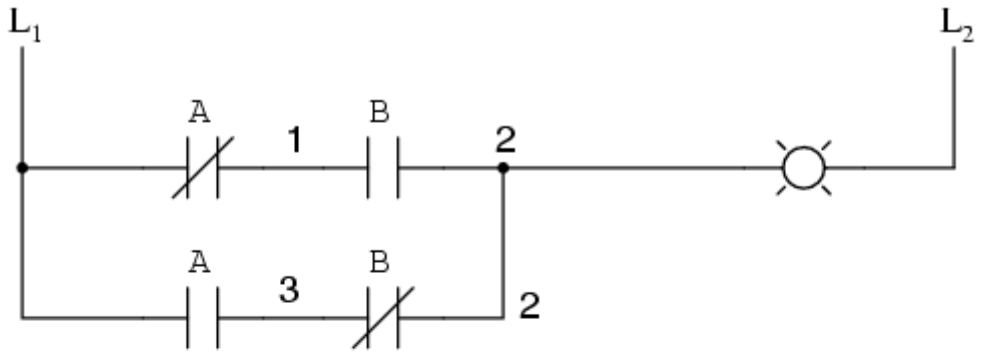


or

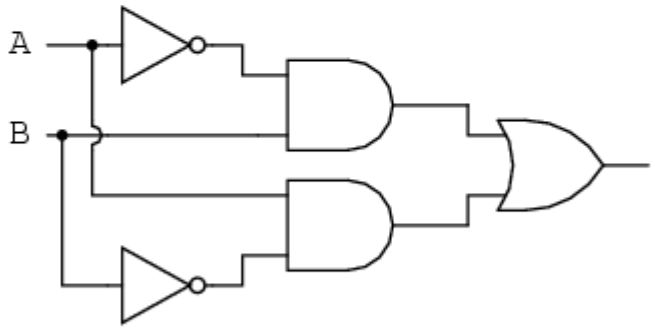
A pattern quickly reveals itself when ladder circuits are compared with their logic gate counterparts:

- Parallel contacts are equivalent to an OR gate.
- Series contacts are equivalent to an AND gate.
- Normally-closed contacts are equivalent to a NOT gate (inverter).

We can build combinational logic functions by grouping contacts in series-parallel arrangements, as well. In the following example, we have an Exclusive-OR function built from a combination of AND, OR, and inverter (NOT) gates:



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0



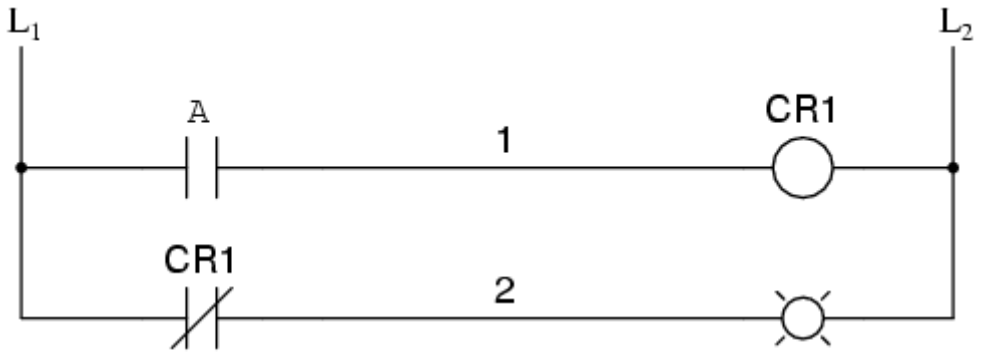
or

The top rung (NC contact A in series with NO contact B) is the equivalent of the top NOT/AND gate combination. The bottom rung (NO contact A in series with NC contact B) is the equivalent of the bottom NOT/AND gate combination. The parallel connection between the two rungs at wire number 2 forms the equivalent of the OR gate, in allowing either rung 1 *or* rung 2 to energize the lamp.

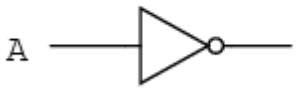
To make the Exclusive-OR function, we had to use two contacts per input: one for direct input and the other for "inverted" input. The two "A" contacts are physically actuated by the same mechanism, as are the two "B" contacts. The common association between contacts is denoted by the label of the contact. There is no limit to how many contacts per switch can be represented in a ladder diagram, as each new contact on any switch or relay (either normally-open or normally-closed) used in the diagram is simply marked with the same label.

Sometimes, multiple contacts on a single switch (or relay) are designated by a compound labels, such as "A-1" and "A-2" instead of two "A" labels. This may be especially useful if you want to specifically designate which set of contacts on each switch or relay is being used for which part of a circuit. For simplicity's sake, I'll refrain from such elaborate labeling in this lesson. If you see a common label for multiple contacts, you know those contacts are all actuated by the same mechanism.

If we wish to invert the *output* of any switch-generated logic function, we must use a relay with a normally-closed contact. For instance, if we want to energize a load based on the inverse, or NOT, of a normally-open contact, we could do this:

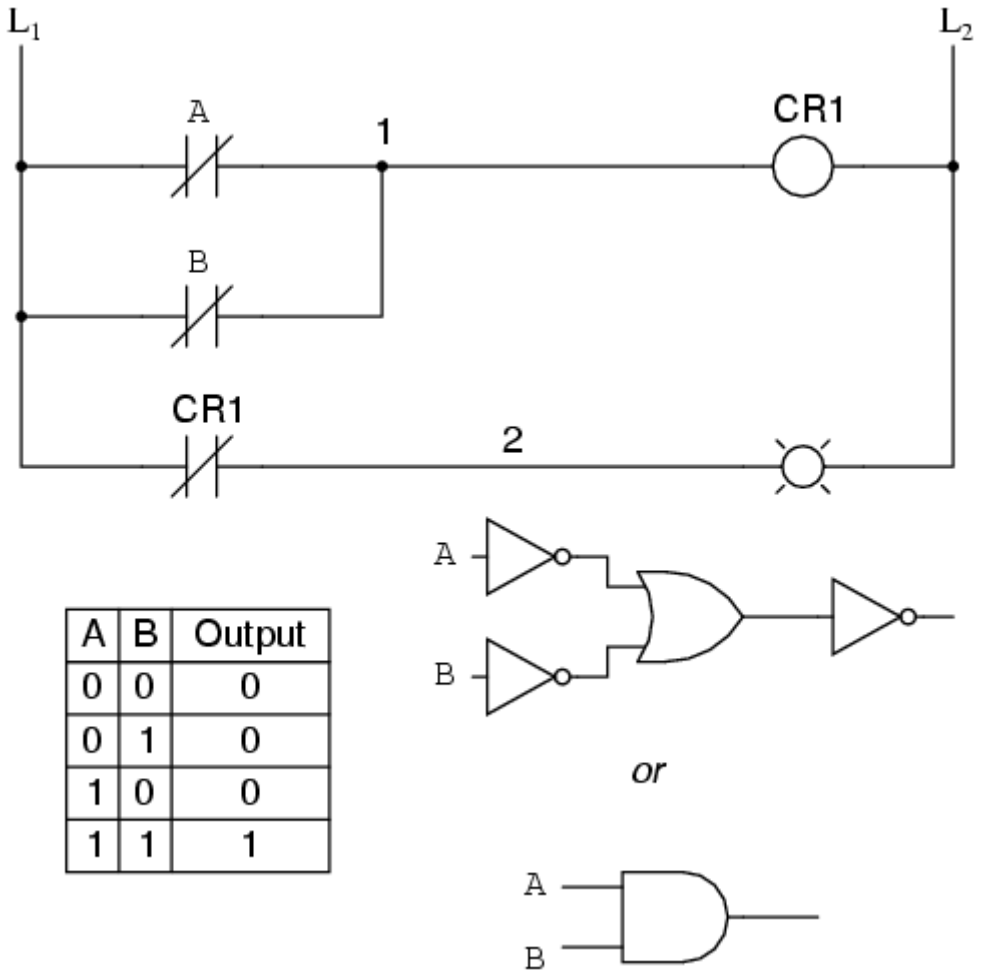


A	CR1	Output
0	0	1
1	1	0



We will call the relay, "control relay 1", or CR_1. When the coil of CR_1 (symbolized with the pair of parentheses on the first rung) is energized, the contact on the second rung *opens*, thus de-energizing the lamp. From switch A to the coil of CR_1, the logic function is noninverted. The normally-closed contact actuated by relay coil CR_1 provides a logical inverter function to drive the lamp opposite that of the switch's actuation status.

Applying this inversion strategy to one of our inverted-input functions created earlier, such as the OR-to-NAND, we can invert the output with a relay to create a noninverted function:



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

From the switches to the coil of CR_1, the logical function is that of a NAND gate. CR_1's normally-closed contact provides one final inversion to turn the NAND function into an AND function.

REVIEW:

- Parallel contacts are logically equivalent to an OR gate.
- Series contacts are logically equivalent to an AND gate.
- Normally closed (N.C.) contacts are logically equivalent to a NOT gate.
- A relay must be used to invert the *output* of a logic gate function, while simple normally-closed switch contacts are sufficient to represent inverted gate *inputs*.

Source: Tony R. Kuphaldt and Workforce LibreTexts, https://workforce.libretexts.org/Bookshelves/Electronics_Technology/Book%3A_Electric_Circuits_IV_-_Digital_Circuitry_(Kuphaldt)/06%3A_Ladder_Logic

Karnaugh Mapping

Read this chapter on Karnaugh mapping, which is a tabular way for simplifying Boolean logic. There are several ways for representing Boolean logic: algebraic expressions, which use symbols and Boolean operations; Venn diagrams, which use distinct and overlapping circles; and tables relating inputs to outputs (for combinational logic) or tables relating inputs and current state to outputs and next state (for sequential logic). When designing sequential logic, some of the components are memory devices. Cost and processing time are considerations in using memory devices, which can be expensive. To reduce the cost or processing time the logic can be simplified. This simplification can be done using algebraic rules to manipulate the symbols and operations, analysis of the areas inside the circles for Venn diagrams, or Karnaugh maps for input/output tables.

Introduction to Karnaugh Mapping

Why learn about *Karnaugh* maps? The Karnaugh map, like Boolean algebra, is a simplification tool applicable to digital logic. See the "Toxic waste incinerator" in the Boolean algebra chapter for an example of Boolean simplification of digital logic. The Karnaugh Map will simplify logic faster and more easily in most cases.

Boolean simplification is actually faster than the Karnaugh map for a task involving two or fewer Boolean variables. It is still quite usable at three variables, but a bit slower. At four input variables, Boolean algebra becomes tedious. Karnaugh maps are both faster and easier. Karnaugh maps work well for up to six input variables, are usable for up to eight variables. For more than six to eight variables, simplification should be by *CAD* (computer automated design).

Recommended logic simplification vs number of inputs			
Variables	Boolean algebra	Karnaugh map	computer automated
1-2	X		?
3	X	X	?
4	?	X	?
5-6		X	X
7-8		?	X
>8			X

In theory any of the three methods will work. However, as a practical matter, the above guidelines work well. We would not normally resort to computer automation to simplify a three input logic block. We could sooner solve the problem with pencil and paper. However, if we had seven of these problems to solve, say for a *BCD* (Binary Coded Decimal) to *seven segment decoder*, we might want to automate the process. A BCD to seven segment decoder generates the logic signals to drive a seven segment LED (light emitting diode) display.

Examples of computer automated design languages for simplification of logic are PALASM, ABEL, CUPL, Verilog, and VHDL. These programs accept a *hardware descriptor*

language input file which is based on Boolean equations and produce an output file describing a *reduced* (or simplified) Boolean solution. We will not require such tools in this chapter. Let's move on to Venn diagrams as an introduction to Karnaugh maps.

Source: Tony R.
Kuphaldt, https://workforce.libretexts.org/Bookshelves/Electronics_Technology/Book%3A_Electric_Circuits_IV_-_Digital_Circuitry_(Kuphaldt)/08%3A_Karnaugh_Mapping

3.2: Combinational Logic

Combinational Logic Functions

Read this chapter, which describes the design of several components using logic gates, including adders, encoders and decoders, multiplexers, and demultiplexers. This chapter also mentions ladder logic. If you are not familiar with ladder logic, you may also optionally read Chapter 6.1 and 6.2 as a reference. Note that ladder logic is not generally used in computer design and may be omitted. For the one input decoder, note that for a 0 input, the D0 output is a 1 and when the input is a 1, the D1 output is a 1. All the other decoders work the same way. One output line is a 1and the rest are 0's, indicating which binary number has been placed on the input lines. So an input in binary of the number 6 would cause D6 to be a 1.

Introduction to Combinational Logic Functions

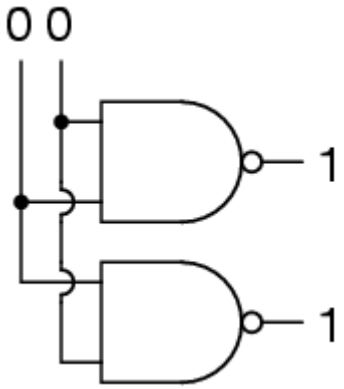
The term "combinational" comes to us from mathematics. In mathematics a combination is an unordered set, which is a formal way to say that nobody cares which order the items came in. Most games work this way, if you rolled dice one at a time and get a 2 followed by a 3 it is the same as if you had rolled a 3 followed by a 2. With combinational logic, the circuit produces the same output regardless of the order the inputs are changed.

There are circuits which depend on the when the inputs change, these circuits are called sequential logic. Even though you will not find the term "sequential logic" in the chapter titles, the next several chapters will discuss sequential logic.

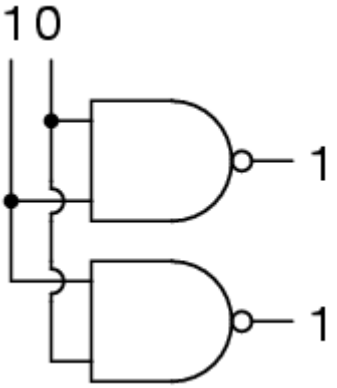
Practical circuits will have a mix of combinational and sequential logic, with sequential logic making sure everything happens in order and combinational logic performing functions like arithmetic, logic, or conversion.

You have already used combinational circuits. Each logic gate discussed previously is a combinational logic function. Let's follow how two NAND gate works if we provide them inputs in different orders.

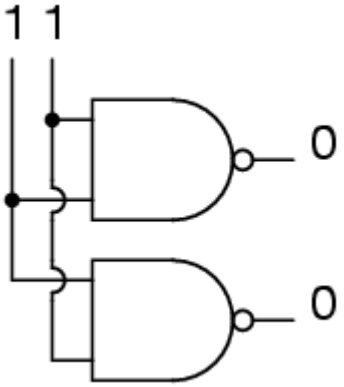
We begin with both inputs being 0.



We then set one input high.



We then set the other input high.



So NAND gates do not care about the order of the inputs, and you will find the same true of all the other gates covered up to this point (AND, XOR, OR, NOR, XNOR, and NOT).

Source: Tony R.
Kuphaldt, https://workforce.libretexts.org/Bookshelves/Electronics_Technology/Book%3A_Electric_Circuits_IV_-_Digital_Circuitry_(Kuphaldt)/09%3A_Combinational_Logic_Functions

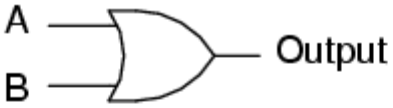
3.3: Flip-Flops, Latches, and Registers

Multivibrators

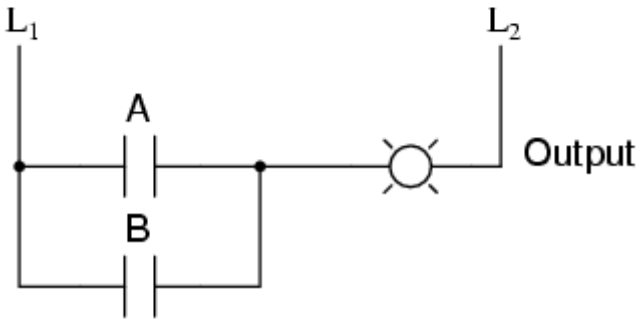
Read this chapter, which discusses how logic gates are connected to store bits (0s and 1s). Combinational circuits, described in the previous section, do not have memory. Using logic gates, latches and flip flops are designed for storing bits. Groups of flip flops are used to build registers which hold strings of bits. For each storage device in Chapter 10, focus on the overview at the beginning of the section and the review of the device's characteristics at the end of its section. While you do not absolutely need to know the details of how latches and flip flops work, you might find the material of interest. We strongly recommend that you read the details of the design of each storage device, because it will give you a stronger background.

Digital Logic With Feedback

With simple gate and combinational logic circuits, there is a definite output state for any given input state. Take the truth table of an OR gate, for instance:

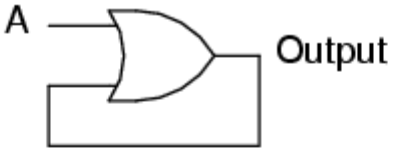


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

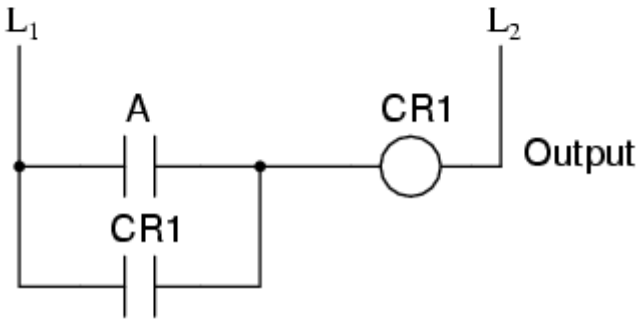


For each of the four possible combinations of input states (0-0, 0-1, 1-0, and 1-1), there is one, definite, unambiguous output state. Whether we're dealing with a multitude of cascaded gates or a single gate, that output state is determined by the truth table(s) for the gate(s) in the circuit, and nothing else.

However, if we alter this gate circuit so as to give signal feedback from the output to one of the inputs, strange things begin to happen:



A	Output
0	?
1	1

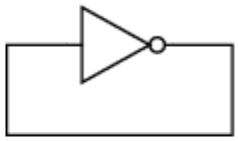


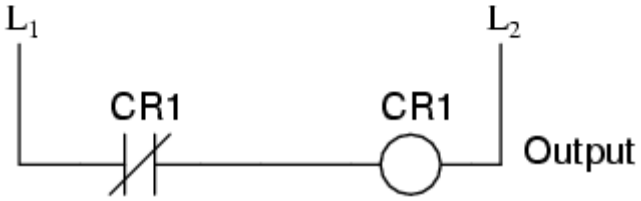
We know that if A is 1, the output *must* be 1, as well. Such is the nature of an OR gate: any "high" (1) input forces the output "high" (1). If A is "low" (0), however, we cannot guarantee the logic level or state of the output in our truth table. Since the output feeds back to one of the OR gate's inputs, and we know that any 1 input to an OR gates makes the output 1, this circuit will "latch" in the 1 output state after any time that A is 1. When A is 0, the output could be either 0 or 1, *depending on the circuit's prior state!* The proper way to complete the above truth table would be to insert the word *latch* in place of the question mark, showing that the output maintains its last state when A is 0.

Any digital circuit employing feedback is called a *multivibrator*. The example we just explored with the OR gate was a very simple example of what is called a *bistable* multivibrator. It is called "bistable" because it can hold stable in one of *two* possible output states, either 0 or 1. There are also *monostable* multivibrators, which have only *one* stable output state (that other state being momentary), which we'll explore later; and *astable* multivibrators, which have no stable state (oscillating back and forth between an output of 0 and 1).

A very simple astable multivibrator is an inverter with the output fed directly back to the input:

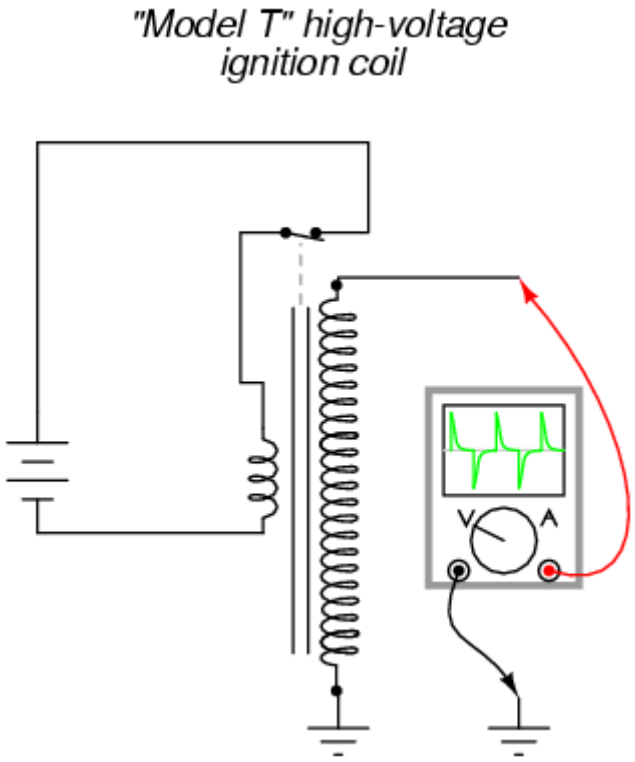
Inverter with feedback





When the input is 0, the output switches to 1. That 1 output gets fed back to the input as a 1. When the input is 1, the output switches to 0. That 0 output gets fed back to the input as a 0, and the cycle repeats itself. The result is a high frequency (several megahertz) oscillator, if implemented with a solid-state (semiconductor) inverter gate:

If implemented with relay logic, the resulting oscillator will be considerably slower, cycling at a frequency well within the audio range. The *buzzer* or *vibrator* circuit thus formed was used extensively in early radio circuitry, as a way to convert steady, low-voltage DC power into pulsating DC power which could then be stepped up in voltage through a transformer to produce the high voltage necessary for operating the vacuum tube amplifiers. Henry Ford's engineers also employed the buzzer/transformer circuit to create continuous high voltage for operating the spark plugs on Model T automobile engines:



Borrowing terminology from the old mechanical buzzer (vibrator) circuits, solid-state circuit engineers referred to any circuit with two or more vibrators linked together as a *multivibrator*. The astable multivibrator mentioned previously, with only one "vibrator," is more commonly implemented with multiple gates, as we'll see later.

The most interesting and widely used multivibrators are of the bistable variety, so we'll explore them in detail now.

Source: Tony R. Kuphaldt, https://workforce.libretexts.org/Bookshelves/Electronics_Technology/Book%3A_Electric_Circuits_IV_-_Digital_Circuitry_(Kuphaldt)/10%3A_Multivibrators

3.4: Sequential Logic Design

Sequential Circuits

Read this chapter on sequential circuits. Combinatorial circuits have outputs that depend on the inputs. Sequential circuits and finite state machines have outputs that depend on the inputs AND the current state, values stored in memory.

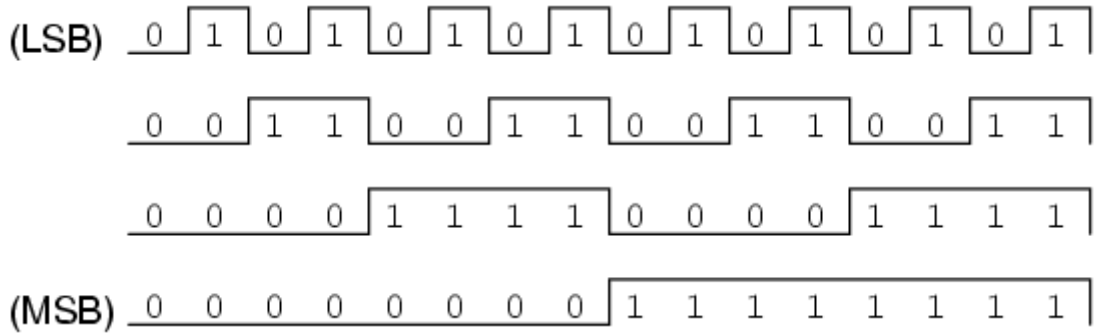
Binary Count Sequence

If we examine a four-bit binary count sequence from 0000 to 1111, a definite pattern will be evident in the "oscillations" of the bits between 0 and 1:

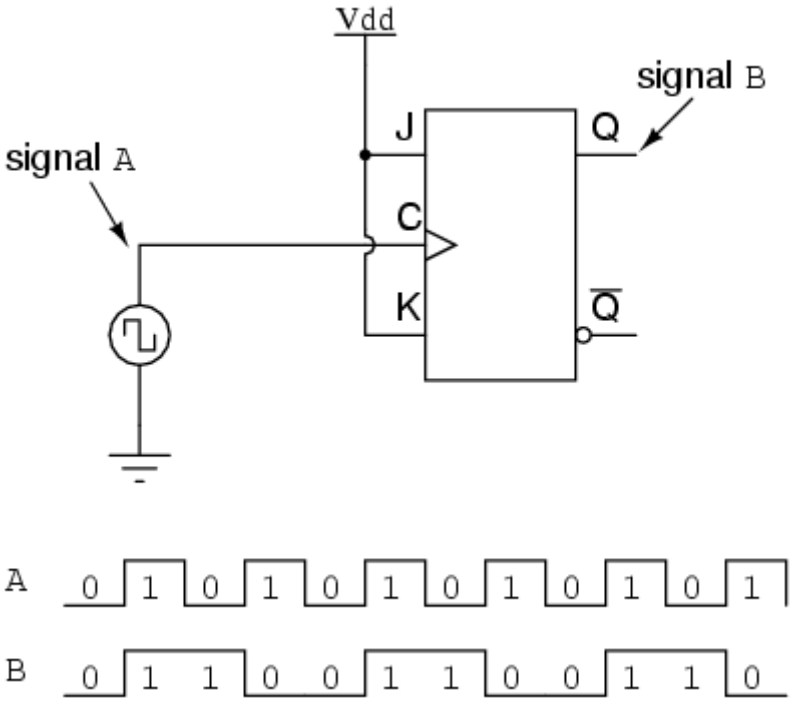
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

Note how the least significant bit (LSB) toggles between 0 and 1 for every step in the count sequence, while each succeeding bit toggles at one-half the frequency of the one before it. The most significant bit (MSB) only toggles once during the entire sixteen-step count sequence: at the transition between 7 (0111) and 8 (1000).

If we wanted to design a digital circuit to "count" in four-bit binary, all we would have to do is design a series of frequency divider circuits, each circuit dividing the frequency of a square-wave pulse by a factor of 2:



J-K flip-flops are ideally suited for this task, because they have the ability to "toggle" their output state at the command of a clock pulse when both J and K inputs are made "high" (1):



If we consider the two signals (A and B) in this circuit to represent two bits of a binary number, signal A being the LSB and signal B being the MSB, we see that the count sequence is backward: from 11 to 10 to 01 to 00 and back again to 11. Although it might not be counting in the direction we might have assumed, at least it counts!

The following sections explore different types of counter circuits, all made with J-K flip-flops, and all based on the exploitation of that flip-flop's toggle mode of operation.

REVIEW:

- Binary count sequences follow a pattern of octave frequency division: the frequency of oscillation for each bit, from LSB to MSB, follows a divide-by-two pattern. In other words, the LSB will oscillate at the highest frequency, followed by the next bit at one-half the LSB's frequency, and the next bit at one-half the frequency of the bit before it, etc.
- Circuits may be built that "count" in a binary sequence, using J-K flip-flops set up in the "toggle" mode.

Source: Tony R. Kuphaldt, https://workforce.libretexts.org/Bookshelves/Electronics_Technology/Book%3A_Electric_Circuits_IV_-_Digital_Circuitry_(Kuphaldt)/11%3A_Sequential_Circuits

3.5: Case Study: Design of a Finite State Machine (FSM) to Control a Vending Machine

Finite State Automata

Read this article for an example of a finite state machine design of a simple vending machine.A sequential circuit is also called a sequential machine or a finite state machine (FSM) or a finite state automaton. This case study gives an example of the design of a sequential circuit. Each state is assigned a distinct set of 1's and 0's, not using more variables than necessary. Thus 4 states require 2 variables, 00, 01, 10 and 11. A binary table represents the input/output behavior of the circuit. We use a sequential circuit, because the output also depends on the state. Recall that state requires memory (that is, flip flops). Thus, a binary table with entries that give the output and next state for given inputs and current state represents the design of the machine. A finite state machine diagram can also represent the design: circles represent states; arrows represent transitions next to states; and inputs and outputs label the arrows (sometimes written as input/ output). Finally, Boolean equations can also represent the design. Lastly, Karnaugh maps or Boolean logic rules can be used to simplify (or minimize) the equations, and thus the design.

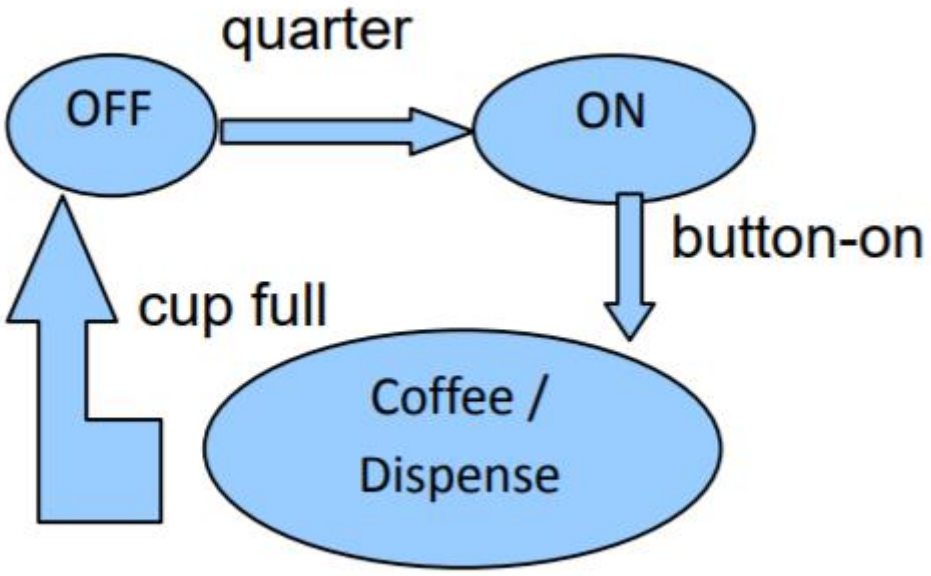
Finite State Automaton, also named finite-state machines, are just a special type of graph. In defining them we will again use set theory, showing the wide utility of sets.

A Finite State Automaton is a tuple (S, I, O, ns, o), where S is a finite set of state, I is a finite set of input symbols, O is a finite set of output symbols, ns a function from S x I to S is the next state function, and o is a function from S to O is the output function. There is a distinguished state, so, which is the initial or starting state. An example of a finite state automaton is given in the next paragraph.

Assume a simple vending machine that takes a quarter, button-on, and "cup full" as inputs, and coffee as output. This is a minimal 'no frills' vending machine. We give a graph representation of this machine, its state transition diagram which describes the behavior of the machine.

Finite State Automaton Given by a Transition Diagram

Instruction: For each state we draw a vertex or node. The input function maps a state and an input to a next state. For each such state and input we draw an arc to the node that the input function maps to. The resulting diagram is as follows:



Note that the "/" in the Coffee / Dispense state. The state is written above the "/" and the output is written below. Dispense is short-hand for dispense the cup and the coffee into the cup.

Generating Finite State Automaton from Next-State Tables

A finite state machine can be represented via a graph diagram, as above, or equivalently using a table. Given the diagram one can construct the next state table; conversely, given the next state table one can draw the state diagram.

The next state table for the above diagram is as follows:

Current State	Input	Next State	Output
OFF	quarter	ON	None
ON	button-on	Coffee	Dispense
COFFEE	cup full	OFF	None

We constructed the table by reading the entries from the next-state diagram. For example, in the diagram, there is an arc that connects the "OFF" state with the "ON" state, labeled "quarter." Hence, we enter "quarter" into the cell having row labeled OFF and column labeled Input.

The reverse process of drawing the transition diagram from a given next-state table is similar, but in reverse. Draw a node for each state in the table; for the above table, the states are the labels of the rows and the labels of the Next State column. Draw an arc for each entry in a cell whose row is a Current State and whose column is Next State. The label on the arc is the entry in the Input column for that Current State and Next State. In

each Current State node for which there is an entry in the Output column, write the corresponding entry below the name of the Current State node after a "/.

Source: Saylor Academy

Unit 4: Computer Arithmetic

In this unit, you will build upon your knowledge of computer instructions and digital logic design to discuss the role of computer arithmetic in hardware design. We will also discuss the designs of adders, multipliers, and dividers. You will learn that there are two types of arithmetic operations performed by computers: integer and floating point. Finally, we will discuss floating point details for carrying out operations with real numbers.

- Upon successful completion of this unit, you will be able to:
    - show the block diagram for a simple adder and show the outputs for a simple 4 bit addition;
    - show how to use a 4 bit adder to perform multiplication, subtraction, and division; and
    - design a full adder from a half adder.

- 4.1: Number Representation

Integers and the Representation of Real Numbers

Read these sections on the representation of integers and real numbers. Earlier, you read about number systems and the representation of numbers used for computing. This will give you a chance to review that material. Computer architecture comprises components which perform the functions of storage of data, transfer of data from one component to another, computations, and interfacing to devices external to the computer. Data is stored in terms of units, called words. A word is made up of a number of bits, typically, depending on the computer, 32 bits or 64 bits. Words keep getting longer, with larger numbers of bits. Instructions are also stored in words. Before, you saw examples of how instructions are stored in a word or words. Now, you will see how numbers are stored in words.

Integers

In scientific computing, most operations are on real numbers. Computations on integers rarely add up to any serious computation load. It is mostly for completeness that we start with a short discussion of integers.

Integers are commonly stored in 16, 32, or 64 bits, with 16 becoming less common and 64 becoming more and more so. The main reason for this increase is not the changing nature of computations, but the fact that integers are used to index arrays. As the size of data sets grows (in particular in parallel computations), larger indices are needed. For instance, in 32 bits one can store the numbers zero through $2_{32}-1\approx4\cdot10_{9}2_{32}-1\approx4\cdot10_{9}$.

In other words, a 32 bit index can address 4 gigabytes of memory. Until recently this was enough for most purposes; these days the need for larger data sets has made 64 bit indexing necessary.

When we are indexing an array, only positive integers are needed. In general integer computations, of course, we need to accommodate the negative integers too. There are several ways of implementing negative integers. The simplest solution is to reserve one bit as a *sign bit*, and use the remaining 31 (or 15 or 63; from now on we will consider 32 bits the standard) bits to store the absolute magnitude.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as bitstring	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as integer	0	$\cdots$	$2^{31}-1$	-0	$\cdots$	-2^{31}

This scheme has some disadvantages, one being that there is both a positive and negative number zero. This means that a test for equality becomes more complicated than simply testing for equality as a bitstring.

The scheme that is used most commonly is called 2's complement, where integers are represented as follows.

- If $0 \leq m \leq 2^{31}-1$0≤m≤231−1, the normal bit pattern for *m* is used.
- If $1 \leq n \leq 2^{31}$1≤n≤231, then $-n$−n is represented by the bit pattern for $2^{32}-n$232−n.

bitstring	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
interpretation as bitstring	0	$\ldots$	$2^{31}-1$	2^{31}	$\ldots$	$2^{32}-1$
interpretation as integer	0	$\cdots$	$2^{31}-1$	$2^{32}-2^{31}$	$\cdots$	-1

Some observations:

- There is no overlap between the bit patterns for positive and negative integers, in particular, there is only one pattern for zero.
- The positive numbers have a leading bit zero, the negative numbers have the leading bit set.

Exercise 3.1. For the 'naive' scheme and the 2's complement scheme for negative numbers, give pseudocode for the comparison test $m < n$, where *m* and *n* are integers. Be careful to distinguish between all cases of *m*; *n* positive, zero, or negative.

Adding two numbers with the same sign, or multiplying two numbers of any sign, may lead to a result that is too large or too small to represent. This is called *overflow*.

Exercise 3.2. Investigate what happens when you perform such a calculation. What does your compiler say if you try to write down a nonrepresentible number explicitly, for instance in an assignment statement?

In exercise 1 above you explored comparing two integers. Let us know explore how subtracting numbers in two's complement is implemented.
Consider $0 \leq m \leq 2^{31} - 1$ and $1 \leq n \leq 2^{31}$ a nd let us see what happens in the computation of $m - n$.

Suppose we have an algorithm for adding and subtracting unsigned 32-bit numbers. Can we use that to subtract two's complement integers? We start by observing that the integer subtraction $m - n$ becomes the unsigned addition $m + (2^{32} - n)$.

- Case: $m < n$.
  Since $m + (2^{32} - n) = 2^{32} - (n - m)$ and $1 \leq n - m \leq 2^{31}$, we conclude that $(2^{32} - (n - m)$ $1 \leq n - m \leq 231$, we conclude that (232−(n−m) is a valid bit pattern. Moreover, it is the bit pattern representation of the negative number $m - n$, so we cnan indeed compute $m - n$ as an unsigned operation on the bitstring representations of *m* and *n*.
- Case: $m < n$. Here we observe that $m + (2^{32} - n) = 2^{32} + m - n$. Since $m - n > 0$, this is a number $> 2^{32}$ and therefore not a legitimate expression of a negative number. However, if we store this number in 33 bits, we see that it is the correct result $m - n$, plus a single bit in the 33-rd position. Thus, by performing the unsigned addition, and ignoring the *overflow bit*, we again get the correct result.

In both cases we conclude that we can perform subtraction by adding the bitstrings that represent the positive and negative number as unsigned integers, and ignoring overflow if it occurs.

Representation of real numbers

In this section we will look at how various kinds of numbers are represented in a computer, and the limitations of various schemes. The next section will then explore the ramifications of this on arithmetic involving computer numbers.

Real numbers are stored using a scheme that is analogous to what is known as 'scientific notation', where a number is represented as a *significant* and an *exponent*, for instance 6.022×10^{23}, which has a significant 6.022 with a *radix* point after the first digit, and an exponent 23. This number stands for

$$6.022 \cdot 2023 = [6 \times 10^0 + 0 \times 10^{-2} + 2 \times 10^{-2} + 2 \times 10^{-3}] \cdot 1024$$

We introduce a base, a small integer number, 10 in the preceding example, and 2 in computer numbers, and write numbers in terms of it as a sum of *t* terms:

$$x = \pm 1 \times [d_1\beta_0 + d_2\beta_{-1} + d_3\beta_2 + \ldots] \times \beta_e = \pm \sum_{ti=1} d_i\beta_{1-i} \times \beta_e x = \pm 1 \times [d1\beta 0 + d2\beta - 1 + d3\beta 2 + \ldots] \times \beta e = \pm \sum i=1 tdi\beta 1 - i \times \beta e$$

where the components are

- the *sign bit*: a single bit storing whether the number is positive or negative;
- $\beta\beta$ is the base of the number system;
- $0 \leq d_i \leq \beta - 10 \leq di \leq \beta - 1$ the digits of the *mantissa* or *significant* – the location of the radix point (decimal point in decimal numbers) is implicitly assumed to the immediately following the first digit;
- *t* is the length of the mantissa;
- $e \in [L,U]e \in [L,U]$ exponent; typically $L < 0 < UL < 0 < U$ and $L \approx -UL \approx -U$

Note that there is an explicit sign bit for the whole number; the sign of the exponent is handled differently. For reasons of efficiency, *e* is not a signed number; instead it is considered as an unsigned number in excess of a certain minimum value. For instance, the bit pattern for the number zero is interpreted as *e* = *L*.

Some examples

Let us look at some specific examples of floating point representations. Base 10 is the most logical choice for human consumption, but computers are binary, so base 2 predominates there. Old IBM mainframes grouped bits to make for a base 16 representation.

	β	t	L	U
IEEE single precision (32 bit)	2	24	-126	127
IEEE double precision (64 bit)	2	53	-1022	1023
Old Cray 64 bit	2	48	-16383	16384
IBM mainframe 32 bit	16	6	-64	63
packed decimal	10	50	-999	999
Setun	3			

Of these, the single and double precision formats are by far the most common. We will discuss these in section 3.2.4 and further.

Binary coded decimal

Decimal numbers are not relevant in scientific computing, but they are useful in financial calculations, where com-putations involving money absolutely have to be exact. Binary arithmetic is at a disadvantage here, since numbers such as 1/10 are repeating fractions in binary. With a finite number of bits in the mantissa, this means that the number 1/10 can not be represented exactly in binary. For this reason, *binary-coded-decimal* schemes were used in old IBM mainframes, and are in fact being standardized in revisions of IEEE754 [4]; see also section 3.2.4. Few processors these days have hardware support for BCD; one example is the IBM Power6.

In BCD schemes, one or more decimal digits are encoded in a number of bits. The simplest scheme would encode the digits 0 . . . 9 in four bits. This has the advantage that in a BCD number each digit is readily identified; it has the disadvantage that about 1/3 of all bits are wasted, since 4 bits can encode the numbers 0 . . . 15. More efficient encodings would encode 0 . . . 999 in ten bits, which could in principle store the numbers 0 . . . 1023. While this is efficient in the sense that few bits are wasted, identifying individual digits in such a number takes some decoding.

Ternary computers

There have been some experiments with ternary arithmetic [2, 8, 9].

Limitations

Since we use only a finite number of bits to store floating point numbers, not all numbers can be represented. The ones that can not be represented fall into two categories: those that are too large or too small (in some sense), and those that fall in the gaps. Numbers can be too large or too small in the following ways.

Overflow: The largest number we can store is $(1-\beta^{-t-1})\beta^{U}(1-\beta-t-1)\beta U$, and the smallest number (in an absolute sense) is $-(1-\beta^{-t-1})\beta^{U}-(1-\beta-t-1)\beta U$; anything larger than the former or smaller than the latter causes a condition called *overflow*.

Underflow: The number closest to zero is $\beta^{-t-1}\cdot\beta^{L}\beta-t-1\cdot\beta L$. A computation that has a result less than that (in absolute value) causes a condition called underflow. In fact, most computers use *normalized floating point numbers*: the first digit d_1 is taken to be nonzero; see section 3.2.3 for more about this. In this case, any number less than $\beta^{-1}\cdot\beta^{L}\beta-1\cdot\beta L$ causes underflow. Trying to compute a number less than that is sometimes handled by using *unnormalized floating point numbers* (a process known

as *gradual underflow*), but this is typically tens or hundreds of times slower than computing with regular floating point numbers. At the time of this writing, only the IBM Power6 has hardware support for gradual underflow.

The fact that only a small number of real numbers can be represented exactly is the basis of the field of round-off error analysis. We will study this in some detail in the following sections.

Normalized numbers and machine precision

The general definition of floating point numbers, equation (3.1), leaves us with the problem that numbers have more than one representation. For instance, $.5\times 10^2 = .05\times 10^3$ $.5\times 10^2 = .05\times 10^3$. Since this would make computer arithmetic needlessly complicated, for instance in testing equality of numbers, we use *normalized floating point numbers*. A number is normalized if its first digit is nonzero. The implies that the mantissa part is $\beta > x_m \geq 1$ $\beta > x_m \geq 1$.

A practical implication in the case of binary numbers is that the first digit is always 1, so we do not need to store it explicitly. In the IEEE 754 standard, this means that every floating point number is of the form

$$1.d_1 d_2 \ldots d_t \times 2^{\text{exp}}$$ $$1.d_1 d_2 \ldots d_t \times 2^{\text{exp}}$$

We can now be a bit more precise about the *representation error*. A machine number $\bar{x}$ $\bar{x}$ is the representation for all x in an interval around it. With t digits in the mantissa, this is the interval of numbers that differ from x in the t + 1st digit. For the mantissa part we get:

$$\begin{cases} x \in [\tilde{x}, \tilde{x} + \beta^{-2}] & \text{truncation} \\ x \in [\tilde{x} - \frac{1}{2}\beta^{-t}, \tilde{x} + \frac{1}{2}\beta^{-t}] & \text{rounding} \end{cases}$$ $$\begin{cases} x \in [\tilde{x}, \tilde{x} + \beta^{-2}] & \text{truncation} \\ x \in [\tilde{x} - \frac{1}{2}\beta^{-t}, \tilde{x} + \frac{1}{2}\beta^{-t}] & \text{rounding} \end{cases}$$

Often we are only interested in the order of magnitude of the error, and we will write $\tilde{x} = x(1+\epsilon)$ $\tilde{x} = x(1+\epsilon)$, where $|\epsilon| \leq \beta^{-t}$ $|\epsilon| \leq \beta^{-t}$. This maximum relative error is called the *machine precision*, or sometimes machine epsilon. Typical values are:

$$\begin{cases} \epsilon \approx 10^{-7} & 32\text{-bit single precision} \\ \epsilon \approx 10^{-16} & 64\text{-bit double precision} \end{cases}$$ $$\begin{cases} \epsilon \approx 10^{-7} & 32\text{-bit single precision} \\ \epsilon \approx 10^{-16} & 64\text{-bit double precision} \end{cases}$$

sign	exponent	mantissa
s	$e_1 \cdots e_8$	$s_1 \ldots s_{23}$
31	$30 \cdots 23$	$22 \cdots 0$

$(e_1 \cdots e_8)$	numerical value
$(0 \cdots 0) = 0$	$\pm 0.s_1 \cdots s_{23} \times 2^{-126}$
$(0 \cdots 01) = 1$	$\pm 1.s_1 \cdots s_{23} \times 2^{-126}$
$(0 \cdots 010) = 2$	$\pm 1.s_1 \cdots s_{23} \times 2^{-125}$
$\cdots$	
$(01111111) = 127$	$\pm 1.s_1 \cdots s_{23} \times 2^{0}$
$(10000000) = 128$	$\pm 1.s_1 \cdots s_{23} \times 2^{1}$
$\cdots$	
$(11111110) = 254$	$\pm 1.s_1 \cdots s_{23} \times 2^{127}$
$(11111111) = 255$	$\pm\infty$ if $s_1 \cdots s_{23} = 0$, otherwise NaN

Figure 3.1: Single precision arithmetic

Machine precision can be defined another way: is the smallest number that can be added to 1 so that 1 + has a different representation than 1. A small example shows how aligning exponents can shift a too small operand so that it is effectively ignored in the addition operation:

$$\begin{array}{rll} & 1.0000 & \times 10^{0} \\ + & 1.0000 & \times 10^{-5} \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{rll} & 1.0000 & \times 10^{0} \\ + & 0.00001 & \times 10^{0} \\ \hline = & 1.0000 & \times 10^{0} \end{array}$$

Yet another way of looking at this is to observe that, in the addition $x+y$, if the ratio of x and y is too large, the result will be identical to x.

The machine precision is the maximum attainable accuracy of computations: it does not make sense to ask for more than 6-or-so digits accuracy in single precision, or 15 in double.

Exercise 3.3. Write a small program that computes the machine epsilon. Does it make any difference if you set the compiler optimization levels low or high? Can you find other ways in which this computation goes wrong?

The IEEE 754 standard for floating point numbers

Some decades ago, issues like the length of the mantissa and the rounding behaviour of operations could differ between computer manufacturers, and even between models from one manufacturer. This was obviously a bad situation from a point of portability of codes and reproducibility of results. The IEE standard 754[2] codified all this, for instance

stipulating 24 and 53 bits for the mantissa in single and double precision arithmetic, using a storage sequence of sign bit, exponent, mantissa. This for instance facilitates comparison of numbers.

The standard also declared the rounding behaviour to be 'exact rounding': the result of an operation should be the rounded version of the exact result.

Above (section 3.2.2), we have seen the phenomena of overflow and underflow, that is, operations leading to un-representible numbers. There is a further exceptional situation that needs to be dealt with: what result should be returned if the program asks for illegal operations such as $-4---\sqrt{-4}$? The IEEE 754 standard has two special quantities for this: Inf and NaN for 'infinity' and 'not a number'. If NaN appears in an expression, the whole expression will evaluate to that value. The rule for computing with Inf is a bit more complicated [43].

An inventory of the meaning of all bit patterns in IEEE 754 double precision is given in figure 3.1. Note that for normalized numbers the first nonzero digit is a 1, which is not stored, so the bit pattern $d_1d_2...d_td1d2...dt$ is interpreted as

$1.d_1d_2...d_t1.d1d2...dt$

These days, almost all processors adhere to the IEEE 754 standard, with only occasional exceptions. For instance, Nvidia Tesla GPU s are not standard-conforming in single precision. The justification for this is that double precision is the 'scientific' mode, while single precision is mostly likely used for graphics, where exact compliance matters less.

Source: Victor Eijkhout, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

4.2: Addition and Subtraction Hardware

Add and Subtract Blocks

Study this article to learn how addition is implemented and carried out at the gate level. Nowadays, computers are architected using larger components. For example, to perform addition and subtraction, computer architects utilize ALUs, arithmetic logic units. You can design a computer without knowing the details of an ALU or of an adder, similar to using a calculator to find the square root of a number without knowing how to manually compute the square root (or in computer science terminology, without knowing the algorithm that the calculator performs to find the square root). However, we want you to have the strongest foundation in your study of computer architecture. Knowing the underlying algorithm for larger components, you will be able to better use them in constructing larger components, for example, using half adders to construct a full adder.

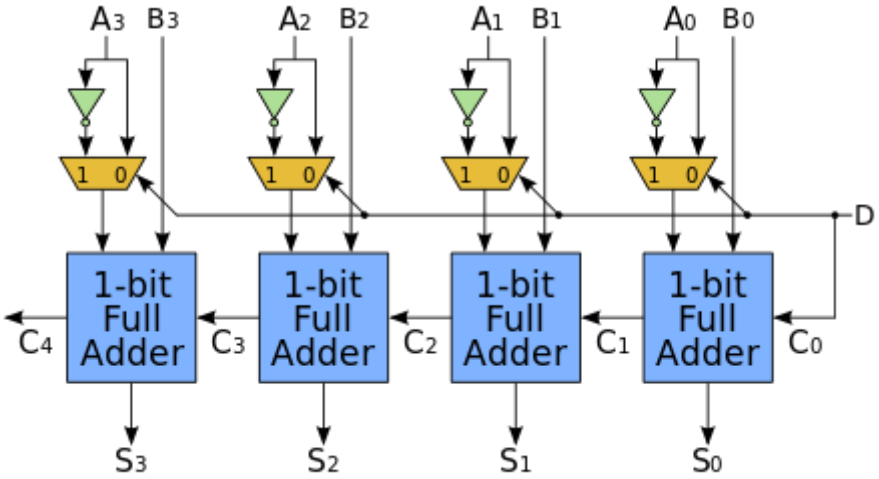
A half adder takes 2 bits as input and outputs a sum and a carry bit; a full adder takes 2 operand bits and a carry bit as input, and outputs a sum bit and a carry bit. To add two floating point numbers, one or both need to be put in a form such that they have the same exponent. Then, the mantissas are added. Lastly, the result is normalized. We will cover floating point addition later. Subtraction is not discussed explicitly, because it is done via addition by reversing the sign of the number to be subtracted (subtrahend) and adding the result to the number subtracted from (minuend). You have to be careful when subtracting floating point numbers, because of the possible large round off error in the result.

Addition and Subtraction

Addition and subtraction are similar algorithms. Taking a look at subtraction, we can see that:

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

Using this simple relationship, we can see that addition and subtraction can be performed using the same hardware. Using this setup, however, care must be taken to invert the value of the second operand if we are performing subtraction. Note also that in twos-compliment arithmetic, the value of the second operand must not only be inverted, but 1 must be added to it. For this reason, when performing subtraction, the carry input into the LSB should be a 1 and not a zero.



Our goal on this page, then, is to find suitable hardware for performing addition.

Bit Adders

Half Adder

A half adder is a circuit that performs binary addition on two bits. A half adder does not explicitly account for a carry input signal.

In verilog, a half-adder can be implemented as follows:

```
module half_adder(a, b, c, s);
 input a, b;
 output s, c;
 s = a ^ b;
 c = a & b;
endmodule
```

Full Adder

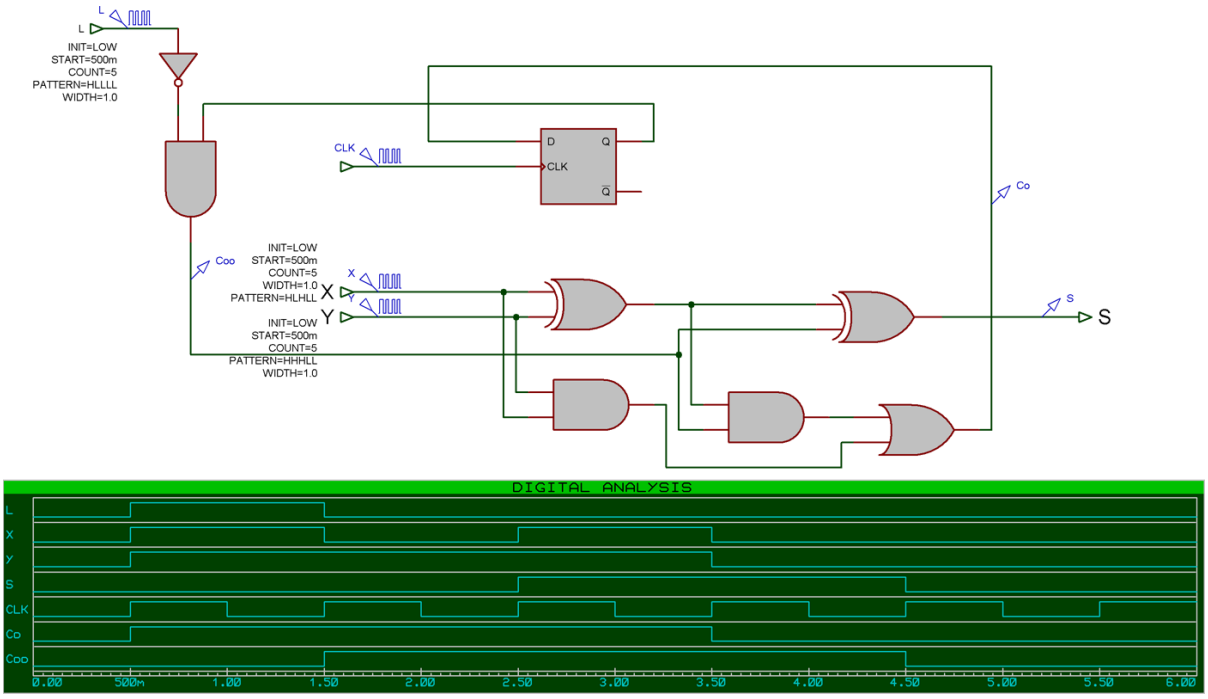
Full adder circuits are similar to the half-adder, except that they do account for a carry input and a carry output. Full adders can be treated as a 3-bit adder with a 2-bit result, or they can be treated as a single stage (a 3:2 compressor) in a larger adder.

As can be seen below, the number of gate delays in a full-adder circuit is 3:

We can use verilog to implement a full adder module:

```
module full_adder(a, b, cin, cout, s);
 input a, b, cin;
 output cout, s;
 wire temp;
 temp = a ^ b;
 s = temp ^ cin;
 cout = (cin & temp) | (a & b);
endmodule
```

Serial Adder[edit]



A serial adder is a kind of ALU that calculates each bit of the output, one at a time, re-using one full adder (total). This image shows a 2-bit serial adder, and the associated waveforms.

Serial adders have the benefit that they require the least amount of hardware of all adders, but they suffer by being the slowest.

Parallel Adder

A parallel adder is a kind of ALU that calculates every bit of the output more or less simultaneously, using one full adder for each output bit. The 1947 Whirlwind computer was the first computer to use a parallel adder.

In many CPUs, the CPU latches the final carry-out of the parallel adder in an external "carry flag" in a "status register".

In a few CPUs, the latched value of the carry flag is always wired to the first carry-in of the parallel adder; this gives "Add with carry" with 2s' complement addition. (In a very few CPUs, an end-around carry -- the final carry-out of the parallel adder is directly connected to the first carry-in of the same parallel adder -- gives 1's complement addition).

Ripple Carry Adder

Numbers of more than 1 bit long require more then just a single full adder to manipulate using arithmetic and bitwise logic instructions[citation needed]. A simple way of operating on larger numbers is to cascade a number of full-adder blocks together into a **ripple-carry adder**, seen above. Ripple Carry adders are so called because the carry value "ripples" from one block to the next, down the entire chain of full adders. The output values of the higher-order bits are not correct, and the arithmetic is not complete, until the carry signal has completely propagated down the chain of full adders.

If each full adder requires 3 gate delays for computation, then an *n*-bit ripple carry adder will require *3n* gate delays. For 32 or 64 bit computers (or higher) this delay can be overwhelmingly large.

Ripple carry adders have the benefit that they require the least amount of hardware of all adders (except for serial adders), but they suffer by being the slowest (except for serial adders).

With the full-adder verilog module we defined above, we can define a 4-bit ripple-carry adder in Verilog. The adder can be expanded logically:

```
wire [4:0] c;
wire [3:0] s;
full_adder fa1(a[0], b[0], c[0], c[1], s[0]);
full_adder fa2(a[1], b[1], c[1], c[2], s[1]);
full_adder fa3(a[2], b[2], c[2], c[3], s[2]);
full_adder fa4(a[3], b[3], c[3], c[4], s[3]);
```

At the end of this module, *s* contains the 4 bit sum, and c[4] contains the final carry out.

This "ripple carry" arrangement makes "add" and "subtract" take much longer than the other operations of an ALU (AND, NAND, shift-left, divide-by-two, etc). A few CPUs use a ripple carry ALU, and require the programmer to insert NOPs to give the "add" time to settle.[1] A few other CPUs use a ripple carry adder, and simply set the clock rate slow enough that there is plenty of time for the carry bits to ripple through the adder. A few CPUs use a ripple carry adder, and make the "add" instruction take more clocks than the "XOR" instruction, in order to give the carry bits more time to ripple through the adder on an "add", but without unnecessarily slowing down the CPU during a "XOR". However, it makes pipelining much simpler if every instruction takes the same number of clocks to execute.

Carry Skip Adder
Carry Lookahead Adder

Carry-lookahead adders use special "look ahead" blocks to compute the carry from a group of 4 full-adders, and passes this carry signal to the next group of 4 full adders. Lookahead units can also be cascaded, to minimize the number of gate delays to completely propagate the carry signal to the end of the chain. Carry lookahead adders are some of the fastest adder circuits available, but they suffer from requiring large amounts of hardware to implement. The number of transistors needed to implement a carry-lookahead adder is proportional to the number of inputs cubed.

The addition of two 1-digit inputs *A* and *B* is said to *generate* if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition 52 + 67, the addition of the tens digits 5 and 6 *generates* because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit clearly does not carry).

In the case of binary addition, $A+B$A+B generates if and only if both *A* and *B* are 1. If we write $G(A,B)$G(A,B) to represent the binary predicate that is true if and only if $A+B$A+B generates, we have:

$$G(A,B)=A\cdot B$$G(A,B)=A·B

The addition of two 1-digit inputs *A* and *B* is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

In the case of binary addition, $A+B$A+B propagates if and only if at least one of *A* or *B* is 1. If we write $P(A,B)$P(A,B) to represent the binary predicate that is true if and only if $A+B$A+B propagates, we have:

$$P(A,B)=A+B$$P(A,B)=A+B

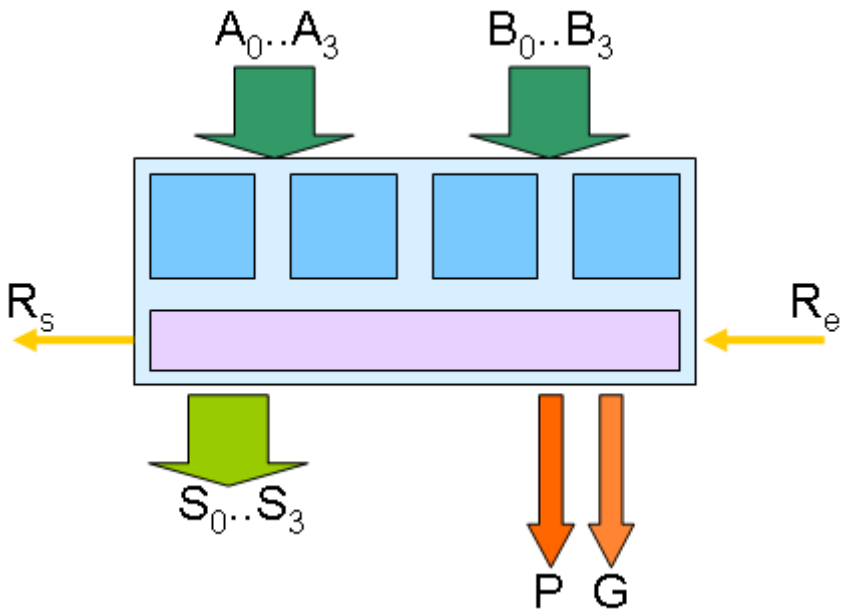
Cascading Adders

The power of carry-lookahead adders is that the bit-length of the adder can be expanded without increasing the propagation delay too much. By cascading lookahead modules, and passing "propagate" and "generate" signals to the next level of the lookahead module. For instance, once we have 4 adders combined into a simple lookahead module, we can use that to create a 16-bit and a 64-bit adder through cascading:

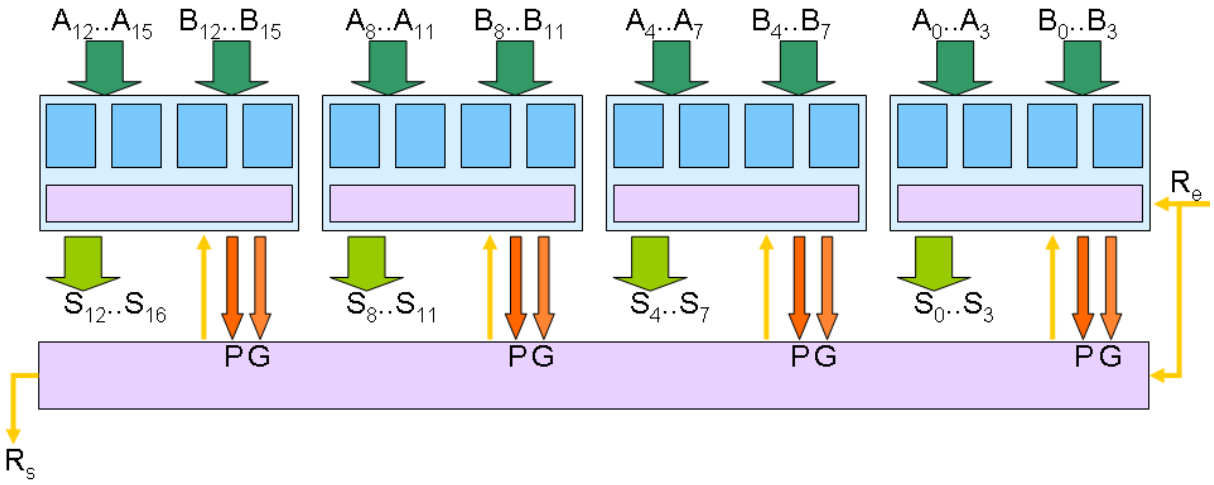
The 16-Bit carry lookahead unit is exactly the same as the 4-bit carry lookahead adder.

The 64-bit carry lookahead unit is exactly the same as the 4-bit and 16-bit units. This means that once we have designed one carry lookahead module, we can cascade it to any large size.

Generalized Cascading



A generalized CLA block diagram. Each of the turquoise blocks represents a smaller CLA adder.



We can cascade the generalized CLA block above to form a larger CLA block. This larger block can then be cascaded into a larger CLA block using the same method.

Source:
Wikibooks, https://en.wikibooks.org/wiki/Microprocessor_Design/Add_and_Subtract_Blocks

4.3: Multiplication

Binary Multipliers

Study this article to learn how multiplication is performed using basic steps that are carried out by elemental logic components, such as an adder and shifter. Two numbers are loaded into registers (fast word storage) and multiplication is carried out by repeated addition and shifting (moving the bits in a register to the right or left). Multiplication by an integer is performed by shifting the bits in the word to the left. Each shift multiplys by 2. Note that the ALU consists of the adder and associated circuitry that can also do ands, ors, inversions and other operations, so multiplications can be done by the adder or even by programs steps which can do the same process. A hardware multiplier can also be used, which is faster than using the adder and shifter or using programmming. To multiply two floating point numbers, add the exponents, multiply the mantissas, normalize the mantissa of the product, and adjust the exponent accordingly. Floating point multiplication will be covered later.

A **binary multiplier** is an electronic circuit used in digital electronics, such as a computer, to multiply two binary numbers. It is built using binary adders.

A variety of computer arithmetic techniques can be used to implement a digital multiplier. Most techniques involve computing a set of *partial products*, and then summing the partial products together. This process is similar to the method taught to primary schoolchildren for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system.

History

Between 1947-1949 Arthur Alec Robinson worked for English Electric Ltd, as a student apprentice, and then as a development engineer. Crucially during this period he studied for a PhD degree at the University of Manchester, where he worked on the design of the hardware multiplier for the early Mark 1 computer. However, until the late 1970s, most minicomputers did not have a multiply instruction, and so programmers used a "multiply routine" which repeatedly shifts and accumulates partial results, often written using loop unwinding. Mainframe computers had multiply instructions, but they did the same sorts of shifts and adds as a "multiply routine".

Early microprocessors also had no multiply instruction. Though the multiply instruction is usually associated with the 16-bit microprocessor generation, at least two "enhanced" 8-bit micro have a multiply instruction: the Motorola 6809, introduced in 1978, and Intel MCS-51 family, developed in 1980, and later the modern Atmel AVR 8-bit microprocessors present in the ATMega, ATTiny and ATXMega microcontrollers.

As more transistors per chip became available due to larger-scale integration, it became possible to put enough adders on a single chip to sum all the partial products at once, rather than reuse a single adder to handle each partial product one at a time.

Because some common digital signal processing algorithms spend most of their time multiplying, digital signal processor designers sacrifice considerable chip area in order to make the multiply as fast as possible; a single-cycle multiply–accumulate unit often used up most of the chip area of early DSPs.

Basics

The method taught in school for multiplying decimal numbers is based on calculating partial products, shifting them to the left and then adding them together. The most difficult part is to obtain the partial products, as that involves multiplying a long number by one digit (from 0 to 9):

```
    123
  x 456
  =====
    738  (this is 123 x 6)
   615   (this is 123 x 5, shifted one position to the left)
+ 492    (this is 123 x 4, shifted two positions to the left)
  =====
  56088
```

Binary numbers

A binary computer does exactly the same multiplication as decimal numbers do, but with binary numbers. In binary encoding each long number is multiplied by one digit (either 0 or 1), and that is much easier than in decimal, as the product by 0 or 1 is just 0 or the same number. Therefore, the multiplication of two binary numbers comes down to calculating partial products (which are 0 or the first number), shifting them left, and then adding them together (a binary addition, of course):

```
      1011   (this is 11 in binary)
    x 1110   (this is 14 in binary)
    ======
      0000   (this is 1011 x 0)
     1011    (this is 1011 x 1, shifted one position to the left)
    1011     (this is 1011 x 1, shifted two positions to the left)
+ 1011       (this is 1011 x 1, shifted three positions to the left)
=========
 10011010   (this is 154 in binary)
```

This is much simpler than in the decimal system, as there is no table of multiplication to remember: just shifts and adds.

This method is mathematically correct and has the advantage that a small CPU may perform the multiplication by using the shift and add features of its arithmetic logic unit rather than a specialized circuit. The method is slow, however, as it involves many intermediate additions. These additions are time-consuming. Faster multipliers may be engineered in order to do fewer additions; a modern processor can multiply two 64-bit numbers with 6 additions (rather than 64), and can do several steps in parallel. [*citation needed*]

The second problem is that the basic school method handles the sign with a separate rule ("+ with + yields +", "+ with − yields −", etc.). Modern computers embed the sign of

the number in the number itself, usually in the two's complement representation. That forces the multiplication process to be adapted to handle two's complement numbers, and that complicates the process a bit more. Similarly, processors that use ones' complement, sign-and-magnitude, IEEE-754 or other binary representations require specific adjustments to the multiplication process.

Unsigned numbers

For example, suppose we want to multiply two unsigned eight bit integers together: *a*[7:0] and *b*[7:0]. We can produce eight partial products by performing eight one-bit multiplications, one for each bit in multiplicand *a*:

```
p0[7:0] = a[0] × b[7:0] = {8{a[0]}} & b[7:0]
p1[7:0] = a[1] × b[7:0] = {8{a[1]}} & b[7:0]
p2[7:0] = a[2] × b[7:0] = {8{a[2]}} & b[7:0]
p3[7:0] = a[3] × b[7:0] = {8{a[3]}} & b[7:0]
p4[7:0] = a[4] × b[7:0] = {8{a[4]}} & b[7:0]
p5[7:0] = a[5] × b[7:0] = {8{a[5]}} & b[7:0]
p6[7:0] = a[6] × b[7:0] = {8{a[6]}} & b[7:0]
p7[7:0] = a[7] × b[7:0] = {8{a[7]}} & b[7:0]
```

where {8{a[0]}} means repeating a[0] (the 0th bit of a) 8 times (Verilog notation).

To produce our product, we then need to add up all eight of our partial products, as shown here:

```
                                                p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]
                                      + p1[7] p1[6] p1[5] p1[4] p1[3] p1[2] p1[1] p1[0] 0
                            + p2[7] p2[6] p2[5] p2[4] p2[3] p2[2] p2[1] p2[0] 0     0
                  + p3[7] p3[6] p3[5] p3[4] p3[3] p3[2] p3[1] p3[0] 0     0     0
        + p4[7] p4[6] p4[5] p4[4] p4[3] p4[2] p4[1] p4[0] 0     0     0     0
  + p5[7] p5[6] p5[5] p5[4] p5[3] p5[2] p5[1] p5[0] 0     0     0     0     0
+ p6[7] p6[6] p6[5] p6[4] p6[3] p6[2] p6[1] p6[0] 0     0     0     0     0     0
+ p7[7] p7[6] p7[5] p7[4] p7[3] p7[2] p7[1] p7[0] 0     0     0     0     0     0     0
-------------------------------------------------------------------------------------
P[15] P[14] P[13] P[12] P[11] P[10]  P[9]  P[8]  P[7]  P[6]  P[5]  P[4]  P[3]  P[2]  P[1]  P[0]
```

In other words, *P*[15:0] is produced by summing *p*0, *p*1 << 1, *p*2 << 2, and so forth, to produce our final unsigned 16-bit product.

Signed integers

If *b* had been a signed integer instead of an unsigned integer, then the partial products would need to have been sign-extended up to the width of the product before summing. If *a* had been a signed integer, then partial product *p7* would need to be subtracted from the final sum, rather than added to it.

The above array multiplier can be modified to support two's complement notation signed numbers by inverting several of the product terms and inserting a one to the left of the first partial product term:

```
                                                1  ~p0[7]  p0[6]  p0[5]  p0[4]  p0[3]  p0[2]
p0[1]  p0[0]
```

```
                                                ~p1[7] +p1[6] +p1[5] +p1[4] +p1[3] +p1[2] +p1[1]
+p1[0]   0
                                          ~p2[7] +p2[6] +p2[5] +p2[4] +p2[3] +p2[2] +p2[1] +p2[0]
0      0
                                   ~p3[7] +p3[6] +p3[5] +p3[4] +p3[3] +p3[2] +p3[1] +p3[0]   0
0      0
                            ~p4[7] +p4[6] +p4[5] +p4[4] +p4[3] +p4[2] +p4[1] +p4[0]   0      0
0      0
                     ~p5[7] +p5[6] +p5[5] +p5[4] +p5[3] +p5[2] +p5[1] +p5[0]   0      0      0
0      0
              ~p6[7] +p6[6] +p6[5] +p6[4] +p6[3] +p6[2] +p6[1] +p6[0]   0      0      0      0
0      0
   1  +p7[7] ~p7[6] ~p7[5] ~p7[4] ~p7[3] ~p7[2] ~p7[1] ~p7[0]   0      0      0      0      0
0      0
 ------------------------------------------------------------------------------------------------
------------
P[15]  P[14]  P[13]  P[12]  P[11]  P[10]   P[9]   P[8]   P[7]   P[6]   P[5]   P[4]   P[3]   P[2]
P[1]  P[0]
```

Where ~p represents the complement (opposite value) of p.

There are many simplifications in the bit array above that are not shown and are not obvious. The sequences of one complemented bit followed by noncomplemented bits are implementing a two's complement trick to avoid sign extension. The sequence of p7 (noncomplemented bit followed by all complemented bits) is because we're subtracting this term so they were all negated to start out with (and a 1 was added in the least significant position). For both types of sequences, the last bit is flipped and an implicit -1 should be added directly below the MSB. When the +1 from the two's complement negation for p7 in bit position 0 (LSB) and all the -1's in bit columns 7 through 14 (where each of the MSBs are located) are added together, they can be simplified to the single 1 that "magically" is floating out to the left. For an explanation and proof of why flipping the MSB saves us the sign extension, see a computer arithmetic book.

Floating point numbers

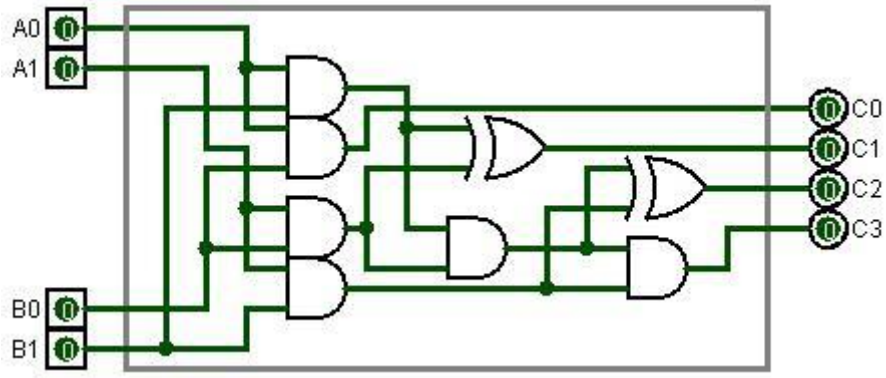
A binary floating number contains a sign bit, significant bits (known as the significand) and exponent bits (for simplicity, we don't consider base and combination field). The sign bits of each operand are XOR'd to get the sign of the answer. Then, the two exponents are added to get the exponent of the result. Finally, multiplication of each operand's significand will return the significand of the result. However, if the result of the binary multiplication is higher then the total number of bits for a specific precision (e.g. 32, 64, 128), rounding is required and the exponent is changed appropriately.

Implementations

Older multiplier architectures employed a shifter and accumulator to sum each partial product, often one partial product per cycle, trading off speed for die area. Modern multiplier architectures use the (Modified) Baugh–Wooley algorithm, Wallace trees, or Dadda multipliers to add the partial products together in a single cycle. The performance of the Wallace tree implementation is sometimes improved

by *modified* Booth encoding one of the two multiplicands, which reduces the number of partial products that must be summed.

Example circuits



2-bit by 2-bit binary multiplier

Source: Wikipedia, https://en.wikipedia.org/wiki/Binary_multiplier

Arithmetic Circuits

Read this article which gives another overview of multipliers.

Introduction

ALU

In this project, we will design arithmetic circuits using an FPGA. We will build a 4-bit magnitude comparator, a ripple-carry adder, and a multiplier circuit. You can challenge yourself by integrating all of those circuits together with some multiplexers to build an arithmetic logic unit (ALU).

Before you begin, you should:

- Have the Xilinx® ISE WebPACK™ installed.
- Have your FPGA board set up.
- Be able to describe digital circuits using logic operators.
- Be able to write test bench and simulate circuit using ISim.

After you're done, you should:

- Understand how magnitude comparators, ripple-carry adders, and multipliers work.
- Be able to describe magnitude comparators, ripple-carry adders, and multipliers structurally.

Inventory:

Qty	Description
1	Digilent Nexys™4, Nexys™3, Nexys™2, or Basys™2 FPGA Board
1	Xilinx ISE Design Suite: WebPACK (14.6 Recommended)
1	Digilent Adept

Comparator

A magnitude comparator is a device that receives two N-bit inputs and asserts one of three possible outputs depending on whether one input is greater than, less than, or equal to the other (simpler comparators, called equality comparators, provide a single output that is asserted whenever the two inputs are equal). The truth table of a bit-sliced magnitude comparator and the block diagram of a magnitude comparator are shown in Figs. 1 and 2 below.

Figure 1. Truth table for a bit-sliced magnitude comparator.

Figure 2. Block diagram of an 8-bit magnitude comparator using a bit-sliced magnitude comparator.

Step 1: Design a 4-bit Comparator

1. Create a Verilog module for a bit-sliced magnitude comparator according to the truth table presented in Fig. 1 above.

```
1
2
3  modulecmp_bitslice(
4  inputA,
5  inputB,
6  inputLT_I,
7  inputEQ_I,
8  inputGT_I,
9  outputLT_O,
10 outputEQ_O,
   outputGT_O
11 );
12 assignGT_O = ( A & ~B ) | GT_I;
13 assignEQ_O = EQ_I & (( A & B ) | (~A & ~B));
14 assignLT_O = ( B & ~A ) | LT_I;
15 endmodule
16
17
```

2. Connect the bit-sliced magnitude comparator according to the block diagram shown in Fig. 2 above.

```
modulecmp(
input[3:0] A,
input[3:0] B,
outputLT_O,
outputEQ_O,
outputGT_O
);
wire[3:0] GT_int;
wire[3:0] EQ_int;
wire[3:0] LT_int;
cmp_bitslice slice_0 (
.A(A[0]),
.B(B[0]),
.LT_I(1'b0),
.EQ_I(1'b1),
.GT_I(1'b0),
.LT_O(LT_int[0]),
.EQ_O(EQ_int[0]),
.GT_O(GT_int[0])
);
cmp_bitslice slice_1 (
.A(A[1]),
.B(B[1]),
.LT_I(LT_int[0]),
.EQ_I(EQ_int[0]),
.GT_I(GT_int[0]),
.LT_O(LT_int[1]),
.EQ_O(EQ_int[1]),
.GT_O(GT_int[1])
);

```

```
30 cmp_bitslice slice_2 (
31 .A(A[2]),
32 .B(B[2]),
33 .LT_I(LT_int[1]),
34 .EQ_I(EQ_int[1]),
35 .GT_I(GT_int[1]),
36 .LT_O(LT_int[2]),
37 .EQ_O(EQ_int[2]),
38 .GT_O(GT_int[2])
39 );
40
41 cmp_bitslice slice_3 (
42 .A(A[3]),
43 .B(B[3]),
44 .LT_I(LT_int[2]),
45 .EQ_I(EQ_int[2]),
46 .GT_I(GT_int[2]),
47 .LT_O(LT_int[3]),
48 .EQ_O(EQ_int[3]),
49 .GT_O(GT_int[3])
50 );
51
52 assignLT_O = LT_int[3];
53 assignEQ_O = EQ_int[3];
54 assignGT_O = GT_int[3];
55 endmodule
56
57
58
59
60
```

```
61
62
```

3. Create a test bench on your own to test the comparator. Please make sure that all possible cases are covered.
4. To implement a comparator, it is more convenient to use behavioral descriptions in Verilog. The following codes can implement the same circuit as the structural implementation above:

```
1
2  modulecmp(
3  input[3:0] A,
4  input[3:0] B,
5  outputLT_O,
6  outputEQ_O,
7  outputGT_O
8  );
9
10 assignLT_O = (A < B) ? 1'b1 : 1'b0;
11 assignEQ_O = (A == B) ? 1'b1 : 1'b0;
12 assignGT_O = (A > B) ? 1'b1 : 1'b0;
13 endmodule
14
```

5. Apply the same test bench on the second comparator and check the result.

Adders

Adder circuits add two N-bit operands to produce an N-bit result and a carry out signal (the carry out is a '1' only when the addition result requires more than N-bits). The logic graph in Fig. 3 below shows the eight different cases that may be encountered when adding two binary numbers. The highlighted bit pairs and the associated carries show that a bit-slice adder circuit must process three inputs (the two addend bits and a carry-in from the previous stage) and produce two outputs (the sum bit and a carry out bit). The circuit block that implements the bit-sliced adder is called a "full adder" (FA).

Figure 3. Truth table for a bit-slice adder (full adder).

Note the carry-in of the RCA LSB is connected directly to ground, because (by definition) the carry-in to the LSB of any adder must be logic '0'. It is possible to capitalize on this observation, and create a smaller bit-slice circuit for use in the LSB position that does not have a carry-in input. Called a half adder (HA), this reduced-function adder circuit is often used in the LSB position. The ripple carry adder block diagram is displayed in Fig. 4 below.

Figure 4. Ripple-carry adder block diagram.

Step 2: Design a 4-bit Binary Adder

1. Create a Verilog module for a full adder.

```
1 moduleFA(
2 inputA,
3 inputB,
4 inputCin,
5 outputS,
```

```
outputCout
);

assignS = A ^ B ^ Cin;
assignCout = (A & B) | ((A ^ B) & Cin);
endmodule


```

2. Create another Verilog module for a half adder.

```

moduleHA(
inputA,
inputB,
outputS,
outputCout
);
assignS = A ^ B;
assignCout = A & B;
endmodule

```

3. Create a Verilog module called "*adder*" to wrap three FAs with one HA to form a 4-bit adder:

```
moduleadder(
input[3:0] A,
input[3:0] B,
output[3:0] S,
outputCout
);
wire[3:0] Carry;
```

```
8  HA add_0 (
9  .A(A[0]),
10 .B(B[0]),
11 .S(S[0]),
12 .Cout(Carry[0])
   );
13 FA add_1 (
14 .A(A[1]),
15 .B(B[1]),
16 .Cin(Carry[0]),
17 .S(S[1]),
18 .Cout(Carry[1])
19 );
20 FA add_2 (
21 .A(A[2]),
22 .B(B[2]),
23 .Cin(Carry[1]),
24 .S(S[2]),
   .Cout(Carry[2])
25 );
26 FA add_3 (
27 .A(A[3]),
28 .B(B[3]),
29 .Cin(Carry[2]),
30 .S(S[3]),
31 .Cout(Carry[3])
32 );
33 assignCout = Carry[3];
34 endmodule
35
36
37
38
```

```
39
40
41
42
43
44
```

4. Create a test bench that covers all of the possible inputs to the adder and verify whether the result is correct. After the verification, you can tie SW3-0 to A and SW7-4 to B as two operands in UCF file, and tie LED4-0 to result of adder, you can test out your adder on your FPGA board.
5. Here, we provide you with another way of implementing adder in Verilog behaviorally. In the codes below, we use operator "+" to indicate an adder. It is very important to remember that when you put down a "+" in Verilog HDL, the tools will instantiate an adder for you, instead of just a operation as you might be familiar with in software programming language.

```
1
2
3  moduleadder(
4  input[3:0] A,
5  input[3:0] B,
6  output[3:0] S,
7  outputCout
8  );
9  wire[4:0] Result;
10 assignResult = A + B;
11 assignS = Result[3:0];
12 assignCout = Result[4];
13 endmodule
14
15
```

6.

Multipliers

Hardware multipliers, based directly on adder architectures, have become indispensable in modern computers. Multiplier circuits are modeled after the "shift and add" algorithm, as shown below. In this algorithm, one partial product is created for each bit in the multiplier—the first partial product is created by the LSB of the multiplier, the second partial product is created by the second bit in the multiplier, and so forth. The partial product bits need to be fed to an array of full adders (and half adders where appropriate), with the adders shifted to the left as indicated by the multiplication example. The final partial products are added with a CLA circuit. Note that some full-adder circuits bring signal values into the carry-in inputs (instead of carry's from the neighboring stage). This is a valid use of the full-adder circuit; the full adder simply adds any three bits applied to its inputs. The circuit for a partial product and the block diagram of the multiplier is shown in Fig. 5 below.

Figure 5. Hardware multiplier.

Step 3: Design a 4-bit Multiplier

Up to this point, you are expected to be able to describe circuits structurally. Based on the block diagram shown above in Fig. 5, you can use assignment statements to implement the circuit for partial products, and use the FA, HA, and the adder you implemented in previous steps to add those partial products together. Before deploying your circuit on your board, write a test bench to verify that your circuit is correct. Use SW3-0 as A and SW7-4 as B (A and B are multiplicants), and show the result of multiplication on 8-bit LEDs.

Unlike the adder and subtractor, multipliers do not have an operator support in Verilog, mostly due to the fact that there are various ways to implement a multiplier which trade off power, hardware resource, and speed. So implementing a multiplier structurally is the only solution.

Test Your Knowledge!

Now that you've completed this project, try these modifications:

1. Implement a 4-bit borrow ripple subtractor using bit-sliced design methodology and describe it structurally in Verilog.
2. Modify the adder you implemented in step 2 into an add_subtract circuit. Add an input "subtract". When "subtract" is '1' the output of add_subtract equals A−B. When "subtract" is "1", the output of add_subtract equals A+B. Inputs and outputs of the add_subtract circuit are represented in 2's complement. Instead of cout, an output "ERR" is needed as overflow/underflow indicator, i.e., ERR is "1" when overflow/underflow occurs.
3. Can you combine add_subtract with other logics to implement an ALU according to the following opcode table? (Inputs and Output of the ALU are 4-bit binary numbers in 2's complement).
4.

OpCode	Description	Output F
000	Addition	A+B
001	Increment	A+1
010	Subtract	A−B
011	Bit- wise XOR	A⊕B
100	Bit- wise OR	A|B
101	Bit-wise AND	A&B

Source: Digilent Inc., https://learn.digilentinc.com/Documents/261

Floating Point Arithmetic and Error Analysis

Read these sections, which explain error analysis when numbers are represented using a fixed number of digits. This issue mostly arises when using floating-point numbers. Real numbers are represented using a fixed number of bits. The number of bits is the precision of the representation. The accuracy of the representation is described in terms of the difference between the actual number and its representation using a fixed number of bits. This difference is the error of the representation. The accuracy becomes more significant because computations can cause the error to get so large that the result is meaningless and potentially even high risk depending on the application. These sections also explain how programming languages approach number representation.

Round-off error analysis

The fact that floating point numbers can only represent a small fraction of all real numbers, means that in practical circumstances a computation will hardly ever be exact. In this section we will study the phenomenon that most real numbers can not be represented, and what it means for the accuracy of computations. This is commonly called *round-off error analysis*.

Representation error

Numbers that are too large or too small to be represented are uncommon: usually computations can be arranged so that this situation will not occur. By contrast, the case that the result of a computation between computer numbers (even something as simple as a single addition) is not representable is very common. Thus, looking at the implementation of an algorithm, we need to analyze the effect of such small errors propagating through the computation. We start by analyzing the error between numbers that can be represented exactly, and those close by that can not be.

If x is a number and $\tilde{x}$ its representation in the computer, we call $x-\tilde{x}$ the *representation error* or *absolute representation error*, and $\frac{x-\tilde{x}}{x}$ the *relative representation error*. Often we are not interested in the sign of the error, so we may apply the terms error and relative error to $|x-\tilde{x}|$ and $\left|\frac{x-\tilde{x}}{x}\right|$ respectively.

Often we are only interested in bounds on the error. If is a bound on the error, we will write

$$\tilde{x}=x\pm\epsilon\equiv D|x-\tilde{x}|\leq\epsilon\Leftrightarrow\tilde{x}\in[x-\epsilon,x+\epsilon]$$

For the relative error we note that

$$\tilde{x}=x(1+\epsilon)\Leftrightarrow\left|\frac{\tilde{x}-x}{x}\right|\leq\epsilon$$

Let us consider an example in decimal arithmetic, that is $\beta=10$β=10, and with a 3-digit mantissa: $t=3$t=3. The number $x=.1256$x=.1256 has a representation that depends on whether we round or
truncate: $x\sim_{\text{round}}=.126$x~round=.126, $x\sim_{\text{truncate}}=.125$x~truncate=.125. The error is in the 4th digit: if $\epsilon=x-x\sim$ϵ=x−x~ then $|\epsilon|<\beta^{t}$|ϵ|<βt

Exercise. The number in this example had no exponent part. What are the error and relative error if there had been one?

Correct rounding
The IEEE 754 standard, mentioned in section 3.2.4, does not only declare the way a floating point number is stored, it also gives a standard for the accuracy of operations such as addition, subtraction, multiplication, division. The model for arithmetic in the standard is that of *correct rounding*: the result of an operation should be as if the following procedure is followed:

- The exact result of the operation is computed, whether this is representable or not;
- This result is then rounded to the nearest computer number.

In short: the representation of the result of an operation is the rounded exact result of that operation. (Of course, after two operations it no longer needs to hold that the computed result is the exact rounded version of the exact result.)

If this statement sounds trivial or self-evident, consider subtraction as an example. In a decimal number system with two digits in the mantissa, the computation $.10-.94\cdot10^{-1}=.10-.094=.006=.06\cdot10^{-2}$.10−.94·10−1=.10−.094=.006=.06·10−2. Note that in an intermediate step the mantissa .094 appears, which has one more digit than the two we declared for our number system. The extra digit is called a *guard digit*.

Without a guard digit, this operation would have proceeded as $.10-.94\cdot10^{-1}$.10−.94·10−1, where $.94\cdot10^{-1}$.94·10−1 would be rounded to .09, giving a final result of .01, which is almost double the correct result.

Exercise 3.5. Consider the computation $.10-.95\cdot10^{-1}$.10−.95·10−1, and assume again that numbers are rounded to fit the 2-digit mantissa. Why is this computation in a way a lot worse than the example?

One guard digit is not enough to guarantee correct rounding. An analysis that we will not reproduce here shows that three extra bits are needed.

Addition

Addition of two floating point numbers is done in a couple of steps. First the exponents are aligned: the smaller of the two numbers is written to have the same exponent as the larger number. Then the mantissas are added. Finally, the result is adjusted so that it again is a normalized number.

As an example, consider $.100+.200\times10^{-2}$.100+.200×10−2. Aligning the exponents, this becomes $.100+.002=.102$.100+.002=.102, and this result requires no final adjustment. We note that this computation was exact, but the sum $.100+.255\times10^{-2}$.100+.255×10−2 has the same result, and here the computation is clearly not exact. The error is $|.10255-.102|<10^{-3}$|.10255−.102|<10−3, and we note that the mantissa has 3 digits, so there clearly is a relation with the machine precision here.

In the example $.615\times10^{1}+.398\times10^{1}=1.013\times10^{2}=.101\times10^{1}$.615×101+.398×101=1.013×102=.101×101 we see that after addition of the mantissas an adjustment of the exponent is needed. The error again comes from truncating or rounding the first digit of the result that does not fit in the mantissa: if xx is the true sum and $\tilde{x}$x~ the computed sum, then $\tilde{x}=x(1+\epsilon)$x~=x(1+ϵ) where, with a 3-digit mantissa $|\epsilon|<10^{-3}$|ϵ|<10−3.

Formally, let us consider the computation of $s=x_1+x_2$s=x1+x2, and we assume that the numbers $x_i$xi are represented as $\tilde{x}_i(1+\epsilon_i)$x~i(1+ϵi). Then the sum ss is represented as

$$\tilde{s}=(\tilde{x}_1+\tilde{x}_2)(1+\epsilon_3=x_1(1+\epsilon_1)(1+\epsilon_3)+x_2(1+\epsilon_2)(1+\epsilon_3)\approx x_1(1+\epsilon_1+\epsilon_3)+x_2(1+\epsilon_1+\epsilon_3)\approx s(1+2\epsilon)$$
s~=(x~1+x~2)(1+ϵ3=x1(1+ϵ1)(1+ϵ3)+x2(1+ϵ2)(1+ϵ3)≈x1(1+ϵ1+ϵ3)+x2(1+ϵ1+ϵ3)≈s(1+2ϵ)

sign under the assumptions that all ϵ_iϵi are small and of roughly equal size, and that both $x_i>0$xi>0. We see that the relative errors are added under addition.

Multiplication

Floating point multiplication, like addition, involves several steps. In order to multiply two numbers $.m_1\times\beta^{e_1}$.m1×βe1 and $.m_2\times\beta^{e_2}$.m2×βe2, the following steps are needed.

- The exponents are added: $e\leftarrow e_1+e_2$e←e1+e2.
- The mantissas are multiplied: $m\leftarrow m_1\times m_2$m←m1×m2.
- The mantissa is normalized, and the exponent adjusted accordingly.

For example: $.123\cdot10^{0}\times.567\cdot10^{1}=.069741\cdot10^{1}\rightarrow.69741\cdot10^{0}\rightarrow.697\cdot10^{0}$.123·100×.567·101=.069741·101→.69741·100→.697·100.

What happens with relative errors?

Subtraction

Subtraction behaves very differently from addition. Whereas in addition errors are added, giving only a gradual increase of overall roundoff error, subtraction has the potential for greatly increased error in a single operation.

For example, consider subtraction with 3 digits to the mantissa: $.124-.123=.001\rightarrow.100\cdot10^{-2}$.124−.123=.001→.100·10−2. While the resultis exact, it has only one significant digit4. To see this, consider the case where the first operand .124 is actually the rounded result of a computation that should have resulted in .1235. In that case, the result of the subtraction should have been $.050\cdot10^{-2}$.050·10−2, that is, there is a 100% error, even though the relative error of the inputs was as small as could be expected. Clearly, subsequent operations involving the result of this subtraction will also be inaccurate. We conclude that subtracting almost equal numbers is a likely cause of numerical roundoff.

There are some subtleties about this example. Subtraction of almost equal numbers is exact, and we have the correct rounding behaviour of IEEE arithmetic. Still, the correctness of a single operation does not imply that a sequence of operations containing it will be accurate. While the addition example showed only modest decrease of numerical accuracy, the cancellation in this example can have disastrous effects.

Examples

From the above, the reader may got the impression that roundoff errors only lead to serious problems in exceptional circumstances. In this section we will discuss some very practical examples where the inexactness of computer arithmetic becomes visible in the result of a computation. These will be fairly simple examples; more complicated examples exist that are outside the scope of this book, such as the instability of matrix inversion.

The 'abc-formula'

As a practical example, consider the quadratic equation $ax^2+bx+c=0$ax2+bx+c=0 which has solutions $x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$x=−b±b2−4ac2a. Suppose $b>0$b>0 and $b^2\gg4ac$b2≫4ac and the '+' solution will be inaccurate. In this case it is better to compute $x_-=\frac{-b-\sqrt{b^2-4ac}}{2a}$x−=−b−b2−4ac2a and use $x_+\cdot x_-=c/a$x+·x−=c/a.

Summing series

The previous example was about preventing a large roundoff error in a single operation. This example shows that even gradual buildup of roundoff error can be handled in different ways.

Consider the sum $\sum_{n=1}^{10000} 1/n^2 = 1.644834$ and assume we are working with single precision, which on most computers means a machine precision of 10^{-7}. The problem with this example is that both the ratio between terms, and the ratio of terms to partial sums, is ever increasing. In section 3.2.3 we observed that a too large ratio can lead to one operand of an addition in effect being ignored.

If we sum the series in the sequence it is given, we observe that the first term is 1, so all partial sums $\sum_{n=1}^{N}$ where N < 10000 are at least 1. This means that any term where $1/n^2 < 10^{-7}$ gets ignored since it is less than the machine precision. Specifically, the last 7000 terms are ignored, and the computed sum is 1.644725. The first 4 digits are correct.

However, if we evaluate the sum in reverse order we obtain the exact result in single precision. We are still adding small quantities to larger ones, but now the ratio will never be as bad as one-to- , so the smaller number is never ignored. To see this, consider the ratio of two terms subsequent terms:

$$n^2/(n-1)^2 = n^2/n^2 - 2n + 1 = 1/1 - 2/n + 1/n^2 \approx 1 + 2/n$$

Since we only sum 10^5 terms and the machine precision is 10^{-7}, in the addition $1/n^2 + 1/(n-1)^2$ the second term will not be wholly ignored as it is when we sum from large to small.

Exercise. There is still a step missing in our reasoning. We have shown that in adding two subsequent terms, the smaller one is not ignored. However, during the calculation we add partial sums to the next term in the sequence. Show that this does not worsen the situation.

The lesson here is that series that are monotone (or close to monotone) should be summed from small to large, since the error is minimized if the quantities to be added are closer in magnitude. Note that this is the opposite strategy from the case of subtraction, where operations involving similar quantities lead to larger errors. This implies that if an application asks for adding and subtracting series of numbers, and we know a priori which terms are positive and negative, it may pay off to rearrange the algorithm accordingly.

Unstable algorithms

We will now consider an example where we can give a direct argument that the algorithm can not cope with problems due to inexactly represented real numbers.

Consider the recurrence $y_n = \int_0^1 \frac{x^n}{x-5}dx = \frac{1}{n} - 5y_{n-1}$ $y_n = \int_0^1 \frac{x^n}{x-5}dx = \frac{1}{n} - 5y_{n-1}$. This is easily seen to be monotonically decreasing; the first 0 x 5 n term can be computed as $y_0 = \ln 6 - \ln 5$ $y_0 = \ln 6 - \ln 5$. Performing the computation in 3 decimal digits we get:

computation		correct result
$y_0 = \ln 6 - \ln 5 = .182\|322 \times 10^1 \ldots$		1.82
$y_1 = .900 \times 10^{-1}$		.884
$y_2 = .500 \times 10^{-1}$		.0580
$y_3 = .830 \times 10^{-1}$	going up?	.0431
$y_4 = -.165$	negative?	.0343

We see that the computed results are quickly not just inaccurate, but actually nonsensical. We can analyze why this is the case.

If we define the error ϵ_n ϵ_n in the *n*-th step as

$\tilde{y}_n - y_n = \epsilon_n$ $\tilde{y}_n - y_n = \epsilon_n$

then

$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1}$ $\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1}$

so $\epsilon_n \geq 5\epsilon_{n-1}$ $\epsilon_n \geq 5\epsilon_{n-1}$. The error made by this computation shows exponential growth.

Linear system solving

Sometimes we can make statements about the numerical precision of a problem even without specifying what algorithm we use. Suppose we want to solve a linear system, that is, we have an $n \times n$ $n \times n$ matrix *A* and a vector *b* of size *n*, and we want to compute the vector *x* such that *Ax* = *b*. (We will actually considering algorithms for this in chapter 5.) Since the vector $\tilde{b}$ $\tilde{b}$ will the result of some computation or measurement, we are actually dealing with a vector $\tilde{b}$ $\tilde{b}$, which is some perturbation of the ideal *b*:

$\tilde{b} = b + \Delta b$ $\tilde{b} = b + \Delta b$

The perturbation vector ΔbΔb can be of the order of the machine precision if it only arises from representation error,or it can be larger, depending on the calculations that produced $b\sim$b~.

We now ask what the relation is between the exact value of *x*, which we would have obtained from doing an exact calculation with *A* and *b*, which is clearly impossible, and the computed value $b\sim$b~, which we get from computing with A and $b\sim$b~. (In this discussion we will assume that *A* itself is exact, but this is a simplification.)

Writing $x\sim=x+\Delta x$x~=x+Δx, the result of our computation is now

$Ax\sim=b\sim$Ax~=b~

or

$A(x+\Delta x)=b+\Delta b$A(x+Δx)=b+Δb

Since *Ax = b*, we get $A\Delta x=\Delta b$AΔx=Δb. From this, we get

$$\left\{ \begin{array}{ll} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{rl} \|A\|\|x\| & \geq \|b\| \\ \|\Delta x\| & \leq \|A^{-1}\|\|\Delta b\| \end{array} \right. \Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta b\|}{\|b\|}$$

The quantity $\|A\|\|A^{-1}\|$‖A‖‖A−1‖ is called the *condition number* of a matrix. The bound then says that any perturbation in the right hand side can lead to a perturbation in the solution that is at most larger by the condition number of the matrix *A*. Note that it does not say that the perturbation in *x* needs to be anywhere close to that size, but we can not rule it out, and in some cases it indeed happens that this bound is attained.

Suppose that *b* is exact up to machine precision, and the condition number of *A* is $10^4$104. The bound is often interpreted as saying that the last 4 digits of x are unreliable, or that the computation 'loses 4 digits of accuracy'.

Roundoff error in parallel computations

From the above example of summing a series we saw that addition in computer arithmetic is not associative. A similar fact holds for multiplication. This has an interesting consequence for parallel computations: the way a computation is spread over parallel processors influences the result. For instance, consider computing the sum of a large number *N* of terms. With *P* processors at our disposition, we can let each compute *N/P* terms, and combine the partial results. We immediately see that for no two values of *P* will the results be identical. This means that reproducibility of results in a parallel context is elusive.

More about floating point arithmetic

Programming languages

Different languages have different approaches to storing integers and floating point numbers. In Fortran it is possible to specify the number of bytes that a number takes up: INTEGER*2, REAL*8. Often it is possible to write a code using only INTEGER, REAL, and use compiler flags to indicate the size of an integer and real number.

In C, the type identifiers have no standard length. For integers there is short int, int, long int, and for floating point float, double. The sizeof() operator gives the number of bytes used to store a datatype.C99, Fortran2003 Recent standards of the C and Fortran languages incorporate the C/Fortran interoperability stan-dard, which can be used to declare a type in one language so that it is compatible with a certain type in the other language.

Other computer arithmetic systems

Other systems have been proposed to dealing with the problems of inexact arithmetic on computers. One solution is extended precision arithmetic, where numbers are stored in more bits than usual. A common use of this is in the calculation of inner products of vectors: the accumulation is internally performed in extended precision, but returned as a regular floating point number. Alternatively, there are libraries such as GMPlib that allow for any calculation to be performed in higher precision.

Another solution to the imprecisions of computer arithmetic is 'interval arithmetic', where for each calculation interval bounds are maintained. While this has been researched for considerable time, it is not practically used other than through specialized libraries.

Fixed-point arithmetic

A fixed-point number can be represented as $(N, F\}$⟨N, F} where $N \geq \beta_0$N≥β0 is the integer part and $F < 1$ is the fractional part.

Another way of looking at this, is that a fixed-point number is an integer stored in $N + F$ digits, with an implied decimal point after the first N digits.

Fixed-point calculations can overflow, with no possibility to adjust an exponent. Consider the multiplication$\langle N_1, F_1\rangle \times \langle N_2, F_2\rangle$⟨N1,F1⟩×⟨N2,F2⟩,
where $N_1 \geq \beta_{n_1}$N1≥βn1 and $N_2 \geq \beta_{n_2}$N2≥βn2 . This overflows if $n_1 + n_2$n1+n2 is more than the number of positions available for the integer part. (Informally, the number of digits of the product is the sum of the digits of the operands.) This means that, in a program that uses fixed-point, numbers will need to have a number of zero digits, if you are ever going to multiply them, which lowers the numerical accuracy. It also means that the

programmer has to think harder about calculations, arranging them in such a way that overflow will not occur, and that numerical accuracy is still preserved to a reasonable extent.

So why would people use fixed-point numbers? One important application is in embedded low-power devices, think a battery-powered digital thermometer. Since fixed-point calculations are essentially identical to integer cal-culations, they do not require a floating-point unit, thereby lowering chip size and lessening power demands. Also, many early video game systems had a processor that either had no floating-point unit, or where the integer unit was considerably faster than the floating-point unit. In both cases, implementing non-integer calculations as fixed-point, using the integer unit, was the key to high throughput.

Another area where fixed point arithmetic is still used, is in signal processing. In modern CPUs, integer and floating point operations are of essentially the same speed, but converting between them is relatively slow. Now, if the sine function is implemented through table lookup, this means that in sin(sin *x*) the output of a function is used to index the next function application. Obviously, outputting the sine function in fixed point obviates the need for conversion between real and integer quantities, which simplifies the chip logic needed, and speeds up calculations.

Complex numbers

Some programming languages have complex numbers as a native data type, others not, and others are in between. For instance, in Fortran you can declare

```
COMPLEX z1,z2, z(32)

COMPLEX*16 zz1, zz2, zz(36)
```

A complex number is a pair of real numbers, the real and imaginary part, allocated adjacent in memory. The first declaration then uses 8 bytes to store to REAL*4 numbers, the second one has REAL*8s for the real and imaginary part. (Alternatively, use DOUBLE COMPLEX or in Fortran90 COMPLEX(KIND=2) for the second line.)

By contrast, the C language does not natively have complex numbers, but both C99 and C++ have a complex.h header file5. This defines as complex number as in Fortran, as two real numbers.

Storing a complex number like this is easy, but sometimes it is computationally not the best solution. This becomes apparent when we look at arrays of complex numbers. If a computation often relies on access to the real (or imagi-nary) parts of complex numbers exclusively, striding through an array of complex numbers, has a stride two, which is disadvantageous (see section 1.2.4.3). In this case, it is better to allocate one array for the real parts, and another for the imaginary parts.

Exercise. Suppose arrays of complex numbers are stored the Fortran way. Analyze the memory access pattern of pairwise multiplying the arrays, that is, $\forall i: c_i \leftarrow a_i \cdot b_i$∀i:ci←ai·bi where a(), b(), c() are arrays of complex numbers.

Exercise. Show that an $n \times n$n×n linear system *Ax* = *b* over the complex numbers can be written as a $2n \times 2n$2n×2n system over the real numbers. Hint: split the matrix and the vectors in their real and imaginary parts. Argue for the efficiency of storing arrays of complex numbers as separate arrays for the real and imaginary parts.

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

4.5: Division

Division

Read this article to get the basics of division. Again one can divide by an integer by shifting right. Each shift is a divide by 2. Note also that using the add and shift hardware or using an algorithm to do the same thing is slower that using a hardware divider.

A **division algorithm** is an algorithm which, given two integers N and D, computes their quotient and/or remainder, the result of Euclidean division. Some are applied by hand, while others are employed by digital circuit designs and software.

Division algorithms fall into two main categories: slow division and fast division. Slow division algorithms produce one digit of the final quotient per iteration. Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division. Fast division methods start with a close approximation to the final quotient and produce twice as many digits of the final quotient on each iteration. Newton–Raphson and Goldschmidt algorithms fall into this category.

Variants of these algorithms allow using fast multiplication algorithms. It results that, for large integers, the computer time needed for a division is the same, up to a constant factor, as the time needed for a multiplication, whichever multiplication algorithm is used.

Discussion will refer to the form $N/D = (Q,R)$N/D=(Q,R), where

- *N* = Numerator (dividend)
- *D* = Denominator (divisor)

is the input, and

- *Q* = Quotient
- *R* = Remainder

is the output.

Division by repeated subtraction

The simplest division algorithm, historically incorporated into a greatest common divisor algorithm presented in Euclid's *Elements*, Book VII, Proposition 1, finds the remainder given two positive integers using only subtractions and comparisons:

```
while N ≥ D do
  N := N − D
end
return N
```

The proof that the quotient and remainder exist and are unique (described at Euclidean division) gives rise to a complete division algorithm using additions, subtractions, and comparisons:

```
function divide(N, D)
  if D = 0 then error(DivisionByZero) end
  if D < 0 then (Q, R) := divide(N, -D); return (-Q, R) end
  if N < 0 then
    (Q,R) := divide(-N, D)
    if R = 0 then return (-Q, 0)
    else return (-Q - 1, D - R) end
  end
  -- At this point, N ≥ 0 and D > 0
  return divide_unsigned(N, D)
end
function divide_unsigned(N, D)
  Q := 0; R := N
  while R ≥ D do
    Q := Q + 1
    R := R - D
  end
  return (Q, R)
end
```

This procedure always produces R ≥ 0. Although very simple, it takes Ω(Q) steps, and so is exponentially slower than even slow division algorithms like long division. It is useful if Q is known to be small (being an output-sensitive algorithm), and can serve as an executable specification.

Long division

Long division is the standard algorithm used for pen-and-paper division of multi-digit numbers expressed in decimal notation. It shifts gradually from the left to the right end of the dividend, subtracting the largest possible multiple of the divisor (at the digit level) at each stage; the multiples then become the digits of the quotient, and the final difference is then the remainder.

When used with a binary radix, this method forms the basis for the (unsigned) integer division with remainder algorithm below. Short division is an abbreviated form of long division suitable for one-digit divisors. Chunking – also known as the partial quotients method or the hangman method – is a less-efficient form of long division which may be

easier to understand. By allowing one to subtract more multiples than what one currently has at each stage, a more freeform variant of long division can be developed as well.

Integer division (unsigned) with remainder

The following algorithm, the binary version of the famous long division , will divide *N* by *D*, placing the quotient in *Q* and the remainder in *R*. In the following code, all values are treated as unsigned integers.

```
if D = 0 then error(DivisionByZeroException) end
Q := 0                  -- Initialize quotient and remainder to zero
R := 0
for i := n − 1 .. 0 do  -- Where n is number of bits in N
  R := R << 1           -- Left-shift R by 1 bit
  R(0) := N(i)          -- Set the least-significant bit of R equal to bit i of the numerator
  if R ≥ D then
    R := R − D
    Q(i) := 1
  end
end
```

Example

If we take $N=1100_2$ (12_{10}) and $D=100_2$ (4_{10})

Step 1: Set R=0 and Q=0
Step 2: Take i=3 (one less than the number of bits in N)
Step 3: R=00 (left shifted by 1)
Step 4: R=01 (setting R(0) to N(i))
Step 5: R<D, so skip statement

Step 2: Set i=2
Step 3: R=010
Step 4: R=011
Step 5: R<D, statement skipped

Step 2: Set i=1
Step 3: R=0110
Step 4: R=0110
Step 5: R>=D, statement entered
Step 5b: R=10 (R−D)
Step 5c: Q=10 (setting Q(i) to 1)

Step 2: Set i=0
Step 3: R=100
Step 4: R=100
Step 5: R>=D, statement entered

Step 5b: R=0 (R−D)
Step 5c: Q=11 (setting Q(i) to 1)

end
$Q=11_2$ (3_{10}) and R=0.

Slow division methods
Slow division methods are all based on a standard recurrence equation

$$R_{j+1}=B\times R_j - q_{n-(j+1)}\times D \quad Rj+1=B\times Rj-qn-(j+1)\times D\ ,$$

where:

- R_j is the *j*-th partial remainder of the division
- *B* is the radix (base, usually 2 internally in computers and calculators)
- $q_{n-(j+1)}$ is the digit of the quotient in position $n-(j+1)$, where the digit positions are numbered from least-significant 0 to most significant $n-1$
- *n* is number of digits in the quotient
- *D* is the divisor

Restoring division
Restoring division operates on fixed-point fractional numbers and depends on the assumption $0 < D < N$.

The quotient digits *q* are formed from the digit set {0,1}.

The basic algorithm for binary (radix 2) restoring division is:

```
R := N
D := D << n            -- R and D need twice the word width of N and Q
for i := n − 1 .. 0 do  -- For example 31..0 for 32 bits
  R := 2 * R − D          -- Trial subtraction from shifted value (multiplication
by 2 is a shift in binary representation)
  if R ≥ 0 then
    q(i) := 1          -- Result-bit 1
  else
    q(i) := 0          -- Result-bit 0
    R := R + D         -- New partial remainder is (restored) shifted value
  end
end

-- Where: N = Numerator, D = Denominator, n = #bits, R = Partial remainder, q(i) =
bit #i of quotient
```

The above restoring division algorithm can avoid the restoring step by saving the shifted value $2R$ before the subtraction in an additional register *T* (i.e., $T = R << 1$) and copying register *T* to *R* when the result of the subtraction $2R - D$ is negative.

Non-performing restoring division is similar to restoring division except that the value of 2R is saved, so *D* does not need to be added back in for the case of R < 0.

Non-restoring division

Non-restoring division uses the digit set {−1, 1} for the quotient digits instead of {0, 1}. The algorithm is more complex, but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction, which potentially cuts down the numbers of operations by up to half and lets it be executed faster. The basic algorithm for binary (radix 2) non-restoring division of non-negative numbers is:

```
R := N
D := D << n            -- R and D need twice the word width of N and Q
for i = n − 1 .. 0 do  -- for example 31..0 for 32 bits
  if R >= 0 then
    q[i] := +1
    R := 2 * R − D
  else
    q[i] := −1
    R := 2 * R + D
  end if
end

-- Note: N=Numerator, D=Denominator, n=#bits, R=Partial remainder, q(i)=bit #i of
quotient.
```

Following this algorithm, the quotient is in a non-standard form consisting of digits of −1 and +1. This form needs to be converted to binary to form the final quotient. Example:

Convert the following quotient to the digit set {0,1}:

Start: $Q=1111^{-}11^{-}11^{-}$Q=1111-11-11-

1. Form the positive term: $P=11101010$P=11101010

2. Mask the negative term*: $M=00010101$M=00010101

3. Subtract: $P-M$P−M : $Q=11010101$Q=11010101

*.(Signed binary notation with One's complement without Two's Complement)

If the −1 digits of QQ are stored as zeros (0) as is common, then PP **is** QQ and computing MM is trivial: perform a one's complement (bit by bit complement) on the original QQ.

```
Q := Q − bit.bnot(Q)      * Appropriate if −1 Digits in Q are Represented as zeros
as is common.
```

Finally, quotients computed by this algorithm are always odd, and the remainder in R is in the range −D ≤ R < D. For example, 5 / 2 = 3 R −1. To convert to a positive remainder, do a single restoring step *after* Q is converted from non-standard form to standard form:

```
if R < 0 then
   Q := Q - 1
   R := R + D  -- Needed only if the Remainder is of interest.
end if
```

The actual remainder is R >> n. (As with restoring division, the low-order bits of R are used up at the same rate as bits of the quotient Q are produced, and it is common to use a single shift register for both.)

SRT division

Named for its creators (Sweeney, Robertson, and Tocher), SRT division is a popular method for division in many microprocessor implementations. SRT division is similar to non-restoring division, but it uses a lookup table based on the dividend and the divisor to determine each quotient digit.

The most significant difference is that a *redundant representation* is used for the quotient. For example, when implementing radix-4 SRT division, each quotient digit is chosen from *five* possibilities: { −2, −1, 0, +1, +2 }. Because of this, the choice of a quotient digit need not be perfect; later quotient digits can correct for slight errors. (For example, the quotient digit pairs (0, +2) and (1, −2) are equivalent, since 0×4+2 = 1×4−2.) This tolerance allows quotient digits to be selected using only a few most-significant bits of the dividend and divisor, rather than requiring a full-width subtraction. This simplification in turn allows a radix higher than 2 to be used.

Like non-restoring division, the final steps are a final full-width subtraction to resolve the last quotient bit, and conversion of the quotient to standard binary form.

The Intel Pentium processor's infamous floating-point division bug was caused by an incorrectly coded lookup table. Five of the 1066 entries had been mistakenly omitted.

Fast division methods

Newton–Raphson division

Newton–Raphson uses Newton's method to find the reciprocal of DD and multiply that reciprocal by NN to find the final quotient QQ

The steps of Newton–Raphson division are:

1. Calculate an estimate X_0X0 for the reciprocal $1/D$1/D of the divisor DD.
2. Compute successively more accurate estimates $X_1, X_2, \ldots, X_S$X1,X2,…,XS of the reciprocal. This is where one employs the Newton–Raphson method as such.
3. Compute the quotient by multiplying the dividend by the reciprocal of the divisor: $Q = NX_S$Q=NXS.

In order to apply Newton's method to find the reciprocal of DD, it is necessary to find a function $f(X)$f(X) that has a zero at $X=1/D$X=1/D. The obvious such function is $f(X)=DX-1$f(X)=DX−1 , but the Newton–Raphson iteration for this is unhelpful, since it cannot be computed without already knowing the reciprocal of DD (moreover it attempts to compute the exact reciprocal in one step, rather than allow for iterative improvements). A function that does work is $f(X)=(1/X)-D$f(X)=(1/X)−D, for which the Newton–Raphson iteration gives

$$X_{i+1}=X_i-\frac{f(X_i)}{f'(X_i)}=X_i-\frac{1/X_i-D}{-1/X_i^2}=X_i+X_i(1-DX_i)=X_i(2-DX_i)$$Xi+1=Xi−f(Xi)f′(Xi)
=Xi−1/Xi−D−1/Xi2=Xi+Xi(1−DXi)=Xi(2−DXi),

which can be calculated from X_iXi using only multiplication and subtraction, or using two fused multiply–adds.

From a computation point of view, the expressions $X_{i+1}=X_i+X_i(1-DX_i)$Xi+1=Xi+Xi(1−DXi) and $X_{i+1}=X_i(2-DX_i)$Xi+1=Xi(2−DXi) are not equivalent. To obtain a result with a precision of 2*n* bits while making use of the second expression, one must compute the product between X_iXi and $(2-DX_i)$(2−DXi) with double the given precision of X_iXi (*n* bits). In contrast, the product between X_iXi and $(1-DX_i)$(1−DXi) need only be computed with a precision of *n* bits, because the leading *n* bits (after the binary point) of $(1-DX_i)$(1−DXi) are zeros.

If the error is defined as $\varepsilon_i=1-DX_i\varepsilon i=1-DXi$, then:

$$\varepsilon_{i+1}=1-DX_{i+1}=1-D(X_i(2-DX_i))=1-2DX_i+D^2X_i^2=(1-DX_i)^2=\varepsilon_i^2.\varepsilon i+1=1-DXi$$
+1=1−D(Xi(2−DXi))=1−2DXi+D2Xi2=(1−DXi)2=εi2.

This squaring of the error at each iteration step – the so-called quadratic convergence of Newton–Raphson's method – has the effect that the number of correct digits in the result roughly *doubles for every iteration*, a property that becomes extremely valuable when the numbers involved have many digits (e.g. in the large integer domain). But it also means that the initial convergence of the method can be comparatively slow, especially if the initial estimate X_0X0 is poorly chosen.

For the subproblem of choosing an initial estimate X_0X0, it is convenient to apply a bit-shift to the divisor *D* to scale it so that 0.5 ≤ *D* ≤ 1; by applying the same bit-shift to the numerator *N*, one ensures the quotient does not change. Then one could use a linear approximation in the form

$$X_{0}=T_{1}+T_{2}D\approx {\frac {1}{D}}$$X_{0}=T_{1}+T_{2}D\approx {\frac {1}{D}}

to initialize Newton–Raphson. To minimize the maximum of the absolute value of the error of this approximation on interval $[0.5,1]$[0.5,1], one should use

$X_0 = 48/17 - 32/17 D$\].</dd></dl><p>Thecoefficientsofthelinearapproximationaredeterminedasfollows.Theabsolutevalueoftheerroris\($|\varepsilon_0| = |1 - D(T_1 + T_2 D)|X0 = 4817 - 3217D$\].</dd> </dl> <p>Thecoefficientsofthelinearapproximationaredeterminedasfollows.Theabsolutevalueoftheerroris\($|\varepsilon 0| = |1 - D(T1 + T2D)|$. The minimum of the maximum absolute value of the error is determined by the Chebyshev equioscillation theorem applied to $F(D) = 1 - D(T_1 + T_2 D)$F(D)=1−D(T1+T2D). The local minimum of $F(D)$F(D) occurs when $F'(D) = 0$F′(D)=0, which has solution $D = -T_1/(2T_2)$D=−T1/(2T2). The function at that minimum must be of opposite sign as the function at the endpoints, namely, $F(1/2) = F(1) = -F(-T_1/(2T_2))$F(1/2)=F(1)=−F(−T1/(2T2)). The two equations in the two unknowns have a unique solution $T_1 = 48/17$T1=48/17 and $T_2 = -32/17$T2=−32/17, and the maximum error is $F(1) = 1/17$F(1)=1/17. Using this approximation, the absolute value of the error of the initial value is less than

$|\varepsilon_0| \leq 117 \approx 0.059$|ε0|≤117≈0.059.

It is possible to generate a polynomial fit of degree larger than 1, computing the coefficients using the Remez algorithm. The trade-off is that the initial guess requires more computational cycles but hopefully in exchange for fewer iterations of Newton–Raphson.

Since for this method the convergence is exactly quadratic, it follows that

$S = \lceil \log_2 2P + 1 \log_2 17 \rceil$S=⌈log2P+1log217⌉

steps are enough to calculate the value up to PP, binary places. This evaluates to 3 for IEEE single precision and 4 for both double precision and double extended formats.

Pseudocode

The following computes the quotient of *N* and *D* with a precision of *P* binary places:

```
Express D as M × 2^e where 1 ≤ M < 2 (standard floating point representation)
D' := D / 2^(e+1)   // scale between 0.5 and 1, can be performed with bit shift / exponent subtraction
N' := N / 2^(e+1)
X := 48/17 − 32/17 × D'   // precompute constants with same precision as D
repeat ⌈log2P+1log217⌉⌈log2P+1log217⌉, times   // can be precomputed based on fixed P
    X := X + X × (1 - D' × X)
end
return N' × X
```

For example, for a double-precision floating-point division, this method uses 10 multiplies, 9 adds, and 2 shifts.

Variant Newton–Raphson division

The Newton-Raphson division method can be modified to be slightly faster as follows. After shifting *N* and *D* so that *D* is in [0.5, 1.0], initialize with

$$X := 14033 + D \cdot (-6411 + D \cdot 25699)$$

This is the best quadratic fit to 1/*D* and gives an absolute value of the error less than or equal to 1/99. It is chosen to make the error equal to a re-scaled third order Chebyshev polynomial of the first kind. The coefficients should be pre-calculated and hard-coded.

Then in the loop, use an iteration which cubes the error.

$$E := 1 - D \cdot X$$

$$Y := X \cdot E$$

$$X := X + Y + Y \cdot E$$

The *Y*·*E* term is new.

If the loop is performed until X agrees with 1/*D* on its leading *P* bits, then the number of iterations will be no more than

$$\lceil \log_3 (P + 1 \log_2 99) \rceil$$

which is the number of times 99 must be cubed to get to 2^{P+1}. Then

$$Q := N \cdot X$$

is the quotient to *P* bits.

Using higher degree polynomials in either the initialization or the iteration results in a degradation of performance because the extra multiplications required would be better spent on doing more iterations.

Goldschmidt division

Goldschmidt division (after Robert Elliott Goldschmidt) uses an iterative process of repeatedly multiplying both the dividend and divisor by a common factor F_i, chosen such that the divisor converges to 1. This causes the dividend to converge to the sought quotient *Q*:

$$Q = NDF_1F_1F_2F_2F \ldots F \ldots$$

The steps for Goldschmidt division are:

1. Generate an estimate for the multiplication factor F_i .
2. Multiply the dividend and divisor by F_i .
3. If the divisor is sufficiently close to 1, return the dividend, otherwise, loop to step 1.

Assuming N/D has been scaled so that $0 < D < 1$, each F_i is based on D:

$$F_{i+1}=2-D_iF_i+1=2-D_i.$$

Multiplying the dividend and divisor by the factor yields:

$$N_{i+1}D_{i+1}=N_iD_iF_{i+1}F_{i+1}N_i+1D_i+1=N_iD_iF_i+1F_i+1.$$

After a sufficient number k of iterations $Q=N_kQ=N_k$.

The Goldschmidt method is used in AMD Athlon CPUs and later models. It is also known as Anderson Earle Goldschmidt Powers (AEGP) algorithm and is implemented by various IBM processors.

Binomial theorem

The Goldschmidt method can be used with factors that allow simplifications by the binomial theorem. Assume N/D has been scaled by a power of two such that $D\in(12,1]D\in(12,1]$. We choose $D=1-xD=1-x$ and $F_i=1+x2_iF_i=1+x2_i$. This yields

$$N1-x=N\cdot(1+x)1-x2=N\cdot(1+x)\cdot(1+x2)1-x4=\cdots=Q'=N'=N\cdot(1+x)\cdot(1+x2)\cdots(1+x2(n-1))D'=1-x2n\approx1N1-x=N\cdot(1+x)1-x2=N\cdot(1+x)\cdot(1+x2)1-x4=\cdots=Q'=N'=N\cdot(1+x)\cdot(1+x2)\cdots(1+x2(n-1))D'=1-x2n\approx1.$$

After nn steps $(x\in[0,12))(x\in[0,12))$, the denominator $1-x2_n1-x2n$ can be rounded to 1 with a relative error

$$\varepsilon n=Q'-N'Q'=x2_n\varepsilon n=Q'-N'Q'=x2n$$

which is maximum at $2-2_n2-2n$ when $x=12x=12$, thus providing a minimum precision of 2_n2n binary digits.

Large-integer methods

Methods designed for hardware implementation generally do not scale to integers with thousands or millions of decimal digits; these frequently occur, for example, in modular reductions in cryptography. For these large integers, more efficient division algorithms transform the problem to use a small number of multiplications, which can then be done using an asymptotically

efficient multiplication algorithm such as the Karatsuba algorithm, Toom–Cook multiplication or the Schönhage–Strassen algorithm. The result is that the computational complexity of the division is of the same order (up to a multiplicative constant) as that of the multiplication. Examples include reduction to multiplication by Newton's method as described above, as well as the slightly faster Barrett reduction and Montgomery reduction algorithms. Newton's method is particularly efficient in scenarios where one must divide by the same divisor many times, since after the initial Newton inversion only one (truncated) multiplication is needed for each division.

Division by a constant

The division by a constant D is equivalent to the multiplication by its reciprocal. Since the denominator is constant, so is its reciprocal $(1/D)$. Thus it is possible to compute the value of $(1/D)$ once at compile time, and at run time perform the multiplication $N\cdot(1/D)$ rather than the division N/D. In floating-point arithmetic the use of $(1/D)$ presents little problem, but in integer arithmetic the reciprocal will always evaluate to zero (assuming $|D| > 1$).

It is not necessary to use specifically $(1/D)$; any value (X/Y) that reduces to $(1/D)$ may be used. For example, for division by 3, the factors 1/3, 2/6, 3/9, or 194/582 could be used. Consequently, if Y were a power of two the division step would reduce to a fast right bit shift. The effect of calculating N/D as $(N\cdot X)/Y$ replaces a division with a multiply and a shift. Note that the parentheses are important, as $N\cdot(X/Y)$ will evaluate to zero.

However, unless D itself is a power of two, there is no X and Y that satisfies the conditions above. Fortunately, $(N\cdot X)/Y$ gives exactly the same result as N/D in integer arithmetic even when (X/Y) is not exactly equal to $1/D$, but "close enough" that the error introduced by the approximation is in the bits that are discarded by the shift operation.

As a concrete fixed-point arithmetic example, for 32-bit unsigned integers, division by 3 can be replaced with a multiply by $2863311531/2^{33}$, a multiplication by 2863311531 (hexadecimal 0xAAAAAAAB) followed by a 33 right bit shift. The value of 2863311531 is calculated as $2^{33}/3$, then rounded up.

Likewise, division by 10 can be expressed as a multiplication by 3435973837 (0xCCCCCCCD) followed by division by 2^{35} (or 35 right bit shift).

In some cases, division by a constant can be accomplished in even less time by converting the "multiply by a constant" into a series of shifts and adds or subtracts. Of particular interest is division by 10, for which the exact quotient is obtained, with remainder if required.

Rounding error
Round-off error can be introduced by division operations due to limited precision.

Source: Wikipedia, https://en.wikipedia.org/wiki/Division_algorithm

Arithmetic for Computers

This article nicely sums up what the ALU does. It includes block diagrams of the ALU, not present in the other articles.

Addition and Subtraction

1. Add (`add`), and immediate (`addi`), and subtract (`sub`) cause exceptions on overflow. MIPS detects overflow with an *exception* (or *interrupt*), which is an unscheduled procedure call. The address of current instruction is saved and the computer jumps to predefined address to invoke the appropriate routine for that exception.

   MIPS uses *exception program counter* (EPC) to contain the address of the instruction that causes the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register.

2. Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do not cause exceptions on overflow. Programmers can trap overflow anyway: when overflow occurs, the sign bit of the result is not properly set. Compairing with sign bits of operands, the sign bit of the result can be determined.

SIMD (single instruction, multiple data): By partitioning the carry chains within a 64-bit adder, a processor could perform simultaneous operations on a short vecters of eight 8-bit operands, four 16-bit operands, etc. Vectors and 8-bit data often appears in multimedia routine.

Multiplication
multiplicand * multiplier = product

Sequential Version of the Multiplication

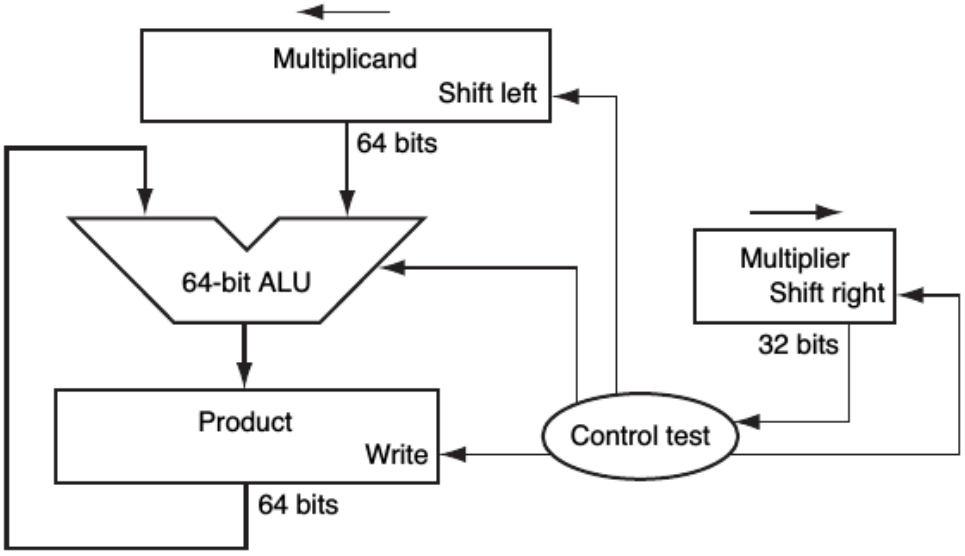


FIGURE 3.4 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix C describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

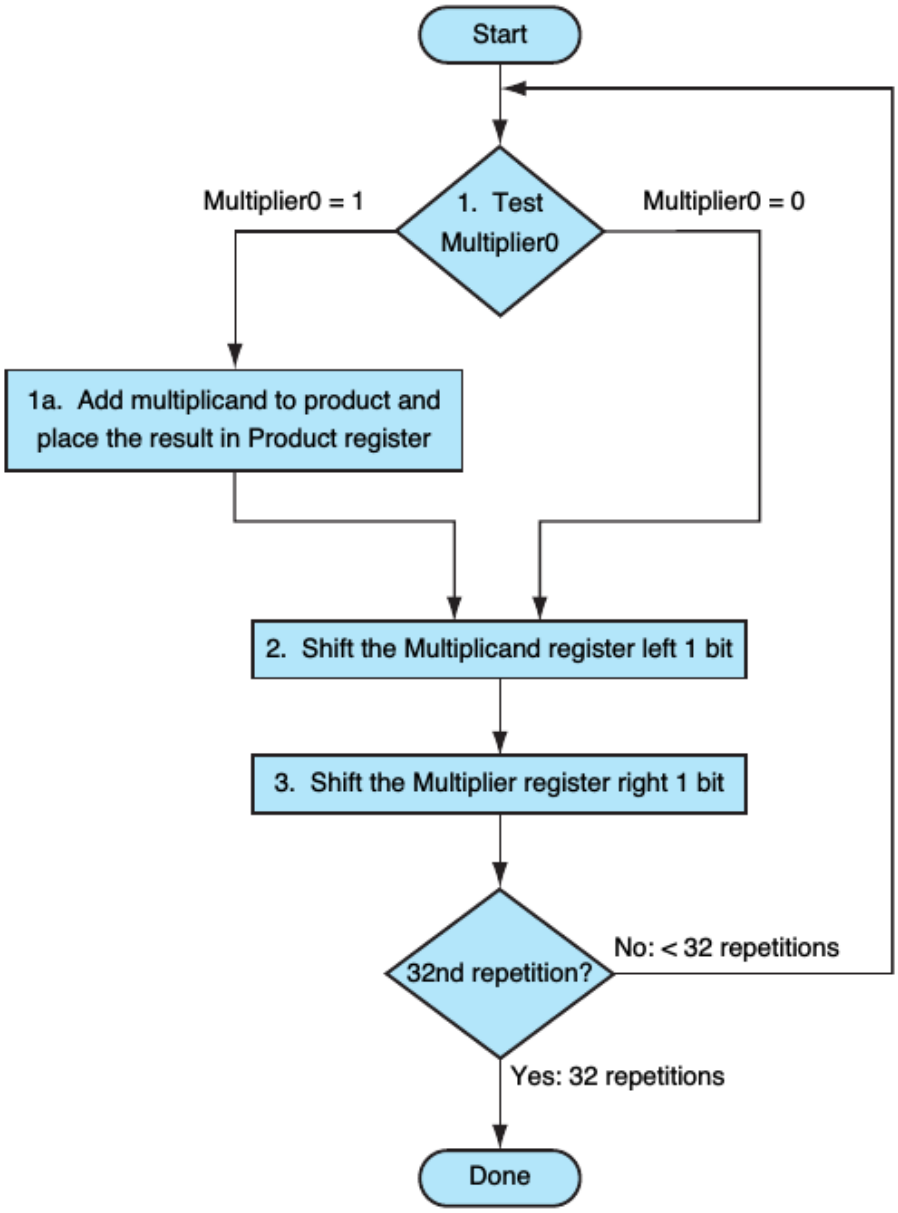


FIGURE 3.5 The first multiplication algorithm, using the hardware shown in Figure 3.4. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

Refined version:

- Init: put multiplier to the left 32-bit of the product register.
- Cycle:
    1. if the last bit of product register is 1, add the left 32-bit with the multiplicand
    2. shift right the product register
- Final: the product register contains the 64-bit product

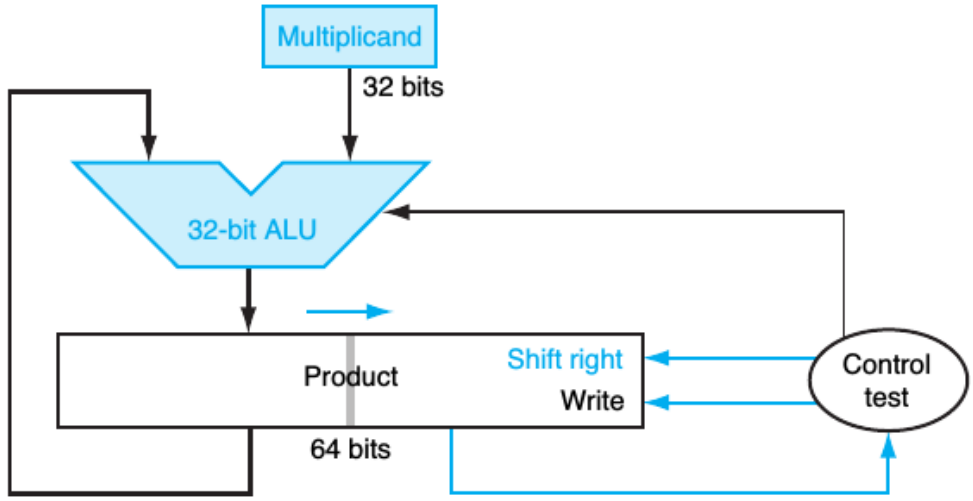


FIGURE 3.6 Refined version of the multiplication hardware. Compare with the first version in Figure 3.4. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.4.)

Faster Multiplication

A way to organize these 32 addtions is in a parallel tree:

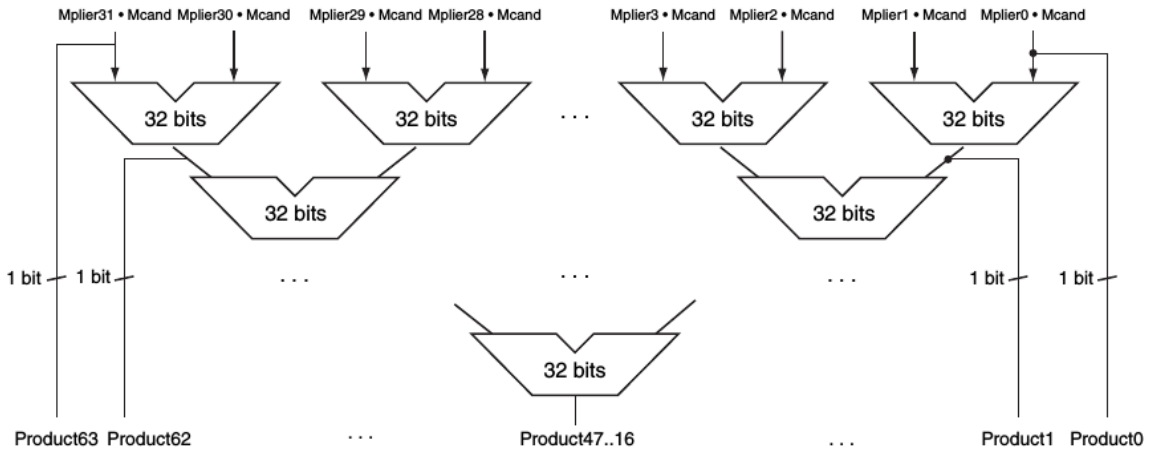


FIGURE 3.8 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay.

Multiply in MIPS

The registers `Hi` and `Lo` contains the 64-bit product. Call `mflo` to fetch the 32-bit product, `mfhi` can be used to get `Hi` to test for overflow.

Division

Dividend = Quotient * Divisor + Remainder

Division Algorithm

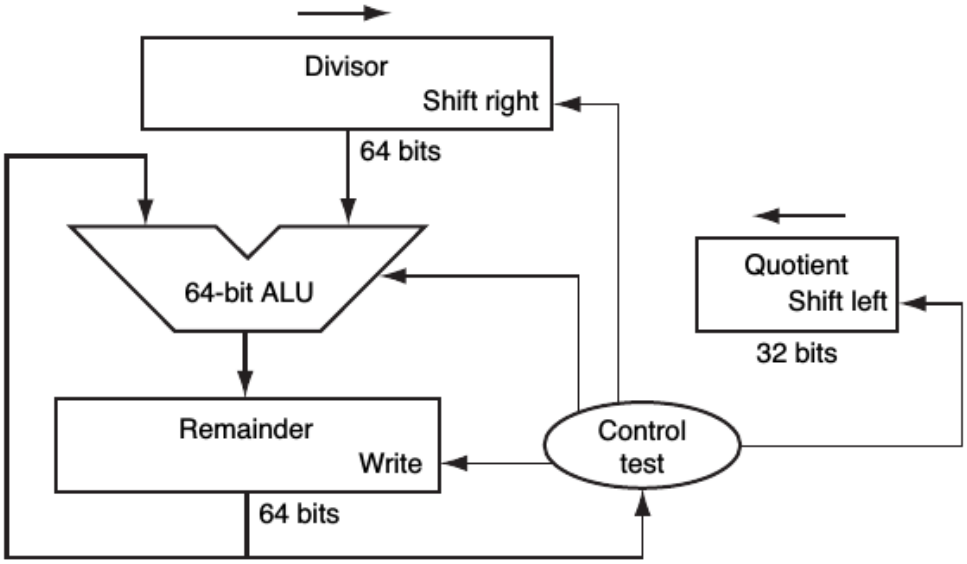


FIGURE 3.9 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

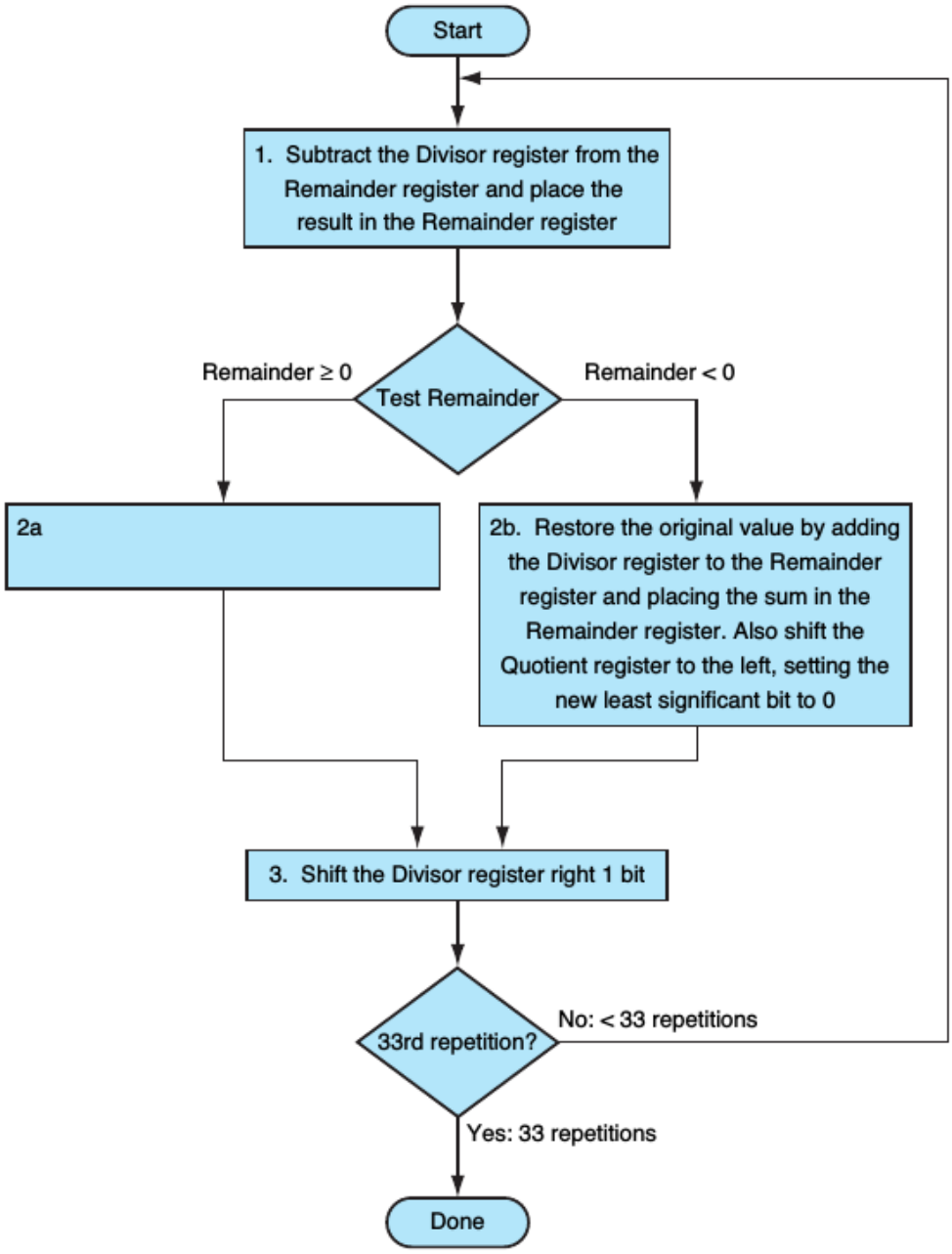


FIGURE 3.10 A division algorithm, using the hardware in Figure 3.9. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

Improved version:

- Init: put the dividend in the right 32-bit of remainder register.
- Cycle:
    1. subtract the left 32-bit of remainder by the divisor
    2. shift left the remaider register
    3. set the last bit as new quotient bit
- Final: the left 32-bit contains the remainder, right 32-bit contains the quotient.

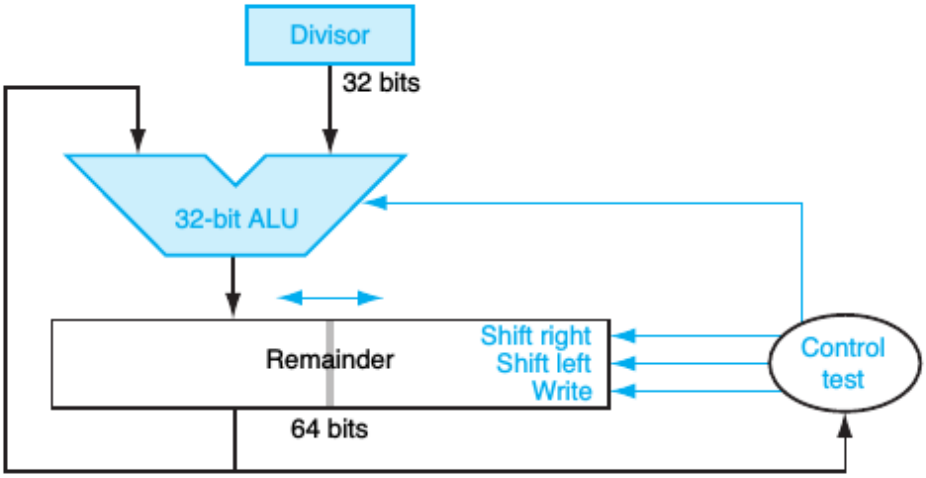


FIGURE 3.12 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.9, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.6, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

Faster Division

SRT division: try to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. The key is guessing the value to subtract.

Divide in MIPS

`Hi` contains the remainder, and `Lo` contains the quotient after the divide instruction complete.

MIPS divide instructions ignore overflow. MIPS software must check the divisor to discover division by 0 as well as overflow.

Floating Point

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

normalized A number in floating-point notation that has no leading 0s.

fraction The value, generally between 0 and 1, placed in the fraction field.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

overflow the exponent is too large to be represented in the exponent field.

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

In general, floating-point numbers are of the form: $(-1)^S \times F \times 2^E$ (−1)S×F×2E

MIPS float: sign(1 bit) + exponent(8 bit) + fraction(23 bit) MIPS double: s(1 bit) + exponent(11 bit) + fraction(52 bit)

IEEE 754 uses a bias of 127 for single precesion, and makes the leading 1 implicit. Since 0 has no leading 1, it's given the reserved exponent 0 so that hardware won't attach a leading 1.

Thus 00…00 represents 0; the representation of the rest are in the following form:

$(-1)^S \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$ (−1)S×(1+Fraction)×2(Exponent−Bias)

The exponent is located left and the bias is for comparison convenience.

Source: Harttle Land, https://harttle.land/2014/02/12/computer-design-arithmetic.html

4.6: Case Study: Floating Point Arithmetic in an x86 Processor

Extended Precision

Read this article to learn about minimizing roundoff and overflow in floating point arithmetic using extended precision.

Extended precision refers to floating point number formats that provide greater precision than the basic floating point formats. Extended precision formats support a basic format by minimizing roundoff and overflow errors in intermediate values of expressions on the base format. In contrast to *extended precision*, arbitrary-precision arithmetic refers to implementations of much larger numeric types (with a storage count that usually is not a power of two) using special software (or, rarely, hardware).

Extended precision implementations

There is a long history of extended floating-point formats reaching back nearly to the middle of the last century. Various manufacturers have used different formats for extended precision for different machines. In many cases the format of the extended precision is not quite the same as a scale-up of the ordinary single- and double-precision formats it is meant to extend. In a few cases the implementation was merely a software-based change in the floating-point data format, but in most cases extended precision was implemented in hardware, either built into the central processor itself, or more often, built into the hardware of an optional, attached processor called a "floating-point unit" (FPU) or "floating-point processor" (FPP), accessible to the CPU as a fast input / output device.

IBM extended precision formats

The IBM 1130, sold in 1965, offered two floating point formats: A 32 bit "standard precision" format and a 40 bit "extended precision" format. Standard precision format contains a 24 bit two's complement significand while extended precision utilizes a 32 bit two's complement significand. The latter format makes full use of the CPU's 32 bit integer operations. The characteristic in both formats is an 8 bit field containing the power of two biased by 128. Floating-point arithmetic operations are performed by software, and double precision is not supported at all. The extended format occupies three 16 bit words, with the extra space simply ignored.

The IBM System/360 supports a 32 bit "short" floating point format and a 64 bit "long" floating point format. The 360/85 and follow-on System/370 add support for a 128 bit "extended" format. These formats are still supported in the current design, where they are now called the "hexadecimal floating point" (HFP) formats.

Microsoft MBF extended precision format

The Microsoft BASIC port for the 6502 CPU, such as in adaptations like Commodore BASIC, AppleSoft BASIC, KIM-1 BASIC or MicroTAN BASIC, supports an extended 40 bit variant of the floating point format *Microsoft Binary Format* (MBF) since 1977.

IEEE 754 extended precision formats

The IEEE 754 floating point standard recommends that implementations provide extended precision formats. The standard specifies the minimum requirements for an extended format but does not specify an encoding. The encoding is the implementor's choice.

The IA32, x86-64, and Itanium processors support an 80 bit "double extended" extended precision format with a 64 bit significand. The Intel 8087 math coprocessor was the first x86 device which supported floating point arithmetic in hardware. It was designed to support a 32 bit "single precision" format and a 64 bit "double precision" format for encoding and interchanging floating point numbers. The temporary real (extended) format was designed not to store data at higher precision as such, but rather primarily to allow for the computation of double results more reliably and accurately by minimising overflow and roundoff-errors in intermediate calculations. For example, many floating point algorithms (e.g. exponentiation) suffer from significant precision loss when computed using the most direct implementations. To mitigate such issues the internal registers in the 8087 were designed to hold intermediate results in an 80 bit "extended precision" format. The 8087 automatically converts numbers to this format when loading floating point registers from memory and also converts results back to the more conventional formats when storing the registers back into memory. To enable

intermediate subexpression results to be saved in extended precision scratch variables and continued across programming language statements, and otherwise interrupted calculations to resume where they were interrupted, it provides instructions which transfer values between these internal registers and memory without performing any conversion, which therefore enables access to the extended format for calculations – also reviving the issue of the accuracy of functions of such numbers, but at a higher precision.

The floating-point units (FPU) on all subsequent x86 processors have supported this format. As a result software can be developed which takes advantage of the higher precision provided by this format. William Kahan, a primary designer of the x87 arithmetic and initial IEEE 754 standard proposal notes on the development of the x87 floating point: "An extended format as wide as we dared (80 bits) was included to serve the same support role as the 13 decimal internal format serves in Hewlett-Packard's 10 decimal calculators". Moreover, Kahan notes that 64 bits was the widest significand across which carry propagation could be done without increasing the cycle time on the 8087, and that the x87 extended precision was designed to be extensible to higher precision in future processors: "For now the 10-byte Extended format is a tolerable compromise between the value of extra-precise arithmetic and the price of implementing it to run fast; very soon two more bytes of precision will become tolerable, and ultimately a 16 byte format. ... That kind of gradual evolution towards wider precision was already in view when IEEE Standard 754 for Floating-Point Arithmetic was framed".

The Motorola 6888x math coprocessors and the Motorola 68040 and 68060 processors support this same 64 bit significand extended precision type (similar to the Intel format although padded to a 96 bit format with 16 unused bits inserted between the exponent and significand fields). The follow-on Coldfire processors do not support this 96 bit extended precision format.

The x87 and Motorola 68881 80 bit formats meet the requirements of the IEEE 754 double extended format, as does the IEEE 754 128 bit format.

x86 extended precision format

The x86 extended precision format is an 80-bit format first implemented in the Intel 8087 math coprocessor and is supported by all processors that are based on the x86 design that incorporate a floating-point unit (FPU). This 80-bit format uses one bit for the sign of the significand, 15 bits for the exponent field (i.e. the same range as the 128-bit quadruple precision IEEE 754 format) and 64 bits for the significand. The exponent field is biased by 16383, meaning that 16383 has to be subtracted from the value in the exponent field to compute the actual power of 2. An exponent field value of 32767 (all fifteen bits **1**) is reserved so as to enable the representation of special states such as infinity and Not a Number. If the exponent field is zero, the value is a denormal number and the exponent of 2 is −16382.

In the following table, "*s*" is the value of the sign bit (0 means positive, 1 means negative), "*e*" is the value of the exponent field interpreted as a positive integer, and "*m*" is the significand interpreted as a positive binary number where the binary point is located between bits 63 and 62. The "*m*" field is the combination of the integer and fraction parts in the above diagram.

Interpretation of the fields of an x86 Extended Precision value

Exponent	Significand		Meaning
	Bit 63	**Bits 62-0**	
All Zeros	Zero	Zero	Zero. The sign bit gives the sign of the zero.
		Non-zero	Denormal. The value is $(-1)^s \times m \times 2^{-16382}$
	One	Anything	Pseudo Denormal. The 80387 and later properly interpret this value but will not generate it. The value is $(-1)^s \times m \times 2^{-16382}$
	Bits 63,62	**Bits 61-0**	
All Ones	00	Zero	Pseudo-Infinity. The sign bit gives the sign of the infinity. The 8087 and 80287 treat this as Infinity. The 80387 and later treat this as an invalid operand.
		Non-zero	Pseudo Not a Number. The sign bit is meaningless. The 8087 and 80287 treat this as a Signaling Not a Number. The 80387 and later treat this as an invalid operand.
	01	Anything	Pseudo Not a Number. The sign bit is meaningless. The 8087 and 80287 treat this as a Signaling Not a Number. The 80387 and later treat this as an invalid operand.
	10	Zero	Infinity. The sign bit gives the sign of the infinity. The 8087 and 80287 treat this as a Signaling Not a Number. The 8087 and 80287 coprocessors used the pseudo-infinity representation for infinities.

	11	Non-zero	Signalling Not a Number, the sign bit is meaningless.
		Zero	Floating-point Indefinite, the result of invalid calculations such as square root of a negative number, logarithm of a negative number, 0/0, infinity / infinity, infinity times 0, and others when the processor has been configured to not generate exceptions for invalid operands. The sign bit is meaningless. This is a special case of a Quiet Not a Number.
		Non-zero	Quiet Not a Number, the sign bit is meaningless. The 8087 and 80287 treat this as a Signaling Not a Number.
	Bit 63	**Bits 62-0**	
All other values	Zero	Anything	Unnormal. Only generated on the 8087 and 80287. The 80387 and later treat this as an invalid operand. The value is $(-1)^s \times m \times 2^{e-16383}$
	One	Anything	Normalized value. The value is $(-1)^s \times m \times 2^{e-16383}$

In contrast to the single and double-precision formats, this format does not utilize an implicit/hidden bit. Rather, bit 63 contains the integer part of the significand and bits 62-0 hold the fractional part. Bit 63 will be 1 on all normalized numbers. There were several advantages to this design when the 8087 was being developed:

- Calculations can be completed a little faster if all bits of the significand are present in the register.
- A 64-bit significand provides sufficient precision to avoid loss of precision when the results are converted back to double precision format in the vast number of cases.
- This format provides a mechanism for indicating precision loss due to underflow which can be carried through further operations. For example, the calculation $2\times10^{-4930} \times 3\times10^{-10} \times 4\times10^{20}$ generates the intermediate result 6×10^{-4940} which is a denormal and also involves precision loss. The product of all of the terms is 24×10^{-4920} which can be represented as a normalized number. The 80287 could complete this calculation and indicate the loss of precision by returning an "unnormal" result (exponent not 0, bit 63 = 0). Processors since the 80387 no longer generate unnormals and do not support unnormal inputs to operations. They will generate a denormal if an underflow occurs but will generate a normalized result if subsequent operations on the denormal can be normalized.

Introduction to use

The 80-bit floating point format was widely available by 1984, after the development of C, Fortran and similar computer languages, which initially offered only the common 32- and 64-bit floating point sizes. On the x86 design most C compilers now support 80-bit extended precision via the long double type, and this was specified in

the C99 / C11 standards (IEC 60559 floating-point arithmetic (Annex F)). Compilers on x86 for other languages often support extended precision as well, sometimes via nonstandard extensions: for example, Turbo Pascal offers an `extended` type, and several Fortran compilers have a `REAL*10` type (analogous to `REAL*4` and `REAL*8`). Such compilers also typically include extended-precision mathematical subroutines, such as square root and trigonometric functions, in their standard libraries.

Working range

The 80-bit floating point format has a range (including subnormals) from approximately 3.65×10^{-4951} to 1.18×10^{4932}. Although $\log_{10}(2^{64}) \cong 19.266$, this format is usually described as giving approximately eighteen significant digits of precision. The use of decimal when talking about binary is unfortunate because most decimal fractions are recurring sequences in binary just as 2/3 is in decimal. Thus, a value such as 10.15 is represented in binary as equivalent to 10.1499996185 etc. in decimal for REAL*4 but 10.150000000000000035527etc. in REAL*8: interconversion will involve approximation except for those few decimal fractions that represent an exact binary value, such as 0.625. For REAL*10, the decimal string is 10.14999999999999999996530553etc. The last 9 digit is the eighteenth fractional digit and thus the twentieth significant digit of the string. Bounds on conversion between decimal and binary for the 80-bit format can be given as follows: if a decimal string with at most 18 significant digits is correctly rounded to an 80-bit IEEE 754 binary floating point value (as on input) then converted back to the same number of significant decimal digits (as for output), then the final string will exactly match the original; while, conversely, if an 80-bit IEEE 754 binary floating point value is correctly converted and (nearest) rounded to a decimal string with at least 21 significant decimal digits then converted back to binary format it will exactly match the original. These approximations are particularly troublesome when specifying the best value for constants in formulae to high precision, as might be calculated via arbitrary precision arithmetic.

Need for the 80-bit format

A notable example of **the need for a minimum of 64 bits of precision in the significand** of the extended precision format is the need to avoid precision loss when performing exponentiation on double precision values.{{efn|"The presence of at least as many extra bits of precision in extended as in the exponent field of the basic format it supports greatly simplifies the accurate computation of the transcendental functions, inner products, and the power function y^x".[p70] The x86 floating-point units do not provide an instruction that directly performs exponentiation. Instead they provide a set of instructions that a program can use in sequence to perform exponentiation using the equation:

$$x^y = 2^{y\cdot\log_2(x)}$$

In order to avoid precision loss, the intermediate results "$\log_2(x)$" and "$y \cdot \log_2(x)$" must be computed with much higher precision, because effectively both the exponent and the significand fields of x must fit into the significand field of the intermediate result. Subsequently the significand field of the intermediate result is split between the exponent and significand fields of the final result when $2^{\text{intermediate result}}$ is calculated. The following discussion describes this requirement in more detail.

With a little unpacking, an IEEE 754 double precision value can be represented as:

$$2(-1)_s \cdot E \cdot M2(-1)s \cdot E \cdot M$$

where s is the sign of the exponent (either 0 or 1), E is the unbiased exponent, which is an integer that ranges from 0 to 1023, and M is the significand which is a 53 bit value that falls in the range $1 \le M < 2$. Negative numbers and zero can be ignored because the logarithm of these values is undefined. For purposes of this discussion M does not have 53 bits of precision because it is constrained to be greater than or equal to one i.e. the hidden bit does not count towards the precision (Note that in situations where M is less than 1, the value is actually a de-normal and therefore may have already suffered precision loss. This situation is beyond the scope of this article).

Taking the log of this representation of a double precision number and simplifying results in the following:

$$\log_2(2(-1)_s \cdot E \cdot M) = (-1)s \cdot E \cdot \log_2(2) + \log_2(M) = \pm E + \log_2(M) \log_2(2(-1)s \cdot E \cdot M) = (-1)s \cdot E \cdot \log_2(2) + \log_2(M) = \pm E + \log_2(M)$$

This result demonstrates that when taking base 2 logarithm of a number, the sign of the exponent of the original value becomes the sign of the logarithm, the exponent of the original value becomes the integer part of the significand of the logarithm, and the significand of the original value is transformed into the fractional part of the significand of the logarithm.

Because E is an integer in the range 0 to 1023, up to 10 bits to the left of the radix point are needed to represent the integer part of the logarithm. Because M falls in the range $1 \le M < 2$, the value of $\log_2 M$ will fall in the range $0 \le \log_2 M < 1$ so at least 52 bits are needed to the right of the radix point to represent the fractional part of the logarithm. Combining 10 bits to the left of the radix point with 52 bits to the right of the radix point means that the significand part of the logarithm must be computed to at least 62 bits of precision. In practice values of M less than $2-\sqrt{2}$ require 53 bits to the right of the radix point and values of M less than $2-\sqrt{4}24$ require 54 bits to the right of the radix point to avoid precision loss. Balancing this requirement for added precision to the right of the radix point, exponents less than 512 only require 9 bits to the left of the radix point and exponents less than 256 require only 8 bits to the left of the radix point.

The final part of the exponentiation calculation is computing $2^{\text{intermediate result}}$. The "intermediate result" consists of an integer part "*I*" added to a fractional part "*F*". If the

intermediate result is negative then a slight adjustment is needed to get a positive fractional part because both "*I*" and "*F*" are negative numbers.

For positive intermediate results:

$$2^{\text{intermediate result}} = 2^{I+F} = 2^I 2^F$$

For negative intermediate results:

$$2^{\text{intermediate result}} = 2^{I+F} = 2^{I+(1-1)+F} = 2^{(I-1)+(1+F)} = 2^{I-1} 2^{1+F}$$

Thus the integer part of the intermediate result ("I" or "*I*−1") plus a bias becomes the exponent of the final result and transformed positive fractional part of the intermediate result: 2^F or 2^{1+F} becomes the significand of the final result. In order to supply 52 bits of precision to the final result, the positive fractional part must be maintained to at least 52 bits.

In conclusion, the exact number of bits of precision needed in the significand of the intermediate result is somewhat data dependent but 64 bits is sufficient to avoid precision loss in the vast majority of exponentiation computations involving double precision numbers.

The number of bits needed for the exponent of the extended precision format follows from the requirement that the product of two double precision numbers should not overflow when computed using the extended format. The largest possible exponent of a double precision value is 1023 so the exponent of the largest possible product of two double precision numbers is 2047 (an 11 bit value). Adding in a bias to account for negative exponents means that the exponent field must be at least 12 bits wide.

Combining these requirements: 1 bit for the sign, 12 bits for the biased exponent, and 64 bits for the significand means that the extended precision format would need at least 77 bits. Engineering considerations resulted in the final definition of the 80 bit format (in particular the IEEE 754 standard requires the exponent range of an extended precision format to match that of the next largest, quad, precision format which is 15 bits).

Another example of calculations that benefit from extended precision arithmetic are iterative refinement schemes, used to indirectly clean out errors accumulated in the direct solution during the typically very large number of calculations made for numerical linear algebra.

Source: Wikipedia, https://en.wikipedia.org/wiki/Extended_precision

Unit 5: Designing a Processor

In this unit, we will discuss various components of MIPS processor architecture and then take a subset of MIPS instructions to create a simplified processor in order to better understand the steps in processor design. This unit will ask you to apply the information you learned in units 2, 3, and 4 to create a simple processor architecture. We will also discuss a technique known as pipelining, which is used to improve processor performance. We will also identify the issues that limit the performance gains that can be achieved from it.

In previous units, you learned about how computer memory stores information, in particular how numbers are represented in a computer memory word (typically, 32 or 64 bits); hardware elements that perform logic functions; the use of these elements to design larger hardware components that perform arithmetic computations, in particular addition and multiplication; and the use of these larger components to design additional components that perform subtraction and division. You also looked at machine language and assembly language instructions that provide control to hardware components in carrying out computations. In this unit, you will learn about how the larger components are used in designing a computer system.

Upon successful completion of this unit, you will be able to:

- list the hardware components used to develop the architecture of a processor and show the data path for a simple operation;
- describe the components and operation of a sequential or Von Neumann computer architecture;
- summarize the design of a simple MIPS processor;
- describe the basic operation of pipelining;
- describe the approaches used to improve processor performance; and
- identify the factors that affect and limit the performance of a processor and its architecture.

5.1: Von Neumann Architecture

The Von Neumann Architecture

Read this section to learn about sequential or Von Neumann computer architecture. Computer architecture is the high-level computer design comprising components that perform the functions of data storage, computations, data transfer, and control.

The Von Neumann architecture

While computers, and most relevantly for this chapter, their processors, can differ in any number of details, they also have many aspects in common. On a very high level of abstraction, many architectures can be described as *von Neumann architectures*. This describes a design with an undivided memory that stores both program and data ('stored program'), and a processing unit that executes the instructions, operating on the data.

This setup distinguishes modern processors for the very earliest, and some special purpose contemporary, designs where the program was hard-wired. It also allows programs to modify themselves, since instructions and data are in the same storage. This allows us to have editors and compilers: the computer treats program code as data to operate on. In this book we will not explicitly discuss compilers, the programs that translate high level languages to machine instructions. However, on occasion we will discuss how a program at high level can be written to ensure efficiency at the low level.

In scientific computing, however, we typically do not pay much attention to program code, focusing almost exclusively on data and how it is moved about during program execution. For most practical purposes it is as if program and data are stored separately. The little that is essential about instruction handling can be described as follows.

The machine instructions that a processor executes, as opposed to the higher level languages users write in, typically specify the name of an operation, as well as of the locations of the operands and the result. These locations are not expressed as memory locations, but as registers: a small number of named memory locations that are part of the CPU. As an example, here is a simple C routine

```
void store(double *a, double *b, double *c) {
  *c = *a + *b;
}
```

and its X86 assembler output, obtained by

```
gcc -O2 -S -o - store.c
  .text
  .p2align 4,,15
.globl store
  .type store, @function
store:
  movsd (%rdi), %xmm0 # Load *a to %xmm0
  addsd (%rsi), %xmm0 # Load *b and add to %xmm0
  movsd %xmm0, (%rdx) # Store to *c
  ret
```

The instructions here are:

- A load from memory to register;
- Another load, combined with an addition;
- Writing back the result to memory.

Each instruction is processed as follows:

- Instruction fetch: the next instruction according to the *program counter* is loaded into the processor. We will ignore the questions of how and from where this happens.
- Instruction decode: the processor inspects the instruction to determine the operation and the operands.
- Memory fetch: if necessary, data is brought from memory into a register.

- Execution: the operation is executed, reading data from registers and writing it back to a register.
- Write-back: for store operations, the register contents is written back to memory.

Complicating this story, contemporary CPUs operate on several instructions simultaneously, which are said to be 'in flight', meaning that they are in various stages of completion. This is the basic idea of the superscalar CPU architecture, and is also referred to as *instruction-level parallelism*. Thus, while each instruction can take several clock cycles to complete, a processor can complete one instruction per cycle in favourable circumstances; in some cases more than one instruction can be finished per cycle.

The main statistic that is quoted about CPUs is their Gigahertz rating, implying that the speed of the processor is the main determining factor of a computer's performance. While speed obviously correlates with performance, the story is more complicated. Some algorithms are *cpu-bound*, and the speed of the processor is indeed the most important factor; other algorithms are *memory-bound*, and aspects such as bus speed and cache size become important.

In scientific computing, this second category is in fact quite prominent, so in this chapter we will devote plenty of attention to the process that moves data from memory to the processor, and we will devote relatively little attention to the actual processor.

Floating point units

Many modern processors are capable of doing multiple operations simultaneously, and this holds in particular for the arithmetic part. For instance, often there are separate addition and multiplication units; if the compiler can find addition and multiplication operations that are independent, it can schedule them so as to be executed simultaneously, thereby doubling the performance of the processor. In some cases, a processor will have multiple addition or multiplication units.

Another way to increase performance is to have a 'fused multiply-add' unit, which can execute the instruction $x \leftarrow ax+b$x←ax+b

in the same amount of time as a separate addition or multiplication. Together with pipelining (see below), this means that a processor has an asymptotic speed of several floating point operations per clock cycle.

Processor	floating point units	max operations per cycle
Pentium4, Opteron	2 add or 2 mul	2
Woodcrest, Barcelona	2 add + 2 mul	4

IBM POWER4, POWER5, POWER6	2 FMA	4
IBM BG/L, BG/P	1 SIMD FMA	4
SPARC IV	1 add + 1 mul	2
Itanium2	2 FMA	4

Table 1.1: Floating point capabilities of several current processor architectures

Pipelining

The floating point add and multiply units of a processor are pipelined, which has the effect that a stream of inde-pendent operations can be performed at an asymptotic speed of one result per clock cycle.

The idea behind a pipeline is as follows. Assume that an operation consists of multiple simpler operations, and that for each suboperation there is separate hardware in the processor. For instance, an multiply instruction can have the following components:

- Decoding the instruction, including finding the locations of the operands. Copying the operands into registers ('data fetch').
- Aligning the exponents; the multiplication $.3\times10^{-1}\times.2\times10^{2}$.3×10−1×.2×102 becomes $.0003\times10^{2}\times.2\times10^{2}$. 0003×102×.2×102.
- Executing the multiplication, in this case giving $.00006\times10^{0}$.00006×100.
- Normalizing the result, in this example to $.6\times10^{0}$.6×100.
- Storing the result.

These parts are often called the 'stages' or 'segments' of the pipeline.

If every component is designed to finish in 1 clock cycle, the whole instruction takes 6 cycles. However, if each has its own hardware, we can execute two multiplications in less than 12 cycles:

- Execute the decode stage for the first operation;
- Do the data fetch for the first operation, and at the same time the decode for the second.
- Execute the third stage for the first operation and the second stage of the second operation simultaneously.
- Et cetera.

You see that the first operation still takes 6 clock cycles, but the second one is finished a mere 1 cycle later. This idea can be extended to more than two operations: the first operation still takes the same amount of time as before, but after that one more result will be produced each cycle. Formally, executing n operations on a s-segment pipeline takes $s + n - 1$ cycles.

Exercise 1.1. Let us compare the speed of a classical floating point unit, and a pipelined one. If the pipeline has s stages, what is the asymptotic speedup? That is, with $T_0(n)$ the time for n operations on a classical CPU, and $T_s(n)$ the time for n operations on an s-segment pipeline, what is $\lim_{n\to\infty}(T_0(n)/T_s(n))$limn→∞(T0(n)/Ts(n))?

Next you can wonder how long it takes to get close to the asymptotic behaviour. Define $S_s(n)$ as the speedup achieved on n operations. The quantity $n_{1/2}$ is defined as the value of n such that S_s(n) is half the asymptotic speedup. Give an expression for $n_{1/2}$.

Since a vector processor works on a number of instructions simultaneously, these instructions have to be independent. The operation $\forall_i: a_i \leftarrow b_i + c_i$∀i:ai←bi+ci has independent additions; the operation $\forall_i: a_{i+1} \leftarrow a_i b_i + c_i$∀i:ai+1←aibi+ci feeds the result of one iteration (a_i)(ai) to the input of the next $(a_{i+1}=\ldots)$(ai+1=...), so the operations are not independent.

[A pipelined processor can speed up operations by a factor of 4; 5; 6 with respect to earlier CPUs. Such numbers were typical in the 1980s when the first successful vector computers came on the market. These days, CPUs can have 20-stage pipelines. Does that mean they are incredibly fast? This question is a bit complicated. Chip designers continue to increase the clock rate, and the pipeline segments can no longer finish their work in one cycle, so they are further spit up. Sometimes there are even segments in which nothing happens: that time is needed to make sure data can travel to a different part of the chip in time.]

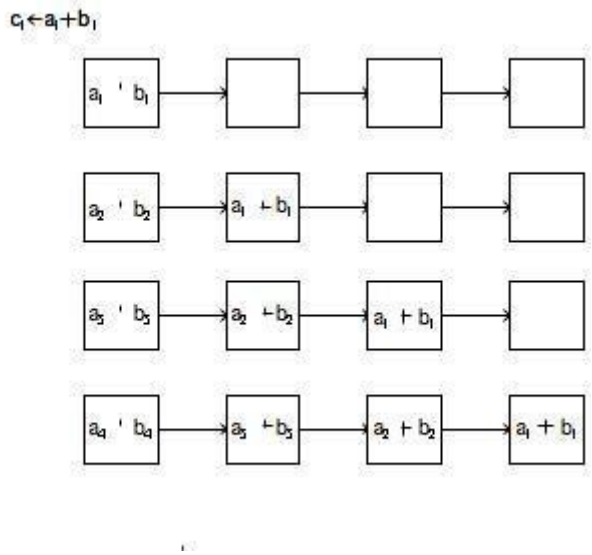


Figure 1.1: Schematic depiction of a pipelined operation

The amount of improvement you can get from a pipelined CPU is limited, so in a quest for ever higher performance several variations on the pipeline design have been tried. For instance, the Cyber 205 had separate addition and multiplication pipelines, and it was possible to feed one pipe into the next without data going back to memory first.

Operations like $\forall_i: a_i \leftarrow b_i + c \cdot d_i$∀i:ai←bi+c·di were called 'linked triads' (because of the number of paths to memory, one input operand had to be scalar).

Exercise 1.2. Analyse the speedup and $n_{1/2}$ of linked triads.

Another way to increase performance is to have multiple identical pipes. This design was perfected by the NEC SX series. With, for instance, 4 pipes, the operation \[\forall_i: a_i \leftarrow b_i + c_i\) would be split module 4, so that the first pipe operated on indices $i = 4 \cdot j$i=4·j, the second on $i = 4 \cdot j + 1$i=4·j+1, et cetera.

Exercise 1.3. Analyze the speedup and $n_{1/2}$ of a processor with multiple pipelines that operate in parallel. That is, suppose that there are p independent pipelines, executing the same instruction, that can each handle a stream of operands.

(The reason we are mentioning some fairly old computers here is that true pipeline supercomputers hardly exist anymore. In the US, the Cray X1 was the last of that line, and in Japan only NEC still makes them. However, the functional units of a CPU these days are pipelined, so the notion is still important.)

Exercise 1.4. The operation

```
for (i) {
  x[i+1] = a[i]*x[i] + b[i];
}
```

can not be handled by a pipeline or SIMD processor because there is a *dependency* between input of one iteration of the operation and the output of the previous. However, you can transform the loop into one that is mathematically equivalent, and potentially more efficient to compute. Derive an expression that computes x[i+2] from x[i] without involving x[i+1]. This is known as *recursive doubling*. Assume you have plenty of temporary storage. You can now perform the calculation by

- Doing some preliminary calculations;
- computing x[i],x[i+2],x[i+4],..., and from these,
- compute the missing terms x[i+1],x[i+3],....

Analyze the efficiency of this scheme by giving formulas for $T_0(n)$ and $T_s(n)$. Can you think of an argument why the preliminary calculations may be of lesser importance in some circumstances?

Peak performance

For marketing purposes, it may be desirable to define a 'top speed' for a CPU. Since a pipelined floating point unit can yield one result per cycle asymptotically, you would calculate the theoretical *peak performance* as the product of the clock speed (in ticks per second), number of floating point units, and the number of cores (see section 1.3). This

top speed is unobtainable in practice, and very few codes come even close to it; see section 2.11. Later in this chapter you will learn the reasons that it is so hard to get this perfect performance.

Pipelining beyond arithmetic: instruction-level parallelism

In fact, nowadays, the whole CPU is pipelined. Not only floating point operations, but any sort of instruction will be put in the instruction pipeline as soon as possible. Note that this pipeline is no longer limited to identical instructions: the notion of pipeline is now generalized to any stream of partially executed instructions that are simultaneously "in flight".

This concept is also known as *instruction-level parallelism*, and it is facilitated by various mechanisms:

- multiple-issue: instructions that are independent can be started at the same time;
- pipelining: already mentioned, arithmetic units can deal with multiple operations in various stages of completion;
- branch prediction and speculative execution: a compiler can 'guess' whether a conditional instruction will evaluate to true, and execute those instructions accordingly;
- out-of-order execution: instructions can be rearranged if they are not dependent on each other, and if the resulting execution will be more efficient;
- prefetching: data can be speculatively requested before any instruction needing it is actually encountered (this is discussed further in section 1.2.5).

As clock frequency has gone up, the processor pipeline has grown in length to make the segments executable in less time. You have already seen that longer pipelines have a larger $n_{1/2}$, so more independent instructions are needed to make the pipeline run at full efficiency. As the limits to instruction-level parallelism are reached, making pipelines longer (sometimes called 'deeper') no longer pays off. This is generally seen as the reason that chip designers have moved to *multi-core* architectures as a way of more efficiently using the transistors on a chip; section 1.3.

There is a second problem with these longer pipelines: if the code comes to a branch point (a conditional or the test in a loop), it is not clear what the next instruction to execute is. At that point the pipeline can stall. CPUs have taken to *speculative execution*' for instance, by always assuming that the test will turn out true. If the code then takes the other branch (this is called a *branch misprediction*), the pipeline has to be cleared and restarted. The resulting delay in the execution stream is called the branch penalty.

8-bit, 16-bit, 32-bit, 64-bit

Processors are often characterized in terms of how big a chunk of data they can process as a unit. This can relate to

- The width of the path between processor and memory: can a 64-bit floating point number be loaded in one cycle, or does it arrive in pieces at the processor.
- The way memory is addressed: if addresses are limited to 16 bits, only 64,000 bytes can be identified. Early PCs had a complicated scheme with segments to get around this limitation: an address was specified with a segment number and an offset inside the segment.
- The number of bits in a register, in particular the size of the integer registers which manipulate data address; see the previous point. (Floating point register are often larger, for instance 80 bits in the x86 architecture.) This also corresponds to the size of a chunk of data that a processor can operate on simul-taneously.
- The size of a floating point number. If the arithmetic unit of a CPU is designed to multiply 8-byte numbers efficiently ('double precision'; see section 3.2) then numbers half that size ('single precision') can some-times be processed at higher efficiency, and for larger numbers ('quadruple precision') some complicated scheme is needed. For instance, a quad precision number could be emulated by two double precision numbers with a fixed difference between the exponents.

These measurements are not necessarily identical. For instance, the original Pentium processor had 64-bit data busses, but a 32-bit processor. On the other hand, the Motorola 68000 processor (of the original Apple Macintosh) had a 32-bit CPU, but 16-bit data busses.

The first Intel microprocessor, the 4004, was a 4-bit processor in the sense that it processed 4 bit chunks. These days, processors are 32-bit, and 64-bit is becoming more popular.

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

5.2: Simple MIPS Processor Components

An Introduction to Processor Design

Watch this lecture, which introduces the components of MIPS architecture that are required to process a subset of MIPS instructions. This is the first lecture of a series of video lectures on the design of a MIPS processor. This series of six videos incrementally design a processor that implements a subset of eight MIPS five arithmetic instructions, two memory reference instructions, and one flow control instruction. Using hardware components, referred to as building blocks, and a microprogram control, the lecturer executes these eight instructions to develop a simple design of a processor. In this lecture, Kumar discusses the performance of the design and performance improvement

using a multi-cycle design. Also, Kumar identifies an extension of the design to deal with exceptional cases that could occur when executing programs written using the eight instructions (exception handling). Lastly, the lecturer identifies increments to the design to include additional instructions.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=0B-y1RPDXjs

5.3: Designing a Datapath for a Simple Processor

Datapaths

Watch this lecture, which is the second video in a series of six video lectures on the design of a MIPS processor. The previous video lecture presented the processor building blocks that will be used in the series. This video lecture explains how to build a datapath of the MIPS architecture to process a subset of the MIPS instructions. We will take R-format instructions and memory instructions and look into the datapath requirements to process them. Then, the other instructions will be addressed one at a time, and incremental changes to the design will be made to handle them. The data path and controller will be interconnected, and the (micro) control signals to perform the correct hardware operations at the right time will be identified. Later, we will look at the design of the controller.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=Ngu1UbRAeqQ

5.4: Alternative Approach to Datapath Design and Design of a Control for a Simple Processor

Processor Design Control

Watch this lecture, which is the third video of the series of video lectures on the design of a simple processor. This lecture explains how to build the control part of the MIPS architecture that is required to process a subset of MIPS instructions. This builds on the datapath design from the last lecture. That datapath design approach started with a design for the R class instructions with operands in registers, such as add, subtract, 'and', 'or', and 'less than'. It then included the other instructions, one at a time, and incremented the design to accommodate them. In this video lecture, an alternative approach is used to arrive at the same design. Here, the datapath for the arithmetic and logic instructions is designed. Then, the data path for the store, then for the load, then the branch on equal, and, finally, the jump are designed individually. Next, datapath design for all eight instructions is the union of the five individual designs. The control signals for each instruction are identified and combined to form a truth table for a controller, which is implemented using a PLA (program logic array). The video concludes

with a performance/delay analysis of the design to show the limitations of a single cycle datapath. Next, we will look at pipelining for increased performance.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=kW_OuCEq_Iw

5.5: Pipelining and Hazards

Pipelined Processor Design

Watch this lecture, which presents basic ideas on improving processor performance through the use of pipelining. The previous video showed the limitations of a single cycle datapath design. To overcome the limitations and improve performance, a pipeline datapath design is considered. This video lecture explains pipelining. A pipeline datapath is analogous to an assembly line in manufacturing. First, you develop a skeleton design of a pipeline datapath. Performance analysis shows that several types of delays can arise, called hazards: structure, data, or control hazards. Design can address those arising from structure. However, data and control delays cannot always be prevented. The next video lecture will complete the design of a pipeline datapath for the simple processor.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=pIBwr7Rx-1M

5.6: Pipelined Processors

Pipelined Processor Datapaths

Watch this lecture, which explains how to design a pipelined MIPS processor. The previous video introduced pipelining as a way to increase performance. It showed how hazards can limit the performance improvement of a pipeline datapath. This video lecture completes the design of a pipeline datapath. Ignoring hazards, the lecturer designs a control for the pipeline, integrates all the components including the control with the pipeline, and then considers the behavior with respect to hazards.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=SL_djmTegqc

5.7: Instruction-Level Parallelism

Understanding Parallelism

Read this article, which discusses granularity of parallelism. On a uniprocessor, instruction level parallelism includes pipelining techniques and multiple functional units. Parallel processing using multi-processors will be covered in Unit 8.

In a sense, we have been talking about parallelism from the beginning of the book. Instead of calling it "parallelism", we have been using words like "pipelined", "superscalar", and "compiler flexibility". As we move into programming on multiprocessors, we must increase our understanding of parallelism in order to understand how to effectively program these systems. In short, as we gain more parallel resources, we need to find more parallelism in our code.

When we talk of parallelism, we need to understand the concept of granularity. The granularity of parallelism indicates the size of the computations that are being performed at the same time between synchronizations. Some examples of parallelism in order of increasing grain size are:

- When performing a 32-bit integer addition, using a carry lookahead adder, you can partially add bits 0 and 1 at the same time as bits 2 and 3.
- On a pipelined processor, while decoding one instruction, you can fetch the next instruction.
- On a two-way superscalar processor, you can execute any combination of an integer and a floating-point instruction in a single cycle.
- On a multiprocessor, you can divide the iterations of a loop among the four processors of the system.
- You can split a large array across four workstations attached to a network. Each workstation can operate on its local information and then exchange boundary values at the end of each time step.

In this chapter, we start at *instruction-level parallelism* (pipelined and superscalar) and move toward *thread-level parallelism*, which is what we need for multiprocessor systems. It is important to note that the different levels of parallelism are generally not in conflict. Increasing thread parallelism at a coarser grain size often exposes more fine-grained parallelism.

The following is a loop that has plenty of parallelism:

```
DO I=1,16000
  A(I) = B(I) * 3.14159
ENDDO
```

We have expressed the loop in a way that would imply that A(1) must be computed first, followed by A(2), and so on. However, once the loop was completed, it would not have mattered if A(16000), were computed first followed by A(15999), and so on. The loop could have computed the even values of I and then computed the odd values of I. It would not even make a difference if all 16,000 of the iterations were computed simultaneously using a 16,000-way superscalar processor.[1] If the compiler has flexibility in

the order in which it can execute the instructions that make up your program, it can execute those instructions simultaneously when parallel hardware is available.

One technique that computer scientists use to formally analyze the potential parallelism in an algorithm is to characterize how quickly it would execute with an "infinite-way" superscalar processor.

Not all loops contain as much parallelism as this simple loop. We need to identify the things that limit the parallelism in our codes and remove them whenever possible. In previous chapters we have already looked at removing clutter and rewriting loops to simplify the body of the loop.

We looked at the mechanics of compiling code, all of which apply here, but we didn't answer all of the "whys". Basic block analysis techniques form the basis for the work the compiler does when looking for more parallelism. Looking at two pieces of data, instructions, or data and instructions, a modern compiler asks the question, "Do these things depend on each other"? The three possible answers are yes, no, and we don't know. The third answer is effectively the same as a yes, because a compiler has to be conservative whenever it is unsure whether it is safe to tweak the ordering of instructions.

Helping the compiler recognize parallelism is one of the basic approaches specialists take in tuning code. A slight rewording of a loop or some supplementary information supplied to the compiler can change a "we don't know" answer into an opportunity for parallelism. To be certain, there are other facets to tuning as well, such as optimizing memory access patterns so that they best suit the hardware, or recasting an algorithm. And there is no single best approach to every problem; any tuning effort has to be a combination of techniques.

Footnotes

- 1 Interestingly, this is not as far-fetched as it might seem. On a single instruction multiple data (SIMD) computer such as the Connection CM-2 with 16,384 processors, it would take three instruction cycles to process this entire loop.

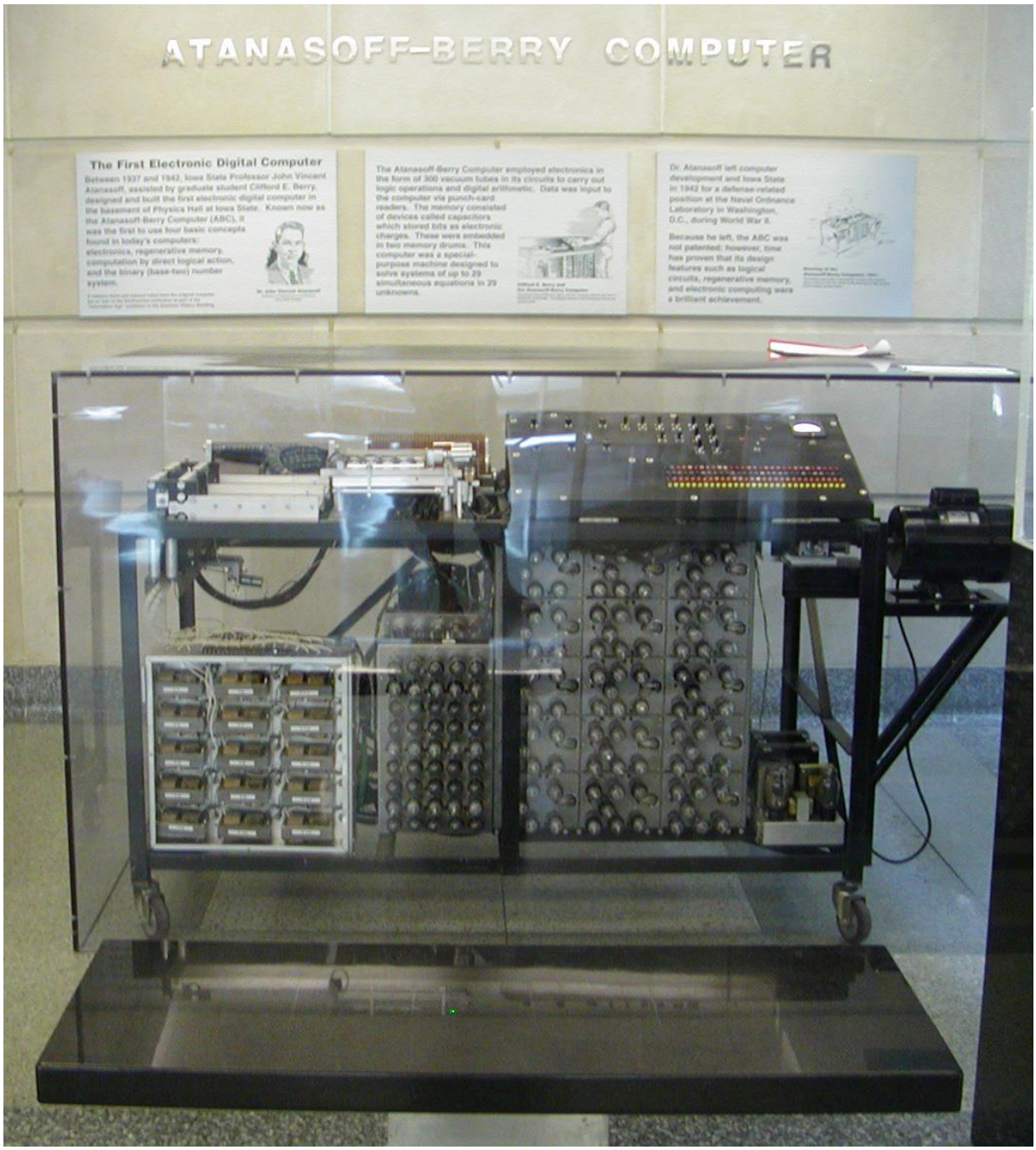
Source: Charles Severance and Kevin Dowd, https://cnx.org/contents/_I9-stX3@3/Understanding-Parallelism-Introduction

Instruction-Level Parallelism

Read the explanation of instruction level parallelism. Parallelism can occur at different levels of granularity, and when discussing parallelism, we need to be clear on exactly what is being done in parallel.

Instruction-level parallelism (**ILP**) is a measure of how many of the instructions in a computer program can be executed simultaneously.

ILP must not be confused with concurrency, since the first is about parallel execution of a sequence of instructions belonging to a specific thread of execution of a process (that is a running program with its set of resources - for example its address space, a set of registers, its identifiers, its state, program counter, and more). Conversely, concurrency regards with the threads of one or different processes being assigned to a CPU's core in a strict alternance or in true parallelism if there are enough CPU's cores, ideally one core for each runnable thread.



Atanasoff–Berry computer, the first computer with parallel processing

There are two approaches to instruction level parallelism: Hardware and Software.

Hardware level works upon dynamic parallelism, whereas the software level works on static parallelism. Dynamic parallelism means the processor decides at run time which instructions to execute in parallel, whereas static parallelism means the compiler decides which instructions to execute in parallel. The Pentium processor works on the dynamic sequence of parallel execution, but the Itanium processor works on the static level parallelism.

Consider the following program:

```
1 e = a + b
2 f = c + d
3 m = e * f
```
Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. However, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time, giving an ILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography may exhibit much less parallelism.

Micro-architectural techniques that are used to exploit ILP include:

- Instruction pipelining where the execution of multiple instructions can be partially overlapped.
- Superscalar execution, VLIW, and the closely related explicitly parallel instruction computing concepts, in which multiple execution units are used to execute multiple instructions in parallel.
- Out-of-order execution where instructions execute in any order that does not violate data dependencies. Note that this technique is independent of both pipelining and superscalar execution. Current implementations of out-of-order execution dynamically (i.e., while the program is executing and without any help from the compiler) extract ILP from ordinary programs. An alternative is to extract this parallelism at compile time and somehow convey this information to the hardware. Due to the complexity of scaling the out-of-order execution technique, the industry has re-examined instruction sets which explicitly encode multiple independent operations per instruction.
- Register renaming which refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations, used to enable out-of-order execution.
- Speculative execution which allows the execution of complete instructions or parts of instructions before being certain whether this execution should take place. A commonly used form of speculative execution is control flow speculation where instructions past a control flow instruction (e.g., a branch) are executed before the target of the control flow instruction is determined. Several other forms of speculative execution have been proposed and are in use including speculative execution driven by value prediction, memory dependence prediction and cache latency prediction.

- Branch prediction which is used to avoid stalling for control dependencies to be resolved. Branch prediction is used with speculative execution.

It is known that the ILP is exploited by both the compiler and hardware support but the compiler also provides inherent and implicit ILP in programs to hardware by compilation optimization. Some optimization techniques for extracting available ILP in programs would include scheduling, register allocation/renaming, and memory access optimization.

Dataflow architectures are another class of architectures where ILP is explicitly specified, for a recent example see the TRIPS architecture.

In recent years, ILP techniques have been used to provide performance improvements in spite of the growing disparity between processor operating frequencies and memory access times (early ILP designs such as the IBM System/360 Model 91 used ILP techniques to overcome the limitations imposed by a relatively small register file). Presently, a cache miss penalty to main memory costs several hundreds of CPU cycles. While in principle it is possible to use ILP to tolerate even such memory latencies, the associated resource and power dissipation costs are disproportionate. Moreover, the complexity and often the latency of the underlying hardware structures results in reduced operating frequency further reducing any benefits. Hence, the aforementioned techniques prove inadequate to keep the CPU from stalling for the off-chip data. Instead, the industry is heading towards exploiting higher levels of parallelism that can be exploited through techniques such as multiprocessing and multithreading.

Source: Wikipedia, https://en.wikipedia.org/wiki/Instruction-level_parallelism

Unit 6: The Memory Hierarchy

In prior units, you have studied elementary hardware components like combinational circuits and sequential circuits, functional hardware components like adders, arithmetic logical units, and data buses, and computational components like processors.

This unit will address the memory hierarchy of a computer and will identify different types of memory and how they interact with one another. This unit will look into a memory type known as cache and will discuss how caches improve computer performance. This unit will then discuss the main memory, DRAM (or the Dynamic Random Access Memory), and the associated concept of virtual memory. You will take a look at the common framework for memory hierarchy. The unit concludes with a review of the design of a cache hierarchy for an industrial microprocessor.

- Upon successful completion of this unit, you will be able to:

- describe cache memory and calculate how much it speeds up processing times using various cache structures;
- calculate miss rate for various cache configurations; and
- explain the importance of memory hierarchy in computer design, and explain how memory design impacts overall hardware performance.

- 6.1: Elements of Memory Hierarchy and Caches

The Basics of Memory Hierarchy

Watch this lecture. Previously, we focused on processor design to increase performance. Now, we will turn to memory. This video introduces various methods of improving processor performance through the use of memory hierarchy. The lecture discusses memory technologies, which vary in cost and speed. We have to assume that memory is flat, but with current technology, flat memory does not meet performance demands placed on it. You will take a look at hierarchical memory and the use of caches. This video will also discuss the analysis of memory hierarchies and cache performance with respect to miss rates and block size. Finally, the lecturer considers cache policy.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=_eAL-v5oNOw

Sequential Computer Architecture

Read section 1.2 to learn about memory hierarchies, and read section 1.4 to learn about locality and data reuse.

1.2 Memory Hierarchies

We will now refine the picture of the Von Neuman architecture, in which data is loaded immediately from memory to the processors, where it is operated on. This picture is unrealistic because of the so-called *memory wall*: the processor is much too fast to load data this way. Specifically, a single load can take 1000 cycles, while a processor can perform several operations per cycle. (After this long wait for a load, the next load can come faster, but still too slow for the processor. This matter of wait time versus throughput will be addressed below in section 1.2.2.)

In reality, there will be various memory levels in between the floating point unit and the main memory: the registers and the caches. Each of these will be faster to a degree than main memory; unfortunately, the faster the memory on a certain level, the slower it will be. This leads to interesting programming problems, which we will discuss in the rest of this chapter, and particularly section 1.5.

The use of registers is the first instance you will see of measures taken to counter the fact that loading data from memory is slow. Access to data in registers, which are built into the processor, is almost instantaneous, unlike main memory, where hundreds of clock cycles can pass between requesting a data item, and it being available to the processor.

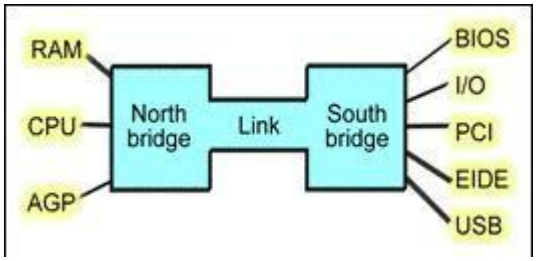
One advantage of having registers is that data can be kept in them during a computation, which obviates the need for repeated slow memory loads and stores. For example, in

```
s = 0;
for (i=0; i<n; i++)
  s += a[i]*b[i];
```

the variable s only needs to be stored to main memory at the end of the loop; all intermediate values are almost immediately overwritten. Because of this, a compiler will keep s in a register, eliminating the delay that would result from continuously storing and loading the variable. Such questions of *data reuse* will be discussed in more detail below; we will first discuss the components of the memory hierarchy.

1.2.1 Busses

The wires that move data around in a computer, from memory to cpu or to a disc controller or screen, are called *busses*. The most important one for us is the *Front-Side Bus (FSB)* which connects the processor to memory. In one popular architecture, this is called the 'north bridge', as opposed to the 'south bridge' which connects to external devices, with the exception of the graphics controller.



The bus is typically much slower than the processor, operating between 500MHz and 1GHz, which is the main reason that caches are needed.

1.2.2 Latency and Bandwidth

Above, we mentioned in very general terms that operating on data in registers is almost instantaneous, whereas loading data from memory into the registers, a necessary step before any operation, incurs a substantial delay. We will now make this story slightly more precise.

There are two important concepts to describe the movement of data: *latency* and *bandwidth*. The assumption here is that requesting an item of data incurs an initial delay; if this item was the first in a stream of data, usually a consecutive range of memory addresses, the remainder of the stream will arrive with no further delay at a regular amount per time period.

Latency is the delay between the processor issuing a request for a memory item, and the item actually arriving. We can distinguish between various latencies, such as the transfer from memory to cache, cache to register, or summarize them all into the latency between memory and processor. Latency is measured in (nano) seconds, or clock periods.

If a processor executes instructions in the order they are found in the assembly code, then execution will often *stall* while data is being fetched from memory; this is also called *memory stall*. For this reason, a low latency is very important. In practice, many processors have 'out-of-order execution' of instructions, allowing them to perform other operations while waiting for the requested data. Programmers can take this into account, and code in a way that achieves *latency hiding*.

Bandwidth is the rate at which data arrives at its destination, after the initial latency is overcome. Bandwidth is measured in bytes (kilobyes, megabytes, gigabyes) per second or per clock cycle. The bandwidth between two memory levels is usually the product of the cycle speed of the channel (the *bus speed*) and width of the bus: the number of bits that can be sent simultaneously in every cycle of the bus clock.

The concepts of latency and bandwidth are often combined in a formula for the time that a message takes from start to finish:

$$T(n)=\alpha+\beta n$$

where α is the latency and β is the inverse of the bandwidth: the time per byte.

Typically, the further away from the processor one gets, the longer the latency is, and the lower the bandwidth. These two factors make it important to program in such a way that, if at all possible, the processor uses data from cache or register, rather than from main memory. To illustrate that this is a serious matter, consider a vector addition

```
for (i)
  a[i] = b[i]+c[i]
```

Each iteration performs one floating point operation, which modern CPUs can do in one clock cycle by using pipelines. However, each iteration needs two numbers loaded and one written, for a total of 24 bytes4 of memory traffic. Typical memory bandwidth figures (see for instance figure 1.2) are nowhere near 32 bytes per cycle. This means that, without caches, algorithm performance can be bounded by memory performance.

The concepts of latency and bandwidth will also appear in parallel computers, when we talk about sending data from one processor to the next.

1.2.3 Registers

The registers are the memory level that is closest to the processor: any operation acts on data in register and leaves its result in register. Programs written in assembly language explicitly use these registers:

```
addl %eax, %edx
```

which is an instruction to add the content of one register to another. As you see in this sample instruction, registers are not numbered in memory, but have distinct names that are referred to in the assembly instruction.

Registers have a high bandwidth and low latency, because they are part of the processor. That also makes them a very scarce resource.

1.2.4 Caches

In between the registers, which contain the data that the processor operates on, and the main memory where lots of data can reside for a long time, are various levels of *cache* memory, that have lower latency and higher bandwidth than main memory. Data from memory travels the cache hierarchy to wind up in registers. The advantage to having cache memory is that if a data item is reused shortly after it was first needed, it will still be in cache, and therefore it can be accessed much faster than if it would have to be brought in from memory.

The caches are called 'level 1' and 'level 2' (or, for short, L1 and L2) cache; some processors can have an L3 cache. The L1 and L2 caches are part of the die, the processor chip, although for the L2 cache that is a recent development; the L3 cache is off-chip. The L1 cache is small, typically around 16Kbyte. Level 2 (and, when present, level 3) cache is more plentiful, up to several megabytes, but it is also slower. Unlike main memory, which is expandable, caches are fixed in size. If a version of a processor chip exists with a larger cache, it is usually considerably more expensive.

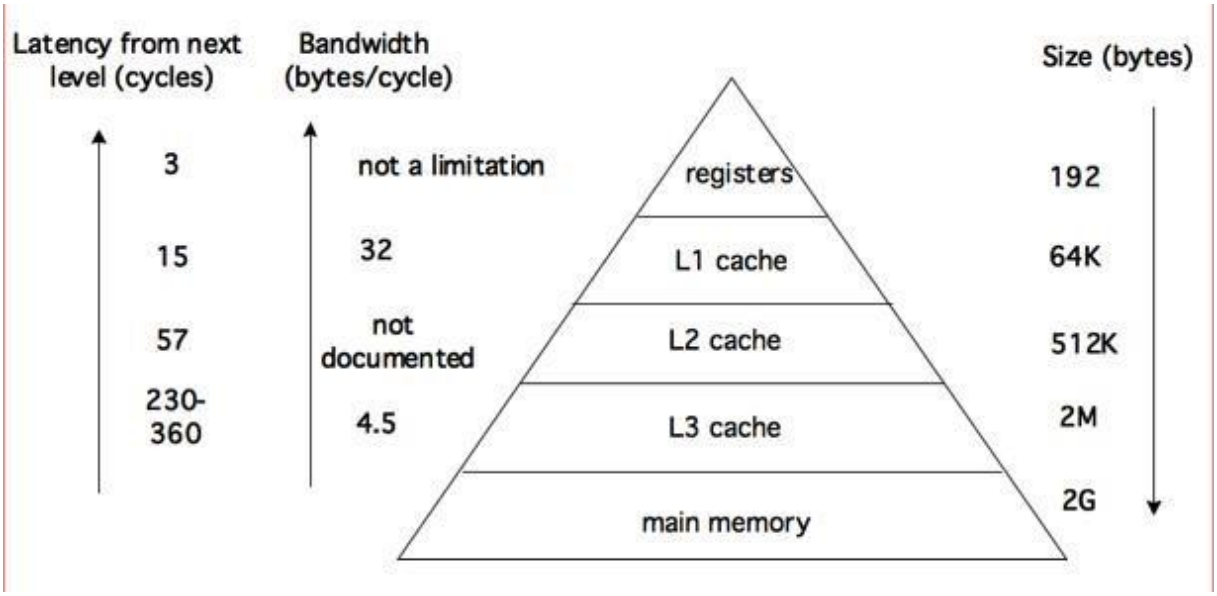
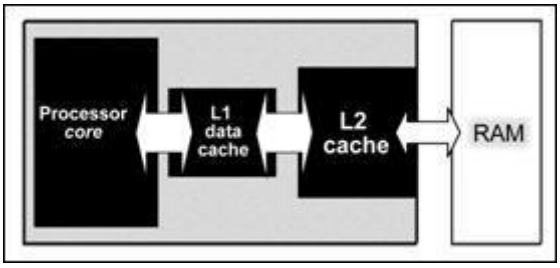


Figure 1.2: Memory hierarchy, characterized by speed and size.



Data that is needed in some operation gets copied into the various caches on its way to the processor. If, some instructions later, a data item is needed again, it is first searched for in the L1 cache; if it is not found there, it is searched for in the L2 cache; if it is not found there, it is loaded from main memory. Finding data in cache is called a cache hit, and not finding it a *cache miss*.

Figure 1.2 illustrates the basic facts of caches, in this case for the AMD Opteron chip: the closer caches are to the floating point units, the fast, but also the smaller they are. Some points about this figure.

- Loading data from registers is so fast that it does not constitute a limitation on algorithm execution speed. On the other hand, there are few registers. The Opteron5 has 16 general purpose registers, 8 media and floating point registers, and 16 SIMD registers.
- The L1 cache is small, but sustains a bandwidth of 32 bytes, that is 4 double precision number, per cycle. This is enough to load two operands each for two operations, but note that the Opteron can actually perform 4 operations per cycle. Thus, to achieve peak speed, certain operands need to stay in register. The latency from L1 cache is around 3 cycles.
- The bandwidth from L2 and L3 cache is not documented and hard to measure due to cache policies (see below). Latencies are around 15 cycles for L2 and 50 for L3.
- Main memory access has a latency of more than 100 cycles, and a bandwidth of 4.5 bytes per cycle, which is about 1/7th of the L1 bandwidth. However, this bandwidth is shared by the 4 cores of the opteron chip, so effectively the bandwidth is a quarter of this number. In a machine like Ranger, which has 4 chips per node, some bandwidth is spent on maintaining cache coherence (see section 1.3) reducing the bandwidth for each chip again by half.

On level 1, there are separate caches for instructions and data; the L2 and L3 cache contain both data and instructions.

You see that the larger caches are increasingly unable to supply data to the processors fast enough. For this reason it is necessary to code in such a way that data is kept as much as possible in the highest cache level possible. We will discuss this issue in detail in the rest of this chapter.

Exercise 1.5. The L1 cache is smaller than the L2 cache, and if there is an L3, the L2 is smaller than the L3. Give a practical and a theoretical reason why this is so.

1.2.4.1 Reuse is the name of the game

The presence of one or more caches is not immediately a guarantee for high performance: this largely depends on the *memory access pattern* of the code, and how well this exploits the caches. The first time that an item is referenced, it is copied from memory into cache, and through to the processor registers. The latency and bandwidth for this are not mitigated in any way by the presence of a cache. When the same item is referenced a second time, it may be found in cache, at a considerably reduced cost in terms of latency and bandwidth: caches have shorter latency and higher bandwidth than main memory.

We conclude that, first, an algorithm has to have an opportunity for data reuse. If every data item is used only once (as in addition of two vectors), there can be no reuse, and the presence of caches is largely irrelevant. A code will only benefit from the increased bandwidth and reduced latency of a cache if items in cache are referenced more than once; see section 1.4.1 for a detailed discussion. An example would be the matrix-vector multiplication $y = Ax$ where each element of x is used in n operations, where n is the matrix dimension.

Secondly, an algorithm may theoretically have an opportunity for reuse, but it needs to be coded in such a way that the reuse is actually exposed. We will address these points in section 1.4.3. This second point especially is not trivial.

1.2.4.2 Replacement policies

Data in cache and registers is placed there by the system, outside of programmer control. Likewise, the system decides when to overwrite data in the cache or in registers if it is not referenced in a while, and as other data needs to be placed there. Below, we will go into detail on how caches do this, but as a general principle, a Least Recently Used (LRU) cache replacement policy is used: if a cache is full and new data needs to be placed into it, the data that was least recently used is *flushed*, meaning that it is overwritten with the new item, and therefore no longer accessible. LRU is by far the most common replacement policy; other possibilities are FIFO (first in first out) or random replacement.

Exercise 1.6. Sketch a simple scenario, and give some (pseudo) code, to argue that LRU is preferable over FIFO as a replacement strategy.

1.2.4.3 Cache lines

Data is not moved between memory and cache, or between caches, in single bytes. Instead, the smallest unit of data moved is called a *cache line*. A typical cache line is 64 or 128 bytes long, which in the context of scientific computing implies 8 or 16 double precision floating point numbers. The cache line size for data moved into L2 cache can be larger than for data moved into L1 cache.

It is important to acknowledge the existence of cache lines in coding, since any memory access costs the transfer of several words (see section 1.5.3 for some practical discussion). An efficient program then tries to use the other items on the cache line, since access to

them is effectively free. This phenomenon is visible in code that accesses arrays by stride: elements are read or written at regular intervals.

Stride 1 corresponds to sequential access of an array:

```
for (i=0; i<N; i++)
  ... = ... x[i] ...
```

A larger stride

```
for (i=0; i<N; i+=stride)
  ... = ... x[i] ...
```

implies that in every cache line only certain elements are used; a stride of 4 with a cache line size of 4 double precision numbers implies that in every cache line only one number is used, and therefore that 3/4 of the used bandwidth has been wasted.

Some applications naturally lead to strides greater than 1, for instance, accessing only the real parts of an array of complex numbers; section 3.4.4. Also, methods that use recursive doubling often have a code structure that exhibits non-unit strides

```
for (i=0; i<N/2; i++)
  x[i] = y[2*i];
```

1.2.4.4 Cache mapping

Caches get faster, but also smaller, the closer to the floating point units they get; even the largest cache is consid-erably smaller than the main memory size. We already noted that this has implications for the cache replacement strategy. Another issue we need to address in this context is that of *cache mapping*, which is the question of 'if an item is placed in cache, where does it get placed'. This problem is generally addressed by mapping the (main memory) address of the item to an address in cache, leading to the question 'what if two items get mapped to the same address'.

1.2.4.5 Direct mapped caches

The simplest cache mapping strategy is *direct mapping*. Suppose that memory addresses are 32 bits long, so that they can address 4G bytes; suppose further that the cache has 8K words, that is, 64k bytes, needing 16 bits to address. Direct mapping then takes from each memory address the last ('least significant') 16 bits, and uses these as the address of the data item in cache.

Direct mapping is very efficient because of its address calculations can be performed very quickly, leading to low latency, but it has a problem in practical applications. If two items are addressed that are separated by 8K words, they will be mapped to the same cache location, which will make certain calculations inefficient. Example:

```
double A[3][8192]; int i;
for (i=0; i<512; i++)
  a[2][i] = ( a[0][i]+a[1][i] )/2.;
```

Here, the locations of a[0][i], a[0][i], and a[2][i] are 8K from each other for every i, so the last 16 bits of their addresses will be the same, and hence they will be mapped to the same location in cache. The execution of the loop will now go as follows:

- The data at a[0][0] is brought into cache and register. This engenders a certain amount of latency. Together with this element, a whole cache line is transferred.
- The data at a[1][0] is brought into cache (and register, as we will not remark anymore from now on), together with its whole cache line, at cost of some latency. Since this cache line is mapped to the same location as the first, the first cache line is overwritten.
- In order to write the output, the cache line containing a[2][0] is brought into memory. This is again mapped to the same location, causing flushing of the cache line just loaded for a[1][0].
- In the next iteration, a[0][1] is needed, which is on the same cache line as a[0][0]. However, this cache line has been flushed, so it needs to be brought in anew from main memory or a deeper cache level. In doing so, it overwrites the cache line that holds a[1][0].
- A similar story hold for a[1][1]: it is on the cache line of a[1][0], which unfortunately has been overwritten in the previous step.

If a cache line holds four words, we see that each four iterations of the loop involve eight transfers of elements of a, where two would have sufficed, if it were not for the cache conflicts.

Exercise 1.7. In the example of direct mapped caches, mapping from memory to cache was done by using the final 16 bits of a 32 bit memory address as cache address. Show that the problems in this example go away if the mapping is done by using the first ('most significant') 16 bits as the cache address. Why is this not a good solution in general?

1.2.4.6 Associative caches

The problem of cache conflicts, outlined in the previous section, would be solved if any data item could go to any cache location. In that case there would be no conflicts, other than the cache filling up, in which case a cache replacement policy (section 1.2.4.2) would flush data to make room for the incoming item. Such a cache is called *fully associative*, and while it seems optimal, it is also very costly to build, and much slower in use.

For this reason, the most common solution is to have a k-way associative cache, where k is at least two. In this case, a data item can go to any of k cache locations. Code would have to have a $k + 1$-way conflict before data would be flushed prematurely as in

the above example. In that example, a value of $k = 2$ would suffice, but in practice higher values are often encountered.

For instance, the Intel Woodcrest processor has

- an L1 cache of 32K bytes, that is 8-way set associative with a 64 byte cache line size;
- an L2 cache of 4M bytes, that is 8-way set associative with a 64 byte cache line size.

On the other hand, the AMD Barcelona chip has 2-way associativity for the L1 cache, and 8-way for the L2. A higher associativity ('way-ness') is obviously desirable, but makes a processor slower, since determining whether an address is already in cache becomes more complicated. For this reason, the associativity of the L1 cache, where speed is of the greatest importance, is typically lower than of the L2.

Exercise 1.8. Write a small cache simulator in your favourite language. Assume a k-way associative cache of 32 entries and an architecture with 16 bit addresses. Run the following experiment for $k = 1, 2, 4$:

1. Let k be the associativity of the simulated cache.
2. Write the translation from 16 bit address to $32/2$k32/2k bit cache address.
3. Generate 32 random machine addresses, and simulate storing them in cache.

Since the cache has 32 entries, optimally the 32 addresses can all be stored in cache. The chance of this actually happening is small, and often the data of one address will be removed from the cache when it conflicts with another address. Record how many addresses, out of 32, are actually stored in the cache. Do step 3 100 times, and plot the results; give median and average value, and the standard deviation. Observe that increasing the associativity improves the number of addresses stored. What is the limit behaviour? (For bonus points, do a formal statistical analysis.)

1.2.5 Prefetch streams

In the traditional von Neumann model (section 1.1), each instruction contains the location of its operands, so a CPU implementing this model would make a separate request for each new operand. In practice, often subsequent data items are adjacent or regularly spaced in memory. The memory system can try to detect such data patterns by looking at cache miss points, and request a *prefetch data stream*.

In its simplest form, the CPU will detect that consecutive loads come from two consecutive cache lines, and au-tomatically issue a request for the next following cache line. This process can be repeated or extended if the code makes an actual request for that third cache line. Since these cache lines are now brought from memory well before they are needed, prefetch has the possibility of eliminating the latency for all but the first couple of data items.

The concept of cache miss now needs to be revisited a little. From a performance point of view we are only interested in *stalls* on cache misses, that is, the case where the computation has to wait for the data to be brought in. Data that is not in cache, but can be brought in while other instructions are still being processed, is not a problem. If an 'L1 miss' is understood to be only a 'stall on miss', then the term 'L1 cache refill' is used to describe all cacheline loads, whether the processor is stalling on them or not.

Since prefetch is controlled by the hardware, it is also described as *hardware prefetch*. Prefetch streams can some-times be controlled from software, though often it takes assembly code to do so.

1.2.6 Memory banks

Above, we discussed issues relating to bandwdith. You saw that memory, and to a lesser extent caches, have a bandwidth that is less than what a processor can maximally absorb. The situation is actually even worse than the above discussion made it seem. For this reason, memory is often divided into *memory banks* that are interleaved: with four memory banks, words 0, 4, 8 . . . are in bank 0, words 1, 5, 9 . . . are in bank 1, et cetera.

Suppose we now access memory sequentially, then such 4-way interleaved memory can sustain four times the bandwidth of a single memory bank. Unfortunately, accessing by stride 2 will halve the bandwidth, and larger strides are even worse. In practice the number of memory banks will be higher, so that strided memory access with small strides will still have the full advertised bandwidth.

This concept of banks can also apply to caches. For instance, the cache lines in the L1 cache of the AMD Barcelona chip are 16 words long, divided into two interleaved banks of 8 words. This means that sequential access to the elements of a cache line is efficient, but strided access suffers from a deteriorated performance.

1.2.7 TLB and virtual memory

All of a program's data may not be in memory simultaneously. This can happen for a number of reasons:

- The computer serves multiple users, so the memory is not dedicated to any one user;
- The computer is running multiple programs, which together need more than the physically available memory;

- One single program can use more data than the available memory.

For this reason, computers use *Virtual memory*: if more memory is needed than is available, certain blocks of memory are written to disc. In effect, the disc acts as an extension of the real memory. This means that a block of data can be anywhere in memory, and in fact, if it is swapped in and out, it can be in different locations at different times. Swapping does not act on individual memory locations, but rather on *memory pages*: contiguous blocks of memory, from a few kilobytes to megabytes in size. (In an earlier generation of operating systems, moving memory to disc was a programmer's responsibility. Pages that would replace each other were called *overlays*.)

For this reason, we need a translation mechanism from the memory addresses that the program uses to the actual addresses in memory, and this translation has to be dynamic. A program has a 'logical data space' (typically starting from address zero) of the addresses used in the compiled code, and this needs to be translated during program execution to actual memory addresses. For this reason, there is a page table that specifies which memory pages contain which logical pages.

However, address translation by lookup in this table is slow, so CPUs have a *Translation Look-aside Buffer (TLB)*. The TLB is a cache of frequently used Page Table Entries: it provides fast address translation for a number of pages. If a program needs a memory location, the TLB is consulted to see whether this location is in fact on a page that is remembered in the TLB. If this is the case, the logical address is translated to a physical one; this is a very fast process. The case where the page is not remembered in the TLB is called a *TLB miss*, and the page lookup table is then consulted, if necessary bringing the needed page into memory. The TLB is (sometimes fully) associative (section 1.2.4.6), using an LRU policy (section 1.2.4.2).

A typical TLB has between 64 and 512 entries. If a program accesses data sequentially, it will typically alternate between just a few pages, and there will be no TLB misses. On the other hand, a program that access many random memory locations can experience a slowdown because of such misses.

Section 1.5.4 and appendix D.5 discuss some simple code illustrating the behaviour of the TLB.

[There are some complications to this story. For instance, there is usually more than one TLB. The first one is associated with the L2 cache, the second one with the L1. In the AMD Opteron, the L1 TLB has 48 entries, and is is fully (48-way) associative, while the L2 TLB has 512 entries, but is only 4-way associative. This means that there can actually be TLB conflicts. In the discussion above, we have only talked about the L2 TLB. The reason that this can be associated with the L2 cache, rather than with main memory, is that the translation from memory to L2 cache is deterministic.]

1.4 Locality and data reuse

By now it should be clear that there is more to the execution of an algorithm than counting the operations: the data
transfer involved is important, and can in fact dominate the cost. Since we have caches and registers, the amount of data transfer can be minimized by programming in such a way that data stays as close to the processor as possible.
Partly this is a matter of programming cleverly, but we can also look at the theoretical question: does the algorithm
allow for it to begin with.

1.4.1 Data reuse

In this section we will take a look at data reuse: are data items involved in a calculation used more than once, so
that caches and registers can be exploited? What precisely is the ratio between the number of operations and the
amount of data transferred.

We define the data reuse of an algorithm as follows:

If n is the number of data items that an algorithm operates on, and f(n) the number of operations it takes, then the reuse factor is f(n)/n.

Consider for example the vector addition

$$\forall_i : x_i \leftarrow x_i + y_i \quad \forall i : x_i \leftarrow x_i + y_i$$

This involves three memory accesses (two loads and one store) and one operation per iteration, giving a data reuse
of 1/3. The *axpy* (for 'a times x plus y') operation

$$\forall_i : x_i \leftarrow x_i + a \cdot y \quad \forall i : x_i \leftarrow x_i + a \cdot y$$

has two operations, but the same number of memory access since the one-time load of *a* is amortized. It is therefore more efficient than the simple addition, with a reuse of 2/3.

The inner product calculation

$$\forall_i : s \leftarrow s + x_i \cdot y_i \quad \forall i : s \leftarrow s + x_i \cdot y_i$$

is similar in structure to the axpy operation, involving one multiplication and addition per iteration, on two vectors and one scalar. However, now there are only two load operations, since *s* can be kept in register and only written
back to memory at the end of the loop. The reuse here is 1.

1.4.2 Matrix-matrix product

Next, consider the matrix-matrix product:

$\forall_{i,j}: c_{ij} = \sum_k a_{ik} b_{kj}$ ∀i,j:cij=∑kaikbkj.

This involves $3n^2$ data items and $2n^3$ operations, which is of a higher order. The data reuse is $O(n)$O(n), meaning that every data item will be used $O(n)$O(n) times. This has the implication that, with suitable programming, this operation has the potential of overcoming the bandwidth/clock speed gap by keeping data in fast cache memory.

Exercise 1.10. The matrix-matrix product, considered as operation, clearly has data reuse by the above definition. Argue that this reuse is not trivially attained by a simple implementation. What determines whether the naive implementation has reuse of data that is in cache?

[In this discussion we were only concerned with the number of operations of a given implementation, not the
mathematical operation. For instance, there are ways or performing the matrix-matrix multiplication and Gaussian
elimination algorithms in fewer than $O(n^3)$O(n3) operations. However, this requires a different implementation, which has its own analysis in terms of memory access and reuse.]

The matrix-matrix product is the heart of the 'LINPACK benchmark'. The benchmark may give an optimistic view of the performance of a computer: the matrix-matrix product is an operation that has considerable data reuse, so it is relatively insensitive to memory bandwidth and, for parallel computers, properties of the network. Typically, computers will attain 60–90% of their peak performance on the Linpack benchmark. Other benchmarks may give considerably lower figures.

1.4.3 Locality

Since using data in cache is cheaper than getting data from main memory, a programmer obviously wants to code in such a way that data in cache is reused. While placing data in cache is not under explicit programmer control, even from assembly language, in most CPUs, it is still possible, knowing the behaviour of the caches, to know what data is in cache, and to some extent to control it.

The two crucial concepts here are temporal locality and spatial locality. Temporal locality is the easiest to explain: this describes the use of a data element within a short time of its last use. Since most caches have a LRU replacement policy (section 1.2.4.2), if in between the two references less data has been referenced than the cache size, the element will still be in cache and therefore quickly accessible. With other replacement policies, such as random replacement, this guarantee can not be made.

As an example, consider the repeated use of a long vector:

```
for (loop=0; loop<10; loop++) {
  for (i=0; i<N; i++) {
    ... = ... x[i] ...
  }
}
```
Each element of x will be used 10 times, but if the vector (plus other data accessed) exceeds the cache size, each element will be flushed before its next use. Therefore, the use of x[i] does not exhibit temporal locality: subsequent uses are spaced too far apart in time for it to remain in cache.

If the structure of the computation allows us to exchange the loops:

```
for (i=0; i<N; i++) {
  for (loop=0; loop<10; loop++) {
    ... = ... x[i] ...
  }
}
```
the elements of x are now repeatedly reused, and are therefore more likely to remain in the cache. This rearranged code displays better temporal locality in its use of x[I].

The concept of spatial locality is slightly more involved. A program is said to exhibit spatial locality if it references memory that is 'close' to memory it already referenced. In the classical von Neumann architecture with only a processor and memory, spatial locality should be irrelevant, since one address in memory can be as quickly retrieved as any other. However, in a modern CPU with caches, the story is different. Above, you have seen two examples of spatial locality:

- Since data is moved in cache lines rather than individual words or bytes, there is a great benefit to coding in such a manner that all elements of the cacheline are used. In the loop

  ```
  for (i=0; i<N*s; i+=s) {
     ... x[i] ...
  }
  ```

  spatial locality is a decreasing function of the stride *s*.

  Let S be the cacheline size, then as s ranges from 1 . . . S, the number of elements used of each cacheline goes down from S to 1. Relatively speaking, this increases the cost of memory traffic in the loop: if $s = 1$, we load $1/S$ cachelines per element; if $s = S$, we load one cacheline for each element. This effect is demonstrated in section 1.5.3.

- A second example of spatial locality worth observing involves the TLB (section 1.2.7). If a program references elements that are close together, they are likely on the same memory page, and address translation will be fast. On the other hand, if a program references many widely disparate elements, it will also be referencing

many different pages. The resulting TLB misses are very costly; see also section 1.5.4.

Let us examine locality issues for a realistic example. The matrix-matrix multiplicatiom $C \leftarrow C+A \cdot BC \leftarrow C+A \cdot B$ can be computed in several ways. We compare two implementations, assuming that all matrices are stored by rows, and that the cache size is insufficient to store a whole row or column.

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] = c[i,j]
               + a[i,k]*b[k,j]

for i=1..n
  for k=1..n
    for j=1..n
      c[i,j] = c[i,j]
               + a[i,k]*b[k,j]
```

Our first observation is that both implementations indeed compute $C \leftarrow C+A \cdot BC \leftarrow C+A \cdot B$, and that they both take roughly $2n^3$ operations. However, their memory behaviour, including spatial and temporal locality is very different.

c[i,j]: In the first implementation, c[i,j] is invariant in the inner iteration, which constitutes temporal locality, so it can be kept in register. As a result, each element of *C* will be loaded and stored only once. In the second implementation, c[i,j] will be loaded and stored in each inner iteration. In particular, this implies that there are now n^3 store operations, a factor of n more than the first implementation.

a[i,k]: In both implementations, a[i,j] elements are accessed by rows, so there is good spatial locality, as each loaded cacheline will be used entirely. In the second implementation, a[i,k] is invariant in the inner loop, which constitutes temporal locality; it can be kept in register. As a result, in the second case *A* will be loaded only once, as opposed to *n* times in the first case.

b[k,j]: The two implementations differ greatly in how they access the matrix *B*. First of all, b[k,j] is never invariant so it will not be kept in register, and *B* engenders n^3 memory loads in both cases. However, the access patterns differ. In second case, b[k,j] is access by rows so there is good spatial locality: cachelines will be fully utilized after they are loaded. In the first implementation, b[k,j] is accessed by columns. Because of the row storage of the matrices, a cacheline contains a part of a row, so for each cacheline loaded, only one element is used in the columnwise traversal. This means that the first implementation has more loads for *B* by a factor of the cacheline length.

Note that we are not making any absolute predictions on code performance for these implementations, or even relative comparison of their runtimes. Such predictions are very

hard to make. However, the above discussion identifies issues that are relevant for a wide range of classical CPUs.

Exercise 1.11. Consider the following pseudocode of an algorithm for summing n numbers x[i] where *n* is a power of 2:

```
for s=2,4,8,...,n/2,n:
  for i=0 to n-1 with steps s:
    x[i] = x[i] + x[i+s/2]
sum = x[0]
```

Analyze the spatial and temporal locality of this algorithm, and contrast it with the standard algorithm

```
sum = 0
for i=0,1,2,...,n-1
  sum = sum + x[i]
```

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

6.2: Cache Architectures and Improving Cache Performance

Cache Organization

Watch this lecture on memory hierarchy design with caches. This lecture discusses the impact that memory operations have on overall processor performance and identifies different cache architectures that can improve overall processor performance. In the memory hierarchy, from top to bottom, there are: processor registers, cache, main memory, and secondary memory. The order goes from faster, smaller capacity, and higher cost to slower, higher capacity, and lower cost. Data moves from the main memory to cache. To do this requires a mapping from addresses in the main memory to cache addresses. How this is done affects performance. Performance analysis of cache and memory organization involves miss rate and miss penalty. Policies for reading, loading, fetching, replacing, and writing memory also affect cost and performance.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=6F6NP1lrRpc

Programming Strategies for High Performance

Read section 1.5, which discusses the relationship of pipelining and caching to programming.

1.5 Programming strategies for high performance

In this section we will look at how different ways of programming can influence the performance of a code. This will only be an introduction to the topic; for further discussion see the book by Goedeker and Hoisie [42].

The full listings of the codes and explanations of the data graphed here can be found in appendix D. All performance results were obtained on the AMD processors of the Ranger computer [10].

1.5.1 Pipelining

In section 1.1.1.1 you learned that the floating point units in a modern CPU are pipelined, and that pipelines require a number of independent operations to function efficiently. The typical pipelineable operation is a vector addition; an example of an operation that can not be pipelined is the inner product accumulation

```
for (i=0; i<N; i++)
  s += a[i]*b[i]
```

The fact that s gets both read and written halts the addition pipeline. One way to fill the *floating point pipeline* is to apply *loop unrolling*:

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += a[2*i] * b[2*i];
  sum2 += a[2*i+1] * b[2*i+1];
}
```

With a little indexing optimization this becomes:

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += *(a + 0) * *(b + 0);
  sum2 += *(a + 1) * *(b + 1);

  a += 2; b += 2;
}
```

A first observation about this code is that we are implicitly using associativity and commutativity of addition: while the same quantities are added, they are now in effect added in a different order. As you will see in chapter 3, in computer arithmetic this is not guaranteed to give the exact same result.

In a further optimization, we disentangle the addition and multiplication part of each instruction. The hope is that while the accumulation is waiting for the result of the multiplication, the intervening instructions will keep the processor busy, in effect increasing the number of operations per second.

```
for (i = 0; i < N/2-1; i ++) {
  temp1 = *(a + 0) * *(b + 0);
  temp2 = *(a + 1) * *(b + 1);

  sum1 += temp1; sum2 += temp2;

  a += 2; b += 2;
}
```

Finally, we realize that the furthest we can move the addition away from the multiplication, is to put it right in front of the multiplication of the next iteration:

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += temp1;
  temp1 = *(a + 0) * *(b + 0);

  sum2 += temp2;
  temp2 = *(a + 1) * *(b + 1);

a += 2; b += 2;
}

s = temp1 + temp2;
```

Of course, we can unroll the operation by more than a factor of two. While we expect an increased performance, large unroll factors need large numbers of registers. Asking for more registers than a CPU has is called *register spill*, and it will decrease performance.

Another thing to keep in mind is that the total number of operations is unlikely to be divisible by the unroll factor. This requires *cleanup code* after the loop to account for the final iterations. Thus, unrolled code is harder to write than straight code, and people have written tools to perform such *source-to-source transformations* automatically.

Cycle times for unrolling the inner product operation up to six times are given in table 1.2. Note that the timings do not show a monotone behaviour at the unrolling by four. This sort of variation is due to various memory-related factors.

1	2	3	4	5	6
6794	507	340	359	334	528

Table 1.2: Cycle times for the inner product operation, unrolled up to six times

1.5.2 Cache size

Above, you learned that data from L1 can be moved with lower latency and higher bandwidth than from L2, and L2 is again faster than L3 or memory. This is easy to demonstrate with code that repeatedly access the same data:

```
for (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[j]+1.2;
```

If the size parameter allows the array to fit in cache, the operation will be relatively fast. As the size of the dataset grows, parts of it will evict other parts from the L1 cache, so the speed of the operation will be determined by the latency and bandwidth of the L2 cache. This can be seen in figure 1.5. The full code is given in section D.2.

Exercise 1.12. Argue that with a LRU replacement policy (section 1.2.4.2) essentially all data in the L1 will be replaced in every iteration of the outer loop. Can you write an example code that will let some of the L1 data stay resident?

Often, it is possible to arrange the operations to keep data in L1 cache. For instance, in our example, we could write

```
for (i=0; i<NRUNS; i++) {
  blockstart = 0;
  for (b=0; b<size/l1size; b++)
    for (j=0; j<l1size; j++)
      array[blockstart+j] = 2.3*array[blockstart+j]+1.2;
```

assuming that the L1 size divides evenly in the dataset size. This strategy is called *cache blocking* or *blocking for cache reuse*.

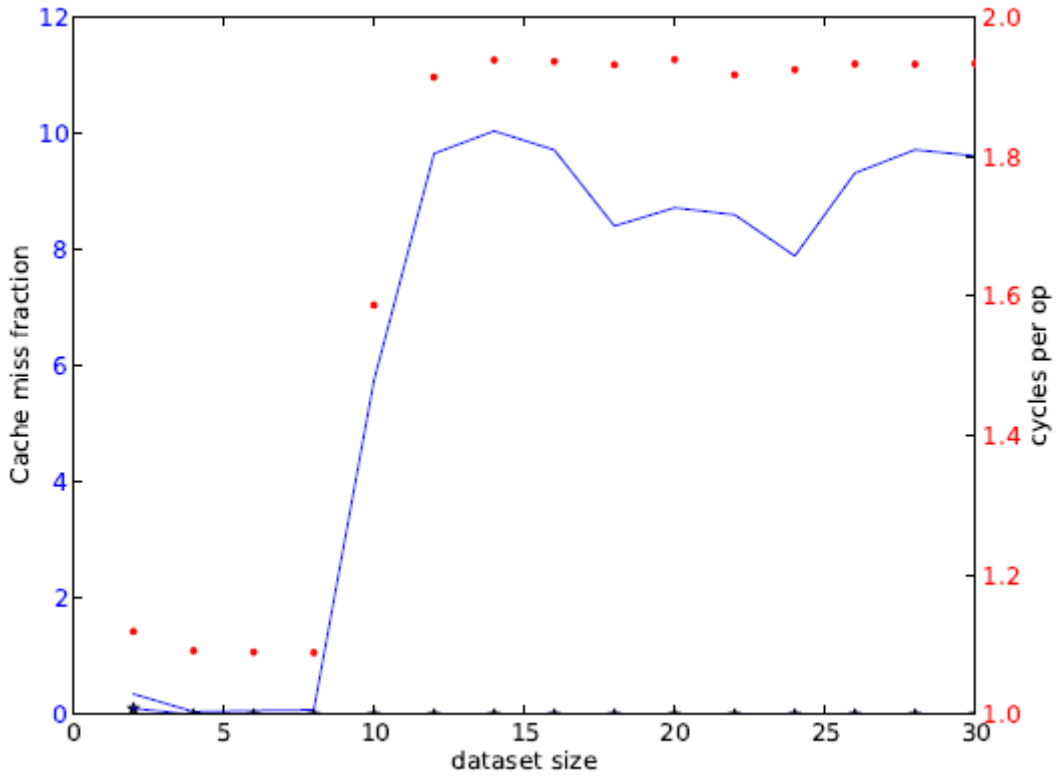


Figure 1.5: Average cycle count per operation as function of the dataset size

1.5.3 Cache lines

Since data is moved from memory to cache in consecutive chunks named cachelines (see section 1.2.4.3), code that does not utilize all data in a cacheline pays a bandwidth penalty. This is born out by a simple code

```
for (i=0,n=0; i<L1WORDS; i++,n+=stride)
  array[n] = 2.3*array[n]+1.2;
```

Here, a fixed number of operations is performed, but on elements that are at distance stride. As this stride increases, we expect an increasing runtime, which is born out by the graph in figure 1.6.

The graph also shows a decreasing reuse of cachelines, defined as the number of vector elements divided by the number of L1 misses (on stall; see section 1.2.5).

The full code is given in section D.3.

1.5.4 TLB

As explained in section 1.2.7, the Translation Look-aside Buffer (TLB) maintains a list of currently in use memory pages, and addressing data that is located on one of these pages is much faster than data that is not. Consequently, one wants to code in such a way that the number of pages in use is kept low.

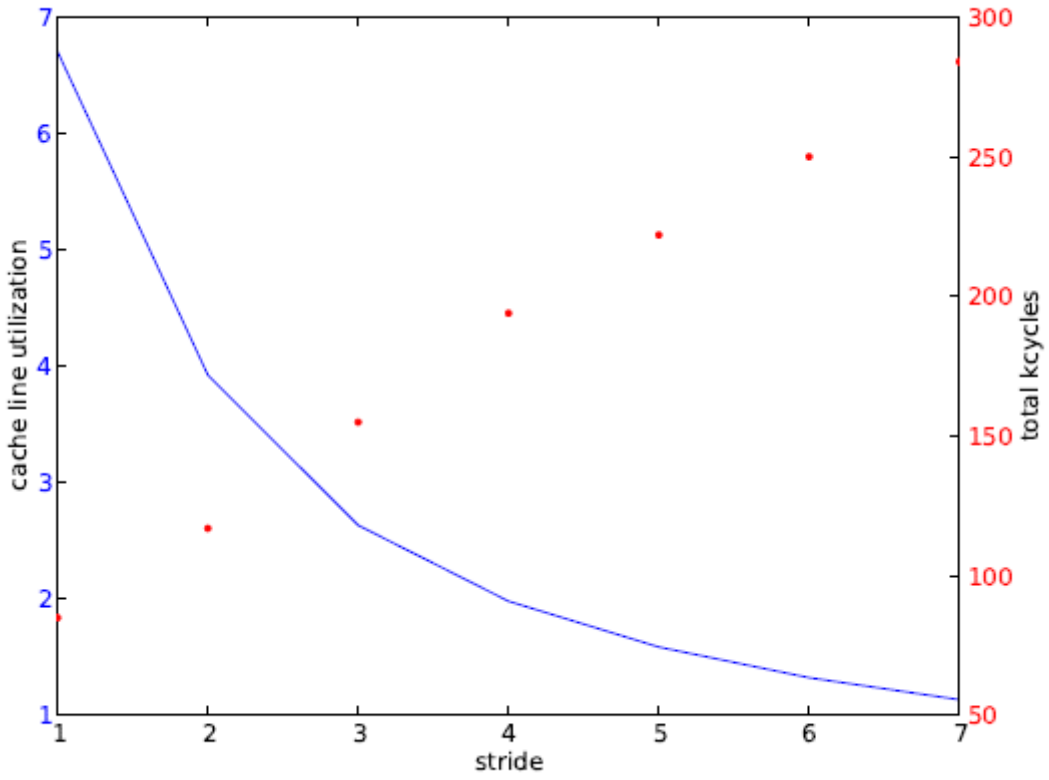


Figure 1.6: Run time in kcycles and L1 reuse as a function of stride

Consider code for traversing the elements of a two-dimensional array in two different ways.

```
#define INDEX(i,j,m,n) i+j*m
  array = (double*) malloc(m*n*sizeof(double));

  /* traversal #1 */
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;

  /* traversal #2 */
    for (i=0; i<m; i++)
      for (j=0; j<n; j++)
        array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```

The results (see Appendix D.5 for the source code) are plotted in figures 1.8 and 1.7.

Using $m = 1000$ means that, on the which has pages of 512 doubles, we need roughly two pages for each column. We run this example, plotting the number 'TLB misses', that is, the number of times a page is referenced that is not recorded in the TLB.

1. In the first traversal this is indeed what happens. After we touch an element, and the TLB records the page it is on, all other elements on that page are used subsequently, so no further TLB misses occur. Figure 1.8 shows that, with increasing n, the number of TLB misses per column is roughly two.
2. In the second traversal, we touch a new page for every element of the first row. Elements of the second row will be on these pages, so, as long as the number of columns is less than the number of TLB entries, these pages will still be recorded in the TLB. As the number of columns grows, the number of TLB increases, and ultimately there will be one TLB miss for each element access. Figure 1.7 shows that, with a large enough number of columns, the number of TLB misses per column is equal to the number of elements per column.

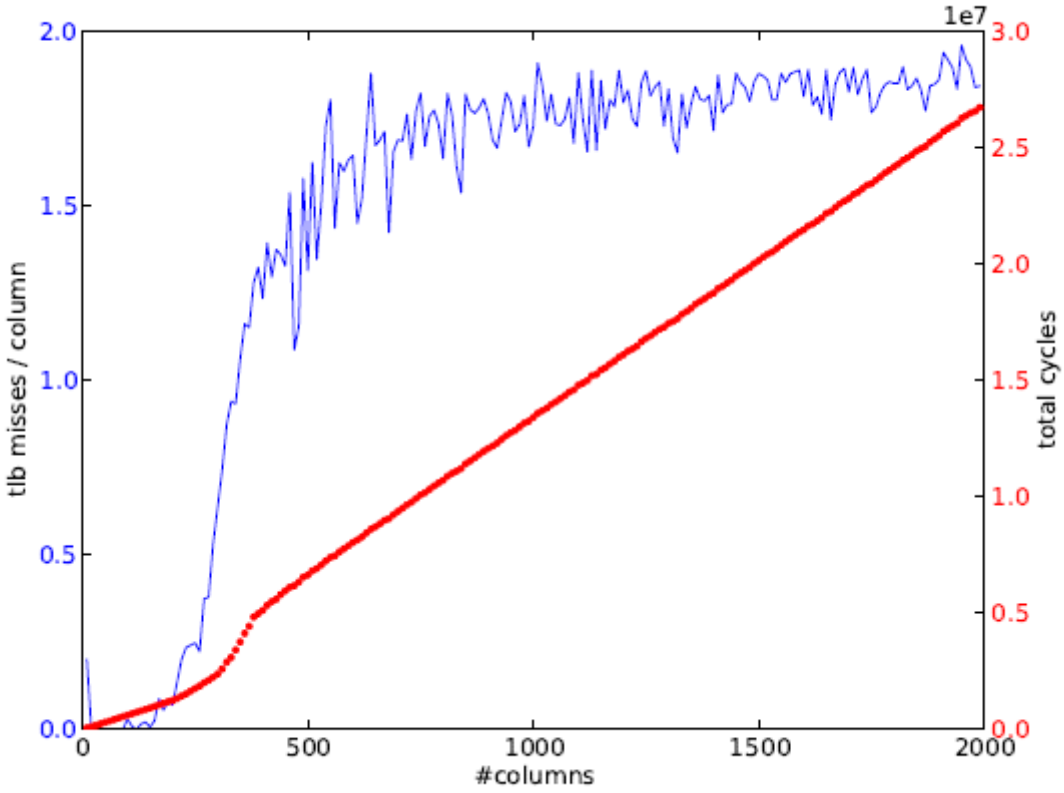


Figure 1.7: Number of TLB misses per column as function of the number of columns; columnwise traversal of the array.

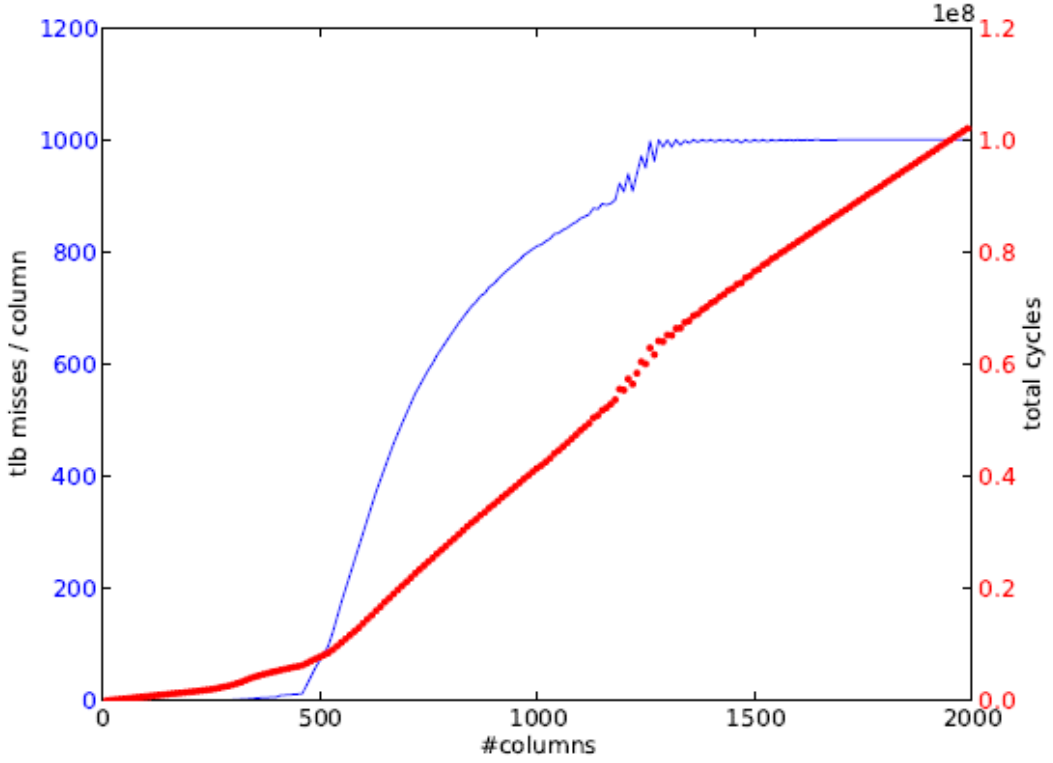


Figure 1.8: Number of TLB misses per column as function of the number of columns; rowwise traversal of the array.

1.5.5 Cache associativity

There are many algorithms that work by recursive division of a problem, for instance the *Fast Fourier Transform (FFT)* algorithm. As a result, code for such algorithms will often operate on vectors whose length is a power of two. Unfortunately, this can cause conflicts with certain architectural features of a CPU, many of which involve powers of two.

Consider the operation of adding a small number of vectors

$$\forall j: y_j = y_j + \sum_{i=1}^{m} x_{i,j} \quad \forall j: y_j = y_j + \sum_{i=1}^{m} x_{i,j}$$

If the length of the vectors y, x_i is precisely the right (or rather, wrong) number, y_j and $x_{i,j}$ will all be mapped to the same location in cache. As an example we take the AMD , which has an L1 cache of 64K bytes, and which is two-way set associative. Because of the set associativity, the cache can handle two addresses being mapped to the same cache location, but not three of more. Thus, we let the vectors be of size $n = 4096$ doubles, and we measure the effect in cache misses and cycles of letting $m = 1, 2 \ldots$

First of all, we note that we use the vectors sequentially, so, with a cacheline of eight doubles, we should ideally see a cache miss rate of 1/8 times the number of vectors m.

Instead, in figure 1.9 we see a rate approximately proportional to *m*, meaning that indeed cache lines are evicted immediately. The exception here is the case = 1, where the two-way associativity allows the cachelines of two vectors to stay in cache.

Compare this to figure 1.10, where we used a slightly longer vector length, so that locations with the same *j* are no longer mapped to the same cache location. As a result, we see a cache miss rate around 1/8, and a smaller number of cycles, corresponding to a complete reuse of the cache lines.

Two remarks: the cache miss numbers are in fact lower than the theory predicts, since the processor will use prefetch streams. Secondly, in figure 1.10 we see a decreasing time with increasing *m*; this is probably due to a progressively more favourable balance between load and store operations. Store operations are more expensive than loads, for various reasons.

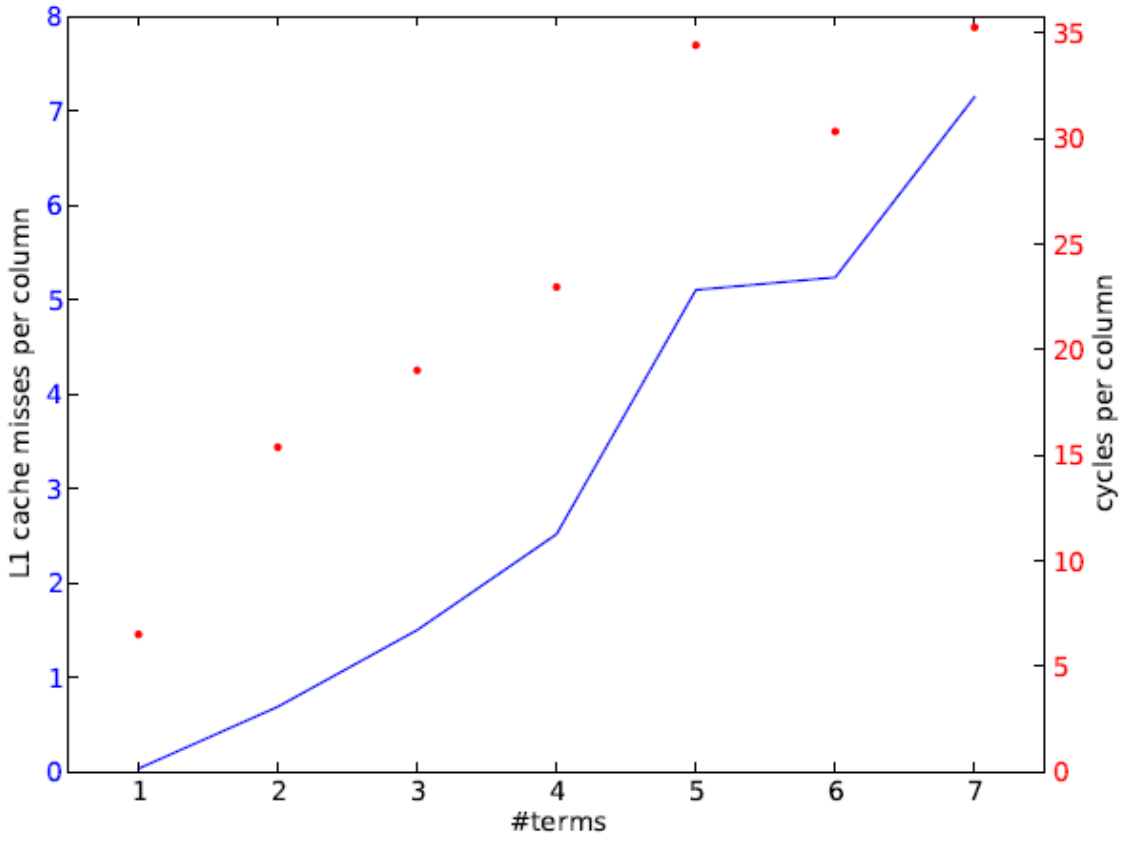


Figure 1.9: The number of L1 cache misses and the number of cycles for each j *column accumulation, vector length 4096*

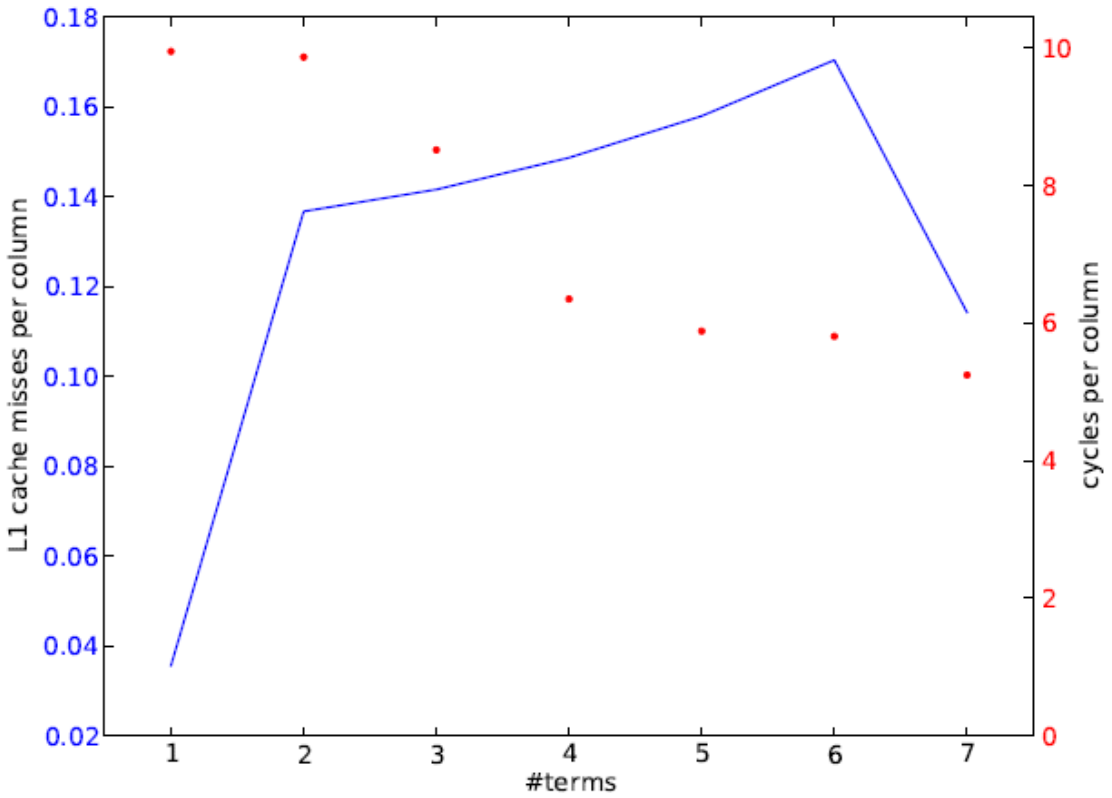


Figure 1.10: The number of L1 cache misses and the number of cycles for each j *column accumulation, vector length 4096 + 8*

1.5.6 Loop tiling

```
for (n=0; n<10; n++)
  for (i=0; i<100000; i++)
    x[i] = ...

for (b=0; b<100; b++)
  for (n=0; n<10; n++)
    for (i=b*1000; i<(b+1)*1000; i++)
      x[i] = ...
```

1.5.7 Case study: Matrix-vector product

Let us consider in some detail the matrix-vector product

$\forall_{i,j}: y_i \leftarrow a_{ij} \cdot x_j \forall_{i,j}: y_i \leftarrow a_{ij} \cdot x_j$

This involves $2n^2 2n^2$ operations on $n^2+2n n^2+2n$ data items, so reuse is $O(1) O(1)$: memory accesses and operations are of the same order. However, we note that there is a double loop involved, and the $x,y x,y$vectors have only a single index, so each element in them is used multiple times.

Exploiting this theoretical reuse is not trivial. In

```
/* variant 1 */
for (i)
  for (j)
    y[i] = y[i] + a[i][j] * x[j];
```
the element y[i] seems to be reused. However, the statement as given here would write y[i] to memory in every inner iteration, and we have to write the loop as

```
/* variant 2 */
for (i) {
  s = 0;
  for (j)
    s = s + a[i][j] * x[j];
  y[i] = s;
}
```
to ensure reuse. This variant uses $2n^2$ loads and n stores.

This code fragment only exploits the reuse of *y* explicitly. If the cache is too small to hold the whole vector *x* plus a column of *a*, each element of *a* is still repeatedly loaded in every outer iteration.

Reversing the loops as

```
/* variant 3 */
for (j)
  for (i)
    y[i] = y[i] + a[i][j] * x[j];
```
exposes the reuse of *x*, especially if we write this as

```
/* variant 3 */
for (j) {
  t = x[j];
  for (i)
    y[i] = y[i] + a[i][j] * t;
}
```
but now *y* is no longer reused. Moreover, we now have $2n^2+n$ loads, comparable to variant 2, but n^2 stores, which is of a higher order.

It is possible to get reuse both of *x* and *y*, but this requires more sophisticated programming. The key here is split the loops into blocks. For instance:

```
for (i=0; i<M; i+=2) {
  s1 = s2 = 0;
  for (j) {
    s1 = s1 + a[i][j] * x[j];
    s2 = s2 + a[i+1][j] * x[j];
  }
  y[i] = s1; y[i+1] = s2;
}
```
This is also called *loop unrolling*, *loop tiling*, or *strip mining*. The amount by which you unroll loops is determined by the number of available registers.

1.5.8 Optimization strategies

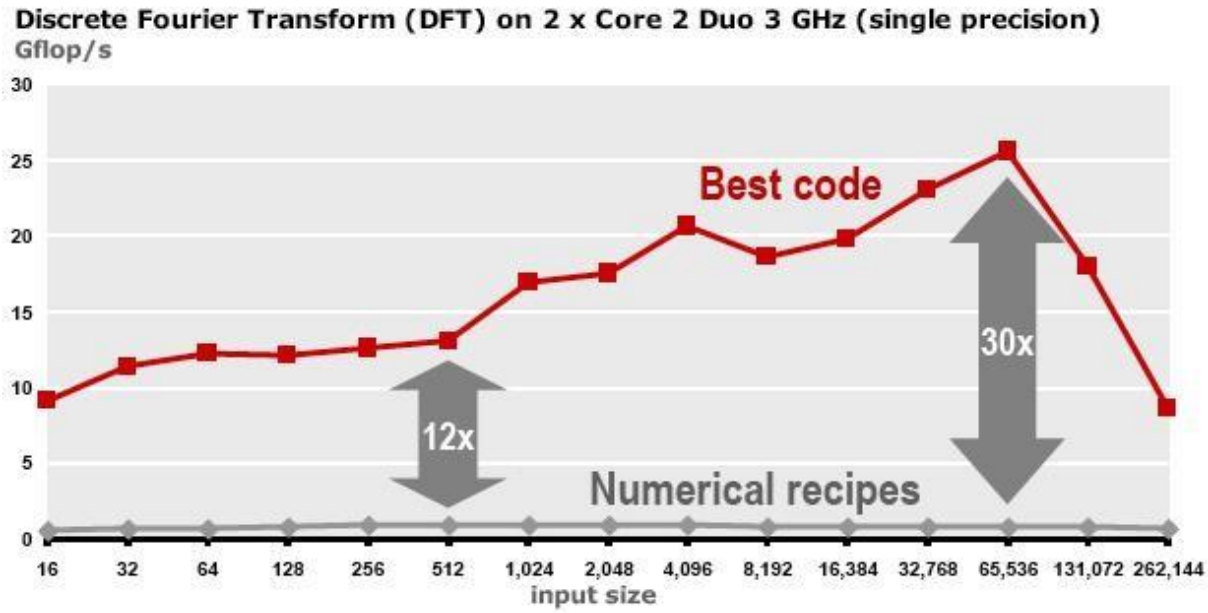


Figure 1.11: Performance of naive and optimized implementations of the Discrete Fourier Transform

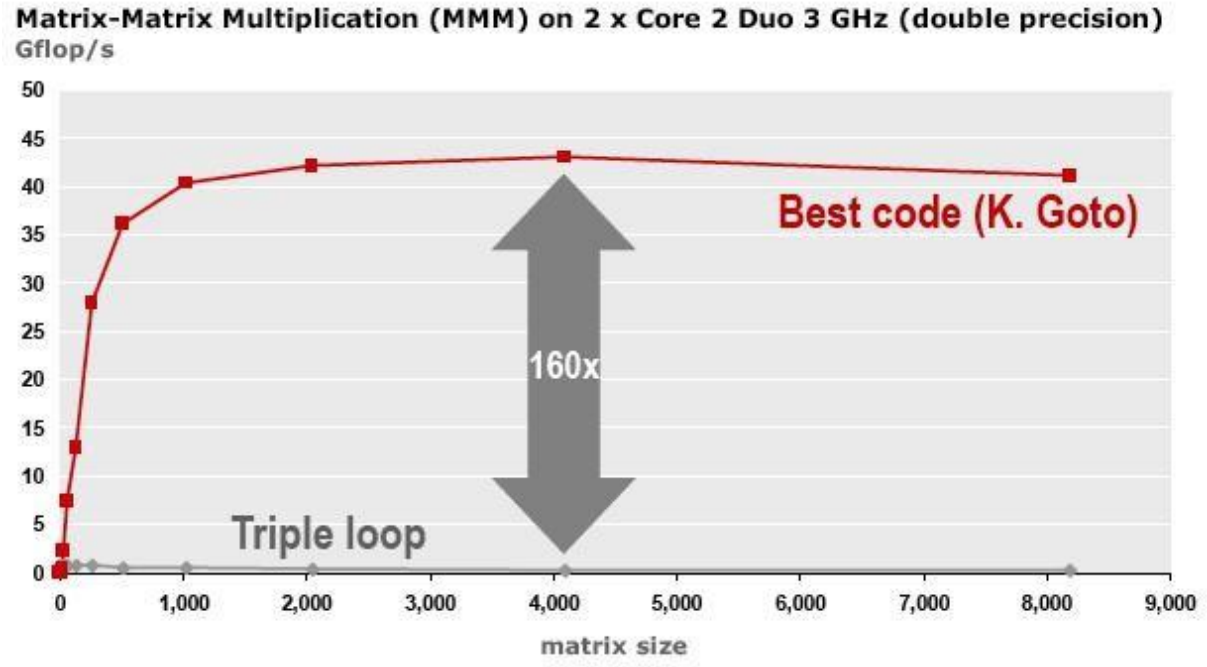


Figure 1.12: Performance of naive and optimized implementations of the matrix-matrix product

Figures 1.11 and 1.12 show that there can be wide discrepancy between the performance of naive implementations of an operation (sometimes called the 'reference implementation'), and optimized implementations. Unfortunately, optimized implementations are not simple to find. For one, since they rely on blocking, their loop

nests are dou-ble the normal depth: the matrix-matrix multiplication becomes a six-deep loop. Then, the optimal block size is dependent on factors like the target architecture.

We make the following observations:

- Compilers are not able to extract anywhere close to optimal performance8.
- There are *autotuning* projects for automatic generation of implementations that are tuned to the architec-ture. This approach can be moderately to very successful. Some of the best known of these projects are Atlas [84] for Blas kernels, and Spiral [73] for transforms.

1.5.9 Cache aware programming

Unlike registers and main memory, both of which can be addressed in (assembly) code, use of caches is implicit. There is no way a programmer can load data explicitly to a certain cache, even in assembly language.

However, it is possible to code in a 'cache aware' manner. Suppose a piece of code repeatedly operates on an amount of data that less data than the cache size. We can assume that the first time the data is accessed, it is brought into cache; the next time it is accessed it will already be in cache. On the other hand, if the amount of data is more than the cache size, it will partly or fully be flushed out of cache in the process of accessing it.

We can experimentally demonstrate this phenomenon. With a very accurate counter, the code fragment

```
for (x=0; x<NX; x++)
  for (i=0; i<N; i++)
    a[i] = sqrt(a[i]);
```

will take time linear in N up to the point where a fills the cache. An easier way to picture this is to compute a normalized time, essentially a time per execution of the inner loop:

```
t = time();
for (x=0; x<NX; x++)
  for (i=0; i<N; i++)
    a[i] = sqrt(a[i]);
t = time()-t;
t_normalized = t/(N*NX);
```

The normalized time will be constant until the array a fills the cache, then increase and eventually level off again.

The explanation is that, as long as a[0]...a[N-1] fit in L1 cache, the inner loop will use data from the L1 cache. Speed of access is then determined by the latency and bandwidth of the L1 cache. As the amount of data grows beyond the L1 cache size, some or all of the data will be flushed from the L1, and performance will be determined by the characteristics of the L2 cache. Letting the amount of data grow even further, performance will again drop to a linear behaviour determined by the bandwidth from main memory.

1.5.10 Arrays and programming languages

In section B.10.1.1 you can find a discussion of the different ways arrays are stored in C/C++ and Fortran. These storage modes have some ramifications on performance. Both from a point of cache line usage (section 1.5.3) and prevention of TLB misses (section 1.5.4) it is best to traverse a multi-dimensional array so as to access sequential memory locations, rather than strided. This means that

- In Fortran you want to loop through an array first by columns (that is, in the inner loop), then by rows (in the outer loop);
- In C/C++ you want to loop first through rows (inner loop), then through columns (outer loop).

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

6.3: Main Memory and Virtual Memory

Virtual Memory

Watch this lecture, which addresses the subject of virtual memory. This topic pertains to the relationship of main memory and secondary memory and has similarities and differences to the relationship of cache and main memory. Differences arise due to the speed of the various memories and their capacities. This lecture discusses virtual memory, mapping from virtual memory addresses to physical addresses in main and paging techniques used with the main memory. The lecture discusses the use of page tables in translating virtual addresses to physical addresses. Issues that arise with page tables include structure, location, and large size.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=cIlKSD8ptAk

6.4: Performance Tuning

Parallel Computing

Read Chapters 1–5. Before reading these chapters, list the factors that you can think of that can affect performance, like memory performance, cache, memory hierarchy, multi-cores, and so on, and what you might suggest as ways to increase performance. After reading these chapters, what might you add, if anything, to your list?

Abstract

This is is the first tutorial in the "Livermore Computing Getting Started" workshop. It is intended to provide only a very quick overview of the extensive and broad topic of

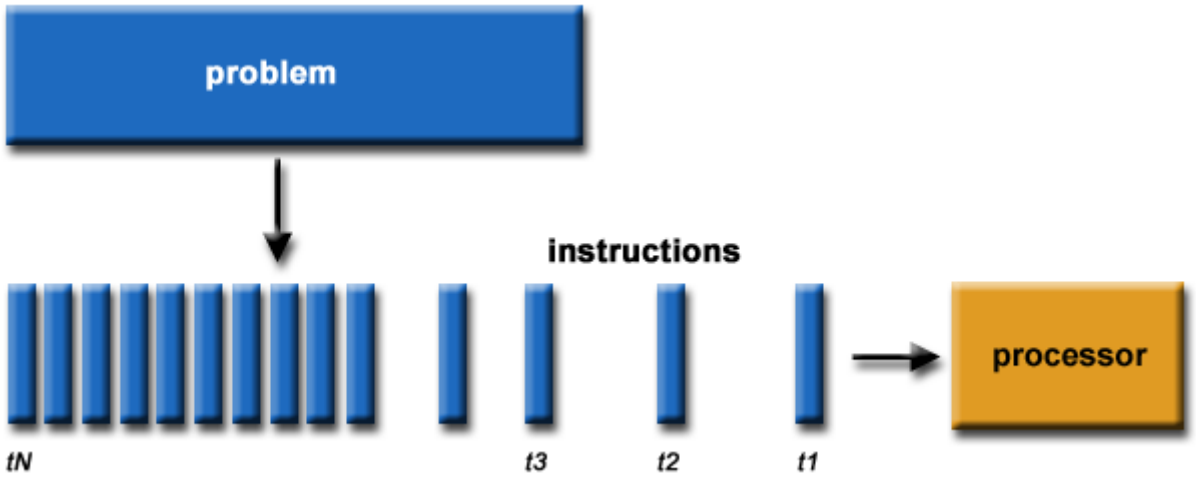
Parallel Computing, as a lead-in for the tutorials that follow it. As such, it covers just the very basics of parallel computing, and is intended for someone who is just becoming acquainted with the subject and who is planning to attend one or more of the other tutorials in this workshop. It is not intended to cover Parallel Programming in depth, as this would require significantly more time. The tutorial begins with a discussion on parallel computing - what it is and how it's used, followed by a discussion on concepts and terminology associated with parallel computing. The topics of parallel memory architectures and programming models are then explored. These topics are followed by a series of practical discussions on a number of the complex issues related to designing and running parallel programs. The tutorial concludes with several examples of how to parallelize simple serial programs.

What is Parallel Computing?

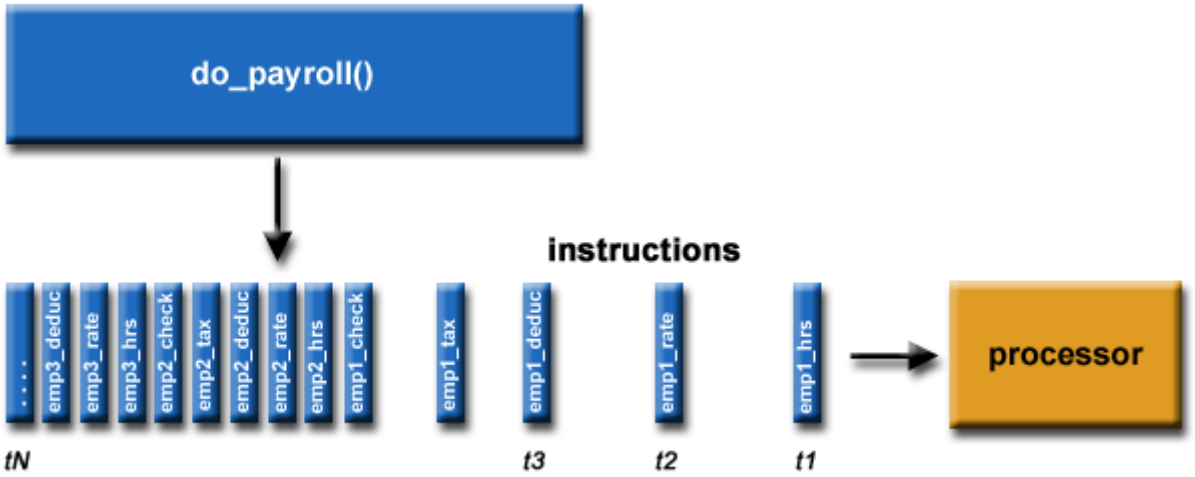
Serial Computing:

Traditionally, software has been written for ***serial*** computation:

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

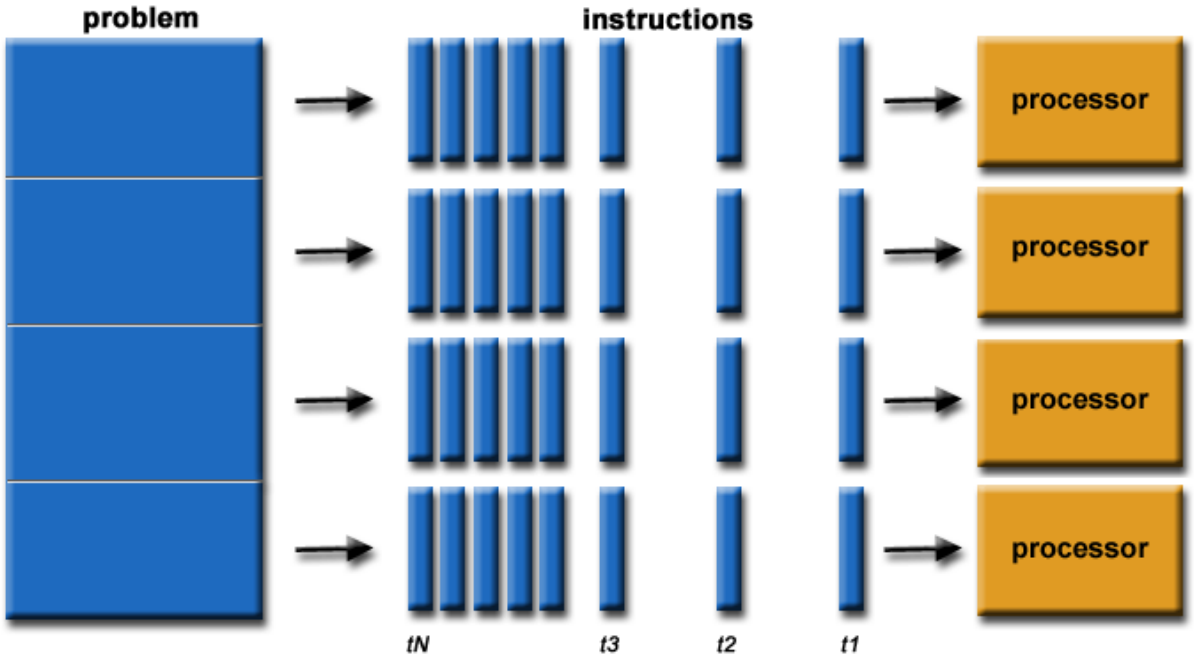


For example:

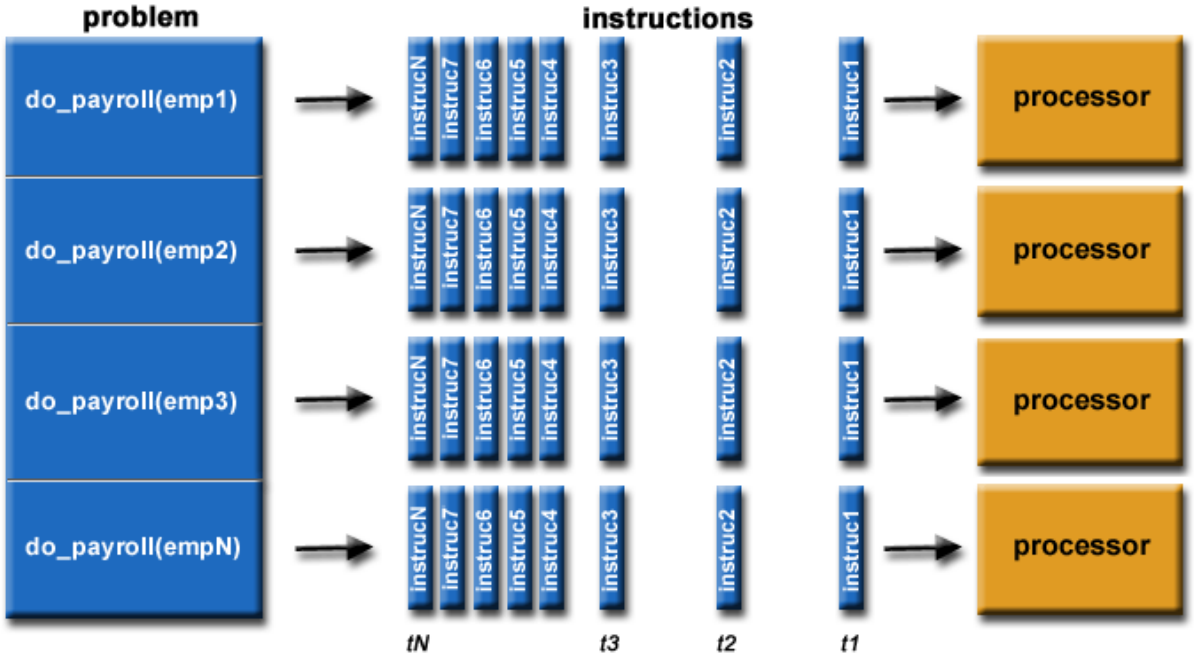


Parallel Computing:

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different processors
  - An overall control/coordination mechanism is employed



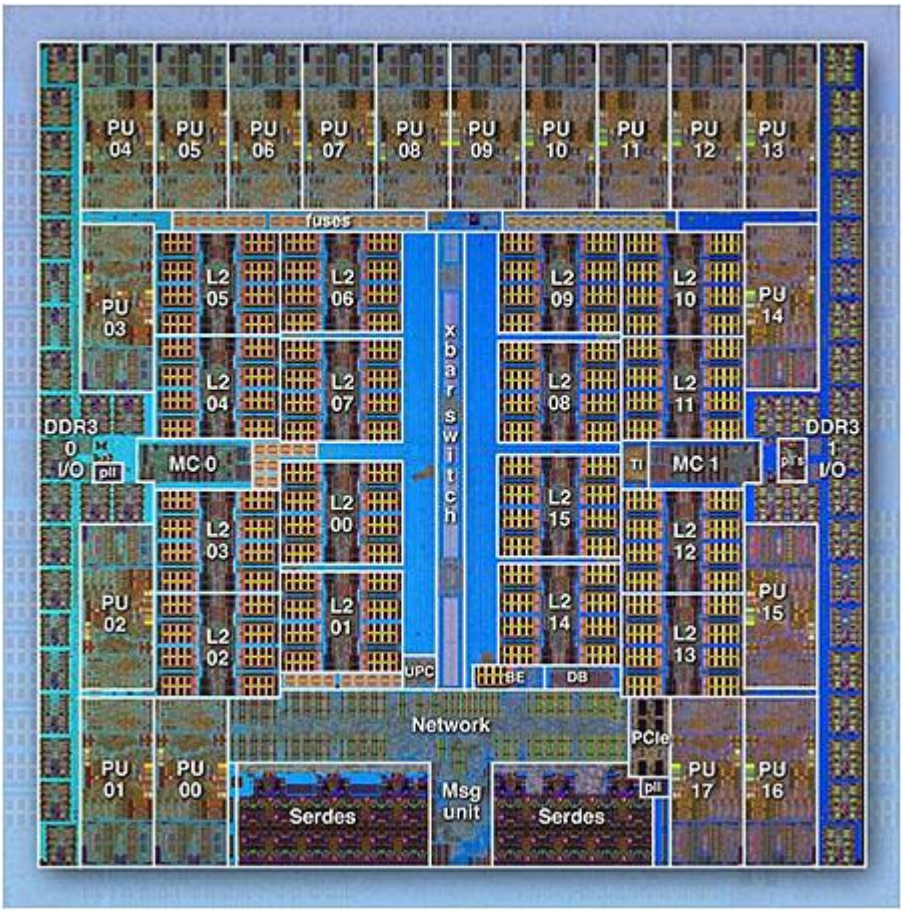
For example:



- The computational problem should be able to:
  -
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

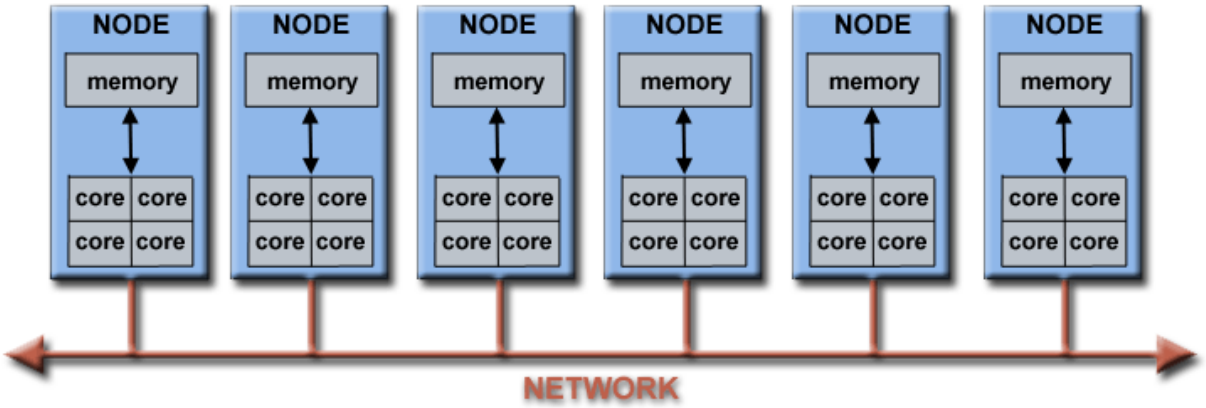
Parallel Computers:

- Virtually all stand-alone computers today are parallel from a hardware perspective:
  - Multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
  - Multiple execution units/cores
  - Multiple hardware threads



IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

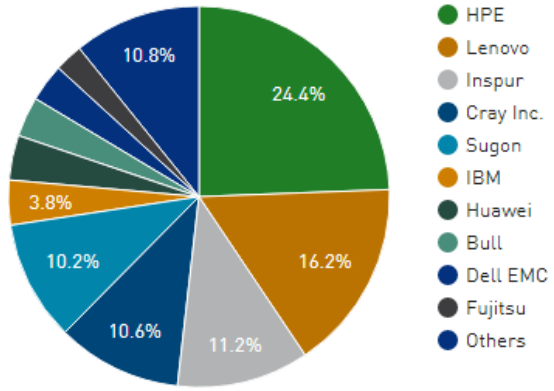
- Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



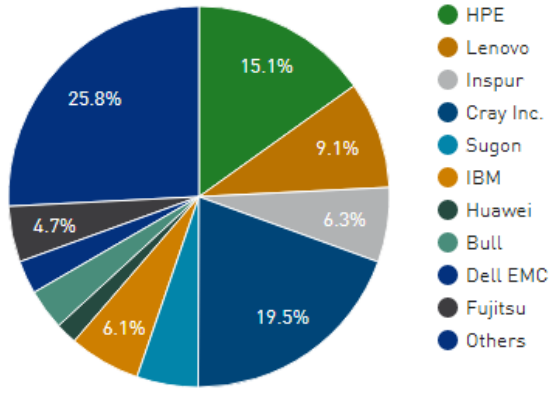
- For example, the schematic below shows a typical LLNL parallel computer cluster:
  - Each compute node is a multi-processor parallel computer in itself
  - Multiple compute nodes are networked together with an Infiniband network
  - Special purpose nodes, also multi-processor, are used for other purposes

- The majority of the world's large parallel computers (supercomputers) are clusters of hardware produced by a handful of (mostly) well known vendors.

Vendors System Share



Vendors Performance Share



Vendors	Count	System Share (%)
HPE	122	24.4
Lenovo	81	16.2
Inspur	56	11.2
Cray Inc.	53	10.6
Sugon	51	10.2
IBM	19	3.8
Huawei	19	3.8
Bull	17	3.4
Dell EMC	16	3.2
Fujitsu	12	2.4
Penguin Computing	10	2
NUDT	4	0.8
PEZY Computing / Exascaler Inc.	4	0.8
NEC	4	0.8
Atipa	3	0.6
Lenovo/IBM	3	0.6
Nvidia	2	0.4
Dell EMC / IBM-GBS	2	0.4
T-Platforms	2	0.4
IBM/Lenovo	2	0.4
Self-made	1	0.2
T-Platforms, Intel, Dell	1	0.2
ClusterVision	1	0.2
Supermicro	1	0.2
ExaScaler	1	0.2
E4 Computer Engineering S.p.A.	1	0.2
RSC Group	1	0.2

Source: Top500.org

Why Use Parallel Computing?

The Real World is Massively Parallel:

- In the natural world, many complex, interrelated events are happening at the same time, yet within a tempor
- Compared to serial computing, parallel computing is much better suited for modeling, simulating and underst real world phenomena.
- For example, imagine modeling these serially

Auto Assembly

Jet Construction

Drive-thru Lunch

Rush Hour Traffic

Plate Tectonics

Weather

Auto Assembly

Jet Construction

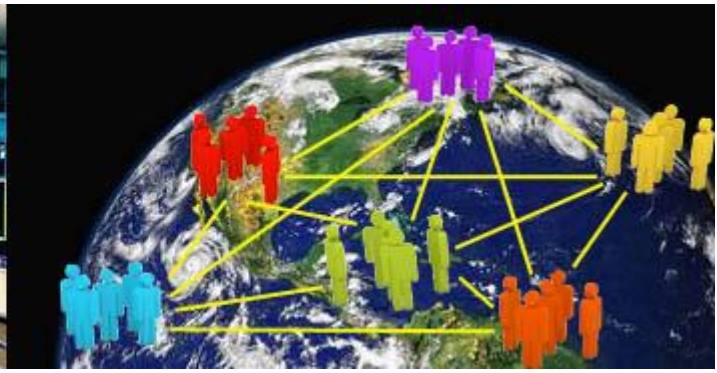
Drive-thru Lunch

Main Reasons:

- **SAVE TIME AND/OR MONEY:**
  - In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings.
  - Parallel computers can be built from cheap, commodity components.


- **SOLVE LARGER / MORE COMPLEX PROBLEMS:**
  - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
  - Example: "Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge) requiring PetaFLOPS and PetaBytes of computing resources.
  - Example: Web search engines/databases processing millions of transactions every second

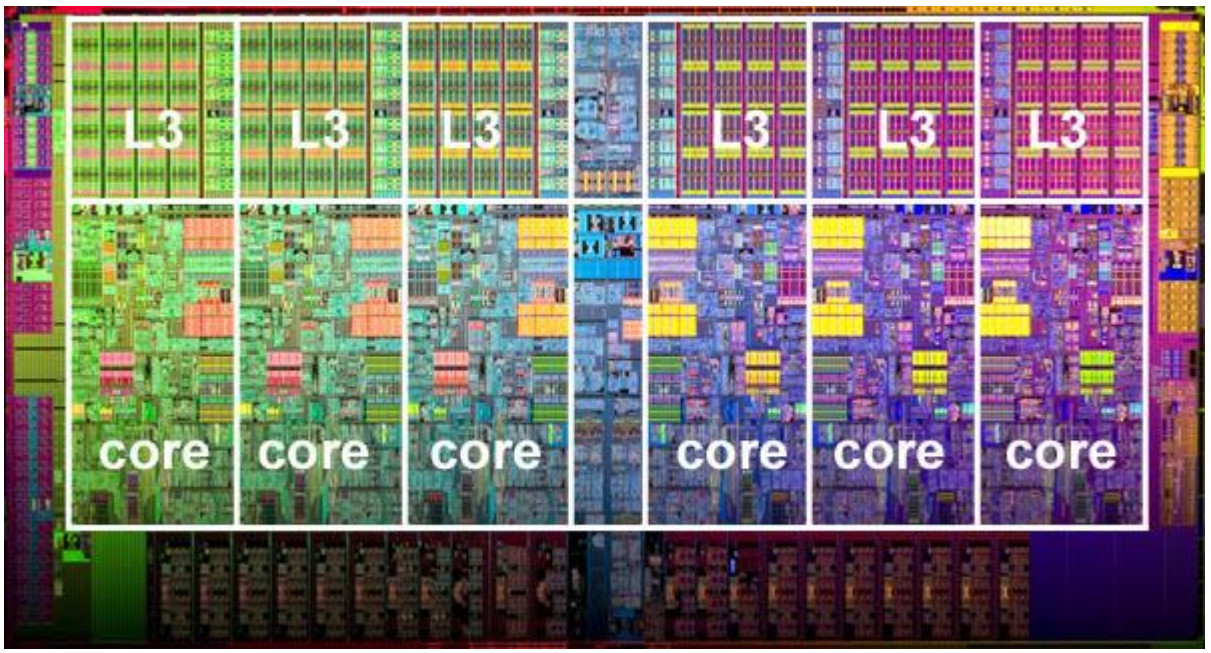
- **PROVIDE CONCURRENCY:**
  - A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously.
  - Example: Collaborative Networks provide a global venue where people from around the world can meet and conduct work "virtually".



- **TAKE ADVANTAGE OF NON-LOCAL RESOURCES:**
  - Using compute resources on a wide area network, or even the Internet when local compute resources are scarce or insufficient. Two examples below, each of which has over 1.7 million contributors globally (May 2018):
  - Example: SETI@home (setiathome.berkeley.edu)

- Example: Folding@home (folding.stanford.edu)

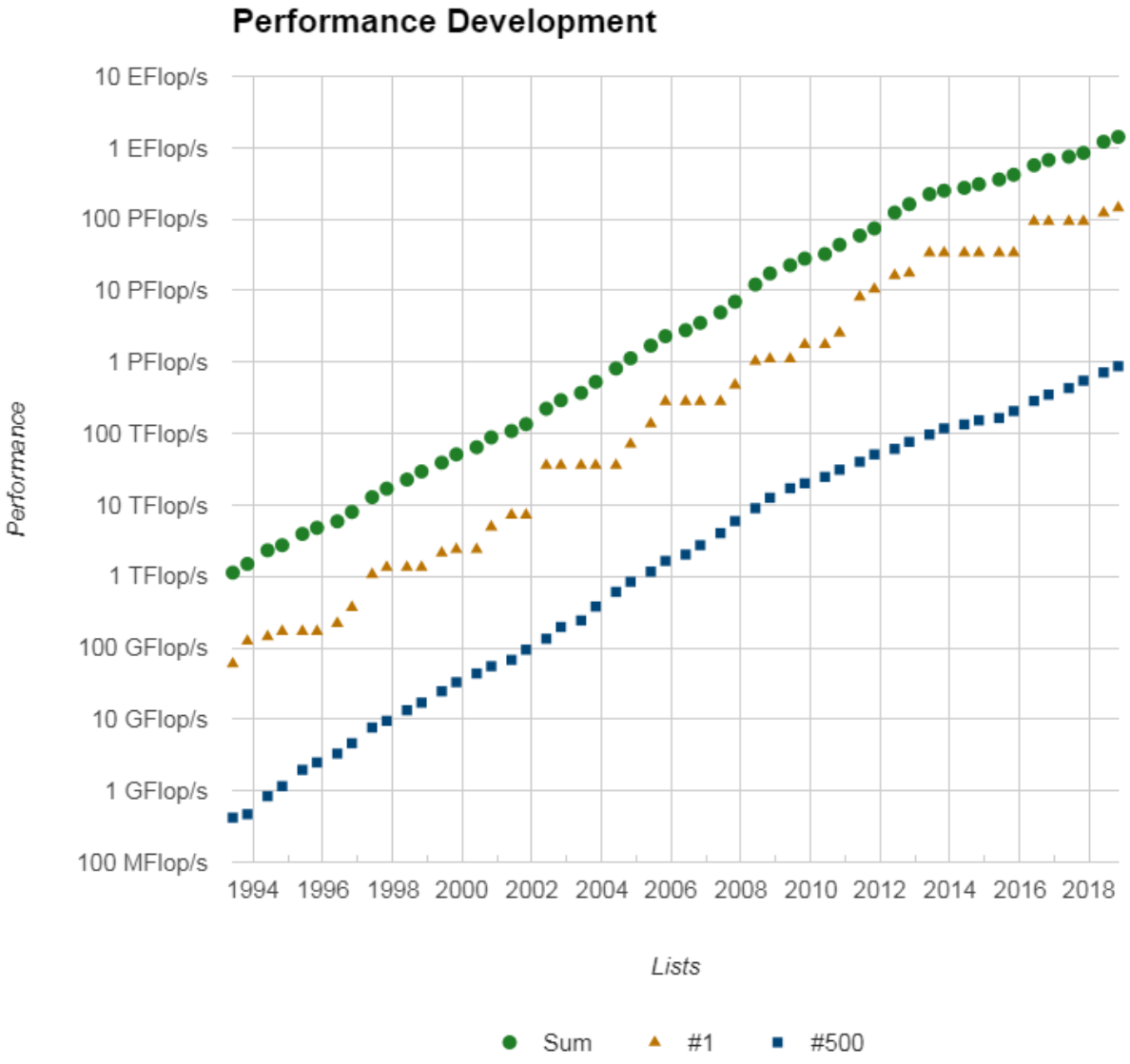
- **MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE:**
  - Modern computers, even laptops, are parallel in architecture with multiple processors/cores.
  - Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.
  - In most cases, serial programs run on modern computers "waste" potential computing power.



Intel Xeon processor with 6 cores and 6 L3 cache units

The Future:

- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***.
- In this same time period, there has been a greater than **500,000x** increase in supercomputer performance, with no end currently in sight.
- ***The race is already on for Exascale Computing!***
  - Exaflop = 10^{18} calculations per second



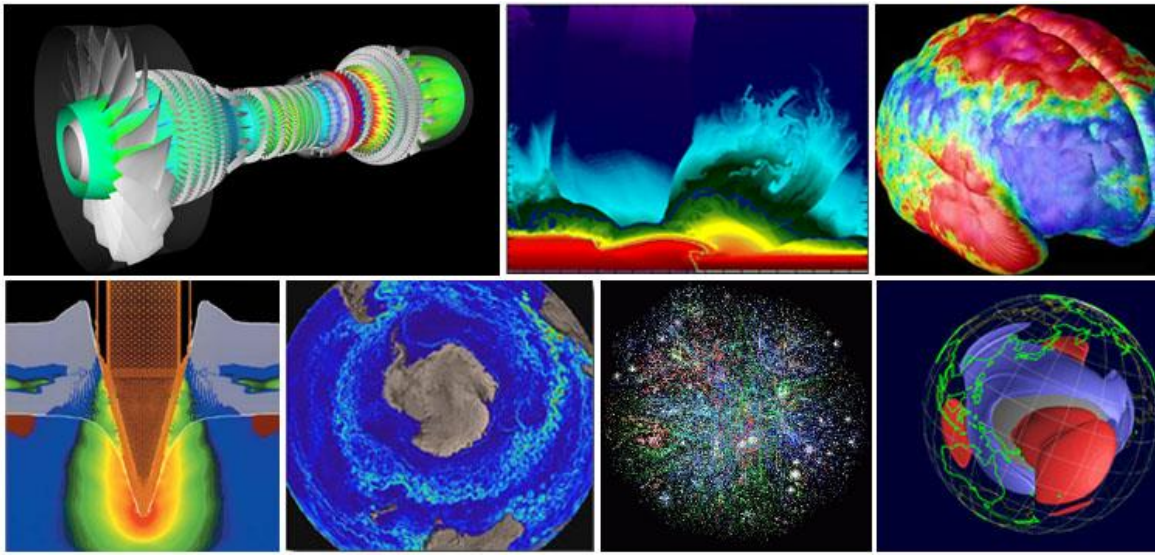
Source: Top500.org

Who is Using Parallel Computing?

Science and Engineering:

- Historically, parallel computing has been considered to be "the high end of computing", and has been used to problems in many areas of science and engineering:

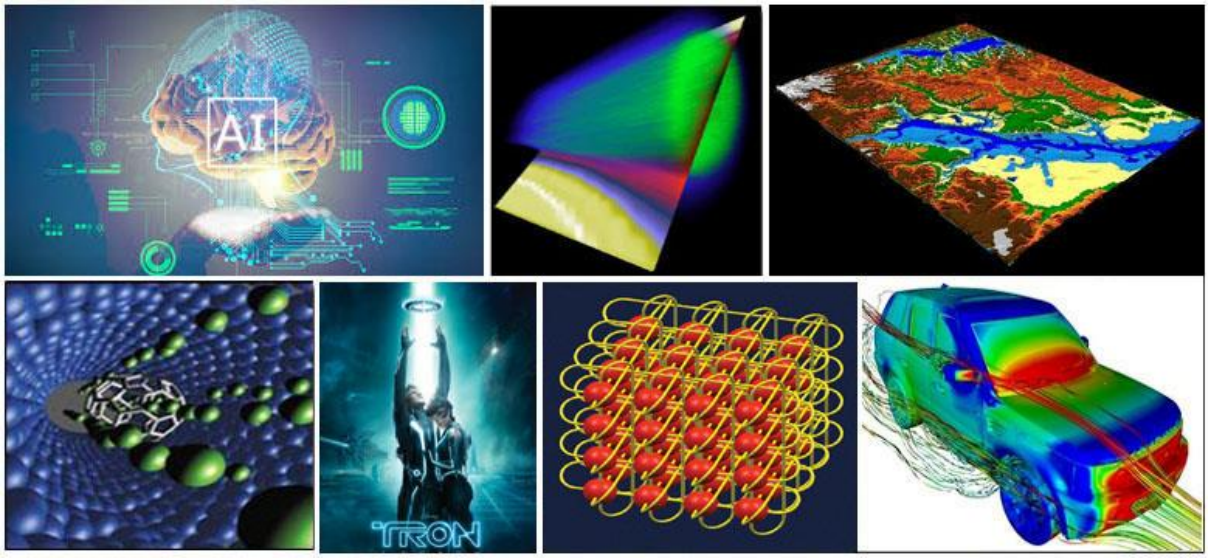
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology
  - Mechanical Engineering - fr to spacecraft
  - Electrical Engineering, Circu Microelectronics
  - Computer Science, Mathem
  - Defense, Weapons



Industrial and Commercial:

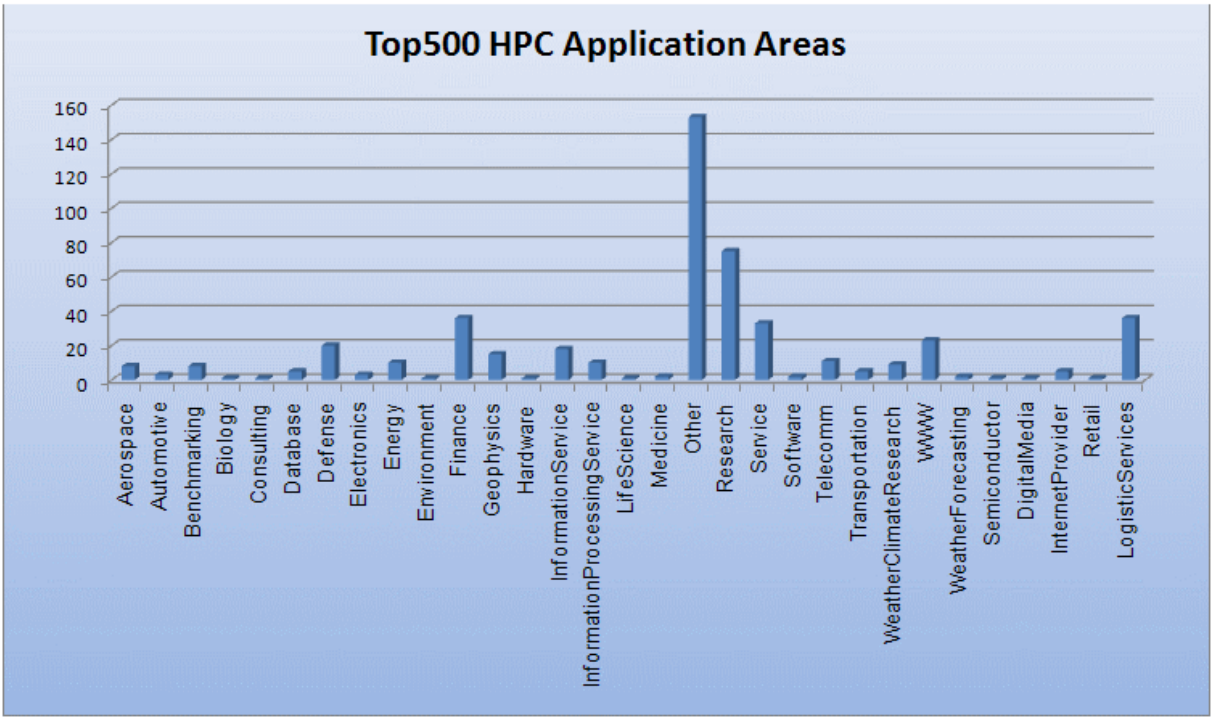
- Today, commercial applications provide an equal or greater driving force in the development of faster compu applications require the processing of large amounts of data in sophisticated ways. For example:

  - "Big Data", databases, data mining
  - Artificial Intelligence (AI)
  - Web search engines, web based business services
  - Medical imaging and diagnosis
  - Pharmaceutical design
  - Financial and economic modeling
  - Management of national and multi-national corpo
  - Advanced graphics and virtual reality, particularly entertainment industry
  - Networked video and multi-media technologies
  - Oil exploration



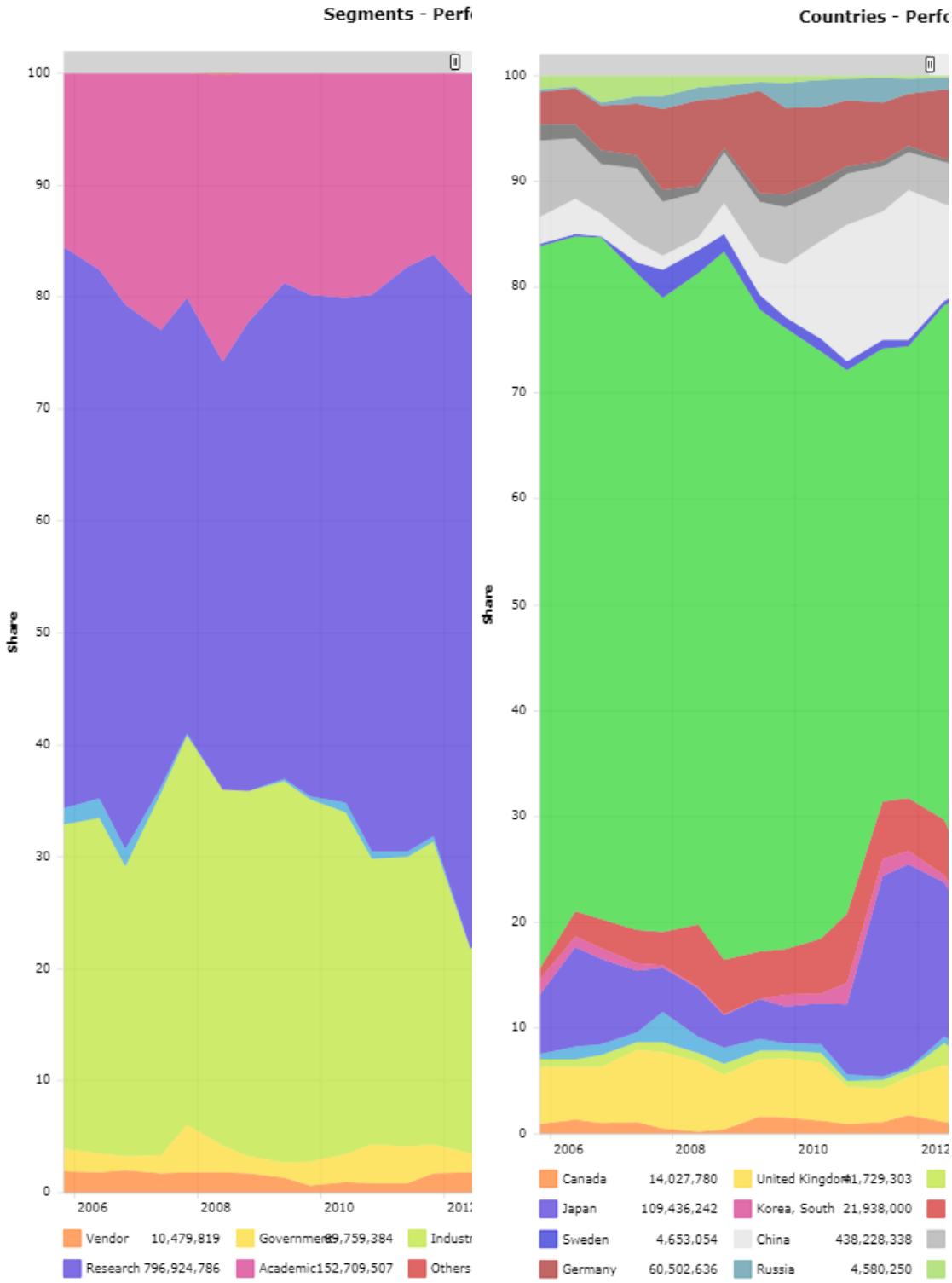
Global Applications:

- Parallel computing is now being used extensively around the world, in a wide variety of applications.



Source: [Top500.org](Top500.org)

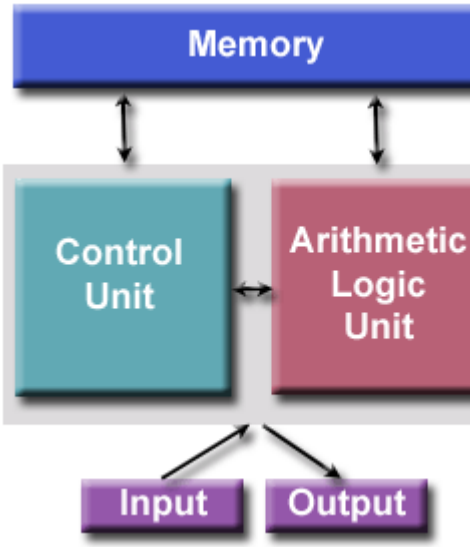
Click on images below for larger version



Source: [Top500.org](Top500.org)

von Neumann Architecture

- Named after the Hungarian mathematician/genius John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
- Also known as "stored-program computer" - both program instructions and data are kept in electronic memory. Differs from earlier computers which were programmed through "hard wiring".
- Since then, virtually all computers have followed this basic design:



- Comprised of four main components:
  - Memory
  - Control Unit
  - Arithmetic Logic Unit
  - Input/Output
- Read/write, random access memory is used to store both program instructions and data
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coor dinates operations to accomplish the programmed task.
- Arithmetic Unit performs basic arithmetic operations
- Input/Output is the interface to the human operator

John von Neumann circa 1940s

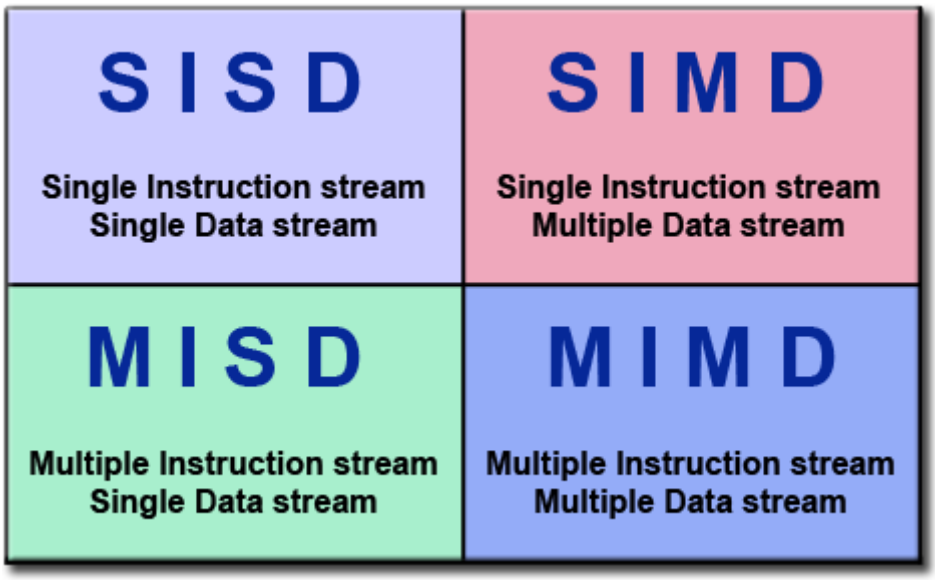
(Source: LANL archives)

- More info on his other remarkable accomplishments: http://en.wikipedia.org/wiki/John_von_Neumann
- So what? Who cares?

- Well, parallel computers still follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same

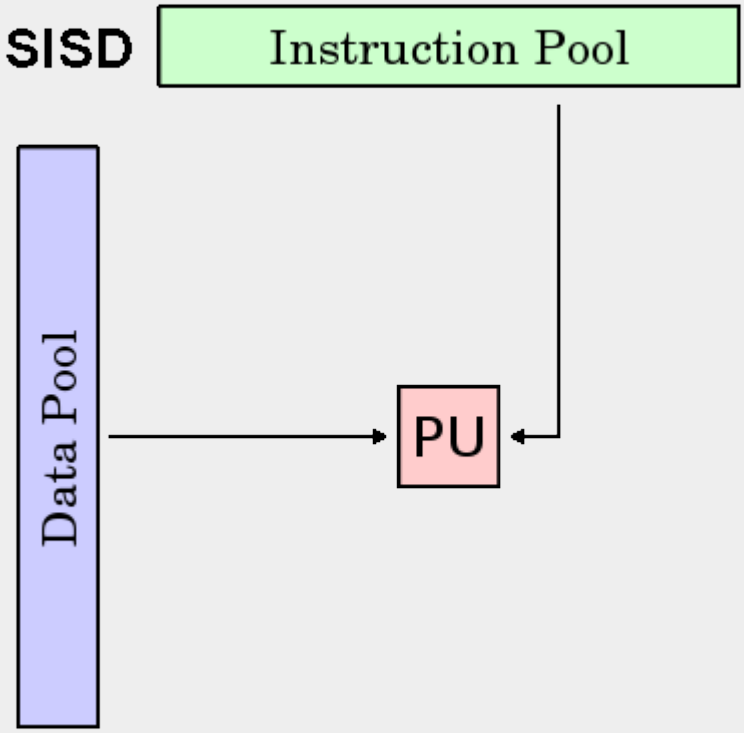
Flynn's Classical Taxonomy

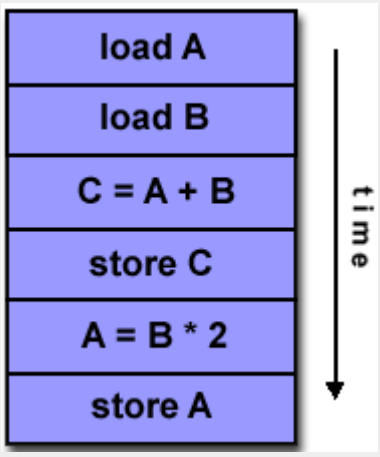
- There are different ways to classify parallel computers. Examples available HERE.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.
- The matrix below defines the 4 possible classifications according to Flynn:



Single Instruction, Single Data (SISD):

- A serial (non-parallel) computer
- **Single Instruction:** Only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single Data:** Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.





UNIVAC1

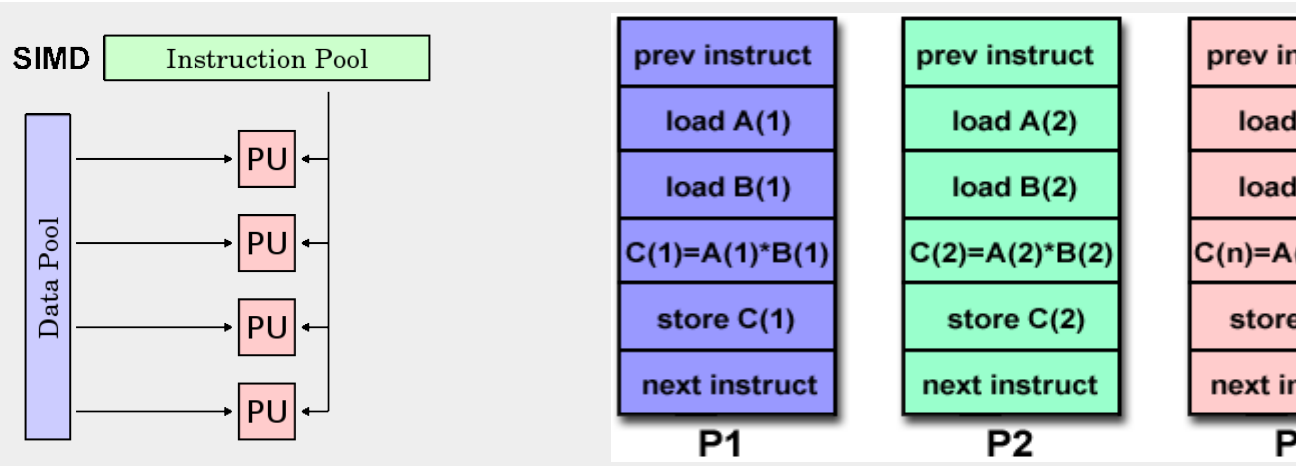
IBM 360

CDC 7600

PDP1

Single Instruction, Multiple Data (SIMD):

- A type of parallel computer
- **Single Instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple Data:** Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
    - Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
    - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.



ILLIAC IV

MasPar

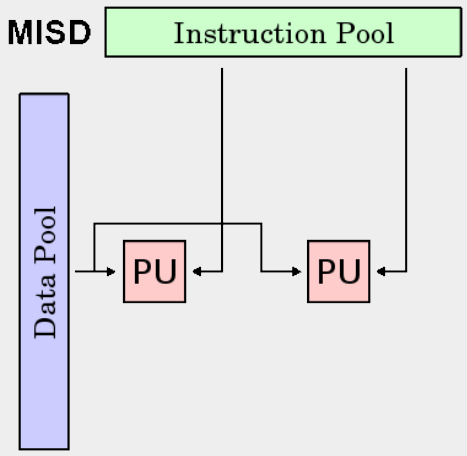
Cray X-MP

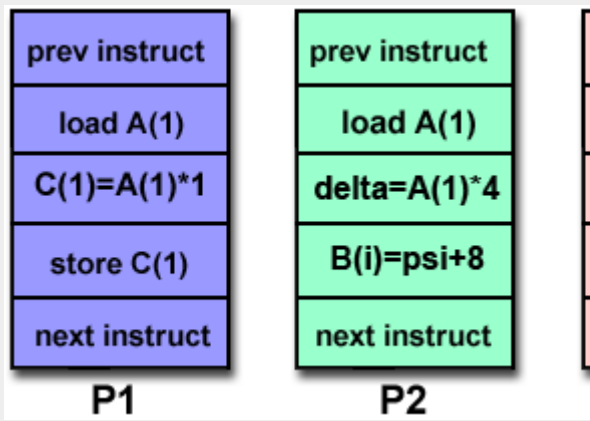
Cray Y-MP

Thinking Machines CM-2

Multiple Instruction, Single Data (MISD):

- A type of parallel computer
- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

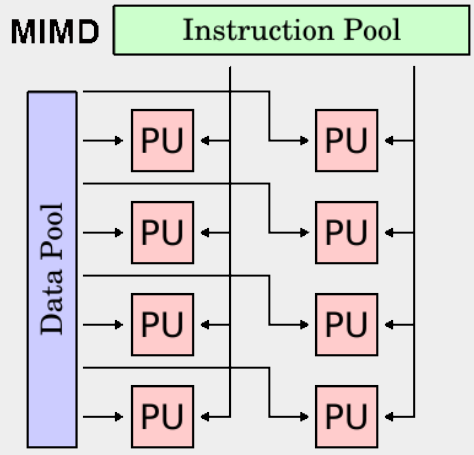


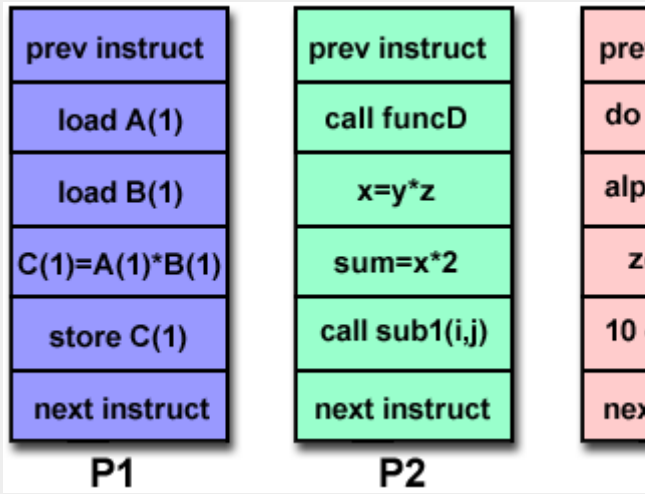


Multiple Instruction, Multiple Data (MIMD):

- A type of parallel computer

- **Multiple Instruction:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components





IBM POWER5

HP/Compaq Alphaserver

AMD Opteron

Cray XT3

Some General Parallel Terminology

- Like everything else, parallel computing has its own "jargon". Some of the more commonly used terms associated with parallel computing are listed below.
- Most of these will be discussed in more detail later.

Supercomputing / High Performance Computing (HPC)

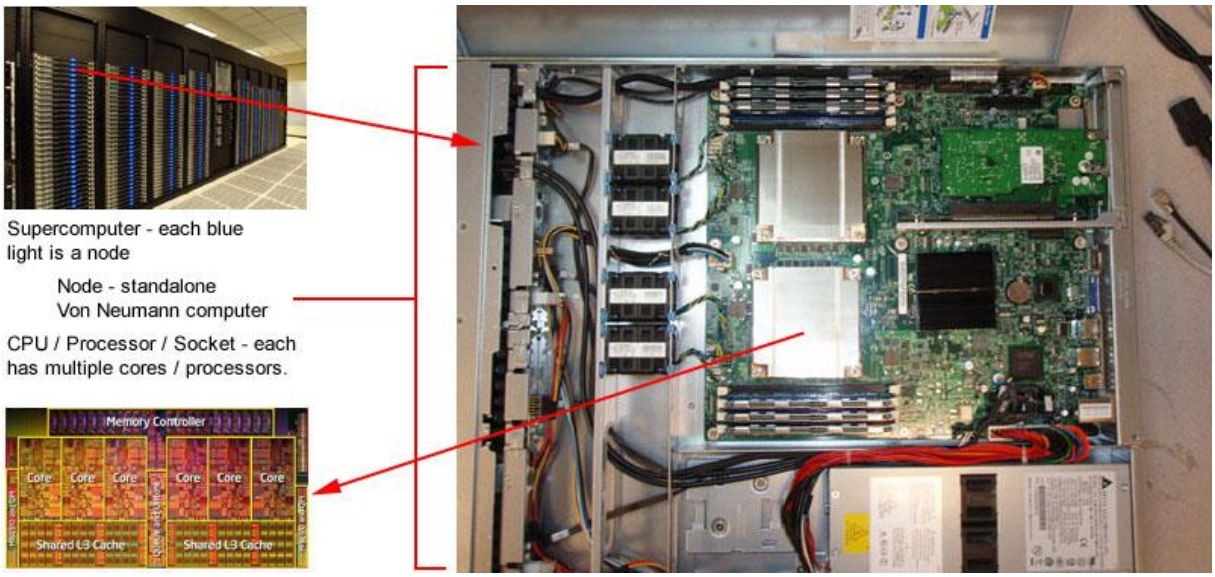
Using the world's fastest and largest computers to solve large problems.

Node

A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer.

CPU / Socket / Processor / Core

This varies, depending upon who you talk to. In the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets" - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confused at times. Wonder why?



Task

A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

Pipelining

Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

Shared Memory

From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Symmetric Multi-Processor (SMP)

Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

Distributed Memory

In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Communications

Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization

The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.

Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

- ***Coarse:*** relatively large amounts of computational work are done between communication events
- ***Fine:*** relatively small amounts of computational work are done between communication events

Observed Speedup

Observed speedup of a code which has been parallelized, defined as:

```
wall-clock time of serial execution
-----------------------------------
 wall-clock time of parallel execution
```

One of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating system, etc.
- Task termination time

Massively Parallel

Refers to the hardware that comprises a given parallel system - having many processing elements. The meaning of "many" keeps increasing, but currently, the largest parallel computers are comprised of processing elements numbering in the hundreds of thousands to millions.

Embarrassingly Parallel

Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks.

Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources. Factors that contribute to scalability include:

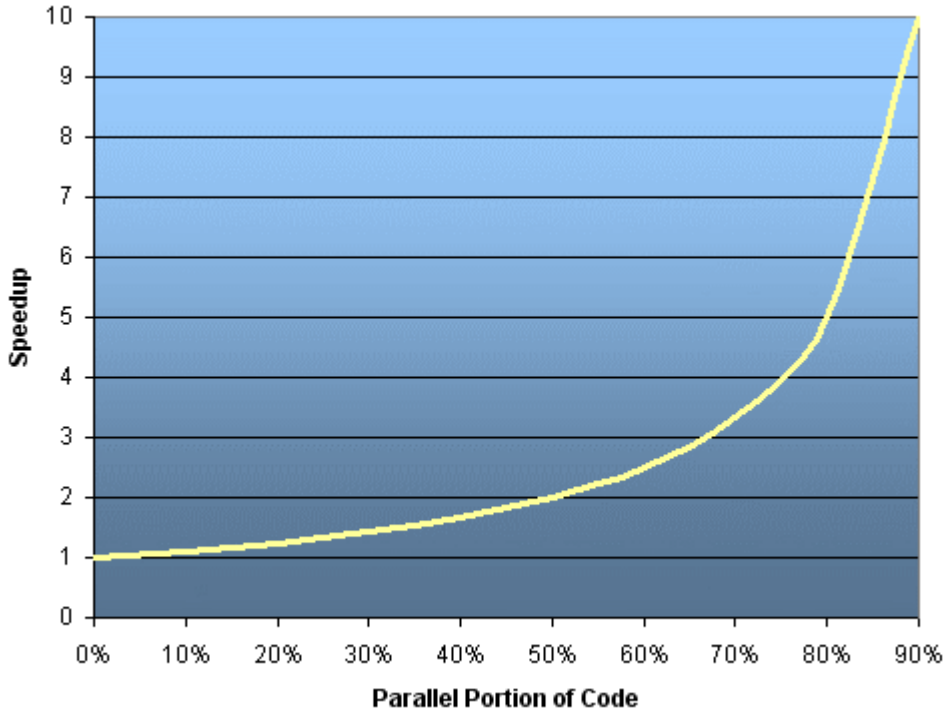
- Hardware - particularly memory-cpu bandwidths and network communication properties
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application

Limits and Costs of Parallel Programming

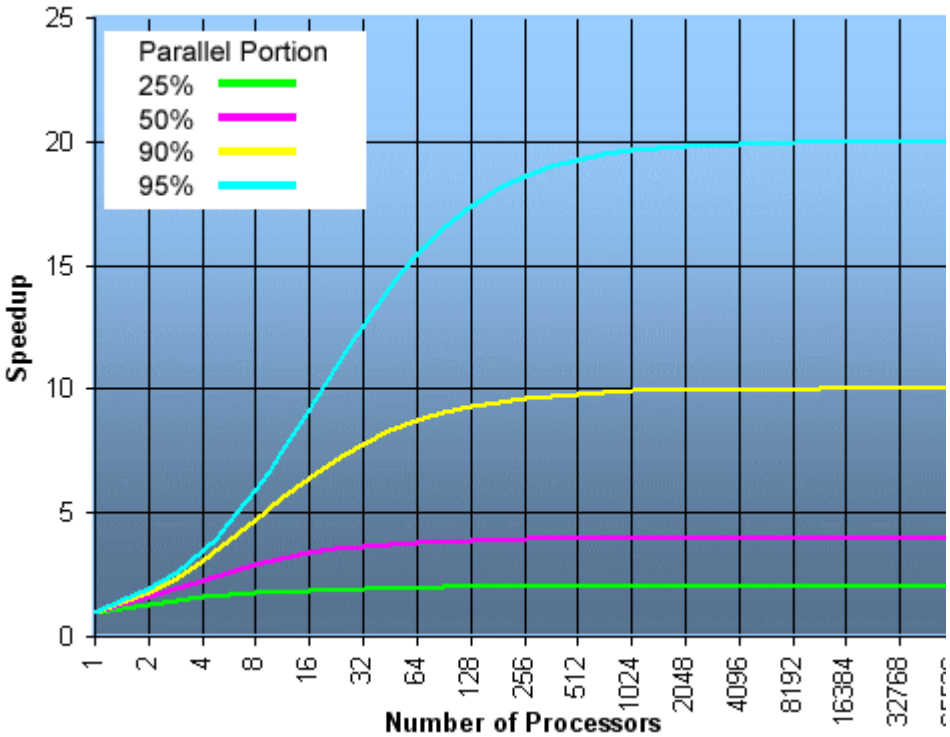
Amdahl's Law:

- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized :

```
    1
speedup
= ------
  1 - P
```



- If none of the code can be parallelized , P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized , P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized , maximum speedup = 2, meaning the code will run twice as fast.
- Introducing the



number of processors performing the parallel fraction of work, the relationship can be modeled by:

```
 1
speedup = ------
 P + S
 ---
 N
```

where P = parallel fraction, N = number of processors and S = serial fraction.

- It soon becomes obvious that there are limits to the scalability of parallelism. For example:

```
 speedup
 ------------------------------------
 N P = .50 P = .90 P = .95 P = .99
 ----- ------- ------- ------- -------
 10 1.82 5.26 6.89 9.17
 100 1.98 9.17 16.80 50.25
 1,000 1.99 9.91 19.62 90.99
 10,000 1.99 9.91 19.96 99.02
100,000 1.99 9.99 19.99 99.90
```

"Famous" quote: *You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!*

- However, certain problems demonstrate increased performance by increasing the problem size. For example:

```
 2D Grid Calculations 85 seconds 85%
```

```
Serial fraction 15 seconds 15%
```

We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

```
2D Grid Calculations 680 seconds 97.84%
Serial fraction 15 seconds 2.16%
```

- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Complexity:

- In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
- The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  - Design
  - Coding
  - Debugging
  - Tuning
  - Maintenance
- Adhering to "good" software development practices is essential when working with parallel applications - especially if somebody besides you will have to work with the software.

Portability:

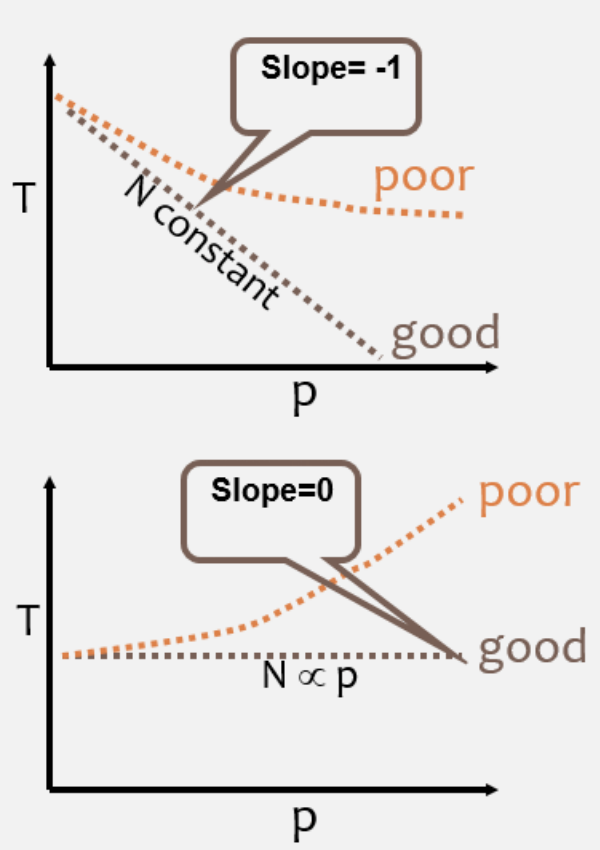
- Thanks to standardization in several APIs, such as MPI, POSIX threads, and OpenMP, portability issues with parallel programs are not as serious as in years past. However...
- All of the usual portability issues associated with serial programs apply to parallel programs. For example, if you use vendor "enhancements" to Fortran, C or C++, portability will be a problem.
- Even though standards exist for several APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effect portability.
- Operating systems can play a key role in code portability issues.
- Hardware architectures are characteristically highly variable and can affect portability.

Resource Requirements:

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
- The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

Scalability:

- Two types of scaling based on time to solution: strong scaling and weak scaling.



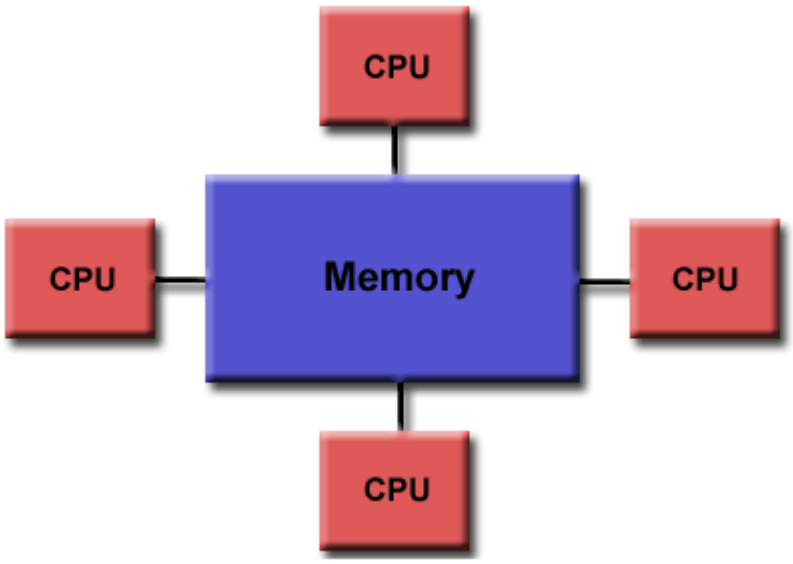
- **Strong scaling:**
  - The total problem size stays fixed as more processors are added.
  - Goal is to run the same problem size faster
  - Perfect scaling means problem is solved in 1/P time (compared to serial)
- **Weak scaling:**
  - The problem size *per processor* stays fixed as more processors are added. The total problem size is proportional to the number of processors used.

    - Goal is to run larger problem in same amount of time
    - Perfect scaling means problem Px runs in same time as single processor run
- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. This is a common situation with many parallel applications.
- Hardware factors play a significant role in scalability. Examples:
    - Memory-cpu bus bandwidth on an SMP machine
    - Communications network bandwidth
    - Amount of memory available on any given machine or set of machines
    - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Shared Memory

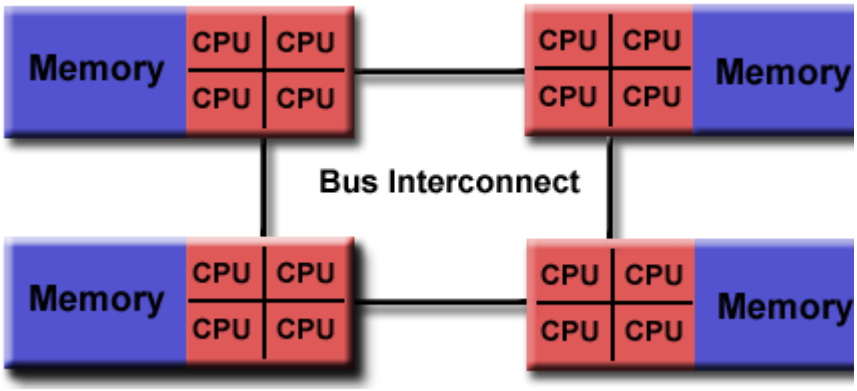
General Characteristics:

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Historically, shared memory machines have been classified as ***UMA*** and ***NUMA***, based upon



Shared Memory (UMA)

memory access times.



Shared Memory (NUMA)

Uniform Memory Access (UMA):

- Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories

- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Advantages:

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

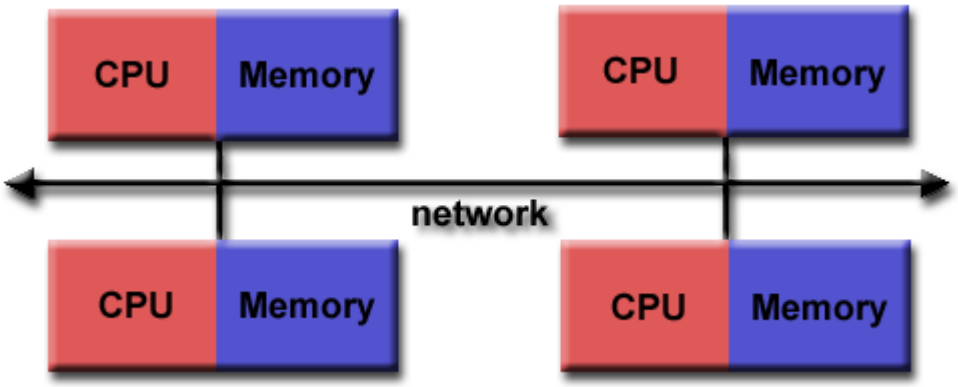
Disadvantages:

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

Distributed Memory

General Characteristics:

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.



- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

Advantages:

- Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

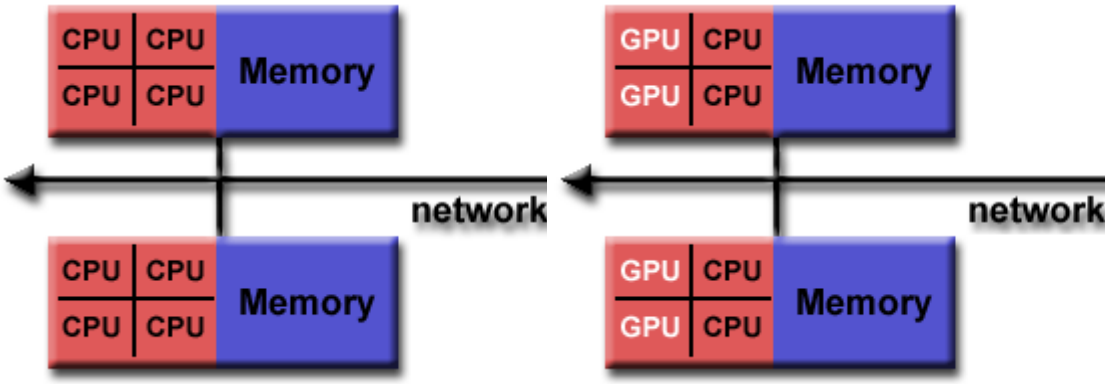
Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

Hybrid Distributed-Shared Memory

General Characteristics:

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one machine to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

Advantages and Disadvantages:

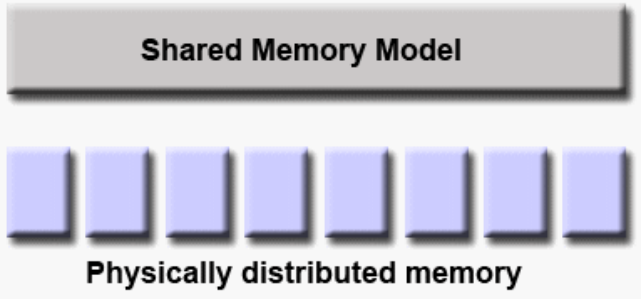
- Whatever is common to both shared and distributed memory architectures.
- Increased scalability is an important advantage
- Increased programmer complexity is an important disadvantage

Parallel Programming Models

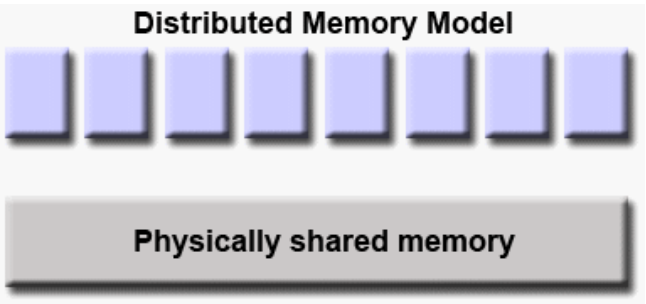
- There are several parallel programming models in common use:
  - Shared Memory (without threads)
  - Threads
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)
- **Parallel programming models exist as an abstraction above hardware and memory architectures.**
- Although it might not seem apparent, these models are **NOT** specific to a particular type of machine or memory architecture. In fact, any of these models

can (theoretically) be implemented on any underlying hardware. Two examples from the past are discussed below.

SHARED memory model on a DISTRIBUTED memory machine:
Kendall Square Research (KSR) ALLCACHE approach. Machine memory was physically distributed across netwo appeared to the user as a single shared memory global address space. Generically, this approach is referred to memory".

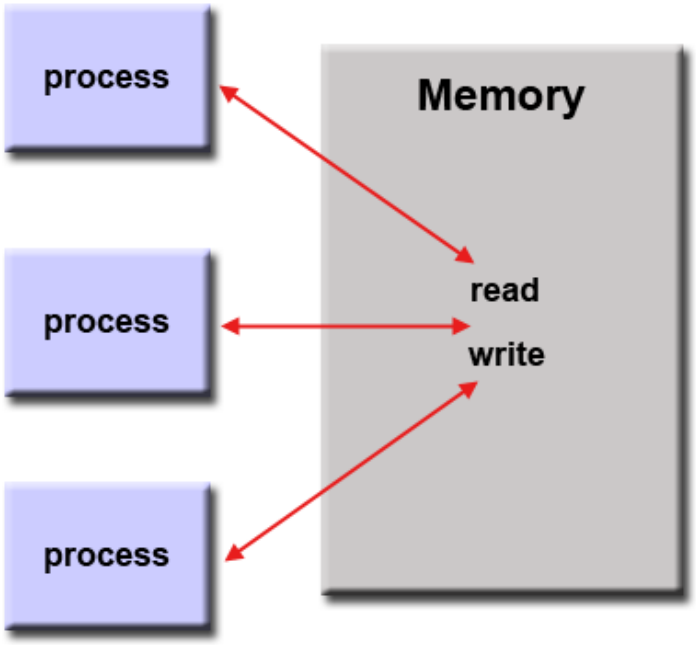


DISTRIBUTED memory model on a SHARED memory machine:
Message Passing Interface (MPI) on SGI Origin 2000. The SGI Origin 2000 employed the CC-NUMA type of sha architecture, where every task has direct access to global address space spread across all machines. However, receive messages using MPI, as is commonly done over a network of distributed memory machines, was impl used.



- **Which model to use?** This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model (without threads)



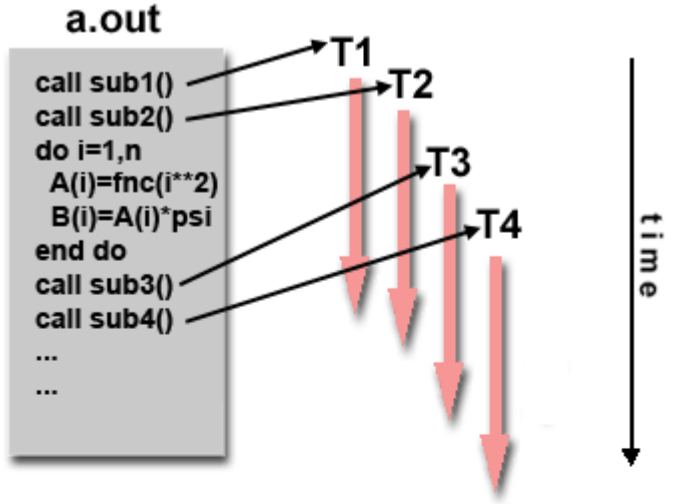
- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- This is perhaps the simplest parallel programming model.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage ***data locality***:
    - Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
    - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

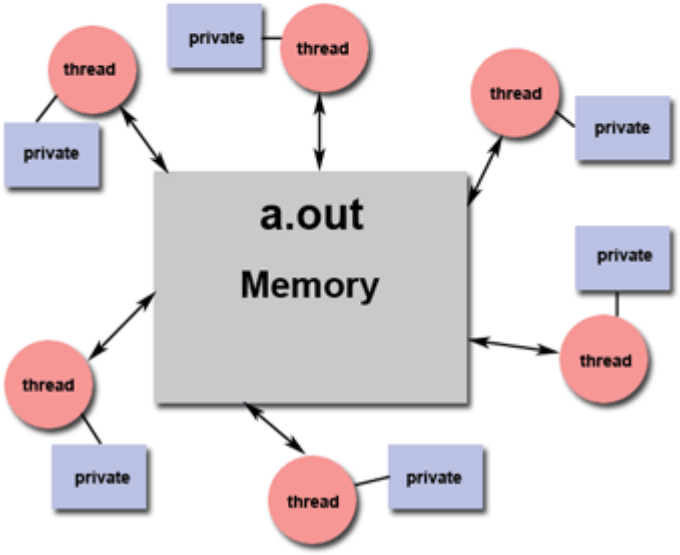
Implementations:

- On stand-alone shared memory machines, native operating systems, compilers and/or hardware provide support for shared memory programming. For example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc).
- On distributed memory machines, memory is physically distributed across a network of machines, but made global through specialized hardware and software. A variety of SHMEM implementations are available: http://en.wikipedia.org/wiki/SHMEM.

Threads Model

- This programming model is a type of shared memory programming.
- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.





- For example:
  - The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.
  - **a.out** performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
  - Each thread has local data, but also, shares the entire resources of **a.out**. This saves the overhead associated with replicating a program's resources for each thread ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of **a.out**.

- A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
- Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
- Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.

Implementations:

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code

  In both cases, the programmer is responsible for determining the parallelism (although compilers can sometimes help).

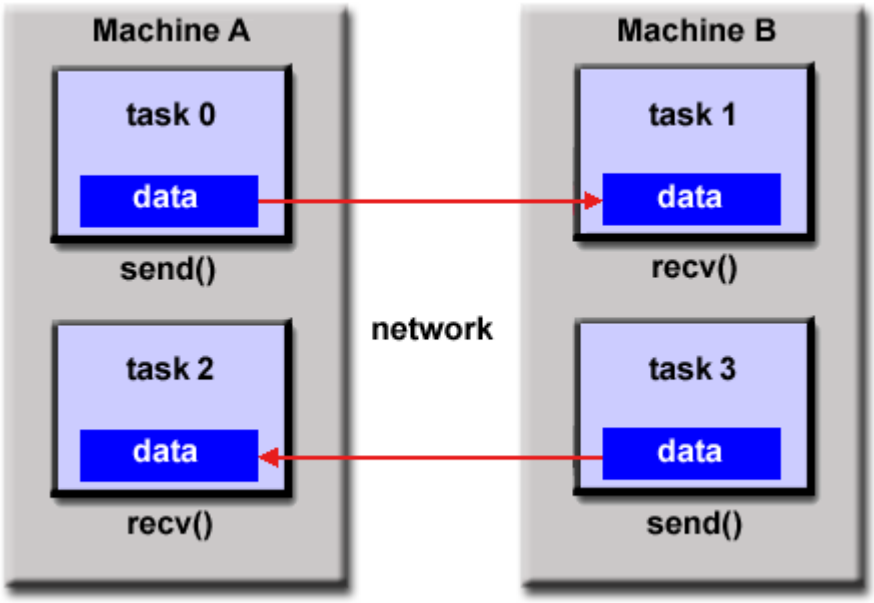
- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.
- **POSIX Threads**
  - Specified by the IEEE POSIX 1003.1c standard (1995). C Language only.
  - Part of Unix/Linux operating systems
  - Library based
  - Commonly referred to as Pthreads.
  - Very explicit parallelism; requires significant programmer attention to detail.
- **OpenMP**
  - Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals.
  - Compiler directive based
  - Portable / multi-platform, including Unix and Windows platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.
- Other threaded implementations are common, but not discussed here:
  - Microsoft threads
  - Java, Python threads
  - CUDA threads for GPUs

More Information:

- POSIX Threads tutorial: computing.llnl.gov/tutorials/pthreads
- OpenMP tutorial: computing.llnl.gov/tutorials/openMP

Distributed Memory / Message Passing Model

- This model demonstrates the following characteristics:



  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Implementations:

- From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996 and MPI-3 in 2012. All MPI specifications are available on the web at http://www.mpi-forum.org/docs/.

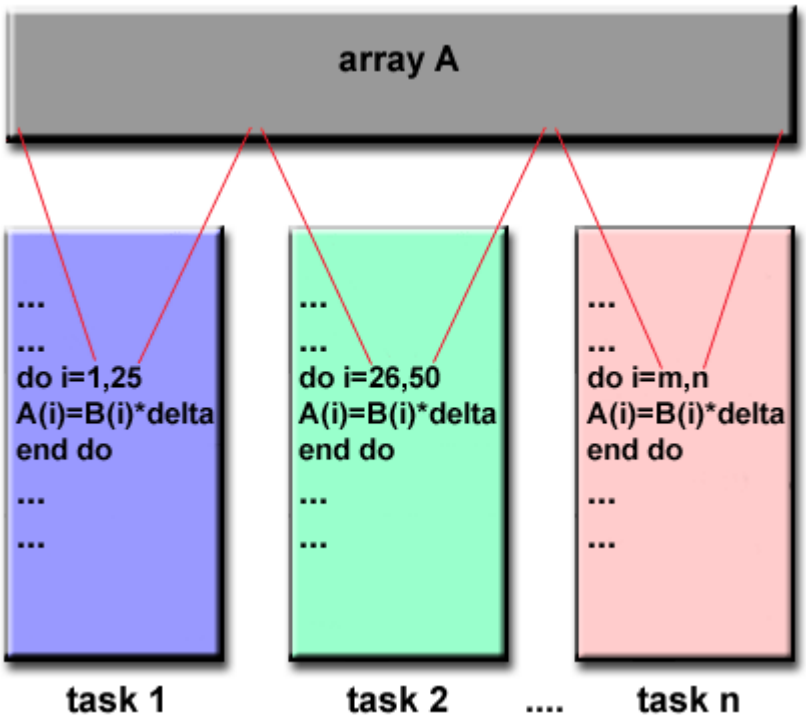
- MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in MPI-1, MPI-2 or MPI-3.

More Information:

- MPI tutorial: computing.llnl.gov/tutorials/mpi

Data Parallel Model

- May also be referred to as the **Partitioned Global Address Space (PGAS)** model.
- The data parallel model demonstrates the following characteristics:



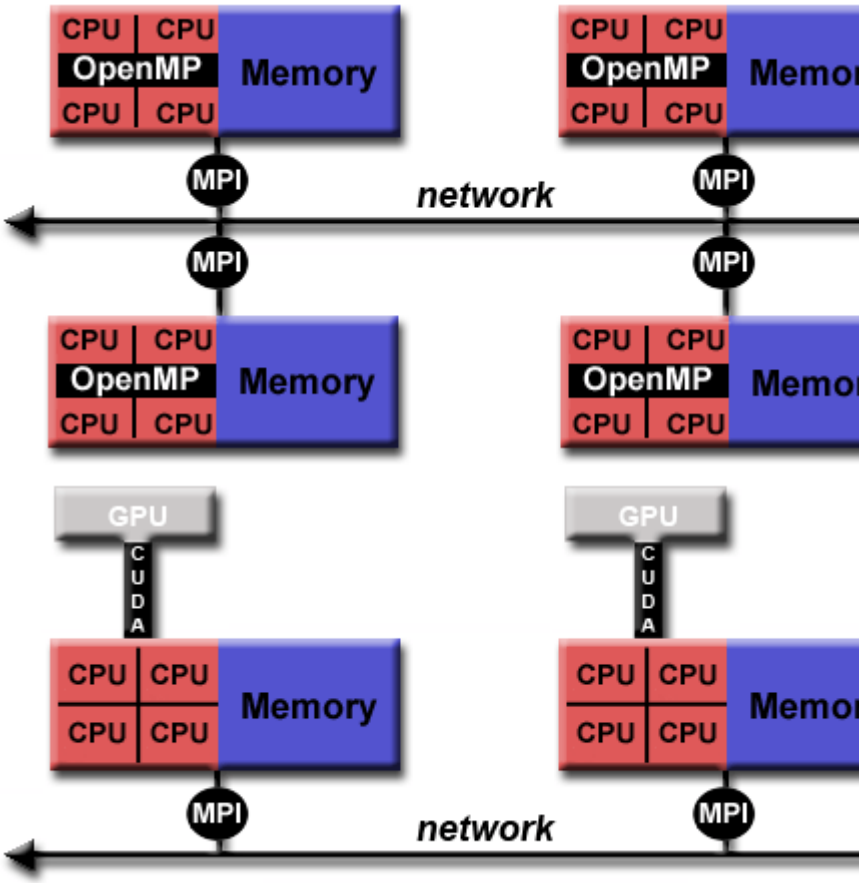
  - Address space is treated globally
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures, the global data structure can be split up logically and/or physically across tasks.

Implementations:

- Currently, there are several relatively popular, and sometimes developmental, parallel programming implementations based on the Data Parallel / PGAS model.
- **Coarray Fortran:** a small set of extensions to Fortran 95 for SPMD parallel programming. Compiler dependent. More information: https://en.wikipedia.org/wiki/Coarray_Fortran
- **Unified Parallel C (UPC):** an extension to the C programming language for SPMD parallel programming. Compiler dependent. More information: http://upc.lbl.gov/
- **Global Arrays:** provides a shared memory style programming environment in the context of distributed array data structures. Public domain library with C and Fortran77 bindings. More information: https://en.wikipedia.org/wiki/Global_Arrays
- **X10:** a PGAS based parallel programming language being developed by IBM at the Thomas J. Watson Research Center. More information: http://x10-lang.org/
- **Chapel:** an open source parallel programming language project being led by Cray. More information: http://chapel.cray.com/

Hybrid Model

- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
    - Threads perform computationally intensive kernels using local, on-node data
    - Communications between processes on different nodes occurs over

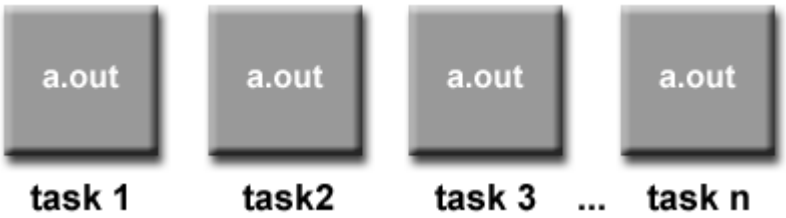


        - the network using MPI
- This hybrid model lends itself well to the most popular hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with CPU-GPU (Graphics Processing Unit) programming.
    - MPI tasks run on CPUs using local memory and communicating with each other over a network.
    - Computationally intensive kernels are off-loaded to GPUs on-node.
    - Data exchange between node-local memory and GPUs uses CUDA (or something equivalent).
- Other hybrid models are common:
    - MPI with Pthreads
    - MPI with non-GPU accelerators
    - ...

SPMD and MPMD

Single Program Multiple Data (SPMD):

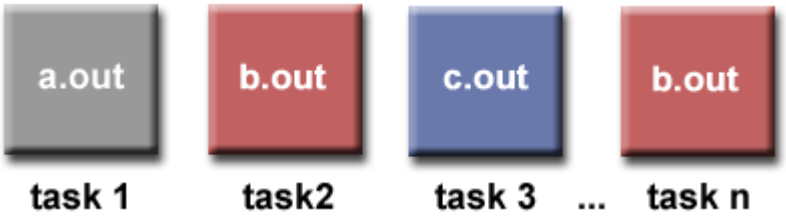
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.



- SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.

Multiple Program Multiple Data (MPMD):

- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.



- MULTIPLE PROGRAM: Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that lend themselves better to functional decomposition than domain decomposition (discussed later under Partitioning).

Source: Blaise Barney, https://computing.llnl.gov/tutorials/parallel_comp/

In this unit, we will discuss the input/output devices that enable communication between computers and the outside world in some form. The reliability of these devices is important; we will accordingly discuss the related issues of dependability, availability, and reliability. You will also take a look at non-volatile storage mediums, such as disk and flash memory, before learning about mechanisms used to connect the computer to input/output devices. This unit will conclude by discussing disk system performance measures.

- Upon successful completion of this unit, you will be able to:
  - calculate the performance of a synchronous bus vs. an asynchronous bus;
  - calculate the bandwidth of a synchronous bus using a given block size; and
  - discuss storage and I/O devices, their performance measurement, and RAID technology.

- 7.1: I/O Devices

Introduction to I/O Subsystems

Watch this lecture, which discusses the basic ideas behind the input/output (I/O) subsystem of a computer system. The lecturer also looks into performance measurement for I/O devices and interfaces used to interconnect I/O devices to the processor. This is the first of two video lectures on I/O. A computer subsystem consists of three major components: processor, memory, and connections. The key words in the previous sentence are subsystem and connections. To be useful, a computer system needs to have connections with external devices to get data and control signals into the computer and to put data and control signals out. The external devices can be other systems or other systems may be connected to the same devices. Thus, our computer system is part of a network of a few or many other subsystems interconnected to perform useful tasks. We are always interested in how well a task is performed, in terms of time, capacity, and cost. In considering performance relative to our useful task, we have to consider processor performance, memory performance, and the performance of the connections, including the performance of the external devices. This first video lecture looks at external or peripheral devices and I/O performance. Next, we will discuss interfaces, buses, and I/O transfer.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=0XybwAbup-w

7.2: Connecting I/O Devices to the Processor

Interfaces and Buses

Watch this lecture for an introduction to the interconnection schemes used for the input/output (I/O) subsystem of a computer system. This is the second of two lectures on I/O devices. The previous lecture looked at the connection of memory, either cache or main memory, with peripheral devices and the transfer and transformation of data

between them. This video lecture analyzes alternative interconnection schemes with a focus on buses. Also, it discusses protocols for the data that flows on the buses: asynchronous and synchronous. A synchronous protocol uses a clock to time sequence the information flow. An asynchronous protocol does not use a clock; the signal carries the sequencing information. Then, the lecture shows a performance comparison of two different protocols.

Be aware that the last example on this resource has an error. When finding the BW, it says 256 x 4 x 1/14400. It should be 256 x 8 x 1/14400. A byte is 8 bits.

Source: Anshul Kumar and the Indian Institute of Technology, Delhi, https://www.youtube.com/watch?v=myVbSSyZtr4

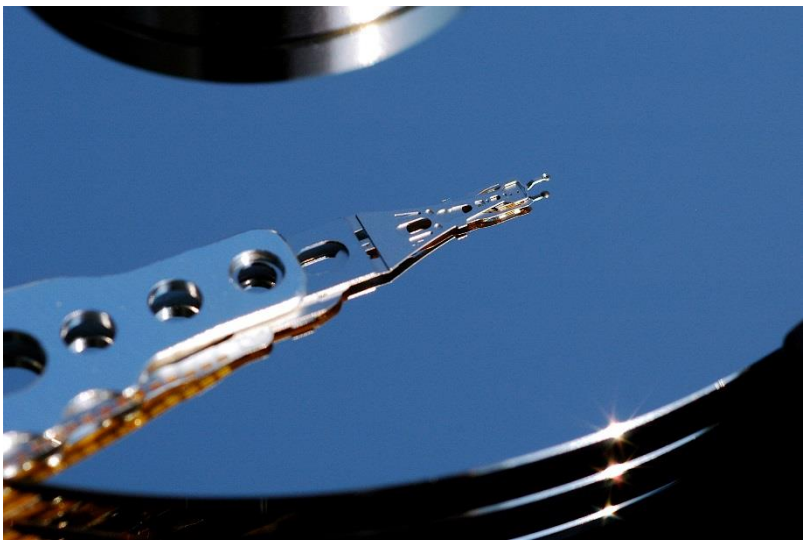
7.3: Measuring Disk Performance

Hard Disk Drive Performance Characteristics

Read this article. Disks have various characteristics, which determine quality attributes, such as reliability, performance, etc. Here, we are interested in performance. Think of some characteristics that affect performance. How can performance of a disk be measured?

Higher performance in hard disk drives comes from devices which have better performance characteristics. These performance characteristics can be grouped into two categories: access time and data transfer time (or rate).

Access time



A hard disk head on an access arm resting on a hard disk platter

The *access time* or *response time* of a rotating drive is a measure of the time it takes before the drive can actually transfer data. The factors that control this time on a rotating drive are mostly related to the mechanical nature of the rotating disks and moving heads. It is composed of a few independently measurable elements that are added together to get a single value when evaluating the performance of a storage device. The access time can vary significantly, so it is typically provided by manufacturers or measured in benchmarks as an average.

The key components that are typically added together to obtain the access time are:

- Seek time
- Rotational latency
- Command processing time
- Settle time

Seek time

With rotating drives, the *seek time* measures the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written. The data on the media is stored in sectors which are arranged in parallel circular tracks (concentric or spiral depending upon the device type) and there is an actuator with an arm that suspends a head that can transfer data with that media. When the drive needs to read or write a certain sector it determines in which track the sector is located. It then uses the actuator to move the head to that particular track. If the initial location of the head was the desired track then the seek time would be zero. If the initial track was the outermost edge of the media and the desired track was at the innermost edge then the seek time would be the maximum for that drive. Seek times are not linear compared with the seek distance traveled because of factors of acceleration and deceleration of the actuator arm.

A rotating drive's *average seek time* is the average of all possible seek times which technically is the time to do all possible seeks divided by the number of all possible seeks, but in practice it is determined by statistical methods or simply approximated as the time of a seek over one-third of the number of tracks.

Seek times & characterstics

The first HDD had an average seek time of about 600 ms. and by the middle 1970s, HDDs were available with seek times of about 25 ms. Some early PC drives used a stepper motor to move the heads, and as a result had seek times as slow as 80–120 ms, but this was quickly improved by voice coil type actuation in the 1980s, reducing seek times to around 20 ms. Seek time has continued to improve slowly over time.

The fastest high-end server drives today have a seek time around 4 ms. Some mobile devices have 15 ms drives, with the most common mobile drives at about 12 ms and the most common desktop drives typically being around 9 ms.

The average seek time is strictly the time to do all possible seeks divided by the number of all possible seeks, but in practice is determined by statistical methods or simply approximated as the time of a seek over one-third of the number of tracks.

Two other less commonly referenced seek measurements are *track-to-track* and *full stroke*. The track-to-track measurement is the time required to move from one track to an adjacent track. This is the shortest (fastest) possible seek time. In HDDs this is typically between 0.2 and 0.8 ms. The full stroke measurement is the time required to move from the outermost track to the innermost track. This is the longest (slowest) possible seek time.

Short stroking

Short stroking is a term used in enterprise storage environments to describe an HDD that is purposely restricted in total capacity so that the actuator only has to move the heads across a smaller number of total tracks. This limits the maximum distance the heads can be from any point on the drive thereby reducing its average seek time, but also restricts the total capacity of the drive. This reduced seek time enables the HDD to increase the number of IOPS available from the drive. The cost and power per usable byte of storage rises as the maximum track range is reduced.

Effect of audible noise and vibration control

Measured in dBA, audible noise is significant for certain applications, such as DVRs, digital audio recording and quiet computers. Low noise disks typically use fluid bearings, lower rotational speeds (usually 5,400 rpm) and reduce the seek speed under load (AAM) to reduce audible clicks and crunching sounds. Drives in smaller form factors (e.g. 2.5 inch) are often quieter than larger drives.

Some desktop- and laptop-class disk drives allow the user to make a trade-off between seek performance and drive noise. For example, Seagate offers a set of features in some drives called Sound Barrier Technology that include some user or system controlled noise and vibration reduction capability. Shorter seek times typically require more energy usage to quickly move the heads across the platter, causing loud noises from the pivot bearing and greater device vibrations as the heads are rapidly accelerated during the start of the seek motion and decelerated at the end of the seek motion. Quiet operation reduces movement speed and acceleration rates, but at a cost of reduced seek performance.

Rotational latency

Typical HDD figures

HDD spindle speed [rpm]	Average rotational latency [ms]
4,200	7.14
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00

Rotational latency (sometimes called *rotational delay* or just *latency*) is the delay waiting for the rotation of the disk to bring the required disk sector under the read-write head. It depends on the rotational speed of a disk (or spindle motor), measured in revolutions per minute (RPM). For most magnetic media-based drives, the *average rotational latency* is typically based on the empirical relation that the average latency in milliseconds for such a drive is one-half the rotational period. *Maximum rotational latency* is the time it takes to do a full rotation excluding any spin-up time (as the relevant part of the disk may have just passed the head when the request arrived).

- ***Maximum latency*** *= 60/rpm*
- ***Average latency*** *= 0.5*Maximum latency*

Therefore, the rotational latency and resulting access time can be improved (decreased) by increasing the rotational speed of the disks. This also has the benefit of improving (increasing) the throughput (discussed later in this article).

The spindle motor speed can use one of two types of disk rotation methods: 1) constant linear velocity (CLV), used mainly in optical storage, varies the rotational speed of the optical disc depending upon the position of the head, and 2) constant angular velocity (CAV), used in HDDs, standard FDDs, a few optical disc systems, and vinyl audio records, spins the media at one constant speed regardless of where the head is positioned.

Another wrinkle occurs depending on whether surface bit densities are constant. Usually, with a CAV spin rate, the densities are not constant so that the long outside tracks have the same number of bits as the shorter inside tracks. When the bit density is constant, outside tracks have more bits than inside tracks and is generally combined with a CLV spin rate. In both these schemes contiguous bit transfer rates are constant. This is not the case with other schemes such as using constant bit density with a CAV spin rate.

Effect of reduced power consumption

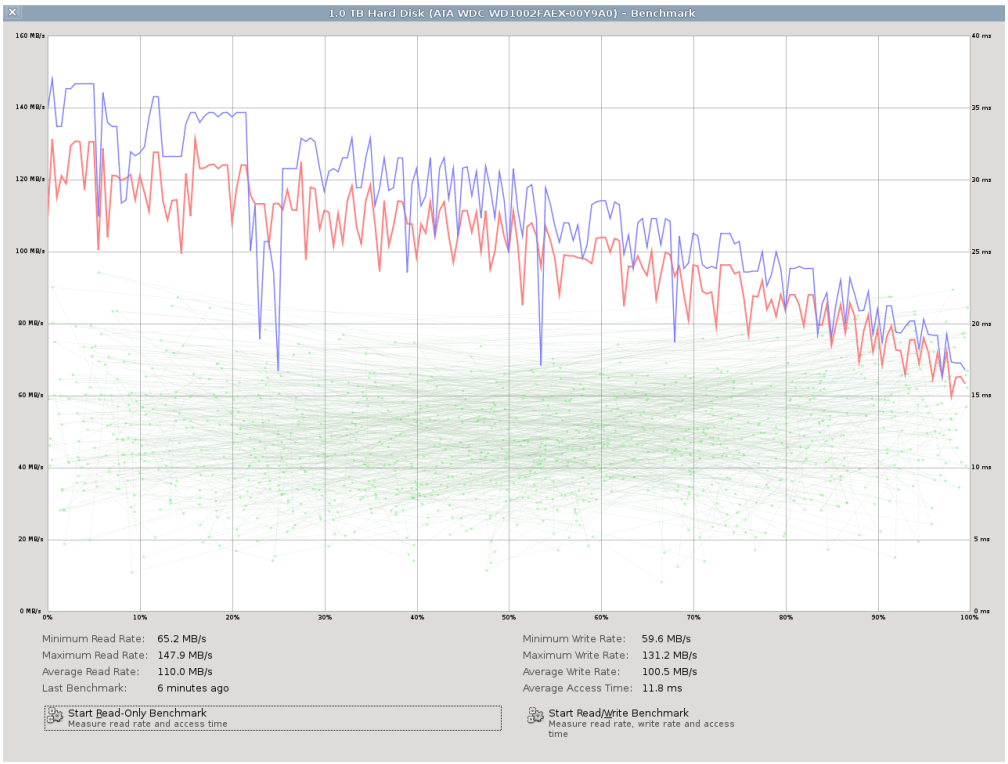
Power consumption has become increasingly important, not only in mobile devices such as laptops but also in server and desktop markets. Increasing data center machine density has led to problems delivering sufficient power to devices (especially for spin-up), and getting rid of the waste heat subsequently produced, as well as environmental and electrical cost concerns (see green computing). Most hard disk drives today support some form of power management which uses a number of specific power modes that save energy by reducing performance. When implemented, an HDD will change between a full power mode to one or more power saving modes as a function of drive usage. Recovery from the deepest mode, typically called Sleep where the drive is stopped or spun down, may take as long as several seconds to be fully operational thereby increasing the resulting latency. The drive manufacturers are also now producing *green drives* that include some additional features that do reduce power, but can adversely affect the latency including lower spindle speeds and parking heads off the media to reduce friction.

Other

The *command processing time* or *command overhead* is the time it takes for the drive electronics to set up the necessary communication between the various components in the device so it can read or write the data. This is of the order of 3 µs, very much less than other overhead times, so it is usually ignored when benchmarking hardware.

The *settle time* is the time it takes the heads to *settle* on the target track and stop vibrating so they do not read or write *off track*. This time is usually very small, typically less than 100 µs, and modern HDD manufacturers account for it in their seek time specifications.

Data transfer rate



A plot showing dependency of transfer rate on cylinder

The *data transfer rate* of a drive (also called *throughput*) covers both the internal rate (moving data between the disk surface and the controller on the drive) and the external rate (moving data between the controller on the drive and the host system). The measurable data transfer rate will be the lower (slower) of the two rates. The *sustained data transfer rate* or *sustained throughput* of a drive will be the lower of the sustained internal and sustained external rates. The sustained rate is less than or equal to the maximum or burst rate because it does not have the benefit of any cache or buffer memory in the drive. The internal rate is further determined by the media rate, sector overhead time, head switch time, and cylinder switch time.

Media rate

Rate at which the drive can read bits from the surface of the media.

Sector overhead time

Additional time (bytes between sectors) needed for control structures and other information necessary to manage the drive, locate and validate data and perform other support functions.

Head switch time

Additional time required to electrically switch from one head to another, re-align the head with the track and begin reading; only applies to multi-head drive and is about 1 to 2 ms.

Cylinder switch time

Additional time required to move to the first track of the next cylinder and begin reading; the name cylinder is used because typically all the tracks of a drive with more than one head or data surface are read before moving the actuator. This time is typically about twice the track-to-track seek time. As of 2001, it was about 2 to 3 ms.

Data transfer rate (read/write) can be measured by writing a large file to disk using special file generator tools, then reading back the file.

- As of 2010, a typical 7200 RPM desktop HDD has a "disk-to-buffer" data transfer rate up to 1030 Mbit/s. This rate depends on the track location, so it will be higher on the outer zones (where there are more data sectors per track) and lower on the inner zones (where there are fewer data sectors per track); and is generally somewhat higher for 10,000 RPM drives.
- Floppy disk drives have sustained "disk-to-buffer" data transfer rates that are one or two orders of magnitude lower than that of HDDs.
- The sustained "disk-to-buffer" data transfer rates varies amongst families of Optical disk drives with the slowest 1x CDs at 1.23 Mbit/s floppy-like while a high performance 12x Blu-ray drive at 432 Mbit/s approaches the performance of HDDs.

A current widely used standard for the "buffer-to-computer" interface is 3.0 Gbit/s SATA, which can send about 300 megabyte/s (10-bit encoding) from the buffer to the computer, and thus is still comfortably ahead of today's disk-to-buffer transfer rates.

SSDs do not have the same internal limits of HDDs, so their internal and external transfer rates are often maximizing the capabilities of the drive-to-host interface.

Effect of file system

Transfer rate can be influenced by file system fragmentation and the layout of the files. Defragmentation is a procedure used to minimize delay in retrieving data by moving related items to physically proximate areas on the disk. Some computer operating systems perform defragmentation automatically. Although automatic defragmentation is intended to reduce access delays, the procedure can slow response when performed while the computer is in use.

Effect of areal density

HDD data transfer rate depends upon the rotational speed of the disks and the data recording density. Because heat and vibration limit rotational speed, increasing density has become the main method to improve sequential transfer rates. *Areal density* (the number of bits that can be stored in a certain area of the disk) has been increased over time by increasing both the number of tracks across the disk, and the number of sectors per track. The latter will increase the data transfer rate for a given RPM speed. Improvement of data transfer rate performance is correlated to the areal density only by increasing a track's linear surface bit density (sectors per track). Simply increasing the number of tracks on a disk can affect seek times but not gross transfer rates. According to industry observers and analysts for 2011 to 2016, "The current roadmap predicts no more than a 20%/yr improvement in bit density". Seek times have not kept up with throughput increases, which themselves have not kept up with growth in bit density and storage capacity.

Interleave

Low-level formatting software from 1987 to find highest performance interleave choice for 10 MB IBM PC XT hard disk drive

Sector interleave is a mostly obsolete device characteristic related to data rate, dating back to when computers were too slow to be able to read large continuous streams of data. Interleaving introduced gaps between data sectors to allow time for slow equipment to get ready to read the next block of data. Without interleaving, the next logical sector would arrive at the read/write head before the equipment was ready, requiring the system to wait for another complete disk revolution before reading could be performed.

However, because interleaving introduces intentional physical delays between blocks of data thereby lowering the data rate, setting the interleave to a ratio higher than required causes unnecessary delays for equipment that has the performance needed to read sectors more quickly. The interleaving ratio was therefore usually chosen by the end-user to suit their particular computer system's performance capabilities when the drive was first installed in their system.

Modern technology is capable of reading data as fast as it can be obtained from the spinning platters, so hard drives usually have a fixed sector interleave ratio of 1:1, which is effectively no interleaving being used.

Power consumption

Power consumption has become increasingly important, not only in mobile devices such as laptops but also in server and desktop markets. Increasing data center machine density has led to problems delivering sufficient power to devices (especially for spin up), and getting rid of the waste heat subsequently produced, as well as environmental and electrical cost concerns (see green computing). Heat dissipation is tied directly to power consumption, and as drives age, disk failure rates increase at higher drive temperatures. Similar issues exist for large companies with thousands of desktop PCs. Smaller form factor drives often use less power than larger drives. One interesting development in this area is actively controlling the seek speed so that the head arrives at its destination only just in time to read the sector, rather than arriving as quickly as possible and then having to wait for the sector to come around (i.e. the rotational latency). Many of the hard drive companies are now producing Green Drives that require much less power and cooling. Many of these Green Drives spin slower (<5,400 rpm compared to 7,200, 10,000 or 15,000 rpm) thereby generating less heat. Power consumption can also be reduced by parking the drive heads when the disk is not in use reducing friction, adjusting spin speeds, and disabling internal components when not in use.

Drives use more power, briefly, when starting up (spin-up). Although this has little direct effect on total energy consumption, the maximum power demanded from the power supply, and hence its required rating, can be reduced in systems with several drives by controlling when they spin up.

- On SCSI hard disk drives, the SCSI controller can directly control spin up and spin down of the drives.
- Some Parallel ATA (PATA) and Serial ATA (SATA) hard disk drives support power-up in standby (PUIS): each drive does not spin up until the controller or system BIOS issues a specific command to do so. This allows the system to be set up to stagger disk start-up and limit maximum power demand at switch-on.
- Some SATA II and later hard disk drives support staggered spin-up, allowing the computer to spin up the drives in sequence to reduce load on the power supply when booting.

Most hard disk drives today support some form of power management which uses a number of specific power modes that save energy by reducing performance. When implemented an HDD will change between a full power mode to one or more power saving modes as a function of drive usage. Recovery from the deepest mode, typically called Sleep, may take as long as several seconds.

Shock resistance

Shock resistance is especially important for mobile devices. Some laptops now include active hard drive protection that parks the disk heads if the machine is dropped, hopefully before impact, to offer the greatest possible chance of survival in such an event. Maximum shock tolerance to date is 350 g for operating and 1,000 g for non-operating.

Comparison to Solid-state drive

Solid-state devices (SSDs) do not have moving parts. Most attributes related to the movement of mechanical components are not applicable in measuring their performance, but they are affected by some electrically based elements that causes a measurable access delay.

Measurement of seek time is only testing electronic circuits preparing a particular location on the memory in the storage device. Typical SSDs will have a seek time between 0.08 and 0.16 ms.

Flash memory-based SSDs do not need defragmentation. However, because SSDs write pages of data that are much larger than the blocks of data managed by the file system, over time, an SSD's write performance can degrade as the drive becomes full of pages which are partial or no longer needed by the file system. This can be ameliorated by a TRIM command from the system or internal garbage collection. Flash memory wears out over time as it is repeatedly written to; the writes required by defragmentation wear the drive for no speed advantage.

Source:
Wikipedia, https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics

7.4: Redundant Array of Inexpensive Disks (RAID)

RAID

This article will provide you with a description of RAID technology and will introduce you to techniques that are used for studying performance and reliability in general.

RAID ("**Redundant Array of Inexpensive Disks**" or "**Redundant Array of Independent Disks**") is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both. This was in contrast to the previous concept of highly reliable mainframe disk drives referred to as "single large expensive disk" (SLED).

Data is distributed across the drives in one of several ways, referred to as RAID levels, depending on the required level of redundancy and performance. The different schemes, or data distribution layouts, are named by the word "RAID" followed by a number, for example RAID 0 or RAID 1. Each scheme, or RAID level, provides a different balance among the key goals: reliability, availability, performance, and capacity. RAID levels greater than RAID 0 provide protection against unrecoverable sector read errors, as well as against failures of whole physical drives.

History

The term "RAID" was invented by David Patterson, Garth A. Gibson, and Randy Katz at the University of California, Berkeley in 1987. In their June 1988 paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)", presented at the SIGMOD conference, they argued that the top-performing mainframe disk drives of the time could be beaten on performance by an array of the inexpensive drives that had been developed for the growing personal computer market. Although failures would rise in proportion to the number of drives, by configuring for redundancy, the reliability of an array could far exceed that of any large single drive.

Although not yet using that terminology, the technologies of the five levels of RAID named in the June 1988 paper were used in various products prior to the paper's publication, including the following:

- Mirroring (RAID 1) was well established in the 1970s including, for example, Tandem NonStop Systems.
- In 1977, Norman Ken Ouchi at IBM filed a patent disclosing what was subsequently named RAID 4.
- Around 1983, DEC began shipping subsystem mirrored RA8X disk drives (now known as RAID 1) as part of its HSC50 subsystem.
- In 1986, Clark et al. at IBM filed a patent disclosing what was subsequently named RAID 5.
- Around 1988, the Thinking Machines' DataVault used error correction codes (now known as RAID 2) in an array of disk drives. A similar approach was used in the early 1960s on the IBM 353.

Industry manufacturers later redefined the RAID acronym to stand for "Redundant Array of *Independent* Disks".

Overview

Many RAID levels employ an error protection scheme called "parity", a widely used method in information technology to provide fault tolerance in a given set of data. Most use simple XOR, but RAID 6 uses two separate parities based respectively on addition and multiplication in a particular Galois field or Reed–Solomon error correction.

RAID can also provide data security with solid-state drives (SSDs) without the expense of an all-SSD system. For example, a fast SSD can be mirrored with a mechanical drive. For this configuration to provide a significant speed advantage an appropriate controller is needed that uses the fast SSD for all read operations. Adaptec calls this "hybrid RAID".

Standard levels

Storage servers with 24 hard disk drives and built-in hardware RAID controllers supporting various RAID levels

A number of standard schemes have evolved. These are called *levels*. Originally, there were five RAID levels, but many variations have evolved, including several nested levels and many non-standard levels (mostly proprietary). RAID levels and their associated data formats are standardized by the Storage Networking Industry Association (SNIA) in the Common RAID Disk Drive Format (DDF) standard:

RAID 0

RAID 0 consists of striping, but no mirroring or parity. Compared to a spanned volume, the capacity of a RAID 0 volume is the same; it is the sum of the capacities of the disks in the set. But because striping distributes the contents of each file among all disks in the set, the failure of any disk causes all files, the entire RAID 0 volume, to be lost. A broken spanned volume at least preserves the files on the unfailing disks. The benefit of RAID 0 is that the throughput of read and write operations to any file is multiplied by the number of disks because, unlike spanned volumes, reads and writes are done concurrently, and the cost is complete vulnerability to drive failures. Indeed, the average failure rate is worse than that of an equivalent single non-RAID drive.

RAID 1

RAID 1 consists of data mirroring, without parity or striping. Data is written identically to two or more drives, thereby producing a "mirrored set" of drives. Thus, any read request can be serviced by any drive in the set. If a request is broadcast to every drive in the set, it can be serviced by the drive that accesses the data first (depending on its seek time and rotational latency), improving performance. Sustained read throughput, if the controller or software is optimized for it, approaches the sum of throughputs of every drive in the set, just as for RAID 0. Actual read throughput of most RAID 1 implementations is slower than the fastest drive. Write throughput is always slower because every drive must be updated, and the slowest drive limits the write performance. The array continues to operate as long as at least one drive is functioning.

RAID 2

RAID 2 consists of bit-level striping with dedicated Hamming-code parity. All disk spindle rotation is synchronized and data is striped such that each sequential bit is on a different drive. Hamming-code parity is calculated across corresponding bits and stored on at least one parity drive. This level is of historical significance only; although it was used on some early machines (for example, the Thinking Machines CM-2), as of 2014 it is not used by any commercially available system.

RAID 3

RAID 3 consists of byte-level striping with dedicated parity. All disk spindle rotation is synchronized and data is striped such that each sequential byte is on a different drive. Parity is calculated across corresponding bytes and stored on a dedicated parity drive. Although implementations exist, RAID 3 is not commonly used in practice.

RAID 4

RAID 4 consists of block-level striping with dedicated parity. This level was previously used by NetApp, but has now been largely replaced by a proprietary implementation of RAID 4 with two parity disks, called RAID-DP. The main advantage of RAID 4 over RAID 2 and 3 is I/O parallelism: in RAID 2 and 3, a single read I/O operation requires reading the whole group of data drives, while in RAID 4 one I/O read operation does not

have to spread across all data drives. As a result, more I/O operations can be executed in parallel, improving the performance of small transfers.

RAID 5

RAID 5 consists of block-level striping with distributed parity. Unlike RAID 4, parity information is distributed among the drives, requiring all drives but one to be present to operate. Upon failure of a single drive, subsequent reads can be calculated from the distributed parity such that no data is lost. RAID 5 requires at least three disks. Like all single-parity concepts, large RAID 5 implementations are susceptible to system failures because of trends regarding array rebuild time and the chance of drive failure during rebuild (see " Increasing rebuild time and failure probability" section, below). Rebuilding an array requires reading all data from all disks, opening a chance for a second drive failure and the loss of the entire array.

RAID 6

RAID 6 consists of block-level striping with double distributed parity. Double parity provides fault tolerance up to two failed drives. This makes larger RAID groups more practical, especially for high-availability systems, as large-capacity drives take longer to restore. RAID 6 requires a minimum of four disks. As with RAID 5, a single drive failure results in reduced performance of the entire array until the failed drive has been replaced. With a RAID 6 array, using drives from multiple sources and manufacturers, it is possible to mitigate most of the problems associated with RAID 5. The larger the drive capacities and the larger the array size, the more important it becomes to choose RAID 6 instead of RAID 5. RAID 10 also minimizes these problems.

Nested (hybrid) RAID

In what was originally termed *hybrid RAID*, many storage controllers allow RAID levels to be nested. The elements of a *RAID* may be either individual drives or arrays themselves. Arrays are rarely nested more than one level deep.

The final array is known as the top array. When the top array is RAID 0 (such as in RAID 1+0 and RAID 5+0), most vendors omit the "+" (yielding RAID 10 and RAID 50, respectively).

- **RAID 0+1:** creates two stripes and mirrors them. If a single drive failure occurs then one of the mirrors has failed, at this point it is running effectively as RAID 0 with no redundancy. Significantly higher risk is introduced during a rebuild than RAID 1+0 as all the data from all the drives in the remaining stripe has to be read rather than just from one drive, increasing the chance of an unrecoverable read error (URE) and significantly extending the rebuild window.

- **RAID 1+0:** (see: RAID 10) creates a striped set from a series of mirrored drives. The array can sustain multiple drive losses so long as no mirror loses all its drives.
- **JBOD RAID N+N:** With JBOD (*just a bunch of disks*), it is possible to concatenate disks, but also volumes such as RAID sets. With larger drive capacities, write delay and rebuilding time increase dramatically (especially, as described above, with RAID 5 and RAID 6). By splitting a larger RAID N set into smaller subsets and concatenating them with linear JBOD, write and rebuilding time will be reduced. If a hardware RAID controller is not capable of nesting linear JBOD with RAID N, then linear JBOD can be achieved with OS-level software RAID in combination with separate RAID N subset volumes created within one, or more, hardware RAID controller(s). Besides a drastic speed increase, this also provides a substantial advantage: the possibility to start a linear JBOD with a small set of disks and to be able to expand the total set with disks of different size, later on (in time, disks of bigger size become available on the market). There is another advantage in the form of disaster recovery (if a RAID N subset happens to fail, then the data on the other RAID N subsets is not lost, reducing restore time).

Non-standard levels

Many configurations other than the basic numbered RAID levels are possible, and many companies, organizations, and groups have created their own non-standard configurations, in many cases designed to meet the specialized needs of a small niche group. Such configurations include the following:

- Linux MD RAID 10 provides a general RAID driver that in its "near" layout defaults to a standard RAID 1 with two drives, and a standard RAID 1+0 with four drives; however, it can include any number of drives, including odd numbers. With its "far" layout, MD RAID 10 can run both striped and mirrored, even with only two drives in `f2` layout; this runs mirroring with striped reads, giving the read performance of RAID 0. Regular RAID 1, as provided by Linux software RAID, does not stripe reads, but can perform reads in parallel.
- Hadoop has a RAID system that generates a parity file by xor-ing a stripe of blocks in a single HDFS file.
- BeeGFS, the parallel file system, has internal striping (comparable to file-based RAID0) and replication (comparable to file-based RAID10) options to aggregate throughput and capacity of multiple servers and is typically based on top of an underlying RAID to make disk failures transparent.
- Declustered RAID scatters dual (or more) copies of the data across all disks (possibly hundreds) in a storage subsystem, while holding back enough spare capacity to allow for a few disks to fail. The scattering is based on algorithms which give the appearance of arbitrariness. When one or more disks fail the missing copies are rebuilt into that spare capacity, again arbitrarily. Because the rebuild is done from and to all the remaining disks, it operates much faster than with traditional RAID, reducing the overall impact on clients of the storage system.

Implementations

The distribution of data across multiple drives can be managed either by dedicated computer hardware or by software. A software solution may be part of the operating system, part of the firmware and drivers supplied with a standard drive controller (so-called "hardware-assisted software RAID"), or it may reside entirely within the hardware RAID controller.

Hardware-based

Configuration of hardware RAID

Hardware RAID controllers can be configured through card BIOS before an operating system is booted, and after the operating system is booted, proprietary configuration utilities are available from the manufacturer of each controller. Unlike the network interface controllers for Ethernet, which can usually be configured and serviced entirely through the common operating system paradigms like ifconfig in Unix, without a need for any third-party tools, each manufacturer of each RAID controller usually provides their own proprietary software tooling for each operating system that they deem to support, ensuring a vendor lock-in, and contributing to reliability issues.

For example, in FreeBSD, in order to access the configuration of Adaptec RAID controllers, users are required to enable Linux compatibility layer, and use the Linux tooling from Adaptec, potentially compromising the stability, reliability and security of their setup, especially when taking the long term view.

Some other operating systems have implemented their own generic frameworks for interfacing with any RAID controller, and provide tools for monitoring RAID volume status, as well as facilitation of drive identification through LED blinking, alarm management and hot spare disk designations from within the operating system without having to reboot into card BIOS. For example, this was the approach taken by OpenBSD in 2005 with its bio(4) pseudo-device and the bioctl utility, which provide volume status, and allow LED/alarm/hotspare control, as well as the sensors (including the drive sensor) for health monitoring; this approach has subsequently been adopted and extended by NetBSD in 2007 as well.

Software-based

Software RAID implementations are provided by many modern operating systems. Software RAID can be implemented as:

- A layer that abstracts multiple devices, thereby providing a single virtual device (e.g. Linux kernel's md and OpenBSD's softraid)

- A more generic logical volume manager (provided with most server-class operating systems, e.g. Veritas or LVM)
- A component of the file system (e.g. ZFS, Spectrum Scale or Btrfs)
- A layer that sits above any file system and provides parity protection to user data (e.g. RAID-F)

Some advanced file systems are designed to organize data across multiple storage devices directly, without needing the help of a third-party logical volume manager:

- ZFS supports the equivalents of RAID 0, RAID 1, RAID 5 (RAID-Z1) single-parity, RAID 6 (RAID-Z2) double-parity, and a triple-parity version (RAID-Z3) also referred to as RAID 7. As it always stripes over top-level vdevs, it supports equivalents of the 1+0, 5+0, and 6+0 nested RAID levels (as well as striped triple-parity sets) but not other nested combinations. ZFS is the native file system on Solaris and illumos, and is also available on FreeBSD and Linux. Open-source ZFS implementations are actively developed under the OpenZFS umbrella project.
- Spectrum Scale, initially developed by IBM for media streaming and scalable analytics, supports declustered RAID protection schemes up to n+3. A particularity is the dynamic rebuilding priority which runs with low impact in the background until a data chunk hits n+0 redundancy, in which case this chunk is quickly rebuilt to at least n+1. On top, Spectrum Scale supports metro-distance RAID 1.
- Btrfs supports RAID 0, RAID 1 and RAID 10 (RAID 5 and 6 are under development).
- XFS was originally designed to provide an integrated volume manager that supports concatenating, mirroring and striping of multiple physical storage devices. However, the implementation of XFS in Linux kernel lacks the integrated volume manager.

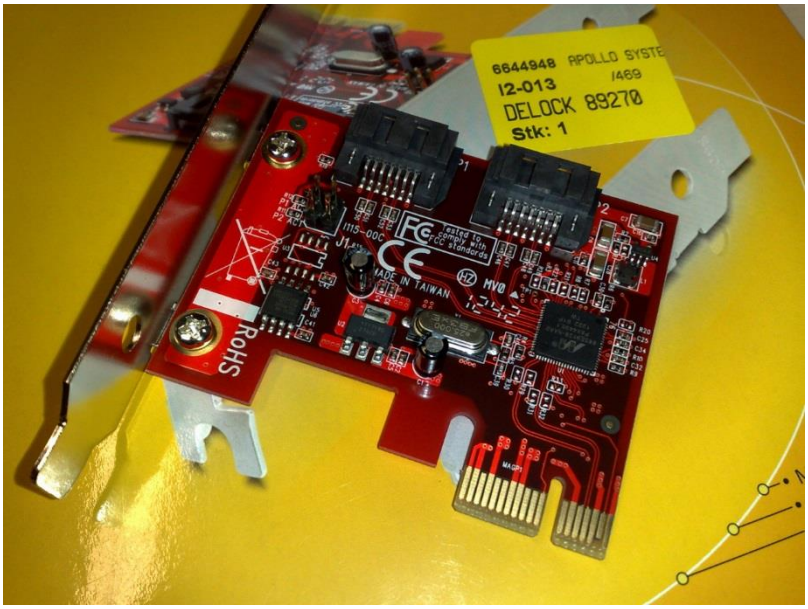
Many operating systems provide RAID implementations, including the following:

- Hewlett-Packard's OpenVMS operating system supports RAID 1. The mirrored disks, called a "shadow set", can be in different locations to assist in disaster recovery.
- Apple's macOS and macOS Server support RAID 0, RAID 1, and RAID 1+0.
- FreeBSD supports RAID 0, RAID 1, RAID 3, and RAID 5, and all nestings via GEOM modules and ccd.
- Linux's md supports RAID 0, RAID 1, RAID 4, RAID 5, RAID 6, and all nestings. Certain reshaping/resizing/expanding operations are also supported.
- Microsoft Windows supports RAID 0, RAID 1, and RAID 5 using various software implementations. Logical Disk Manager, introduced with Windows 2000, allows for the creation of RAID 0, RAID 1, and RAID 5 volumes by using dynamic disks, but this was limited only to professional and server editions of Windows until the release of Windows 8. Windows XP can be modified to unlock support for RAID 0, 1, and 5. Windows 8 and Windows Server 2012 introduced a RAID-like feature known as Storage Spaces, which also allows users to specify mirroring, parity, or no redundancy on a folder-by-folder basis. These options are similar to RAID 1 and RAID 5, but are implemented at a higher abstraction level.

- NetBSD supports RAID 0, 1, 4, and 5 via its software implementation, named RAIDframe.
- OpenBSD supports RAID 0, 1 and 5 via its software implementation, named softraid.

If a boot drive fails, the system has to be sophisticated enough to be able to boot from the remaining drive or drives. For instance, consider a computer whose disk is configured as RAID 1 (mirrored drives); if the first drive in the array fails, then a first-stage boot loader might not be sophisticated enough to attempt loading the second-stage boot loader from the second drive as a fallback. The second-stage boot loader for FreeBSD is capable of loading a kernel from such an array.

Firmware- and driver-based



A SATA 3.0 controller that provides RAID functionality through proprietary firmware and drivers

Software-implemented RAID is not always compatible with the system's boot process, and it is generally impractical for desktop versions of Windows. However, hardware RAID controllers are expensive and proprietary. To fill this gap, inexpensive "RAID controllers" were introduced that do not contain a dedicated RAID controller chip, but simply a standard drive controller chip with proprietary firmware and drivers. During early bootup, the RAID is implemented by the firmware and, once the operating system has been more completely loaded, the drivers take over control. Consequently, such controllers may not work when driver support is not available for the host operating system. An example is Intel Matrix RAID, implemented on many consumer-level motherboards.

Because some minimal hardware support is involved, this implementation is also called "hardware-assisted software RAID", "hybrid model" RAID, or even "fake RAID". If RAID 5 is

supported, the hardware may provide a hardware XOR accelerator. An advantage of this model over the pure software RAID is that—if using a redundancy mode—the boot drive is protected from failure (due to the firmware) during the boot process even before the operating systems drivers take over.

Integrity

Data scrubbing (referred to in some environments as *patrol read*) involves periodic reading and checking by the RAID controller of all the blocks in an array, including those not otherwise accessed. This detects bad blocks before use. Data scrubbing checks for bad blocks on each storage device in an array, but also uses the redundancy of the array to recover bad blocks on a single drive and to reassign the recovered data to spare blocks elsewhere on the drive.

Frequently, a RAID controller is configured to "drop" a component drive (that is, to assume a component drive has failed) if the drive has been unresponsive for eight seconds or so; this might cause the array controller to drop a good drive because that drive has not been given enough time to complete its internal error recovery procedure. Consequently, using consumer-marketed drives with RAID can be risky, and so-called "enterprise class" drives limit this error recovery time to reduce risk. Western Digital's desktop drives used to have a specific fix. A utility called WDTLER.exe limited a drive's error recovery time. The utility enabled TLER (time limited error recovery), which limits the error recovery time to seven seconds. Around September 2009, Western Digital disabled this feature in their desktop drives (e.g. the Caviar Black line), making such drives unsuitable for use in RAID configurations. However, Western Digital enterprise class drives are shipped from the factory with TLER enabled. Similar technologies are used by Seagate, Samsung, and Hitachi. For non-RAID usage, an enterprise class drive with a short error recovery timeout that cannot be changed is therefore less suitable than a desktop drive. In late 2010, the Smartmontools program began supporting the configuration of ATA Error Recovery Control, allowing the tool to configure many desktop class hard drives for use in RAID setups.

While RAID may protect against physical drive failure, the data is still exposed to operator, software, hardware, and virus destruction. Many studies cite operator fault as a common source of malfunction, such as a server operator replacing the incorrect drive in a faulty RAID, and disabling the system (even temporarily) in the process.

An array can be overwhelmed by catastrophic failure that exceeds its recovery capacity and the entire array is at risk of physical damage by fire, natural disaster, and human forces, however backups can be stored off site. An array is also vulnerable to controller failure because it is not always possible to migrate it to a new, different controller without data loss.

Weaknesses

Correlated failures

In practice, the drives are often the same age (with similar wear) and subject to the same environment. Since many drive failures are due to mechanical issues (which are more likely on older drives), this violates the assumptions of independent, identical rate of failure amongst drives; failures are in fact statistically correlated. In practice, the chances for a second failure before the first has been recovered (causing data loss) are higher than the chances for random failures. In a study of about 100,000 drives, the probability of two drives in the same cluster failing within one hour was four times larger than predicted by the exponential statistical distribution—which characterizes processes in which events occur continuously and independently at a constant average rate. The probability of two failures in the same 10-hour period was twice as large as predicted by an exponential distribution.

Unrecoverable read errors during rebuild

Unrecoverable read errors (URE) present as sector read failures, also known as latent sector errors (LSE). The associated media assessment measure, unrecoverable bit error (UBE) rate, is typically guaranteed to be less than one bit in 10^{15} for enterprise-class drives (SCSI, FC, SAS or SATA), and less than one bit in 10^{14} for desktop-class drives (IDE/ATA/PATA or SATA). Increasing drive capacities and large RAID 5 instances have led to the maximum error rates being insufficient to guarantee a successful recovery, due to the high likelihood of such an error occurring on one or more remaining drives during a RAID set rebuild. When rebuilding, parity-based schemes such as RAID 5 are particularly prone to the effects of UREs as they affect not only the sector where they occur, but also reconstructed blocks using that sector for parity computation.

Double-protection parity-based schemes, such as RAID 6, attempt to address this issue by providing redundancy that allows double-drive failures; as a downside, such schemes suffer from elevated write penalty—the number of times the storage medium must be accessed during a single write operation. Schemes that duplicate (mirror) data in a drive-to-drive manner, such as RAID 1 and RAID 10, have a lower risk from UREs than those using parity computation or mirroring between striped sets. Data scrubbing, as a background process, can be used to detect and recover from UREs, effectively reducing the risk of them happening during RAID rebuilds and causing double-drive failures. The recovery of UREs involves remapping of affected underlying disk sectors, utilizing the drive's sector remapping pool; in case of UREs detected during background scrubbing, data redundancy provided by a fully operational RAID set allows the missing data to be reconstructed and rewritten to a remapped sector.

Increasing rebuild time and failure probability

Drive capacity has grown at a much faster rate than transfer speed, and error rates have only fallen a little in comparison. Therefore, larger-capacity drives may take hours if not

days to rebuild, during which time other drives may fail or yet undetected read errors may surface. The rebuild time is also limited if the entire array is still in operation at reduced capacity. Given an array with only one redundant drive (which applies to RAID levels 3, 4 and 5, and to "classic" two-drive RAID 1), a second drive failure would cause complete failure of the array. Even though individual drives' mean time between failure (MTBF) have increased over time, this increase has not kept pace with the increased storage capacity of the drives. The time to rebuild the array after a single drive failure, as well as the chance of a second failure during a rebuild, have increased over time.

Some commentators have declared that RAID 6 is only a "band aid" in this respect, because it only kicks the problem a little further down the road. However, according to the 2006 NetApp study of Berriman et al., the chance of failure decreases by a factor of about 3,800 (relative to RAID 5) for a proper implementation of RAID 6, even when using commodity drives. Nevertheless, if the currently observed technology trends remain unchanged, in 2019 a RAID 6 array will have the same chance of failure as its RAID 5 counterpart had in 2010.

Mirroring schemes such as RAID 10 have a bounded recovery time as they require the copy of a single failed drive, compared with parity schemes such as RAID 6, which require the copy of all blocks of the drives in an array set. Triple parity schemes, or triple mirroring, have been suggested as one approach to improve resilience to an additional drive failure during this large rebuild time.

Atomicity: including parity inconsistency due to system crashes

A system crash or other interruption of a write operation can result in states where the parity is inconsistent with the data due to non-atomicity of the write process, such that the parity cannot be used for recovery in the case of a disk failure (the so-called RAID 5 write hole). The RAID write hole is a known data corruption issue in older and low-end RAIDs, caused by interrupted destaging of writes to disk. The write hole can be addressed with write-ahead logging. Recently mdadm fixed it by introducing a dedicated journaling device (to avoid performance penalty, typically, SSDs and NVMs are preferred) for that purpose.

This is a little understood and rarely mentioned failure mode for redundant storage systems that do not utilize transactional features. Database researcher Jim Gray wrote "Update in Place is a Poison Apple" during the early days of relational database commercialization.

Write-cache reliability

There are concerns about write-cache reliability, specifically regarding devices equipped with a write-back cache, which is a caching system that reports the data as written as

soon as it is written to cache, as opposed to when it is written to the non-volatile medium. If the system experiences a power loss or other major failure, the data may be irrevocably lost from the cache before reaching the non-volatile storage. For this reason good write-back cache implementations include mechanisms, such as redundant battery power, to preserve cache contents across system failures (including power failures) and to flush the cache at system restart time.

Source: Wikipedia, https://en.wikipedia.org/wiki/RAID

Unit 8: Parallel Processing

This unit will address several advanced topics in computer architecture, focusing on the reasons for and the consequences of the recent switch from sequential processing to parallel processing by hardware producers. You will learn that parallel programming is not easy and that parallel processing imposes certain limitations in performance gains, as seen in the well-known Amdahl's law. You will also look into the concepts of shared memory multiprocessing and cluster processing as two common means of improving performance with parallelism. The unit will conclude with a look at some of the programming techniques used in the context of parallel machines.

- Upon successful completion of this unit, you will be able to:
  - explain the basics of parallel programming;
  - use Amdahl's law to predict performance improvements for parallel processing;
  - describe the approaches to parallelism used to improve performance and overcome the limits of current physical devices and their fabrication; and
  - identify reasons for and consequences of the recent switch from sequential processing to parallel processing among hardware manufacturers.

- 8.1: The Reason for the Switch to Parallel Processing

Multi-Core Chips

Read section 1.3 to learn about multi-core chips. These two pages give a summary of processor and chip trends to overcome the challenge of increasing performance and addressing the heat problem of a single core.

1.3 Multi-core chips

In recent years, the limits of performance have been reached for the traditional processor chip design.

- Clock frequency can not increased further, since it increases energy consumption, heating the chips too much. Figure 1.3 gives a dramatic illustration of the heat

that a chip would give off, if single-processor trends had continued. The reason for this is that the power dissipation of a chip is propertional to the voltage squared times the frequency. Since voltage and frequency are proportional, that makes power proportional to the third power of the frequency.

- It is not possible to extract more instruction-level parallelism from codes, either because of compiler lim-itations, because of the limited amount of intrinsically available parallelism, or because branch prediction makes it impossible (see section 1.1.1.3).

One of the ways of getting a higher utilization out of a single processor chip is then to move from a strategy of further sophistication of the single processor, to a division of the chip into multiple processing 'cores.' The separate cores can work on unrelated tasks, or, by introducing what is in effect data parallelism (section 2.2.1), collaborate on a common task at a higher overall efficiency.

While first multi-core chips were simply two processors on the same die, later generations incorporated L2 caches that were shared between the two processor cores. This design makes it efficient for the cores to work jointly on the same problem. The cores would still have their own L1 cache, and these separate caches lead to a *cache coherence* problem; see section 1.3.1 below.

We note that the term 'processor' is now ambiguous: it can refer to either the chip, or the processor core on the chip. For this reason, we mostly talk about a *socket* for the whole chip and *core* for part containing one arithmetic and logic unit and having its own registers. Currently, CPUs with 4 or 6 cores are on the market and 8-core chips will be available shortly. The core count is likely to go up in the future: Intel has already shown an 80-core prototype that is developed into the 48 core 'Single-chip Cloud Computer', illustrated in fig 1.4. This chip has a structure with 24 dual-core 'tiles' that are connected through a 2D mesh network. Only certain tiles are connected to a memory controller, others can not reach memory other than through the on-chip network.

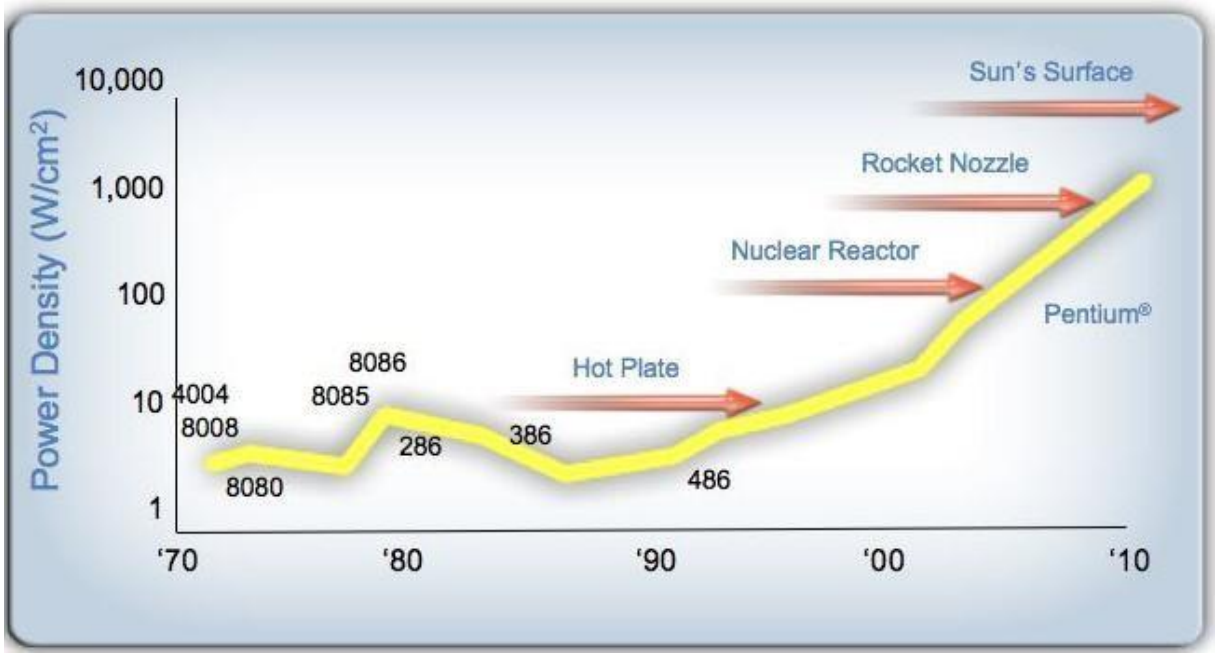


Figure 1.3: Projected heat dissipation of a CPU if trends had continued – this graph courtesy Pat Helsinger

With this mix of shared and private caches, the programming model for multi-core processors is becoming a hybrid between shared and distributed memory:

Core: The cores have their own private L1 cache, which is a sort of distributed memory. The above mentioned Intel 80-core prototype has the cores communicating in a distributed memory fashion.

Socket: On one socket, there is often a shared L2 cache, which is shared memory for the cores.

Node: There can be multiple sockets on a single 'node' or motherboard, accessing the same shared memory.

Network: Distributed memory programming (see the next chapter) is needed to let nodes communicate.

1.3.1 Cache coherence

With parallel processing, there is the potential for a conflict if more than one processor has a copy of the same data item. The problem of ensuring that all cached data are an accurate copy of main memory, is referred to as *cache coherence*.

In distributed memory architectures, a dataset is usually partitioned disjointly over the processors, so conflicting copies of data can only arise with knowledge of the user, and it is up to the user to prevent deal with the problem. The case of shared memory is more subtle: since processes access the same main memory, it would seem that conflicts are in fact impossible. However, processor typically have some private cache, which contains

copies of data from memory, so conflicting copies can occur. This situation arises in particular in multi-core designs.

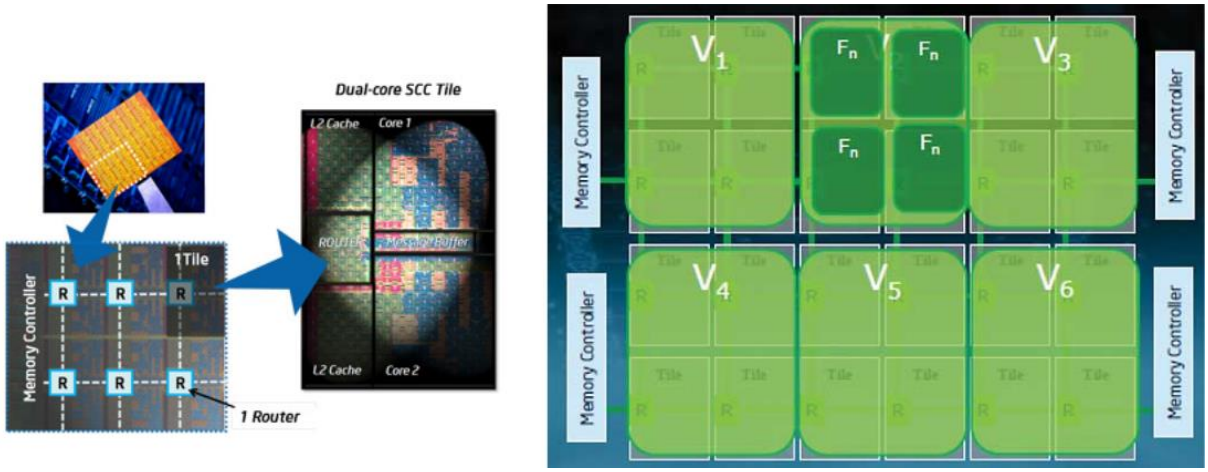


Figure 1.4: Structure of the Intel Single-chip Cloud Computer chip

Suppose that two cores have a copy of the same data item in their (private) L1 cache, and one modifies its copy. Now the other has cached data that is no longer an accurate copy of its counterpart, so it needs to reload that item. This will slow down the computation, and it wastes bandwidth to the core that could otherwise be used for loading or storing operands.

The state of a cache line with respect to a data item in main memory is usually described as one of the following:

Scratch: the cache line does not contain a copy of the item;

Valid: the cache line is a correct copy of data in main memory;

Reserved: the cache line is the *only* copy of that piece of data;

Dirty: the cache line has been modified, but not yet written back to main memory;

Invalid: the data on the cache line is also present on other processors (it is not reserved), and another process has modified its copy of the data.

Exercise 1.9. Consider two processors, a data item x in memory, and cachelines x_1,x_2 in the private caches of the two processors to which x is mapped. Describe the transitions between the states of x_1 and x_2 under reads and writes of x on the two processors. Also indicate which actions cause memory bandwidth to be used. (This list of transitions is a Finite State Automaton (FSA); see section A.3.)

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

8.2: Limitations in Parallel Processing: Amdahl's Law

Introduction to Parallel Computer Architecture

Read sections 2.1 and 2.2 to learn about parallel computer architectures. There are different types of parallelism: there is instruction-level parallelism, where a stream of instructions is simultaneously in partial stages of execution by a single processor; there are multiple streams of instructions, which are simultaneously executed by multiple processors. A quote from the beginning of the chapter states the key ideas:

"In this chapter, we will analyze this more explicit type of parallelism, the hardware that supports it, the programming that enables it, and the concepts that analyze it."

This chapter begins with a simple scientific computation example, followed by a description of SISD, SIMD, MISD, and MIMD architectures."

The largest and most powerful computers are sometimes called 'supercomputers'. For the last few decades, this has, without exception, referred to parallel computers: machines with more than one CPU that can be set to work on the same problem.

Parallelism is hard to define precisely, since it can appear on several levels. In the previous chapter you already saw how inside a CPU several instructions can be 'in flight' simultaneously. This is called *instruction-level parallelism*, and it is outside explicit user control: it derives from the compiler and the CPU deciding which instructions, out of a single instruction stream, can be processed simultaneously. At the other extreme is the sort of parallelism where more than one instruction stream is handled by multiple processors, often each on their own circuit board. This type of parallelism is typically explicitly scheduled by the user.

In this chapter, we will analyze this more explicit type of parallelism, the hardware that supports it, the programming that enables it, and the concepts that analyze it.

For further reading, a good introduction to parallel computers and parallel programming is Wilkinson and Allen [85].

2.1 Introduction

In scientific codes, there is often a large amount of work to be done, and it is often regular to some extent, with the same operation being performed on many data. The question is then whether this work can be sped up by use of a parallel computer. If there are n operations to be done, and they would take time t on a single processor, can they be done in time t/p on p processors?

Let us start with a very simple example. Adding two vectors of length n

```
for (i=0; i<n; i++)
  x[i] += y[i]
```
can be done with up to n processors, and execution time is linearly reduced with the number of processors. If each operation takes a unit time, the original algorithm takes time *n*, and the parallel execution on *p* processors *n/p*. The parallel algorithm is faster by a factor of *p*.

Next, let us consider summing the elements of a vector.

```
s = 0;
for (i=0; i<n; i++)
  s += x[i]
```
This is no longer obviously parallel, but if we recode the loop as

```
for (s=2; s<n; s*=2)
  for (i=0; i<n; i+=s)
    x[i] += x[i+s/2]
```
there is a way to parallelize it: every iteration of the outer loop is now a loop that can be done by *n/s* processors in parallel. Since the outer loop will go through $\log_2 n$log2n iterations, we see that the new algorithm has a reduced runtime of $n/p \cdot \log_2 n$<spanstyle="font-size:0.9375rem;">n/p·log2n<spanstyle="font-size:0.9375rem;">. The parallel algorithm is now faster by a factor of $p/\log_2 n$p/log2n.

Even from these two simple examples we can see some of the characteristics of parallel computing:

- Sometimes algorithms need to be rewritten slightly to make them parallel.
- A parallel algorithm may not show perfect speedup.

There are other things to remark on. In the first case, if there are $p = n$ processors, and each has its x_i, y_i in a local store, the algorithm can be executed without further complications. In the second case, processors need to *communicate* data among each other. What's more, if there is a concept of distance between processors, then each iteration of the outer loop increases the distance over which communication takes place.

These matters of algorithm adaptation, efficiency, and communication, are crucial to all of parallel computing. We will return to this issues in various guises throughout this chapter.

2.2 Parallel Computers Architectures

For quite a while now, the top computers have been some sort of parallel computer, that is, an architecture that allows the simultaneous execution of multiple instructions or instruction sequences. One way of characterizing the various forms this can take is due to Flynn [36]. Flynn's taxonomy distinguishes between whether one or more different instructions are executed simultaneously, and between whether that happens on one or more data items. The following four types result, which we will discuss in more detail below:

SISD: Single Instruction Single Data: this is the traditional CPU architecture: at any one time only a single instruction is executed, operating on a single data item.

SIMD: Single Instruction Multiple Data: in this computer type there can be multiple processors, each operating on its own data item, but they are all executing the same instruction on that data item. Vector computers (section 2.2.1.1) are typically also characterized as SIMD.

MISD: Multiple Instruction Single Data. No architectures answering to this description exist.

MIMD: Multiple Instruction Multiple Data: here multiple CPUs operate on multiple data items, each executing independent instructions. Most current parallel computers are of this type

2.2.1 SIMD

Parallel computers of the SIMD type apply the same operation simultaneously to a number of data items. The design of the CPUs of such a computer can be quite simple, since the arithmetic unit does not need separate logic and instruction decoding units: all CPUs execute the same operation in lock step. This makes SIMD computers excel at operations on arrays, such as

```
for (i=0; i<N; i++) a[i] = b[i]+c[i];
```

and, for this reason, they are also often called *array* processors. Scientific codes can often be written so that a large fraction of the time is spent in array operations.

On the other hand, there are operations that can not can be executed efficiently on an array processor. For instance, evaluating a number of terms of a recurrence $x_{i+1} = ax_i + b_i$ involves that many additions and multiplications, but they alternate, so only one operation of each type can be processed at any one time. There are no arrays of numbers here that are simultaneously the input of an addition or multiplication.

In order to allow for different instruction streams on different parts of the data, the processor would have a 'mask bit' that could be set to prevent execution of instructions. In code, this typically looks like

```
where (x>0) {
  x[i] = sqrt(x[i])
```

The programming model where identical operations are applied to a number of data items simultaneously, is known as *data parallelism*.

Such array operations can occur in the context of physics simulations, but another important source is graphics applications. For this application, the processors in an array processor can be much weaker than the processor in a PC: often they are in fact bit processors, capable of operating on only a single bit at a time. Along these lines, ICL had

the 4096 processor DAP [56] in the 1980s, and Goodyear built a 16K processor MPP [14] in the 1970s.

Later, the Connection Machine (CM-1, CM-2, CM-5) were quite popular. While the first Connection Machine had bit processors (16 to a chip), the later models had traditional processors capable of floating point arithmetic, and were not true SIMD architectures. All were based on a hyper-cube interconnection network; see section 2.6.4. Another manufacturer that had a commercially successful array processor was MasPar.

Supercomputers based on array processing do not exist anymore, but the notion of SIMD lives on in various guises, which we will now discuss.

2.2.1.1 Pipelining

A number of computers have been based on a *vector processor* or *pipeline processor* design. The first commercially successful supercomputers, the Cray-1 and the Cyber-205 were of this type. In recent times, the Cray-X1 and the NEC SX series have featured vector pipes. The 'Earth Simulator' computer [75], which led the TOP500 (section 2.11) for 3 years, was based on NEC SX processors. The general idea behind pipelining was described in section 1.1.1.1.

While supercomputers based on pipeline processors are in a distinct minority, pipelining is now mainstream in the superscalar CPUs that are the basis for clusters. A typical CPU has pipelined floating point units, often with separate units for addition and multiplication; see the previous chapter.

However, there are some important differences between pipelining in a modern superscalar CPU and in, more old-fashioned, vector units. The pipeline units in these vector computers are not integrated floating point units in the CPU, but can better be considered as attached vector units to a CPU that itself has a floating point unit. The vector unit has vector registers[2] with a typical length of 64 floating point numbers; there is typically no 'vector cache'. The logic in vector units is also simpler, often addressable by explicit vector instructions. Superscalar CPUs, on the other hand, are more complicated and geared towards exploiting data streams in unstructured code.

2.2.1.2 True SIMD in CPUs and GPUs

True SIMD array processing can be found in modern CPUs and GPUs, in both cases inspired by the parallelism that is needed in graphics applications.

Modern CPUs from Intel and AMD, as well as PowerPC chips, have instructions that can perform multiple instances of an operation simultaneously. On Intel processors this is known as SSE: Streaming SIMD Extensions. These extensions were originally intended for graphics processing, where often the same operation needs to be performed on a large number of pixels. Often, the data has to be a total of, say, 128 bits, and this can be

divided into two 64-bit reals, four 32-bit reals, or a larger number of even smaller chunks such as 4 bits.

Current compilers can generate SSE instructions automatically; sometimes it is also possible for the user to insert pragmas, for instance with the Intel compiler:

```
void func(float *restrict c, float *restrict a, float *restrict b, int n)
{
#pragma vector always
  for (int i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Use of these extensions often requires data to be aligned with cache line boundaries (section 1.2.4.3), so there are special allocate and free calls that return aligned memory.

For a nontrivial example, see figure 2.1, which describes complex multiplication using SSE3.

Array processing on a larger scale can be found in *Graphics Processing Unit (GPU)*s. A GPU contains a large number of simple processors, ordered in groups of 32, typically. Each processor group is limited to executing the same instruction. Thus, this is true example of Single Instruction Multiple Data (SIMD) processing.

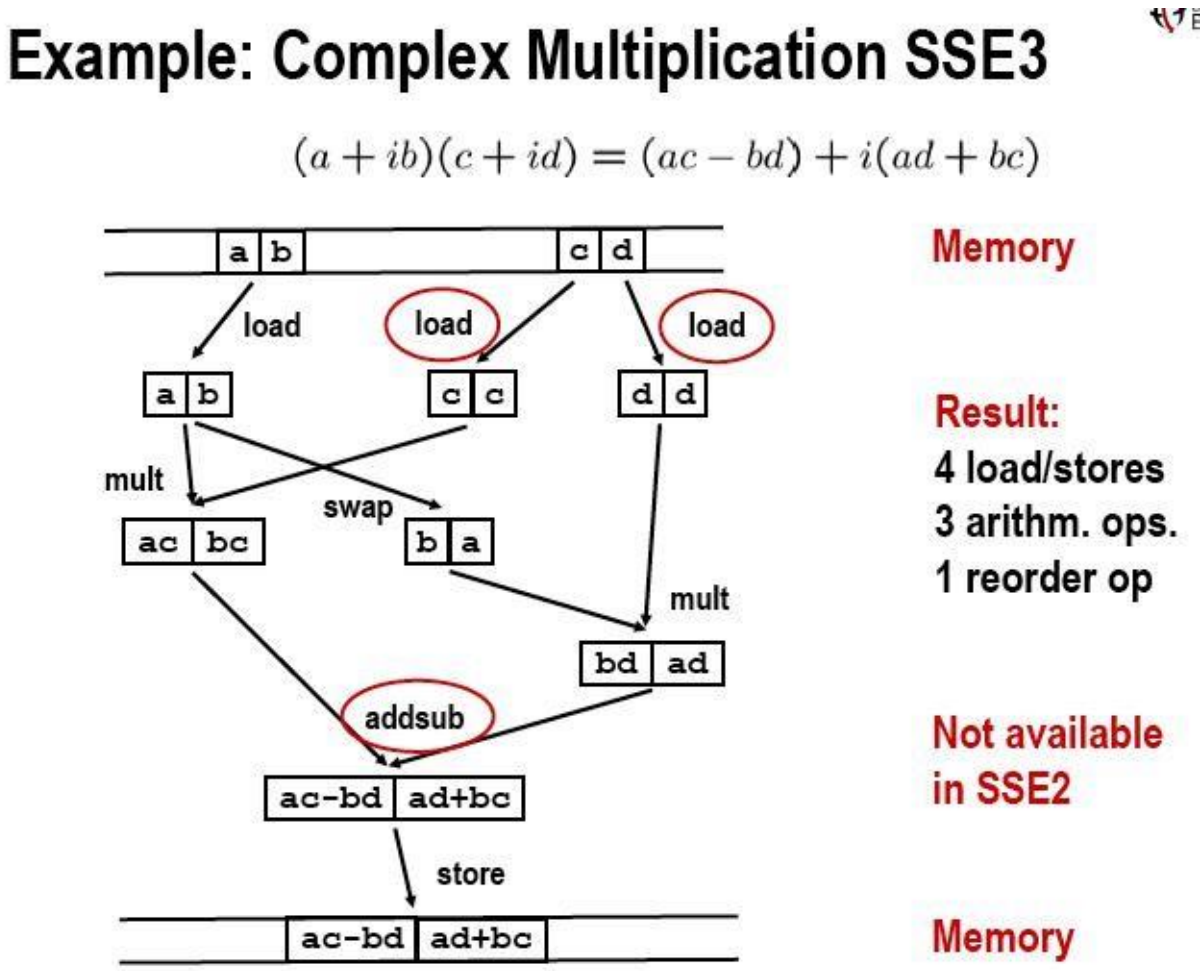


Figure 2.1: Complex multiplication with SSE3

2.2.2 MIMD / SPMD computers

By far the most common parallel computer architecture these days is called Multiple Instruction Multiple Data (MIMD): the processors execute multiple, possibly differing instructions, each on their own data. Saying that the instructions differ does not mean that the processors actually run different programs: most of these machines operate in Single Program Multiple Data (SPMD) mode, where the programmer starts up the same executable on the parallel processors. Since the different instances of the executable can take differing paths through conditional statements, or execute differing numbers of iterations of loops, they will in general not be completely in sync as they were on SIMD machines. This lack of synchronization is called *load unbalance*, and it is a major source of less than perfect *speedup*.

There is a great variety in MIMD computers. Some of the aspects concern the way memory is organized, and the network that connects the processors. Apart from these hardware aspects, there are also differing ways of programming these machines. We will see all these aspects below. Machines supporting the SPMD model are usually called *clusters*. They can be built out of custom or commodity processors; if they consist of PCs, running Linux, and connected with Ethernet, they are referred to as Beowulf clusters [49].

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

Parallelism and Performance

Study the "Amdahl's Law" section. Amdahl's law explains the limitations to performance gains achievable through parallelism. Over the last several decades or so, increases in computer performance have largely come from improvements to current hardware technologies and less from software technologies. Now, however, the limits to these improvements may be near. For significant continued performance improvement, either new physical technology needs to be discovered and/or transitioned to practice, or software techniques will have to be developed to get significant gains in computing performance.

In the equation for Amdahl's law, P is the fraction of code that can be parallelized (that is, the part that must be executed serially); S is the fraction of code that cannot be parallelized; and n is the number of processors. P + S is 1. If there are n processors, then P + S can be executed in the same time that P/n + S can be executed. Thus, the ratio of the time using 1 processor to the time of using n processors is 1/(P/n + S). This is the speedup in going from 1 processor to n processors.

The speedup is limited, even for large n. If n is 1, the speedup is 1. If n is very large, then the speedup is 1/S. If P is very small, then P/n is even smaller, and P/n + S is approximately S. The speedup is 1/S, but S is approximately S + P, which is 1. Therefore, the speed of execution of this code using 1 processor is about the same as using n processors.

Another way of writing Amdahl's law is 1/(P/n + [1 - P]). Thus, if P is close to 1, the speedup is 1/(P/n) or n/P, which is approximately n.

Apply Amdahl's law to better understand how it works by substituting a variety of numeric values into this equation and sketching the graph of the equation.

Speedup

The speed of a program is the time it takes the program to execute. This could be measured in any increment of time. Speedup is defined as the time it takes a program to execute sequentially (with one processor) divided by the time it takes to execute in parallel (with many processors). The formula for speedup is:

$S_p = T_1 / T_p$

Where:

S_p

The speedup obtained from using *p* processors.

T_1

The time it takes the program to be executed sequentially.

T_p

The time it takes the program to be executed in parallel using *p* processors.

Linear Speedup

Linear speedup or **ideal speedup** is obtained when $S_p = p$. When running an algorithm with linear speedup, doubling the number of processors doubles the speed. As this is ideal, it is considered very good scalability.

Amdahl's Law

Amdahl's law states that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

As a formula:

$S_p = 1 / (F + (1 - F) / p)$

Where:

p

The number of processors.

S_p

The speedup obtained from using *p* processors.

F

The fraction of the program that must be executed sequentially. $0 \leq F \leq 1$.

If *p* tends to ∞, the maximum speedup tends to 1 / F.

Limits and Costs of Parallel Programming

From section 4 of Chapter 3 of this resource, study the section titled "Amdahl's Law."

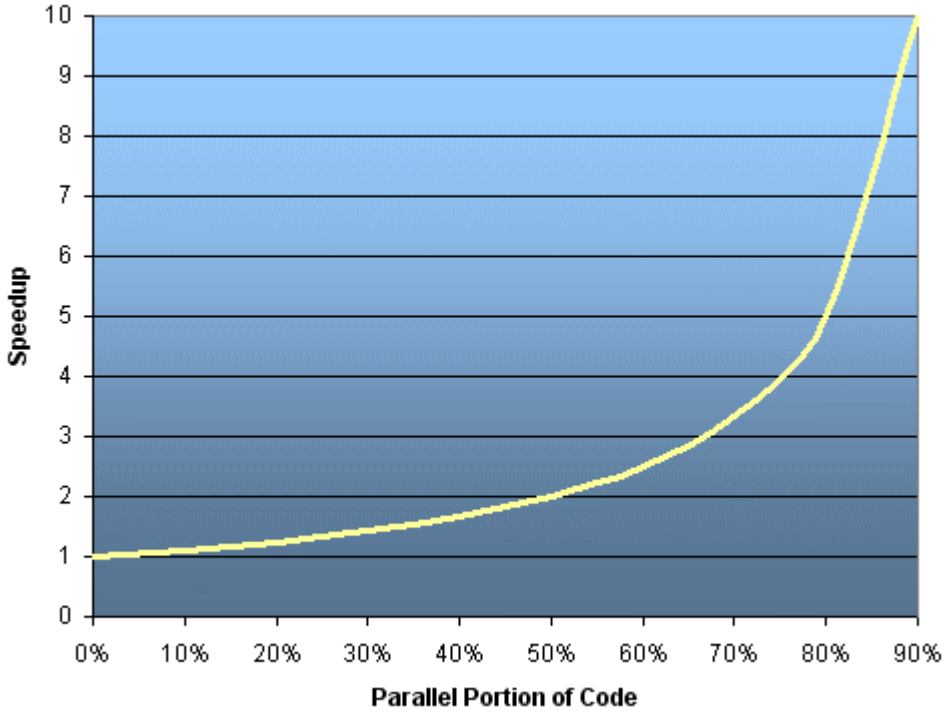
Limits and Costs of Parallel Programming

Amdahl's Law:

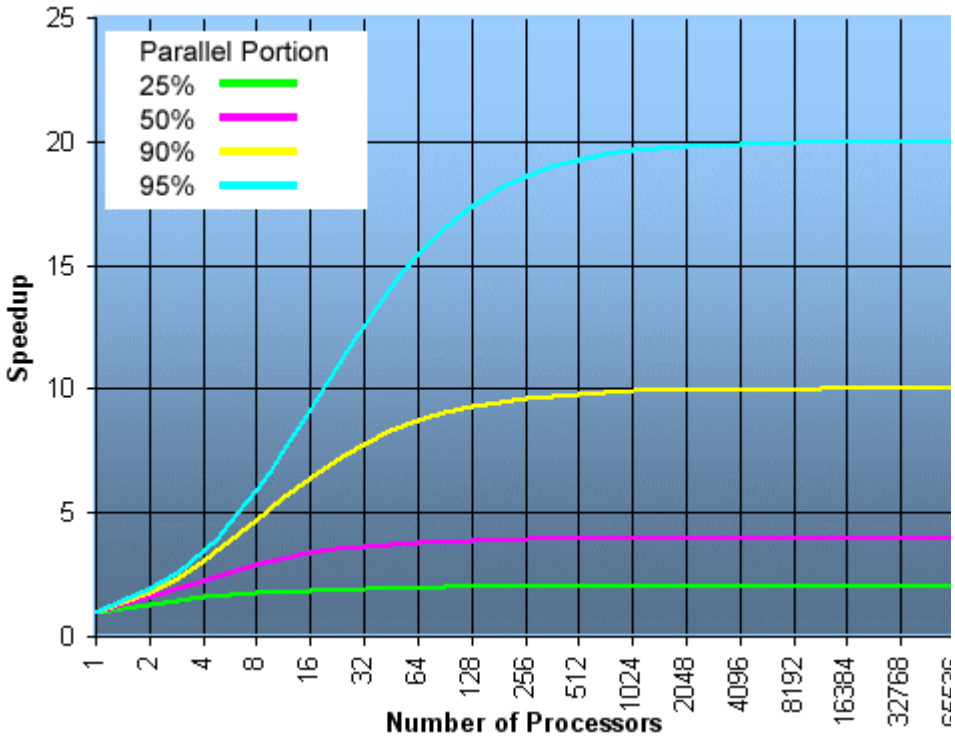
- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

```
1
     speedup     =
--------

1   - P
```



- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.



- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

```
1
    speedup   =
------------

P   +  S

---

N
```

- where P = parallel fraction, N = number of processors and S = serial fraction.

- It soon becomes obvious that there are limits to the scalability of parallelism. For example:

	speedup			
N	P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90

"Famous" quote: *You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!*

- However, certain problems demonstrate increased performance by increasing the problem size. For example:

```
2D Grid Calculations     85 seconds   85%
Serial fraction          15 seconds   15%
```

We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

```
2D Grid Calculations     680 seconds   97.84%
Serial fraction           15 seconds    2.16%
```

- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Source: Blaise Barney, https://computing.llnl.gov/tutorials/parallel_comp/

8.3: Shared Memory and Distributed Memory Multiprocessing

Multiprocessing

Study this material, which focus on the problem of parallel software and discusses scaling by using an example to explain shared memory and message passing, and identify some problems related to cache and memory consistency.

Shared Memory and Distributed Multiprocessing

Bhanu Kapoor, Ph.D.

Issue with Parallelism

- Parallel software is the problem
- Need to get significant performance improvement
    - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
    - Partitioning
    - Coordination
    - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
    - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
    - Speedup = $1 / [(1 - F_{parallelizable}) + F_{parallelizable} / 100] = 90$
    - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = (10 + 100) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 100/10 × t_{add} = 20 × t_{add}
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

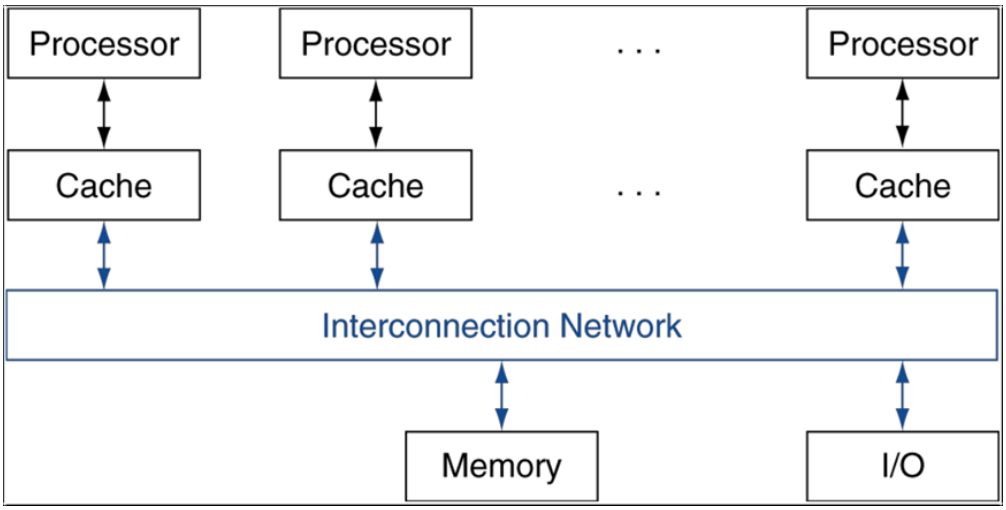
- What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010 × t_{add}
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Strong vs Weak Scaling

- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = 20 × t_{add}
  - 100 processors, 32 × 32 matrix
    - Time = 10 × t_{add} + 1000/100 × t_{add} = 20 × t_{add}
  - Constant performance in this example

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



Example: Sum Reduction

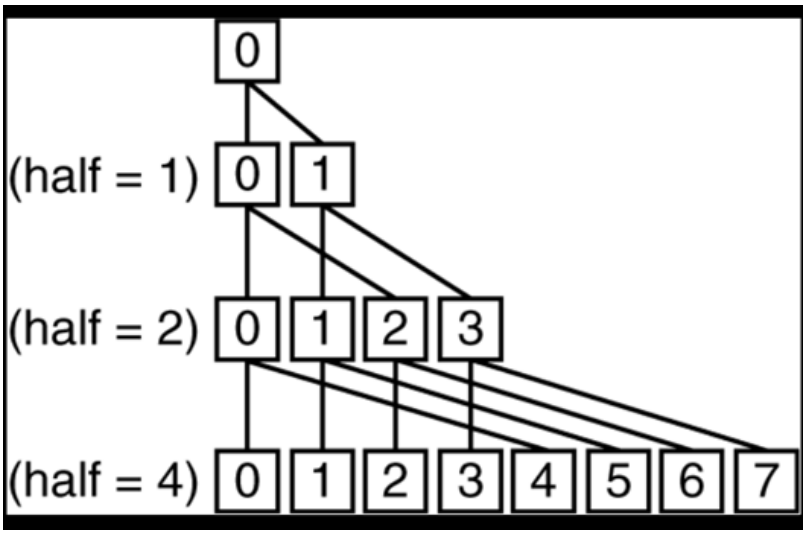
- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
    i < 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

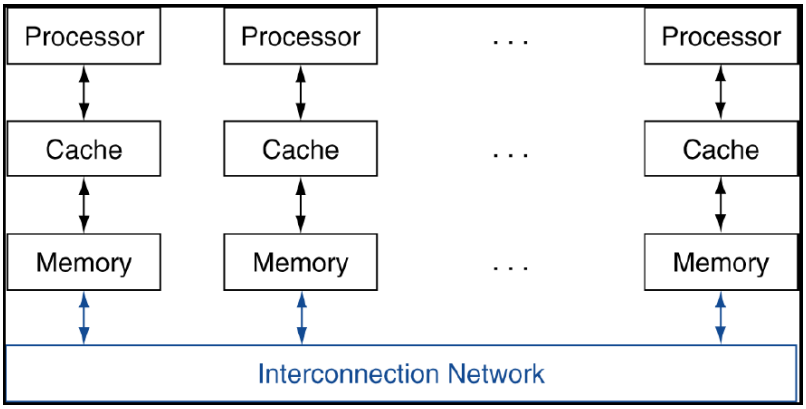
Example: Sum Reduction

```
half = 100;
repeat
  synch();
    if (half%2 != 0 && Pn == 0) sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd; Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */ if (Pn < half) sum[Pn] = sum[Pn]
+ sum[Pn+half];
until (half == 1);
```



Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Sum Reduction (Again)

- Sum 100,000 on 100 processors

- First distribute 1000 numbers to each
    - The do partial sums

```
sum = 0;
  for (i = 0; i<1000; i = i + 1) sum = sum + AN[i];
```

- Reduction
    - Half the processors send, other half receive and add
    - The quarter send, quarter receive and add, …

Sum Reduction (Again)

- Given send() and receive() operations

```
limit = 100; half = 100;/* 100 processors */ repeat
  half = (half+1)/2; /* send vs. receive dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
limit = half; /* upper limit of senders */ until (half == 1); /* exit with final
sum */
```

    - Send/receive also provide synchronization
    - Assumes send/receive take similar time to addition

Cache Coherence Problem

- Suppose two CPU cores share a physical address space
    - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

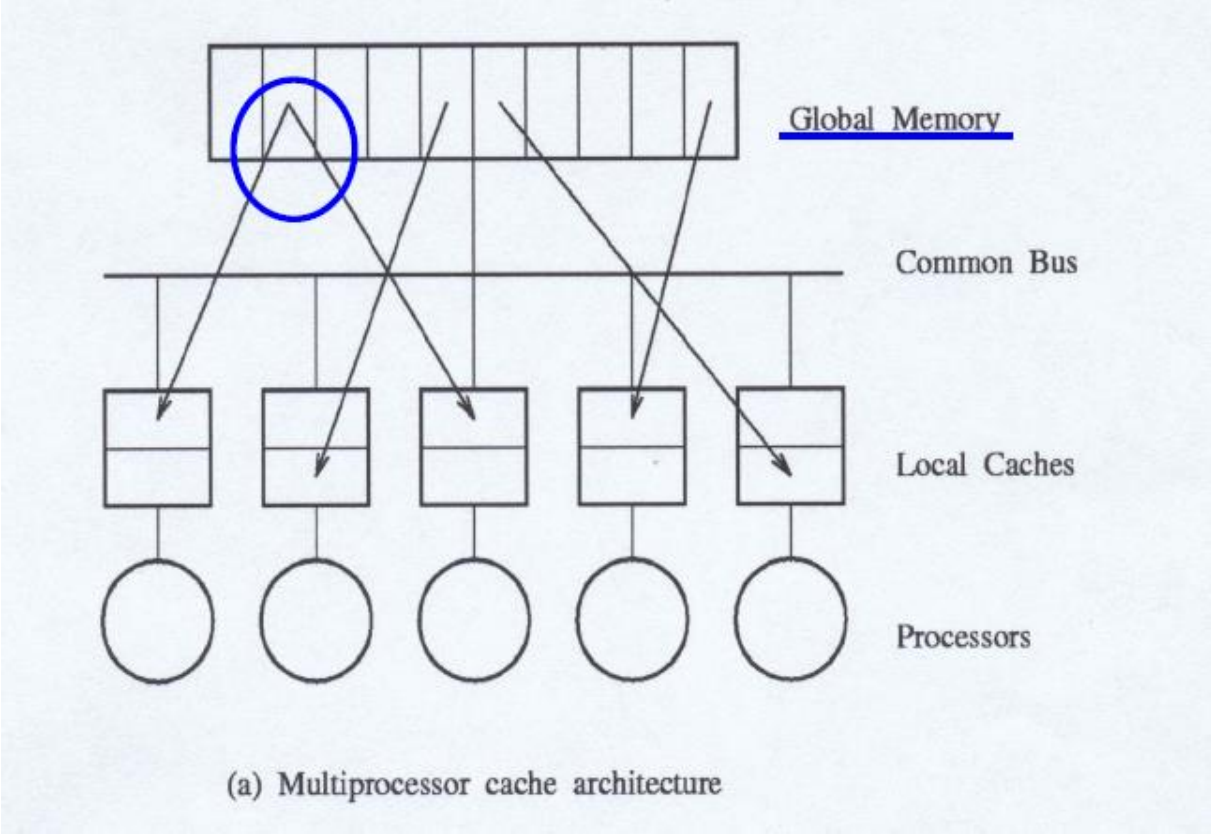
- Informally: Reads return most recently written value
- Formally:
    - P writes X; P reads X (no intervening writes)
        - read returns written value
    - P_1 writes X; P_2 reads X (sufficiently later)
        - read returns written value
            - c.f. CPU B reading X after step 3 in example

- P_1 writes X, P_2 writes X
  - all processors see writes in the same order
    - End up with the same final value for X

Memory Consistency

- When are writes seen by other processors
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    - all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Multiprocessor Caches



(a) Multiprocessor cache architecture

- Caches provide [for shared items]
  - Migration
  - Replication
- Migration Reduces
  - Latency
  - Bandwidth demands

- Replication reduces
    - Latency
    - Contention for a read of shared item

Source: Saylor Academy

8.4: Multicore Processors and Programming with OpenMP and MPI

Parallel Programming

Read section 2.5, which covers two extreme approaches to parallel programming. First, parallelism is handled by the lower software and hardware layers. OpenMP is applicable in this first case. Secondly, parallelism is handled by the programmer. MPI is applicable in the second case.

2.5 Parallel programming

Parallel programming is more complicated than sequential programming. While for sequential programming most programming languages operate on similar principles (some exceptions such as functional or logic languages aside), there is a variety of ways of tackling parallelism. Let's explore some of the concepts and practical aspects.

There are various approaches to parallel programming. One might say that they fall in two categories:

- Let the programmer write a program that is essentially the same as a sequential program, and let the lower software and hardware layers solve the parallelism problem; and
- Expose the parallelism to the programmer and let the programmer manage everything explicitly.

The first approach, which is more user-friendly, is in effect only possible with shared memory, where all processes have access to the same address space. We will discuss this in the section on OpenMP 2.5.1. The second approach is necessary in the case of distributed memory programming. We will have a general discussion of distributed programming in section 2.5.2; section 2.5.3 will discuss the MPI library.

2.5.1 OpenMP

OpenMP is an extension to the programming languages C and Fortran. Its main approach to parallelism is the parallel execution of loops: based on compiler directives, a preprocessor can schedule the parallel execution of the loop iterations.

The amount of parallelism is flexible: the user merely specifies a parallel region, indicating that all iterations are independent to some extent, and the runtime system will then use whatever resources are available. Because of this dynamic nature, and because no data distribution is specified, OpenMP can only work with threads on shared memory.

OpenMP is neither a language nor a library: it operates by inserting directives into source code, which are interpreted by the compiler. Many compilers, such as GCC or the Intel compiler, support the OpenMP extensions. In Fortran, OpenMP directives are placed in comment statements; in C, they are placed in #pragma CPP directives, which indicate compiler specific extensions. As a result, OpenMP code still looks like legal C or Fortran to a compiler that does not support OpenMP. Programs need to be linked to an OpenMP runtime library, and their behaviour can be controlled through environment variables.

OpenMP features *dynamic parallelism*: the number of execution streams operating in parallel can vary from one part of the code to another.

For more information about OpenMP, see [21].

2.5.1.1 Processes and threads

OpenMP is based on 'threads' working in parallel; see section 2.5.1.3 for an illustrative example. A thread is an independent instruction stream, but as part of a Unix process. While processes can belong to different users, or be different programs that a single user is running concurrently, and therefore have their own data space, threads are part of one process and therefore share each other's data. Threads do have a possibility of having private data, for instance, they have their own data stack.

Threads serve two functions:

1. By having more than one thread on a single processor, a higher processor utilization can result, since the instructions of one thread can be processed while another thread is waiting for data.
2. In a shared memory context, multiple threads running on multiple processors or processor cores can be an easy way to parallelize a process. The shared memory allows the threads to all see the same data.

2.5.1.2 Issues in shared memory programming

Shared memory makes life easy for the programmer, since every processor has access to all of the data: no explicit data traffic between the processor is needed. On the other hand, multiple processes/processors can also write to the same variable, which is a source of potential problems.

Suppose that two processes both try to increment an integer variable I by one:

process 1: I=I+2

process 2: I=I+3

If the processes are not completely synchronized, one will read the current value, compute the new value, write it back, and leave that value for the other processor to find.

In this scenario, the parallel program has the same result (I=I+5) as if all instructions were executed sequentially.

However, it could also happen that both processes manage to read the current value simultaneously, compute their own new value, and write that back to the location of I. Even if the conflicting writes can be reconciled, the final result will be wrong: the new value will be either I+2 or I+3, not I+5. Moreover, it will be indeterminate, depending on details of the execution mechanism.

For this reason, such updates of a shared variable are called a *critical section* of code. OpenMP has a mechanism to declare a critical section, so that it will be executed by only one process at a time. One way of implementing this, is to set a temporary lock on certain memory areas. Another solution to the update problem, is to have atomic operations: the update would be implemented in such a way that a second process can not get hold of the data item being updated. One implementation of this is *transactional memory*, where the hardware itself supports atomic operations; the term derives from database transactions, which have a similar integrity problem.

Finally, we mention the semaphore mechanism for dealing with critical sections [26]. Surrounding each critical section there will be two atomic operations controlling a semaphore. The first process to encounter the semaphore will lower it, and start executing the critical section. Other processes see the lowered semaphore, and wait. When the first process finishes the critical section, it executes the second instruction which raises the semaphore, allowing one of the waiting processes to enter the critical section.

2.5.1.3 Threads example

The following example spawns a number of tasks that all update a global counter. Since threads share the same memory space, they indeed see and update the same memory location.

```
#include <stdlib.h>
#include <stdio.h>
#include "pthread.h"
int sum=0;
void adder() {
  sum = sum+1;
  return;
}
#define NTHREADS 50
int main() {
  int i;
  pthread_t threads[NTHREADS];
  printf("forking\n");
  for (i=0; i<NTHREADS; i++)
    if (pthread_create(threads+i,NULL,&adder,NULL)!=0) return i+1;
printf("joining\n");
  for (i=0; i<NTHREADS; i++)
```

```
    if (pthread_join(threads[i],NULL)!=0) return NTHREADS+i+1; printf("Sum
computed: %d\n",sum);
  return 0;
}
```
The fact that this code gives the right result is a coincidence: it only happens because updating the variable is so much quicker than creating the thread. (On a multicore processor the chance of errors will greatly increase.) If we artificially increase the time for the update, we will no longer get the right result:

```
void adder() {
  int t = sum; sleep(1); sum = t+1;
  return;
}
```
Now all threads read out the value of sum, wait a while (presumably calculating something) and then update.

This can be fixed by having a lock on the code region that should be 'mutually exclusive':

```
pthread_mutex_t lock;
void adder() {
  int t,r;
  pthread_mutex_lock(&lock);
    t = sum; sleep(1); sum = t+1;
    pthread_mutex_unlock(&lock);
    return;
  }
int main() {
  ....
  pthread_mutex_init(&lock,NULL);
```
The lock and unlock commands guarantee that no two threads can interfere with each other's update.

2.5.1.4 OpenMP examples

The simplest example of OpenMP use is the parallel loop.

```
#pragma omp for
for (i=0; i<ProblemSize; i++) {
  a[i] = b[i];
}
```
Clearly, all iterations can be executed independently and in any order. The pragma CPP directive then conveys this fact to the compiler.

Some loops are fully parallel conceptually, but not in implementation:

```
for (i=0; i<ProblemSize; i++) {
  t = b[i]*b[i];
  a[i] = sin(t) + cos(t);
}
```
Here it looks as if each iteration writes to, and reads from, a shared variable t. However, t is really a temporary variable, local to each iteration. OpenMP indicates that as follows:

```
#pragma parallel for shared(a,b), private(t)
  for (i=0; i<ProblemSize; i++) {
    t = b[i]*b[i];
    a[i] = sin(t) + cos(t);
  }
```

If a scalar is indeed shared, OpenMP has various mechanisms for dealing with that. For instance, shared variables commonly occur in reduction operations:

```
  s = 0;
#pragma parallel for reduction(+:sum)
  for (i=0; i<ProblemSize; i++) {
    s = s + a[i]*b[i];
  }
```

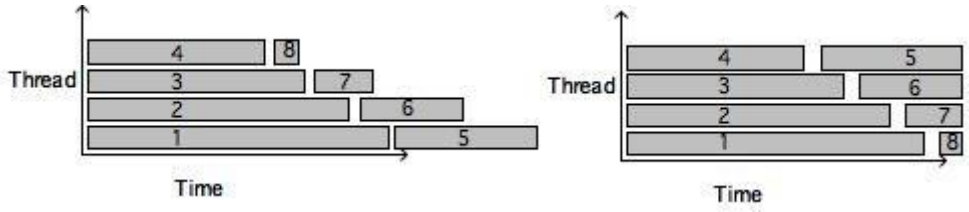


Figure 2.4: Static or round-robin (left) vs dynamic (right) thread scheduling; the task numbers are indicated.

As you see, a sequential code can be easily parallelized this way.

The assignment of iterations to threads is done by the runtime system, but the user can guide this assignment. We are mostly concerned with the case where there are more iterations than threads: if there are P threads and N iterations and $N > P$, how is iteration i going to be assigned to a thread?

The simplest assignment uses *round-robin scheduling*, a *static scheduling* strategy where thread p get iterations p, $p + N$, $p + 2N$, ... This has the advantage that if some data is reused between iterations, it will stay in the data cache of the processor executing that thread. On the other hand, if the iterations differ in the amount of work involved, the process may suffer from *load unbalance* with static scheduling. In that case, a *dynamic scheduling strategy* would work better, where each thread starts work on the next unprocessed iteration as soon as it finishes its current iteration. The example in figure 2.4 shows static versus dynamic scheduling of a number of tasks that gradually decrease in individual running time. In static scheduling, the first thread gets tasks 1 and 4, the second 2 and 5, et cetera. In dynamic scheduling, any thread that finishes its task gets the next task. This clearly gives a better running time in this particular example.

2.5.2 The global versus the local view in distributed programming

There can be a marked difference between how a parallel algorithm looks to an observer, and how it is actually programmed. Consider the case where we have an array of processors $\{P_i\}_{i=0..p-1}$, each containing one element of the arrays x and y, and P_i computes

$$\{y_i \leftarrow y_i + x_{i-1} \quad i > 0 \qquad y_i \text{unchanged} \quad i = 0$$

The global description of this could be

- Every processor P_i except the last sends its x element to P_{i+1};
- every processor P_i except the first receive an x element from their neighbour P_{i-1}, and
- they add it to their y element.

However, in general we can not code in these global terms. In the SPMD model (section 2.2.2) each processor executes the same code, and the overall algorithm is the result of these individual behaviours. The local program has access only to local data – everything else needs to be communicated with send and receive operations – and the processor knows its own number.

A naive attempt at the processor code would look like:

- If I am processor 0, do nothing; otherwise
- receive an x element from my left neighbour, add it to my y element, and
- send my x element to my right neighbour, unless I am the last processor.

This code is correct, but it is also inefficient: processor $i + 1$ does not start working until processor i is done. In other words, the parallel algorithm is in effect sequential, and offers no speedup.
We can easily make our algorithm fully parallel, by changing the processor code to:

- If am not the last processor, send my x element to the right.
- If am not the first processor, receive an x element from the left, and
- add it to my y element.

Now all sends, receives, and additions can happen in parallel.

There can still be a problem with this solution if the sends and receives are so-called blocking communication instructions: a send instruction does not finish until the sent item is actually received, and a receive instruction waits for the corresponding send. In this scenario:

All processors but the last start their send instruction, and consequently block, waiting for their neighbour to execute the corresponding receive.

The last processor is only one not sending, so it executes its receive instruction.

The processor one-before-last now completes its send, and progresses to its receive; Which allows its left neighbour to complete its send, et cetera.

You see that again the parallel execution becomes serialized.

Exercise 2.1. Suggest a way to make the algorithm parallel, even with blocking operations. (Hint: you won't be able to get all processors active at the same time.)

If the algorithm in equation 2.1 had been cyclic:

$$\{y_i \leftarrow y_i + x_{x-1} i=1...n-1 y_0 \leftarrow y_0 + x_{n-1}\ i=0 \{y_i \leftarrow y_i + xx-1 i=1...n-1 y0 \leftarrow y0 + xn-1\ i=0$$

the problem would be even worse. Now the last processor can not start its receive since it is blocked sending xn 1 to processor 0. This situation, where the program can not progress because every processor is waiting for another, is called *deadlock*.

The reason for blocking instructions is to prevent accumulation of data in the network. If a send instruction were to complete before the corresponding receive started, the network would have to store the data somewhere in the mean time. Consider a simple example:

```
buffer = ... ;   // generate some data
send(buffer,0); // send to processor 0
buffer = ... ;   // generate more data
send(buffer,1); // send to processor 1
```

After the first send, we start overwriting the buffer. If the data in it hasn't been received, the first set of values would have to be buffered somewhere in the network, which is not realistic. By having the send operation block, the data stays in the sender's buffer until it is guaranteed to have been copied to the recipient's buffer.

One way out of the problem of sequentialization or deadlock that arises from blocking instruction is the use of *non-blocking communication* instructions, which include explicit buffers for the data. With non-blocking send instruction, the user needs to allocate a buffer for each send, and check when it is safe to overwrite the buffer.

```
buffer0 = ... ; // data for processor 0
send(buffer0,0); // send to processor 0
buffer1 = ... ; // data for processor 1
send(buffer1,1); // send to processor 1
...
// wait for completion of all send operations.
```

2.5.3 MPI

If OpenMP is the way to program shared memory, MPI [77] is the standard solution for programming distributed memory. MPI ('Message Passing Interface') is a specification for a library interface for moving between processes that do not otherwise share data. The MPI routines can be divided roughly in the following categories:

- Process management. This includes querying the parallel environment and constructing subsets of processors.
- Point-to-point communication. This is a set of calls where two processes interact. These are mostly variants of the send and receive calls.
- Collective calls. In these routines, all processors (or the whole of a specified subset) are involved. Exam-ples are the broadcast call, where one processor shares

its data with every other processor, or the gather call, where one processor collects data from all participating processors.

Let us consider how the OpenMP examples can be coded in MPI. First of all, we no longer allocate

```
double a[ProblemSize];
```
but

```
double a[LocalProblemSize];
```
where the local size is roughly a $1/P$ fraction of the global size. (Practical considerations dictate whether you want this distribution to be as evenly as possible, or rather biased in some way.)

The parallel loop is trivially parallel, with the only difference that it now operates on a fraction of the arrays:

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i];
}
```
However, if the loop involves a calculation based on the iteration number, we need to map that to the global value:

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i]+f(i+MyFirstVariable);
}
```
(We will assume that each process has somehow calculated the values of LocalProblemSize and MyFirstVariable. Local variables are now automatically local, because each process has its own instance:

```
for (i=0; i<LocalProblemSize; i++) {
  t = b[i]*b[i];
  a[i] = sin(t) + cos(t);
}
```
However, shared variables are harder to implement. Since each process has its own data, the local accumulation has to be explicitly assembled:

```
for (i=0; i<LocalProblemSize; i++) {
  s = s + a[i]*b[i];
}
MPI_Allreduce(s,globals,1,MPI_DOUBLE,MPI_SUM);
```
The 'reduce' operation sums together all local values s into a variable globals that receives an identical value on each processor. This is known as a collective operation.

Let us make the example slightly more complicated:

```
for (i=0; i<ProblemSize; i++) {
  if (i==0)
    a[i] = (b[i]+b[i+1])/2
  else if (i==ProblemSize-1)
    a[i] = (b[i]+b[i-1])/2
```

```
  else
    a[i] = (b[i]+b[i-1]+b[i+1])/3
```
The basic form of the parallel loop is:

```
for (i=0; i<LocalProblemSize; i++) {
  bleft = b[i-1]; bright = b[i+1];
  a[i] = (b[i]+bleft+bright)/3
```
First we account for the fact that bleft and bright need to be obtained from a different processor for i==0 (bleft), and for i==LocalProblemSize-1 (bright). We do this with a exchange operation with our left and right neighbour processor:

```
get bfromleft and bfromright from neighbour processors, then for (i=0;
i<LocalProblemSize; i++) {
  if (i==0) bleft=bfromleft;
     else bleft = b[i-1]
   if (i==LocalProblemSize-1) bright=bfromright;
     else bright = b[i+1];
a[i] = (b[i]+bleft+bright)/3
```
Obtaining the neighbour values is done as follows. First we need to ask our processor number, so that we can start a communication with the processor with a number one higher and lower.

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID); MPI_Sendrecv
(/* to be sent: */ &b[LocalProblemSize-1],
/* result:       */ &bfromleft,
/* destination */ myTaskID+1, /* some parameters omited */ );
MPI_Sendrecv(&b[0],&bfromright,myTaskID-1 /* ... */ );
```
There are still two problems with this code. First, the sendrecv operations need exceptions for the first and last processors. This can be done elegantly as follows:

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID); MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
if (myTaskID==0) leftproc = MPI_PROC_NULL;
  else leftproc = myTaskID-1;
if (myTaskID==nTasks-1) rightproc = MPI_PROC_NULL;
  else rightproc = myTaskID+1;
MPI_Sendrecv( &b[LocalProblemSize-1], &bfromleft, rightproc );
MPI_Sendrecv( &b[0], &bfromright, leftproc);
```
Exercise 2.2. There is still a problem left with this code: the boundary conditions from the original, global, version have not been taken into account. Give code that solves that problem.

MPI gets complicated if different processes need to take different actions, for example, if one needs to send data to another. The problem here is that each process executes the same executable, so it needs to contain both the send and the receive instruction, to be executed depending on what the rank of the process is.

```
if (myTaskID==0) {
  MPI_Send(myInfo,1,MPI_INT,/* to: */ 1,/* labeled: */,0, MPI_COMM_WORLD);
} else {
  MPI_Recv(myInfo,1,MPI_INT,/* from: */ 0,/* labeled: */,0,
    /* not explained here: */&status,MPI_COMM_WORLD);
}
```

Although MPI is sometimes called the 'assembly language of parallel programming', for its perceived difficulty and level of explicitness, it not all that hard to learn, as evinced by the large number of scientific codes that use it. The main issues that make MPI somewhat intricate to use, are buffer management and blocking semantics.

These issues are related, and stem from the fact that, ideally, data should not be in two places at the same time. Let us briefly consider what happens if processor 1 sends data to processor 2. The safest strategy is for processor 1 to execute the send instruction, and then wait until processor 2 acknowledges that the data was successfully received. This means that processor 1 is temporarily blocked until processor 2 actually executes its receive instruction, and the data has made its way through the network. Alternatively, processor 1 could put its data in a buffer, tell the system to make sure that it gets sent at some point, and later checks to see that the buffer is safe to reuse. This second strategy is called non-blocking communication, and it requires the use of a temporary buffer.

2.5.3.1 Collective operations

In the above examples, you saw the MPI_Allreduce call, which computed a global sum and left the result on each processor. There is also a local version MPI_Reduce which computes the result only on one processor. These calls are examples of collective operations or collectives. The collectives are:

Reduction: each processor has a data item, and these items need to be combined arithmetically with an addition, multiplication, max, or min operation. The result can be left on one processor, or on all, in which case we call this an **allreduce** operation.

Broadcast: one processor has a data item that all processors need to receive.

Gather: each processor has a data item, and these items need to be collected in an array, without combining them in an operations such as an addition. The result can be left on one processor, or on all, in which case we call this an allgather.

Scatter: one processor has an array of data items, and each processor receives one element of that array.

All-to-all: each processor has an array of items, to be scattered to all other processors.

We will analyze the cost of collective operations in detail in section 6.3.1.

2.5.3.2 MPI version 1 and 2

The first MPI standard [68] had a number of notable omissions, which are included in the MPI 2 standard [48]. One of these concerned parallel input/output: there was no facility for multiple processes to access the same file, even if the underlying hardware would allow that. A separate project MPI-I/O has now been rolled into the MPI-2 standard.

A second facility missing in MPI, though it was present in PVM [28, 38] which predates MPI, is process manage-ment: there is no way to create new processes and have them be part of the parallel run.

Finally, MPI-2 has supported for one-sided communication: one process can do a send, without the receiving process actually doing a receive instruction.

2.5.3.3 Non-blocking communication

In a simple computer program, each instruction takes some time to execute, in a way that depends on what goes on in the processor. In parallel programs the situation is more complicated. A send operation, in its simplest form, declares that a certain buffer of data needs to be sent, and program execution will then stop until that buffer has been safely sent and received by another processor. This sort of operation is called a non-local operation since it depends on the actions of other processes, and a *blocking communication* operation since execution will halt until a certain event takes place.

Blocking operations have the disadvantage that they can lead to *deadlock* , if two processes wind up waiting for each other. Even without deadlock, they can lead to considerable *idle time* in the processors, as they wait without performing any useful work. On the other hand, they have the advantage that it is clear when the buffer can be reused: after the operation completes, there is a guarantee that the data has been safely received at the other end.

The blocking behaviour can be avoided, at the cost of complicating the buffer semantics, by using non-blocking operations. A non-blocking send (MPI_Isend) declares that a data buffer needs to be sent, but then does not wait for the completion of the corresponding receive. There is a second operation MPI_Wait that will actually block until the receive has been completed. The advantage of this decoupling of sending and blocking is that it now becomes possible to write:

```
ISend(somebuffer,&handle); // start sending, and
        get a handle to this particular communication { ... } // do useful work on
local data
Wait(handle); // block until the communication is completed; { ... } // do useful
work on incoming data
```

With a little luck, the local operations take more time than the communication, and you have completely eliminated the communication time.

In addition to non-blocking sends, there are non-blocking receives. A typical piece of code then looks like

```
ISend(sendbuffer,&sendhandle);
IReceive(recvbuffer,&recvhandle);
{ ... } // do useful work on local data Wait(sendhandle); Wait(recvhandle);
{ ... }  // do useful work on incoming data
```

Exercise 2.3. Go back to exercise 1 and give pseudocode that solves the problem using non-blocking sends and receives. What is the disadvantage of this code over a blocking solution?

2.5.4 Parallel languages

One approach to mitigating the difficulty of parallel programming is the design of languages that offer explicit support for parallelism. There are several approaches, and we will see some examples.

- Some languages reflect the fact that many operations in scientific computing are data parallel (sec-tion 2.4.1). Languages such as High Performance Fortran (HPF) (section 2.5.4.4) have an array syntax, where operations such addition of arrays can be expressed A = B+C. This syntax simplifies program-ming, but more importantly, it specifies operations at an abstract level, so that a lower level can make specific decision about how to handle parallelism. However, the data parallelism expressed in HPF is only of the simplest sort, where the data are contained in regular arrays. Irregular data parallelism is harder; the *Chapel* language (section 2.5.4.6) makes an attempt at addressing this.
- Another concept in parallel languages, not necessarily orthogonal to the previous, is that of Partitioned Global Address Space (PGAS) model: there is only one address space (unlike in the MPI model), but this address space is partitioned, and each partition has affinity with a thread or process. Thus, this model encompasses both SMP and distributed shared memory. The typical PGAS languages, Unified Parallel C (UPC), allows you to write programs that for the most part looks like regular C code. However, by indicating how the major arrays are distributed over processors, the program can be executed in parallel.

2.5.4.1 Discussion

Parallel languages hold the promise of making parallel programming easier, since they make communication oper-ations appear as simple copies or arithmetic operations. However, by doing so they invite the user to write code that may not be efficient, for instance by inducing many small messages.

As an example, consider arrays a,b that have been horizontally partitioned over the processors, and that are shifted:

```
for (i=0; i<N; i++)
  for (j=0; j<N/np; j++)
    a[i][j+joffset] = b[i][j+1+joffset]
```

If this code is executed on a shared memory machine, it will be efficient, but a naive translation in the distributed case will have a single number being communicated in each iteration of the i loop. Clearly, these can be combined in a single buffer send/receive operation, but compilers are usually unable to make this transformation. As a result, the user is forced to, in effect, re-implement the blocking that needs to be done in an MPI implementation:

```
for (i=0; i<N; i++)
  t[i] = b[i][N/np+joffset]
for (i=0; i<N; i++)
  for (j=0; j<N/np-1; j++) {
    a[i][j] = b[i][j+1]
    a[i][N/np] = t[i]
```

On the other hand, certain machines support direct memory copies through global memory hardware. In that case, PGAS languages can be more efficient than explicit message passing, even with physically distributed memory.

2.5.4.2 Unified Parallel C

Unified Parallel C (UPC) [11] is an extension to the C language. Its main source of parallelism is data parallelism, where the compiler discovers indepdence of operations on arrays, and assigns them to separate processors. The language has an extended array declaration, which allows the user to specify whether the array is partitioned by blocks, or in a *round-robin fashion*.

The following program in UPC performs a vector-vector addition.

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main() {
  int i;
  for(i=MYTHREAD; i<N; i+=THREADS)
    v1plusv2[i]=v1[i]+v2[i];
}
```

The same program with an explicitly parallel loop construct:

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main()
{
  int i;
  upc_forall(i=0; i<N; i++; i)
    v1plusv2[i]=v1[i]+v2[i];
}
```

2.5.4.3 Titanium

Titanium is comparable to UPC in spirit, but based on Java rather than on C.

2.5.4.4 High Performance Fortran

High Performance Fortran3 (HPF) is an extension of Fortran 90 with constructs that support parallel computing, published by the High Performance Fortran Forum (HPFF). The HPFF was convened and chaired by Ken Kennedy of Rice University. The first version of the HPF Report was published in 1993.

Building on the array syntax introduced in Fortran 90, HPF uses a data parallel model of computation to support spreading the work of a single array computation over multiple processors. This allows efficient implementation on both SIMD and MIMD style architectures. HPF features included:

- New Fortran statements, such as FORALL, and the ability to create PURE (side effect free) procedures
- Compiler directives for recommended distributions of array data
- Extrinsic procedure interface for interfacing to non-HPF parallel procedures such as those using message passing
- Additional library routines - including environmental inquiry, parallel prefix/suffix (e.g., 'scan'), data scattering, and sorting operations

Fortran 95 incorporated several HPF capabilities. While some vendors did incorporate HPF into their compilers in the 1990s, some aspects proved difficult to implement and of questionable use. Since then, most vendors and users have moved to OpenMP-based parallel processing. However, HPF continues to have influence. For example the proposed BIT data type for the upcoming Fortran-2008 standard contains a number of new intrinsic functions taken directly from HPF.

2.5.4.5 Co-array Fortran

Co-array Fortran (CAF) is an extension to the Fortran 95/2003 language. The main mechanism to support paral-lelism is an extension to the array declaration syntax, where an extra dimension indicates the parallel distribution. For instance,

```
real,allocatable,dimension(:,:,:)[:,:] :: A
```

declares an array that is three-dimensional on each processor, and that is distributed over a two-dimensional pro-cessor grid.

Communication between processors is now done through copies along the dimensions that describe the processor grid:

```
    COMMON/XCTILB4/ B(N,4)[*]
    SAVE/XCTILB4/
C
    CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
    B(:,3) = B(:,1)[IMG_S]
    B(:,4) = B(:,2)[IMG_N]
    CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
```

The Fortran 2008 standard will include co-arrays.

2.5.4.6 Chapel

Chapel [1] is a new parallel programming language4 being developed by Cray Inc. as part of the DARPA-led High Productivity Computing Systems program (HPCS). Chapel is designed to improve the productivity of high-end computer users while also serving as a portable parallel programming model that can be used on commodity clusters or

desktop multicore systems. Chapel strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI.

Chapel supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel's locale type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality. Chapel supports global-view data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. In contrast to many previous higher-level parallel languages, Chapel is designed around a multiresolution philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require. Chapel supports code reuse and rapid prototyping via object-oriented design, type inference, and features for generic programming.

Chapel was designed from first principles rather than by extending an existing language. It is an imperative block-structured language, designed to be easy to learn for users of C, C++, Fortran, Java, Perl, Matlab, and other popular languages. While Chapel builds on concepts and syntax from many previous languages, its parallel features are most directly influenced by ZPL, High-Performance Fortran (HPF), and the Cray MTA's extensions to C and Fortran.

Here is vector-vector addition in Chapel:

```
const BlockDist= newBlock1D(bbox=[1..m], tasksPerLocale=...);
const ProblemSpace: domain(1, 64)) distributed BlockDist = [1..m];
varA, B, C: [ProblemSpace] real;
forall(a, b, c) in(A, B, C) do
  a = b + alpha * c;
```

2.5.4.7 Fortress

Fortress [7] is a programming language developed by Sun Microsystems. Fortress5 aims to make parallelism more tractable in several ways. First, parallelism is the default. This is intended to push tool design, library design, and programmer skills in the direction of parallelism. Second, the language is designed to be more friendly to paral-lelism. Side-effects are discouraged because side-effects require synchronization to avoid bugs. Fortress provides transactions, so that programmers are not faced with the task of determining lock orders, or tuning their locking code so that there is enough for correctness, but not so much that performance is impeded. The Fortress looping constructions, together with the library, turns "iteration" inside out; instead of the loop specifying how the data is accessed, the data structures specify how the loop is run, and aggregate data structures are designed to break into large parts that can be effectively scheduled for parallel execution. Fortress also includes features from other languages intended to generally help productivity – test code and methods, tied to the code under test; contracts that can optionally be checked when the code is run; and properties, that might be too expensive to run, but can be fed to a theorem prover or model checker. In

addition, Fortress includes safe-language features like checked array bounds, type checking, and garbage collection that have been proven-useful in Java. Fortress syntax is designed to resemble mathematical syntax as much as possible, so that anyone solving a problem with math in its specification can write a program that can be more easily related to its original specification.

2.5.4.8 X10

X10 is an experimental new language currently under development at IBM in collaboration with academic part-ners. The X10 effort is part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems) in the DARPA program on High Productivity Computer Systems. The PERCS project is focused on a hardware-software co-design methodology to integrate advances in chip technology, architecture, operating systems, compilers, pro-gramming language and programming tools to deliver new adaptable, scalable systems that will provide an order-of-magnitude improvement in development productivity for parallel applications by 2010.

X10 aims to contribute to this productivity improvement by developing a new programming model, combined with a new set of tools integrated into Eclipse and new implementation techniques for delivering optimized scalable parallelism in a managed runtime environment. X10 is a type-safe, modern, parallel, distributed object-oriented language intended to be very easily accessible to Java(TM) programmers. It is targeted to future low-end and high-end systems with nodes that are built out of multi-core SMP chips with non-uniform memory hierarchies, and interconnected in scalable cluster configurations. A member of the Partitioned Global Address Space (PGAS) family of languages, X10 highlights the explicit reification of locality in the form of places; lightweight activities embodied in async, future, foreach, and ateach constructs; constructs for termination detection (finish) and phased computation (clocks); the use of lock-free synchronization (atomic blocks); and the manipulation of global arrays and data structures.

2.5.4.9 Linda

As should be clear by now, the treatment of data is by far the most important aspect of parallel programing, far more important than algorithmic considerations. The programming system Linda [39], also called a coordination language, is designed to address the data handling explicitly. Linda is not a language as such, but can, and has been, incorporated into other languages.

The basic concept of Linda is tuple space: data is added to a pool of globally accessible information by adding a label to it. Processes then retrieve data by their label, and without needing to know which processes added them to the tuple space.

Linda is aimed primarily at a different computation model than is relevant for High-Performance Computing (HPC): it addresses the needs of asynchronous communicating processes. However, is has been used for scientific compu-tation [25]. For instance, in

parallel simulations of the heat equation (section 4.3), processors can write their data into tuple space, and neighbouring processes can retrieve their ghost region without having to know its provenance. Thus, Linda becomes one way of implementing .

2.5.5 OS-based approaches

It is possible to design an architecture with a shared address space, and let the data movement be handled by the operating system. The Kendall Square computer [5] had an architecture name 'all-cache', where no data was natively associated with any processor. Instead, all data was considered to be cached on a processor, and moved through the network on demand, much like data is moved from main memory to cache in a regular CPU. This idea is analogous to the numa support in current SGI architectures.

2.5.6 Latency hiding

Communication between processors is typically slow, slower than data transfer from memory on a single processor, and much slower than operating on data. For this reason, it is good to think about the relative volumes of network traffic versus 'useful' operations (see also section 2.9.2) when designing a parallel program. Another way of coping with the relative slowness of communication is to arrange the program so that the communication actually happens while some computation is going on.

For example, consider the parallel execution of a matrix-vector product $y = Ax$ (there will be further discussion of this operation in section 6.2). Assume that the vectors are distributed, so each processor p executes

$$\forall_{i\in I_p}: y_i = \sum_j a_{ij} x_i \quad \forall i \in Ip: yi = \sum_j aijxi$$

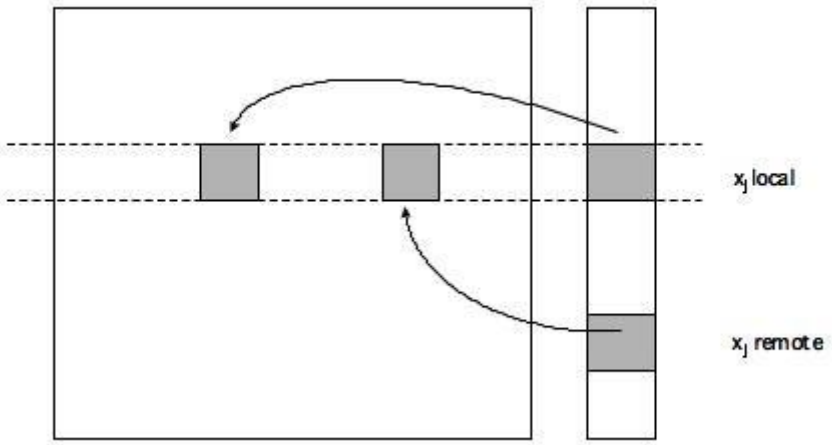


Figure 2.5: The parallel matrix-vector product with a blockrow distribution.

Since x is also distributed, we can write this as

$$\forall_{i\in I_p}: y_i = \left(\sum_{j\text{local}} + \sum_{j\text{notlocal}}\right) a_{ij} x_j \quad \forall i \in Ip: yi = (\sum jlocal + \sum jnotlocal) aijxj$$

This scheme is illustrated in figure 2.5. We can now proceed as follows:

- Start the transfer of non-local elements of *x*;
- Operate on the local elements of *x* while data transfer is going on;
- Make sure that the transfers are finished;
- Operate on the non-local elements of *x*.

Of course, this scenario presupposes that there is software and hardware support for this overlap. MPI allows for this, through so-called asynchronous communication or non-blocking comumnication routines. This does not immediately imply that overlap will actually happen, since hardware support is an entirely separate question.

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

Unit 9: Look Back and Look Ahead

This unit looks back at important concepts of computer architecture that were covered in this course and looks ahead at some additional topics of interest. Computer architecture is both a depth and breadth subject. It is an in depth subject that is of particular interest if you are interested in computer architecture for a professional researcher, designer, developer, tester, manager, manufacturer, etc. and you want to continue with additional study in advanced computer architecture. On the other hand, computer architecture is a rich source of ideas and understanding for other areas of computer science, giving you a broad and stronger foundation for the study of programming, computer languages, compilers, software architecture, domain specific computing (like scientific computing), and more.

In this unit, you will look back at some of the theoretical laws and analysis techniques that were introduced during the course. Looking ahead, you will be introduced to special purpose processors, application specific processing, high volume data storage, and network computing.

- Upon successful completion of this unit, you will be able to:
  - discuss basic laws applicable to computer performance;
  - identify examples of computer architecture for special purpose computing architectures; and
  - identify examples of special purpose applications of parallel computing.

- 9.1: Theory and Laws

Topologies

Read section 2.6 to learn about network topologies. If a task cannot be performed by a computer with one processor, we decompose the task into subtasks, which will be allocated to multiple hardware devices, say processors or arithmetic units or memory. These multiple hardware devices need to communicate so that the original task can be done with acceptable cost and performance. The hardware devices and their interconnections form a network.

Consider another situation: suppose we have a large software system made up of a large number of subsystems, in turn composed of many software components, subcomponents, etc. Suppose we list all the lowest level subcomponent's names across the top of a sheet of paper. These will be our column headings. Also, let's list down the side of the same sheet the same subcomponent names. These will be our row headings. This forms a table or two by two matrix. Finally, suppose we put a 1 in the table whenever or wherever there is a connection between the subcomponent named in the column heading and the subcomponent named in the row heading. Let's put a 0 everywhere else. This table now represents a topology for our network of software components; it could also be done for hardware components. These components and their interconnections are part of the software architecture. Realize that the matrix could be huge: 100 by 100, 1000 by 1000, and so on. The complexity of the interconnections is a factor in the reliability and performance of the architecture.

Read section 2.7, which reviews Amdahl's law, an extension of Amdahl's law that includes communication overhead, and Gustafson's law. These laws express expected performance as the number of processors increases, or as both the size of the problem and number of processors increases.

Then, read section 2.9 to learn about load balancing. This section looks at the situation where a processor is idle and another is busy, which is referred to as a load imbalance. If the work were to be distributed differently among the processors, then the idle time might be able to be eliminated. In this case, the load balance problem is explained as a graph problem.

2.6 Topologies

If a number of processors are working together on a single task, most likely they need to communicate data. For this reason there needs to be a way for data to make it from any processor to any other. In this section we will discuss some of the possible schemes to connect the processors in a parallel machine.

In order to get an appreciation for the fact that there is a genuine problem here, consider two simple schemes that do not ‘scale up’:

- Ethernet is a connection scheme where all machines on a network are on a single cable. If one machine puts a signal on the wire to send a message, and another also wants to send a message, the latter will detect that the sole available communication channel is occupied, and it will wait some time before retrying its

send operation. Receiving data on ethernet is simple: messages contain the address of the intended recipient, so a processor only has to check whether the signal on the wire is intended for it. The problems with this scheme should be clear. The capacity of the communication channel is finite, so as more processors are connected to it, the capacity available to each will go down. Because of the scheme for resolving conflicts, the average delay before a message can be started will also increase.

- In a *fully connected* configuration, each processor has one wire for the communications with each other processor. This scheme is perfect in the sense that messages can be sent in the minimum amount of time, and two messages will never interfere with each other. The amount of data that can be sent from one processor is no longer a decreasing function of the number of processors; it is in fact an increasing function, and if the network controller can handle it, a processor can even engage in multiple simultaneous communications. The problem with this scheme is of course that the design of the network interface of a processor is no longer fixed: as more processors are added to the parallel machine, the network interface gets more con-necting wires. The network controller similarly becomes more complicated, and the cost of the machine increases faster than linearly in the number of processors.

In this section we will see a number of schemes that can be increased to large numbers of processors.

2.6.1 Some graph theory

The network that connects the processors in a parallel computer can conveniently be described with some elementary graph theory concepts. We describe the parallel machine with a graph where each processor is a node, and two nodes are connected if there is a direct connection between them.

We can then analyze two important concepts of this graph.

First of all, the degree of a node in a graph is the number of other nodes it is connected to. With the nodes representing processors, and the edges the wires, it is clear that a high degree is not just desirable for efficiency of computing, but also costly from an engineering point of view. We assume that all processors have the same degree.

Secondly, a message traveling from one processor to another, through one or more intermediate nodes, will most incur some delay at each intermediate node. For this reason, the *diameter* of the graph is important. The diameter is defined as the maximum shortest distance, counting numbers of wires, between any two processors:

$d(G) = \max_{ij} |$ shortest path between i and j $|$:

If d is the diameter, and if sending a message over one wire takes unit time (more about this in the next section), this means a message will always arrive in at most time d.

Exercise 2.4. Find a relation between the number of processors, their degree, and the diameter of the connectivity graph.

In addition to the question 'how long will a message from processor A to processor B take', we often worry about conflicts between two simultaneous messages: is there a possibility that two messages, under way at the same time, will need to use the same network link? This sort of conflict is called *congestion* or *contention*. Clearly, the more links the graph of a parallel comupter has, the smaller the chance of congestion.

A precise way to describe the likelihood of congestion, is to look at the bisection width . This is defined as the minimum number of links that have to be removed to partition the processor graph into two unconnected graphs. For instance, consider processors connected as a linear array, that is, processor P_i is connected to P_{i-1} and P_{i+1}. In this case the bisection width is 1.

The bisection width w describes how many messages can, guaranteed, be under way simultaneously in a parallel computer. Proof: take w sending and w receiving processors. The w paths thus defined are disjoint: if they were not, we could separate the processors into two groups by removing only w - 1 links.

In practice, of course, more than w messages can be under way simultaneously. For instance, in a linear array, which has w = 1, P/2 messages can be sent and received simultaneously if all communication is between neighbours, and if a processor can only send or receive, but not both, at any one time. If processors can both send and receive simultaneously, P messages can be under way in the network.

Bisection width also describes redundancy in a network: if one or more connections are malfunctioning, can a message still find its way from sender to receiver?

Exercise 2.5. What is the diameter of a 3D cube of processors? What is the bisection width? How does that change if you add wraparound torus connections?

While bisection width is a measure express as a number of wires, in practice we care about the capacity through those wires. The relevant concept here is bisection bandwidth : the bandwidth across the bisection width, which is the product of the bisection width, and the capacity (in bits per second) of the wires. Bisection bandwidth can be considered as a measure for the bandwidth that can be attained if an arbitrary half of the processors communicates with the other half. Bisection bandwidth is a more realistic measure than the aggregate bandwidth which is some-times quoted: it is defined as the total data rate if every processor is sending: the number of processors times the bandwidth of a connection times the number of simultaneous sends a processor can perform. This can be quite a high number, and it is typically not representative of the communication rate that is achieved in actual applications.

2.6.2 Linear arrays and rings

A simple way to hook up multiple processors is to connect them in a linear array: every processor has a number i,

and processor P_i is connected to P_{i-1} and P_{i+1}. The first and last processor are possible exceptions: if they are connected to each other, we call the architecture a ring network .

This solution requires each processor to have two network connections, so the design is fairly simple.

Exercise 2.6. What is the bisection width of a linear array? Of a ring?

Exercise 2.7. With the limited connections of a linear array, you may have to be clever about how to program parallel algorithms. For instance, consider a 'broadcast' operation: processor 0 has a data item that needs to be sent to every other processor.

We make the following simplifying assumptions:

- a processor can send any number of messages simultaneously,
- but a wire can can carry only one message at a time; however,
- communication between any two processors takes unit time, regardless the number of processors in between them.

In a fully connected network you can simply write

for i = 1 . . . N - 1:

send the message to processor i

in a fully connected network this means that the operation is done in one step.

Now consider a linear array. Show that, even with this unlimited capacity for sending, the above algorithm runs into trouble because of congestion.

Find a better way to organize the send operations. Hint: pretend that your processors are connected as a binary tree. Assume that there are $N = 2^n$ processors. Show that the broadcast can be done in log N stages, and that processors only need to be able to send a single message simultaneously.

This exercise is an example of embedding a 'logical' communication pattern in a physical one.

2.6.3 2D and 3D arrays

A popular design for parallel computers is to organize the processors in a two-dimensional or three-dimensional cartesian mesh . This means that every processor has a coordinate (i, j) or (i, j, k), and it is connected to its neigh-bours in all coordinate

directions. The processor design is still fairly simple: the number of network connections (the degree of the connectivity graph) is twice the number of space dimensions (2 or 3) of the network.

It is a fairly natural idea to have 2D or 3D networks, since the world around us is three-dimensional, and computers are often used to model real-life phenomena. If we accept for now that the physical model requires nearest neighbour type communications (which we will see is the case in section 4.2.3), then a mesh computer is a natural candidate for running physics simulations.

Exercise 2.8. Analyze the diameter and bisection width of 2D and 3D meshes and toruses.

Exercise 2.9. Your parallel computer has its processors organized in a 2D grid. The chip manufacturer comes out with a new chip with same clock speed that is dual core instead of single core, and that will fit in the existing sockets. Critique the following argument: "the amount work per second that can be done (that does not involve communication) doubles; since the network stays the same, the bisection bandwidth also stays the same, so I can reasonably expect my new machine to become twice as fast."

2.6.4 Hypercubes

Above we gave a hand-waving argument for the suitability of mesh-organized processors, based on the prevalence of nearest neighbour communications. However, sometimes sends and receives between arbitrary processors occur. One example of this is the above-mentioned broadcast. For this reason, it is desirable to have a network with a smaller diameter than a mesh. On the other hand we want to avoid the complicated design of a fully connected network.

A good intermediate solution is the hypercube design. An n-dimensional hypercube computer has 2n processors, with each processor connected to one other in each dimension; see figure 2.6. The nodes of a hypercube are num-bered by bit patterns as in figure 2.7.

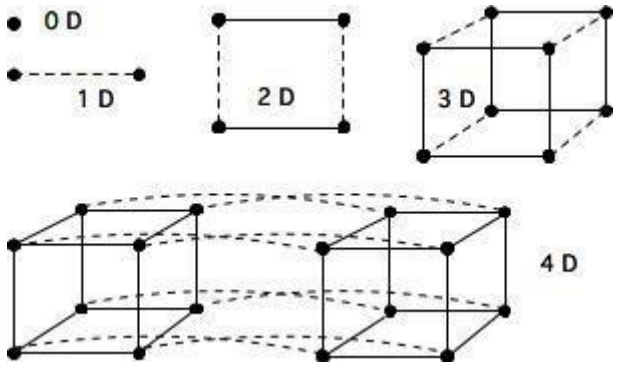


Figure 2.6: Hypercubes

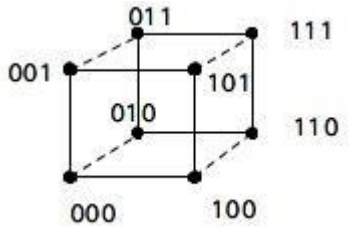


Figure 2.7: Numbering of the nodes of a hypercube

An easy way to describe this is to give each processor an address consisting of d bits. A processor is then connected to all others that have an address that differs by exactly one bit.

The big advantages of a hypercube design are the small diameter and large capacity for traffic through the network.

Exercise 2.10. Diameter? Bisection width?

One disadvantage is the fact that the processor design is dependent on the total machine size. In practice, processors will be designed with a maximum number of possible connections, and someone buying a smaller machine then will be paying for unused capacity. Another disadvantage is the fact that extending a given machine can only be done by doubling it: other sizes than 2^p are not possible.

Exercise 2.11. Consider the parallel summing example of section 2.1, and give the execution time of a parallel implementation, first on a linear array, then on a hypercube. Assume that sending one number from a processor to a neighbour takes time and that a floating point operation takes time . Show that on a linear array the algorithm gives at best a factor speedup over the sequential algorithm; show that the theoretical speedup from the example is attained (up to a factor) for the implementation on a hypercube.

2.6.4.1 Embedding grids in a hypercube

Above we made the argument that mesh-connected processors are a logical choice for many applications that model physical phenomena. How is that for hypercubes? The answer is that a hyercube has enough connections that it can simply pretend to be a mesh by ignoring certain connections. However, we can not use the obvious numbering of nodes as in figure 2.7. For instance, node 1 is directly connected to node 0, but has a distance of 2 to node 2. The left neighbour of node 0 in a ring, node 7, even has the maximum distance of 3. To explain how we can embed a mesh in a hypercube, we first show that it's possible to walk through a hypercube, touching every corner exactly once.

The basic concept here is a (binary reflected) Gray code [46]. This is a way of ordering the binary numbers $0 \ldots 2^d - 1$ as $g_0, \ldots g_{2^d-1}$ so that g_i and g_{i+1} differ in only one bit. Clearly, the ordinary binary numbers do not satisfy this: the binary representations for 1 and 2 already differ in two bits. Why do Gray codes help us? Well, since g_i and g_{i+1} differ only in

one bit, it means they are the numbers of nodes in the hypercube that are directly connected.

Figure 2.8 illustrates how to construct a Gray code. The procedure is recursive, and can be described informally as 'divide the cube into two subcubes, number the one subcube, cross over to the other subcube, and number its nodes in the reverse order of the first one'.

Since a Gray code offers us a way to embed a one-dimensional 'mesh' into a hypercube, we can now work our way up.

Exercise 2.12. Show how a square mesh of 2^{2d} nodes can be embedded in a hypercube by appending the bit patterns of the embeddings of two 2^d node cubes. How would you accomodate a mesh of $2^{d_1+d_2}$ nodes? A three-dimensional mesh of $2^{d_1+d_2+d_3}$ nodes?

2.6.5 Switched networks

Above, we briefly discussed fully connected processors. They are impractical if the connection is made by making a large number of wires between all the processors. There is another possibility, however, by connecting all the processors to a switch or switching network. Some popular network designs are the crossbar butterfly exchange and the fat tree [47].

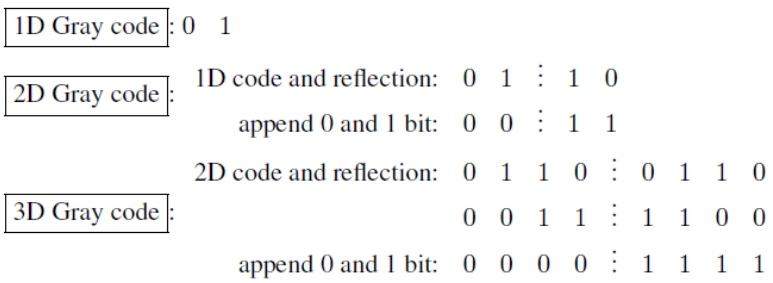


Figure 2.8: Gray codes

Switching networks are made out of switching elements, each of which have a small number (up to about a dozen) of inbound and outbound links. By hooking all processors up to some switching element, and having multiple stages of switching, it then becomes possible to connect any two processors by a path through the network.

2.6.5.1 Cross bar

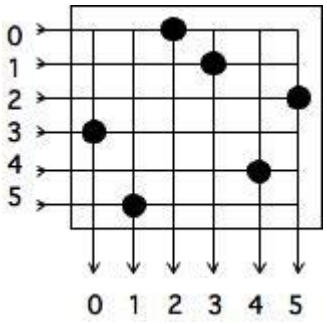


Figure 2.9: A simple cross bar connecting 6 inputs to 6 outputs

The simplest switching network is a cross bar, an arrangement of n horizontal and vertical lines, with a switch element on each intersection that determines whether the lines are connected; see figure 2.9. If we designate the horizontal lines as inputs the vertical as outputs, this is clearly a way of having n inputs be mapped to n outputs. Every combination of inputs and outputs (sometimes called a 'permutation') is allowed.

2.6.5.2 Butterfly exchange

Butterfly exchanges are typically built out of small switching elements, and they have multiple stages; as the number of processors grows, the number of stages grows with it. As you can see in figure 2.11, butterfly exchanges allow several processors to access memory simultaneously. Also, their access times are identical, see exchange networks are a way of implementing a UMA architecture; see section 2.3.1.

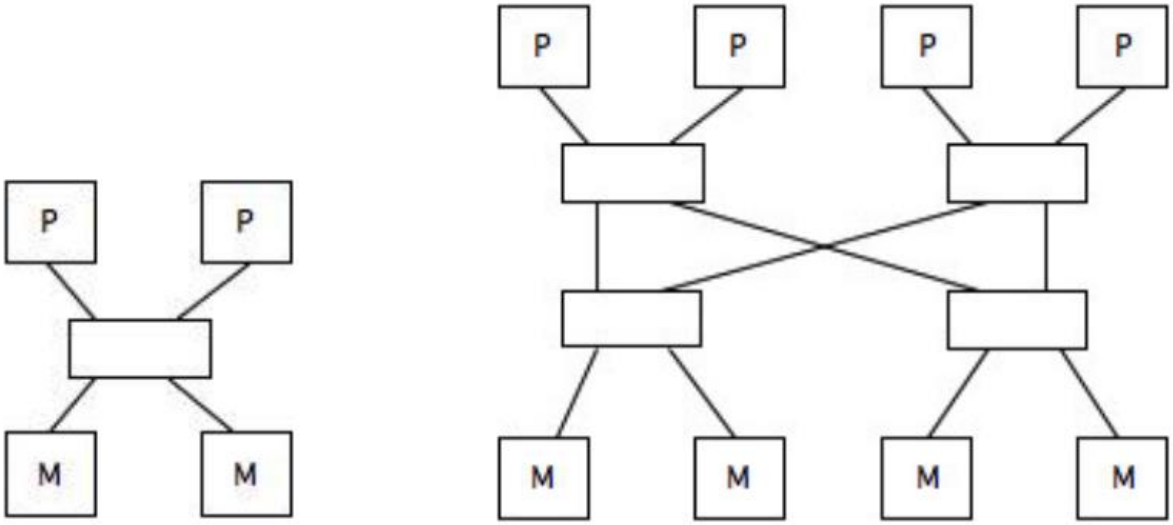


Figure 2.10: A butterfly exchange network for two and four processors/memories

Exercise 2.13. For both the simple cross bar and the butterfly exchange, the network needs to be expanded as the number of processors grows. Give the number of wires (of some unit length) and the number of switching elements that is needed in both cases to connect n processors and memories. What is the time that a data packet needs to go

from memory to processor, expressed in the unit time that it takes to traverse a unit length of wire and the time to traverse a switching element?

2.6.5.3 Fat-trees

If we were to connect switching nodes like a tree, there would be a big problem with congestion close to the root since there are only two wires attached to the root note. Say we have a k-level tree, so there are 2^k leaf nodes. If all leaf nodes in the left subtree try to communicate with nodes in the right subtree, we have 2^{k-1} messages going through just one wire into the root, and similarly out through one wire. A fat-tree is a tree network where each level has the same total bandwidth, so that this congestion problem does not occur: the root will actually have 2^{k-1} incoming and outgoing wires attached.

The first successful computer architecture based on a fat-tree was the Connection Machines CM5.

In fat-trees, as in other switching networks, each message carries its own routing information. Since in a fat-tree the choices are limited to going up a level, or switching to the other subtree at the current level, a message needs to carry only as many bits routing information as there are levels, which is log2 n for n processors.

The theoretical exposition of fat-trees in [64] shows that fat-trees are optimal in some sense: it can deliver messages as fast (up to logarithmic factors) as any other network that takes the same amount of space to build. The underlying assumption of this statement is that switches closer to the root have to connect more wires, therefore take more components, and correspondingly are larger.

This argument, while theoretically interesting, is of no practical significance, as the physical size of the network hardly plays a role in the biggest currently available computers that use fat-tree interconnect. For instance, in the Ranger supercomputer of The University of Texas at Austin, the fat-tree switch connects 60,000 processors, yet takes less than 10 percent of the floor space.

A fat tree, as sketched above, would be costly to build, since for every next level a new, bigger, switch would have to be designed. In practice, therefore, a network with the characteristics of a fat-tree is constructed from simple switching elements; see figure 2.12. This network is equivalent in its bandwidth and routing possibilities to a fat-tree. Routing algorithms will be slightly more complicated: in a fat-tree, a data packet can go

up in only one way, but here a packet has to know to which of the two higher switches to route.

This type of switching network is one case of a Clos network [23].

2.6.6 Bandwidth and latency

The statement above that sending a message can be considered a unit time operation, is of course unrealistic. A large message will take longer to transmit than a short one. There are two concepts to arrive at a more realistic description of the transmission process; we have already seen this in section 1.2.2 in the context of transferring data between cache levels of a processor.

Latency: Setting up a communication between two processors takes an amount of time that is independent of the message size. The time that this takes is known as the latency of a message. There are various causes for this delay.

- The two processors engage in 'hand-shaking', to make sure that the recipient is ready, and that appropriate buffer space is available for receiving the message.
- The message needs to be encoded for transmission by the sender, and decoded by the receiver.
- The actual transmission may take time: parallel computers are often big enough that, even at light-speed, a message can take hundreds of cycles to traverse the distance between two processors.

Bandwidth: After a transmission between two processors has been set up, the main number of interest is the number of bytes per second that can go through the channel. This is known as the bandwidth . The bandwidth can usually be determined by the channel rate, the rate at which a physical link can deliver bits, and the channel width , the number of physical wires in a link. The channel width is typically a multiple of 16, usually 64 or 128. This is also expressed by saying that a channel can send one or two 8-byte words simultaneously.

Bandwidth and latency are formalized in the expression

$$T(n)=\alpha+\beta n$$ T(n)=α+βn

for the transmission time of an n-byte message. Here, $\alpha\alpha$ is the latency and $\beta\beta$ is the time per byte, that is, the inverse of bandwidth.

2.7 Theory

There are two important reasons for using a parallel computer: to have access to more memory or to obtain higher performance. It is easy to characterize the gain in memory, as the total memory is the sum of the individual memo-ries. The speed of a parallel computer is harder to characterize. A simple approach is to let the same program run on a single processor, and on a parallel machine with p processors, and to compare runtimes.

With T_1 the execution time on a single processor and T_p the time on p processors, we define the speedup as $S_p = T_1/T_p$. (Sometimes T_1 is defined as 'the best time to solve the problem on a single processor', which allows for using a different algorithm on a single processor than in parallel.) In the ideal case, $T_p = T_1/p$, but in practice we don't expect to attain that, so $S_P \leq p$ SP≤p. To measure how far we are from the ideal speedup, we introduce the efficiency $E_p = S_p/p$ Ep=Sp/p. Clearly, $0, E_p \leq 1$ 0,Ep≤1.

There is a practical problem with this definition: a problem that can be solved on a parallel machine may be too large to fit on any single processor. Conversely, distributing a single processor problem over many processors may give a distorted picture since very little data will wind up on each processor.

There are various reasons why the actual speed is less than P . For one, using more than one processors necessitates communication, which is overhead. Secondly, if the processors do not have exactly the same amount of work to do, they may be idle part of the time, again lowering the actually attained speedup. Finally, code may have sections that are inherently sequential.

Communication between processors is an important source of a loss of efficiency. Clearly, a problem that can be solved without communication will be very efficient. Such problems, in effect consisting of a number of completely independent calculations, is called embarassingly parallel ; it will have close to a perfect speedup and efficiency.

Exercise 2.14. The case of speedup larger than the number of processors is called superlinear speedup.

Give a theoretical argument why this can never happen.

In practice, superlinear speedup can happen. For instance, suppose a problem is too large to fit in memory, and a single processor can only solve it by swapping data to disc. If the same problem fits in the memory of two processors, the speedup may well be larger than 2 since disc swapping no longer occurs. Having less, or more localized, data may also improve the cache behaviour of a code.

2.7.1 Amdahl's law

One reason for less than perfect speedup is that parts of a code can be inherently sequential. This limits the parallel efficiency as follows. Suppose that 5% of a code is sequential, then the time for that part can not be reduced, no matter how many processors are available. Thus, the speedup on that code is limited to a factor of 20. This phenomenon is known as Amdahl's Law [12], which we will now formulate.

Let F_p be the parallel fraction and F_s be the parallel fraction (or more strictly: the 'parallelizable' fraction) of a code, respectively. Then $F_p + F_s = 1$. The parallel execution time T_p is the sum of the part that is sequential $T_1 F_s$ and the part that can be parallelized $T_1 F_s/p$:

$T_P = T_1(F_s + F_p/P)$TP=T1(Fs+Fp/P)

As the number of processors grows $p \to \infty$p→∞, the parallel execution time now approaches that of the sequential fraction of the code: $T_P \downarrow T_1 F_s$TP↓T1Fs. We conclude that speedup is limited by $S_P \leq 1/F_s$SP≤1/Fs and efficiency is a decreasing function $E \sim 1/p$E∼1/p .

The sequential fraction of a code can consist of things such as I/O operations. However, there are also parts of a code that in effect act as sequential. Consider a program that executes a single loop, where all iterations can be computed independently. Clearly, this code is easily parallelized. However, by splitting the loop in a number of parts, one per processor, each processor now has to deal with loop overhead: calculation of bounds, and the test for completion. This overhead is replicated as many times as there are processors. In effect, loop overhead acts as a sequential part of the code.

In practice, many codes do not have significant sequential parts, and overhead is not important enough to affect parallelization adversely. One reason for this is discussed in the next section.

Exercise 2.15. Investigate the implications of Amdahls's law: if the number of processors P increases, how does the parallel fraction of a code have to increase to maintain a fixed efficiency?

2.7.2 Amdahl's law with communication overhead

In a way, Amdahl's law, sobering as it is, is even optimistic. Parallelizing a code will give a certain speedup, but it also introduces communication overhead that will lower the speedup attained. Let us refine our model of equation (2.2) (see [62, p. 367]):

$T_p = T_1(F_s + F_p/P) + T_c$Tp=T1(Fs+Fp/P)+Tc

where T_c is a fixed communication time.

To assess the influence of this communication overhead, we assume that the code is fully parallelizable, that is, $f_p = 1$. We then find that

$S_p = T_1 T_1/p + T_c$Sp=T1T1/p+Tc

For this to be close to p, we need $T_c \ll T_1/p$Tc≪T1/p or $p \ll T_1/T_c$p≪T1/Tc. In other words, the number of processors should not grow beyond the ratio of scalar execution time and communication overhead.

2.7.3 Scalability

Above, we remarked that splitting a given problem over more and more processors does not make sense: at a certain point there is just not enough work for each processor to operate efficiently. Instead, in practice, users of a parallel code will either choose the number of processors to match the problem size, or they will solve a series of

increasingly larger problems on correspondingly growing numbers of processors. In both cases it is hard to talk about speedup. Instead, the concept of scalability is used.

We distinguish two types of scalability. So-called strong scalability is in effect the same as speedup, discussed above. We say that a program shows strong scalability if, partitioned over more and more processors, it shows perfect or near perfect speedup. Typically, one encounters statements like 'this problem scales up to 500 processors', meaning that up to 500 processors the speedup will not noticeably decrease from optimal.

More interesting, weak scalability is a more vaguely defined term. It describes that, as problem size and number of processors grow in such a way that the amount of data per processor stays constant, the speed in operations per second of each processor also stays constant. This measure is somewhat hard to report, since the relation between the number of operations and the amount of data can be complicated. If this relation is linear, one could state that the amount of data per processor is kept constant, and report that parallel execution time is constant as the number of processors grows.

Scalability depends on the way an algorithm is parallelized, in particular on the way data is distributed. In section 6.3 you will find an analysis of the matrix-vector product operation: distributing a matrix by block rows turns out not to be scalable, but a two-dimensional distribution by submatrices is.

2.7.4 Gustafson's law

Amdahl's law describes speedup in the strong scaling sense discussed above. Gustafson's law is an attempt to formalize weak scaling. Let the computation be normalized again so that $F_p + F_s = 1$, and assume that we keep the amount of parallel work per processor constant. This means that the total amount of work performed by p processors is now Fs + pFp, and the corresponding speedup formula is

$$S(p)=Fs+pFpFs+Fp=Fs+pFp=p+(1−p)FsS(p)=FS+pFPFS+FP=FS+pFP=p+(1−p)FS$$

This is a linearly decreasing function of F_s, rather than the $1/F_s$ function as before.

2.9 Load balancing

In much of this chapter, we assumed that a problem could be perfectly divided over processors, that is, a processor would always be performing useful work, and only be idle because of latency in communication. In practice, however, a processor may be idle because it is waiting for a message, and the sending processor has not even reached the send instruction in its code. Such a situation, where one processor is working and another is idle, I described as load unbalance: there is no intrinsic reason for the one processor to be idle, and it could have been working if we had distributed the work load differently.

2.9.1 Load balancing of independent tasks

Let us consider the case of a job that can be partitioned into independent tasks, for instance computing the pixels of a Mandelbrot set picture, where each pixel is set according to a mathematical function that does not depend on surrounding pixels. If we

could predict the time it would take to draw an arbitrary part of the picture, we could make a perfect division of the work and assign it to the processors. This is known as static load balancing.

More realistically, we can not predict the running time of a part of the job perfectly, and we use an over-decomposition of the work: we divide the work in more tasks than there are processors. These tasks are then assigned to a work pool , and processors take the next job from the pool whenever they finish a job. This is known as dynamic load balancing. Many graph and combinatorial problems can be approached this way.

2.9.2 Load balancing as graph problem

A parallel computation can be formulated as a graph (see Appendix A.6 for an introduction to graph theory) where the processors are the vertices, and there is an edge between two vertices if their processors need to communicate at some point. Such a graph is often derived from an underlying graph of the problem being solved. Let us consider for example the matrix-vector product $y = Ax$ where A is a sparse matrix, and look in detail at the processor that is computing y_i for some i. The statement $y_i \leftarrow y_i + A_{ij}x_j$ $y_i \leftarrow y_i + A_{ij}x_j$ implies that this processor will need the value x_j, so, if this variable is on a different processor, it needs to be sent over.

We can formalize this: Let the vectors x and y be distributed disjointly over the processors, and define uniquely $P(i)$ as the processor that owns index i. Then there is an edge (P, Q) if there is a nonzero element a_{ij} with $P = P(i)$ and $Q = P(j)$. This graph is undirected if the matrix is structurally symmetric, that is $a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$ $a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$.

The distribution of indices over the processors now gives us vertex and edge weights: a processor has a vertex weight that is the number of indices owned by it; an edge (P, Q) has a weight that is the number of vector components that need to be sent from Q to P , as described above.

The load balancing problem can now be formulated as follows:

Find a partitioning $P = \cup_i P_i$ $P = \cup_i P_i$, such the variation in vertex weights is minimal, and simulta-neously the edge weights are as low as possible.

Minimizing the variety in vertex weights implies that all processor have approximately the same amount of work. Keeping the edge weights low means that the amount of communication is low. These two objectives need not be satisfiable at the same time: some trade-off is likely.

Exercise 2.16. Consider the limit case where processors are infinitely fast and bandwidth between pro-cessors is also unlimited. What is the sole remaining factor determining the runtime? What graph problem do you need to solve now to find the optimal load balance? What property of a sparse matrix gives the worst case behaviour?

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-resources/wwwresources/site/textbookuploads/5345_scicompbook.pdf

9.2: Special Purpose Computing Architectures

GPU, Distributed, Grid, and Cloud Computing

Read section 2.8 to learn about GPU computing. GPU stands for Graphics Processing Unit. These are of interest because of the increase in the amount of graphics data handled by popular laptops and desktop computers. Furthermore, it turns out that since a GPU does primarily arithmetic computations, the architecture of a GPU is applicable to other types of applications that involve arithmetic on large amounts of data.

Read section 2.10 to learn about distributed, grid, and cloud computing. These are configurations of multiple computers for increasing performance and/or decreasing cost for high volume database access, sharing of resources, or accessing remote computer resources, respectively.

2.8 GPU computing

A *GPU* (or sometimes *General Purpose Graphics Processing Unit (GPGPU)*) is a special purpose processor, de-signed for fast graphics processing. Howver, since the operations done for graphics are a form of arithmetic, GPUs have gradually evolved a design that is also useful for non-graphics computing. The general design of a GPU is motivated by the 'graphics pipeline': in a form of *data parallelism* (section 2.4.1) identical operations are performed on many data elements, and a number of such blocks of data parallelism can be active at the same time.

Present day GPUs have an architecture that combines SIMD and MIMD parallelism. For instance, an NVidia GPU has 16 Streaming Multiprocessors (SMs), and a SMs consists of 8 Streaming Processors (SPs), which correspond to processor cores; see figure 2.13.

The SIMD, or , nature of GPUs becomes apparent in the way *CUDA* starts processes. A *kernel*, that is, a function that will be executed on the GPU, is started on *mn* cores by:

```
KernelProc<< m,n >>(args)
```

The collection of *mn* cores executing the kernel is known as a *grid*, and it is structured as m thread blocks of n threads each. A thread block can have up to 512 threads.

Recall that threads share an address space (see section 2.5.1.1), so they need a way to identify what part of the data each thread will operate on. For this, the blocks in a thread are numbered with *x*, *y* coordinates, and the threads in a block are numbered with *x*, *y*, *z* coordinates. Each thread knows its coordinates in the block, and its block's coordinates in the grid.

	SM	SP	thread blocks	threads
GPU	16		$8 \times 16 = 128$	
SM		8	8	768

Thread blocks are truly data parallel: if there is a conditional that makes some threads take the *true* branch and others the *false* branch, then one branch will be executed first, with all threads in the other branch stopped. Subsequently, *and not simultaneously*, the threads on the other branch will then execute their code. This may induce a severy performance penalty.

These are some of the differences between GPUs and regular CPUs:

- First of all, as of this writing (late 2010), GPUs are attached processors, so any data they operate on has to be transferred from the CPU. Since the memory *bandwidth* of this transfer is low, sufficient work has to be done on the GPU to overcome this overhead.
- Since GPUs are graphics processors, they put an emphasis on *single precision* floating point arithmetic. To accomodate the scientific computing community, *double precision* support is increasing, but double precision speed is typically half the single precision flop rate.
- A CPU is optimized to handle a single stream of instructions, that can be very heterogeneous in character; a GPU is made explicitly for data parallelism, and will perform badly on traditional codes.
- A CPU is made to handle one thread , or at best a small number of threads. A GPU needs a large number of threads, far larger than the number of computational cores, to perform efficiently.

2.10 Distributed computing, grid computing, cloud computing

In this section we will take a short look at terms such as cloud computing, and an earlier term *distributed computing*. These are concepts that have a relation to parallel computing in the scientific sense, but that differ in certain fundamental ways.

Distributed computing can be traced back as coming from large database servers, such as airline reservations systems, which had to be accessed by many travel agents simultaneously. For a large enough volume of database accesses, a single server will not suffice, so the mechanism of *remote procedure call* was invented, where the central server would call code (the procedure in question) on a different (remote) machine. The remote call could involve transfer of data, the data could be already on the remote machine, or there would be some mechanism that data on the two machines would stay synchronized. This gave rise to the *Storage Area Network (SAN)*. A generation later than distributed database systems, web servers had to deal with the same problem of many simultaneous accesses to what had to act like a single server.

We already see one big difference between distributed computing and high performance parallel computing. Scientific computing needs parallelism because a single simulation

becomes too big or slow for one machine; the business applications sketched above deal with many users executing small programs (that is, database or web queries) against a large data set. For scientific needs, the processors of a parallel machine (the nodes in a cluster) have to have a very fast connection to each other; for business needs no such network is needed, as long as the central dataset stays coherent.

Both in HPC and in business computing, the server has to stay available and operative, but in distributed computing there is considerably more liberty in how to realize this. For a user connecting to a service such as a database, it does not matter what actual server executes their request. Therefore, distributed computing can make use of *virtualization*: a virtual server can be spawned off on any piece of hardware.

An analogy can be made between remote servers, which supply computing power wherever it is needed, and the electric grid, which supplies electric power wherever it is needed. This has led to *grid computing* or *utility computing*, with the Teragrid, owned by the US National Science Foundation, as an example. Grid computing was originally intended as a way of hooking up computers connected by a Local Area Network (LAN) or Wide Area Network (WAN), often the Internet. The machines could be parallel themselves, and were often owned by dif-ferent institutions. More recently, it has been viewed as a way of sharing resources, both datasets and scientific instruments, over the network.

Some of what are now described as 'cloud applications' are of a massively parallel nature. One is Google's search engine, which indexes the whole of the Internet, and another is the GPS capability of Android mobile phones, which combines GIS, GPS, and mashup data. This type of parallelism is different from the scientific kind. One computing model that has been formalized is Google's MapReduce [24], which combines a data parallel aspect (the 'map' part) and a central accumulation part ('reduce'). Neither involves the tightly coupled neighbour-to-neighbour communication that is common in scientific computing. An open source framework for MapReduce computing exists in Hadoop [3]. Amazon offers a commercial Hadoop service.

The concept of having a remote computer serve user needs is attractive even if no large datasets are involved, since it absolves the user from the need of maintaining software on their local machine. Thus, Google Docs offers various 'office' applications without the user actually installing any software. This idea is sometimes called *Software-as-a-Service*, where the user connects to an 'application server', and accesses it through a client such as a web browser. In the case of Google Docs, there is no longer a large central dataset, but each user interacts with their own data, maintained on Google's servers. This of course has the large advantage that the data is available from anywhere the user has access to a web browser.

The term *cloud computing* usually refers to this internet-based model where the data is not maintained by the user. However, it can span some or all of the above concepts, depending on who uses the term. Here is a list of characteristics:

- A cloud is remote, involving applications running on servers that are not owned by the user. The user pays for services on a subscription basis, as pay-as-you-go.
- Cloud computing is typically associated with large amounts of data, either a single central dataset such as on airline database server, or many independent datasets such as for Google Docs, each of which are used by a single user or a small group of users. In the case of large datasets, they are stored distributedly, with concurrent access for the clients.
- Cloud computing makes a whole datacenter appear as a single computer to the user [71].
- The services offered by cloud computing are typically business applications and IT services, rather than scientific computing.
- Computing in a cloud is probably virtualized, or at least the client interfaces to an abstract notion of a server. These strategies often serve to 'move the work to the data'.
- Server processes are loosely coupled, at best synchronized through working on the same dataset.
- Cloud computing can be interface through a web browser; it can involve a business model that is 'pay as you go'.
- The scientific kind of parallelism is not possible or not efficient using cloud computing.

Cloud computing clearly depends on the following factors:

- The ubiquity of the internet;
- Virtualization of servers;
- Commoditization of processors and hard drives.

The infrastructure for cloud computing can be interesting from a computer science point of view, involving dis-tributed file systems, scheduling, virtualization, and mechanisms for ensuring high reliability.

Source: Victor Eijkhout, Edmond Chow, and Robert van de Geijn, https://s3.amazonaws.com/saylordotorg-

TOP500

Read this article, paying particular attention to the rankings of supercomputers, based on performance in running a LINPACK benchmark for computing a dense set of linear equations. Towards the end of the article, note the large number of cores in these supercomputers.

The **TOP500** project ranks and details the 500 most powerful non-distributed computer systems in the world. The project was started in 1993 and publishes an updated list of the supercomputers twice a year. The first of these updates always coincides with the International Supercomputing Conference in June, and the second is presented at the ACM/IEEE Supercomputing Conference in November. The project aims to provide a

reliable basis for tracking and detecting trends in high-performance computing and bases rankings on HPL, a portable implementation of the high-performance LINPACK benchmark written in Fortran for distributed-memory computers.

China currently dominates the list with 229 supercomputers, leading the second place (United States) by a record margin of 121. Since June 2020, the Japanese Fugaku is the world's most powerful supercomputer, reaching 415.53 petaFLOPS on the LINPACK benchmarks.

The TOP500 list is compiled by Jack Dongarra of the University of Tennessee, Knoxville, Erich Strohmaier and Horst Simon of the National Energy Research Scientific Computing Center (NERSC) and Lawrence Berkeley National Laboratory (LBNL), and until his death in 2014, Hans Meuer of the University of Mannheim, Germany.

The TOP500 project lists also Green500 and HPCG benchmark list.

History

Rapid growth of supercomputer performance, based on data from top500.org site. The logarithmic y-axis shows performance in GFLOPS.

In the early 1990s, a new definition of supercomputer was needed to produce meaningful statistics. After experimenting with metrics based on processor count in 1992, the idea arose

at the University of Mannheim to use a detailed listing of installed systems as the basis. In early 1993, Jack Dongarra was persuaded to join the project with his LINPACK benchmarks. A first test version was produced in May 1993, partly based on data available on the Internet, including the following sources:

- "List of the World's Most Powerful Computing Sites" maintained by Gunter Ahrendt
- David Kahaner, the director of the Asian Technology Information Program (ATIP); published a report in 1992, titled "Kahaner Report on Supercomputer in Japan" which had an immense amount of data.

The information from those sources was used for the first two lists. Since June 1993, the TOP500 is produced bi-annually based on site and vendor submissions only.

Since 1993, performance of the No. 1 ranked position has grown steadily in accordance with Moore's law, doubling roughly every 14 months. As of June 2018, *Summit* was fastest with an Rpeak of 187.6593 PFLOPS. For comparison, this is over 1,432,513 times faster than the Connection Machine CM-5/1024 (1,024 cores), which was the fastest system in November 1993 (twenty-five years prior) with an Rpeak of 131.0 G FLOPS.

Architecture and operating systems

Share of processor architecture families in TOP500 supercomputers by time trend

As of November 2019, all supercomputers on TOP500 are 64-bit, mostly based on CPUs using the x86-64 instruction set architecture (of which 474 are Intel EMT64-based

and 6 are AMD AMD64-based). The few exceptions are all based on RISC architectures). Thirteen supercomputers, including the top two, are based on the Power ISA used by IBM POWER microprocessors, three on Fujitsu-designed SPARC64 chips, two on ARM architecture, and one on the Chinese Sunway SW26010 design. One computer uses another non-US design, the Japanese PEZY-SC (based on the British ARM) as an accelerator paired with Intel's Xeon.

In recent years heterogeneous computing, mostly using Nvidia's graphics processing units (GPU) or Intel's x86-based Xeon Phi as coprocessors, has dominated the TOP500 because of better performance per watt ratios and higher absolute performance; it is almost required to make the top 10 or the top spot; the only major recent exception being the aforementioned K computer and Sunway TaihuLight. Tianhe-2 is also an interesting exception, as while it did use accelerators (just not GPUs), i.e. Xeon Phi, US sanctions blocked the upgrade, but still the upgraded Tianhe-2A is faster with non-US-based Matrix-2000, accelerators which where exploited ahead of schedule. Frontera supercomputer, ranked 5th, based on 28-core (56-thread) Intel Xeon Platinum is also an exception, as it was measured without help of GPUs which were later added, but it has two subsystems, both with Nvidia GPUs, and one of them additionally with POWER9 CPUs, and the other liquid immersion cooling.

Two computers which first appeared on the list in 2018 are based on architectures never before seen on the Top500. One was a new x86-64 microarchitecture from Chinese vendor Sugon, using Hygon Dhyana CPUs (these result from a collaboration with AMD, and are a minor variant of Zen-based AMD EPYC) and is ranked 38th, and the other was the first ever ARM-based computer on the list (then upgraded for June 2019) – using Cavium ThunderX2 CPUs. Before the ascendancy of 32-bit x86 and later 64-bit x86-64 in the early 2000s, a variety of RISC processor families made up most TOP500 supercomputers, including RISC architectures such as SPARC, MIPS, PA-RISC, and Alpha.

All the fastest supercomputers in the decade since the Earth Simulator supercomputer have used operating systems based on Linux. Since November 2017, all the listed supercomputers use an operating system based on the Linux kernel.

Since November 2015, no computer on the list runs Windows. In November 2014, Windows Azure cloud computer was no longer on the list of fastest supercomputers (its best rank was 165 in 2012), leaving the Shanghai Supercomputer Center's *Magic Cube* as the only Windows-based supercomputer on the list, until it was also dropped off from the list. It was ranked 436 in its last appearance on the list released in June 2015, while its best rank was 11 in 2008. There are no longer any Mac OS computers on the list. It had at most five such systems at a time, one more than the Windows systems that came later, while the total performance share for Windows was higher. The relative performance share of the whole list was however similar, and never high for either.

It has been well over a decade since MIPS systems dropped entirely off the list but the *Gyoukou* supercomputer that jumped to 4th place in November 2017 (after a huge upgrade) has MIPS as a small part of the coprocessors. Use of 2,048-core coprocessors (plus 8× 6-core MIPS, for each, that "no longer require to rely on an external Intel Xeon E5 host processor") make the supercomputer much more energy efficient than the other top 10 (i.e. it is 5th on Green500 and other such *ZettaScaler-2.2*-based systems take first three spots). At

19.86 million cores, it is by far the biggest system: almost double that of the best manycore system in the TOP500, the Chinese Sunway TaihuLight, ranked 3rd.

TOP 500

From the 52nd list (November 2018) to the 53rd list (June 2019), the Xeon Platinum-based Frontera is the only new supercomputer in the top 10 (at number 5) and the upgraded POWER9-based Lassen moved from 11th to 10th. Titan and Sequoia became the last Blue Gene/Q models to drop out of the top10; they had been ranked 9th and 10th in the 52nd list (and 1st and 2nd in the November 2012, 40th list) and are now 12th and 13th.

"For the first time, all 500 systems deliver a petaflop or more on the High Performance Linpack (HPL) benchmark, with the entry level to the list now at 1.022 petaflops." However, for a different benchmark "Summit and Sierra remain the only two systems to exceed a petaflop on the HPCG benchmark, delivering 2.9 petaflops and 1.8 petaflops, respectively. The average HPCG result on the current list is 213.3 teraflops, a marginal increase from 211.2 six months ago."

Of the top 10 computers in the 54th Top500 list, four are in the top 10 of the November 2019 Green500 list (most energy-efficient supercomputers):

Summit is #5 in the Green500 and #1 in the Top500

AI Bridging Cloud (ABCI) is #6 in the Green500 and #8 in the Top500

PANGEA III is #9 in the Green500 and #11 in the Top500

Sierra is #10 in the Green500 and #2 in the Top500

Top 10 positions of the 55th TOP500 in June 2020

Rank	Rmax Rpeak (PFLOPS)	Name	Model	Processor	Interconnect	Vendor	Site country, year	Operating system
1 ▲	415.530 513.855	Fugaku	Supercomputer Fugaku	A64FX	Tofu interconnect D	Fujitsu	RIKEN Center for Computational Science Japan, 2020	Linux (RHEL)
2 ▼	148.600 200.795	Summit	IBM Power System AC922	POWER9, Tesla V100	InfiniBand EDR	IBM	Oak Ridge National Laboratory United States, 2018	Linux (RHEL)

Top 10 positions of the 55th TOP500 in June 2020

Rank	Rmax Rpeak (PFLOPS)	Name	Model	Processor	Interconnect	Vendor	Site country, year	Operating system
3▼	94.640 125.712	Sierra	IBM Power System S922LC	POWER9, Tesla V100	InfiniBand EDR	IBM	Lawrence Livermore National Laboratory United States, 2018	Linux (RHEL)
4▼	93.015 125.436	Sunway TaihuLight	Sunway MPP	SW26010	Sunway	NRCPC	National Supercomputing Center in Wuxi China, 2016	Linux (Raise)
5▼	61.445 100.679	Tianhe-2A	TH-IVB-FEP	Xeon E5–2692 v2, Matrix-2000	TH Express-2	NUDT	National Supercomputing Center in Guangzhou China, 2013	Linux (Kylin)
6▲	35.450 51.721	HPC5	Dell	Xeon Gold 6252, Tesla V100	Mellanox HDR Infiniband	Dell EMC	Eni Italy, 2020	Linux (CentOS)
7▲	27.580 34.569	Selene	Nvidia	Epyc 7742, Ampere A100	Mellanox HDR Infiniband	Nvidia	Nvidia United States, 2020	Linux (Ubuntu)
8▼	23.516 38.746	Frontera	Dell C6420	Xeon Platinum 8280 (subsystems with e.g. POWER9 CPUs and Nvidia GPUs were added after official benchmarking)	InfiniBand HDR	Dell EMC	Texas Advanced Computing Center United States, 2019	Linux (CentOS)
9▲	21.640 29.354	Marconi-100	IBM Power System AC922	POWER9, Volta V100	Dual-rail Mellanox EDR Infiniband	IBM	CINECA Italy, 2020	Linux (RHEL)

Top 10 positions of the 55th TOP500 in June 2020

Rank	Rmax Rpeak (PFLOPS)	Name	Model	Processor	Interconnect	Vendor	Site country, year	Operating system
10 ▼	21.230 27.154	Piz Daint	Cray XC50	Xeon E5-2690 v3, Tesla P100	Aries	Cray	Swiss National Supercomputing Centre Switzerland, 2016	Linux (CLE)

Legend:

- Rank – Position within the TOP500 ranking. In the TOP500 list table, the computers are ordered first by their Rmax value. In the case of equal performances (Rmax value) for different computers, the order is by Rpeak. For sites that have the same computer, the order is by memory size and then alphabetically.
- Rmax – The highest score measured using the LINPACK benchmarks suite. This is the number that is used to rank the computers. Measured in quadrillions of 64 bit floating point operations per second, i.e., peta FLOPS.
- Rpeak – This is the theoretical peak performance of the system. Computed in petaFLOPS.
- Name – Some supercomputers are unique, at least on its location, and are thus named by their owner.
- Model – The computing platform as it is marketed.
- Processor – The instruction set architecture or processor microarchitecture, alongside GPU and accelerators when available.
- Interconnect – The interconnect between computing nodes. InfiniBand is most used (38%) by performance share, while Gigabit Ethernet is most used (54%) by number of computers.
- Vendor – The manufacturer of the platform and hardware.
- Site – The name of the facility operating the supercomputer.
- Country – The country in which the computer is located.
- Year – The year of installation or last major update.
- Operating system – The operating system that the computer uses.

Other rankings

Top countries

Numbers below represent the number of computers in the TOP500 that are in each of the listed countries or territories.

Distribution of supercomputers in the TOP500 list by country (June 2020)

Country or Territory	Systems

China	226
United States	114
Japan	29
France	19
Germany	16
Netherlands	15
Ireland	14
Canada	12
United Kingdom	10
Italy	7
Singapore	4
Brazil	4
Korea, South	3
Saudi Arabia	3
Norway	3
Australia	2
Russia	2
United Arab Emirates	2
Taiwan	2
Switzerland	2
Sweden	2
India	2
Finland	2
Spain	1
Czechia	1
Hong Kong (China)	1
Poland	1
Austria	1

Systems ranked No. 1 since 1976

- Supercomputer Fugaku (Riken Center for Computational Science Japan, June 2020 – Present)
- IBM Summit (Oak Ridge National Laboratory United States, June 2018 – June2020)
- NRCPC Sunway TaihuLight (National Supercomputing Center in Wuxi China, June 2016 – November 2017)
- NUDT Tianhe-2A (National Supercomputing Center of Guangzhou China, June 2013 – June 2016)
- Cray Titan (Oak Ridge National Laboratory United States, November 2012 – June 2013)
- IBM Sequoia Blue Gene/Q (Lawrence Livermore National Laboratory United States, June 2012 – November 2012)
- Fujitsu K computer (Riken Advanced Institute for Computational Science Japan, June 2011 – June 2012)
- NUDT Tianhe-1A (National Supercomputing Center of Tianjin China, November 2010 – June 2011)
- Cray Jaguar (Oak Ridge National Laboratory United States, November 2009 – November 2010)
- IBM Roadrunner (Los Alamos National Laboratory United States, June 2008 – November 2009)
- IBM Blue Gene/L (Lawrence Livermore National Laboratory United States, November 2004 – June 2008)
- NEC Earth Simulator (Earth Simulator Center Japan, June 2002 – November 2004)
- IBM ASCI White (Lawrence Livermore National Laboratory United States, November 2000 – June 2002)
- Intel ASCI Red (Sandia National Laboratories United States, June 1997 – November 2000)
- Hitachi CP-PACS (University of Tsukuba Japan, November 1996 – June 1997)
- Hitachi SR2201 (University of Tokyo Japan, June 1996 – November 1996)
- Fujitsu Numerical Wind Tunnel (National Aerospace Laboratory of Japan Japan, November 1994 – June 1996)
- Intel Paragon XP/S140 (Sandia National Laboratories United States, June 1994 – November 1994)
- Fujitsu Numerical Wind Tunnel (National Aerospace Laboratory of Japan Japan, November 1993 – June 1994)
- TMC CM-5 (Los Alamos National Laboratory United States, June 1993 – November 1993)
- NEC SX-3/44 (Japan, 1992–1993)
- Fujitsu VP2600/10 (Japan, 1990–1991)
- Cray Y-MP/832 (United States, 1988–1989)
- Cray-2 (United States, 1985–1987)
- Cray X-MP (United States, 1983–1985)
- Cray-1 (United States, 1976–1982)

Number of systems

By number of systems as of November 2018:

Top five processor generations by system quantity (November 2019)

Processor Generation	Systems
Intel Xeon E5 (Broadwell)	183
Intel Xeon Gold	165
Intel Xeon Platinum	30
Intel Xeon E5 (Haswell)	33
Intel Xeon Gold 62xx (Cascade Lake)	19

Top five vendors by system quantity (November 2019)

Vendor	Systems
Lenovo	173
Sugon	71
Inspur	66
Hewlett Packard Enterprise	35
Cray	45

Top five operating systems (November 2019)

Operating System	Systems
Linux	248
CentOS	129
Cray Linux Environment	34
bullx SCS	16
TOSS	11

Note: All operating systems of the TOP500 systems use Linux, but Linux above is *generic* Linux.

New developments in supercomputing

In November 2014, it was announced that the United States was developing two new supercomputers to exceed China's Tianhe-2 in its place as world's fastest supercomputer. The two computers, Sierra and Summit, will each exceed Tianhe-2's 55 peak petaflops. Summit, the more powerful of the two, will deliver 150–300 peak petaflops. On 10 April 2015, US government agencies banned selling chips, from Nvidia to supercomputing centers in China as "acting contrary to the national security ... interests of the United States"; and Intel Corporation from providing Xeon chips to China due to their use, according to the US, in researching nuclear weapons – research to which US export control law bans US companies from contributing – "The Department of Commerce refused, saying it was concerned about nuclear research being done with the machine."

On 29 July 2015, President Obama signed an executive order creating a National Strategic Computing Initiative calling for the accelerated development of an exascale (1000 petaflop) system and funding research into post-semiconductor computing.

In June 2016, Japanese firm Fujitsu announced at the International Supercomputing Conference that its future exascale supercomputer will feature processors of its own design that implement the ARMv8 architecture. The Flagship2020 program, by Fujitsu for RIKEN plans to break the exaflops barrier by 2020 through the Fugaku supercomputer, (and "it looks like China and France have a chance to do so and that the United States is content – for the moment at least – to wait until 2023 to break through the exaflops barrier.") These processors will also implement extensions to the ARMv8 architecture equivalent to HPC-ACE2 that Fujitsu is developing with ARM Holdings.

In June 2016, Sunway TaihuLight became the No. 1 system with 93 petaflop/s (PFLOP/s) on the Linpack benchmark.

In November 2016, Piz Daint was upgraded, moving it from 8th to 3rd, leaving the US with no systems under the TOP3 for only the 2nd time ever.

Inspur has been one of the largest HPC system manufacturer based out of Jinan, China. As of May 2017, Inspur has become the third manufacturer to have manufactured 64-way system – a record which has been previously mastered by IBM and HP. The company has registered over $10B in revenues and have successfully provided a number of HPC systems to countries outside China such as Sudan, Zimbabwe, Saudi Arabia, Venezuela. Inspur was also a major technology partner behind both the supercomputers from China, namely Tianhe-2 and Taihu which lead the top 2 positions of Top500 supercomputer list up to November 2017. Inspur and Supermicro released a few platforms aimed at HPC using GPU such as SR-AI and AGX-2 in May 2017.

In November 2017, for the second time in a row there were no system from the USA under the TOP3. #1 and #2 were installed in China, a system in Switzerland at #3, and a new system in Japan was #4 pushing the top US system to #5.

In June 2018, Summit, an IBM-built system at the Oak Ridge National Laboratory (ORNL) in Tennessee, USA, took the #1 spot with a performance of 122.3 petaflop/s (PFLOP/s), and Sierra, a very similar system at the Lawrence Livermore National Laboratory, CA, USA took #3. These two system took also the first two spots on the HPCG benchmark. Due to Summit and Sierra, the USA took back the lead as consumer of HPC performance with 38.2% of the overall installed performance while China was second with 29.1% of the overall installed performance. For the first time ever, the leading HPC manufacturer is not a US company. Lenovo took the lead with 23.8 percent of systems installed. It is followed by HPE with 15.8 percent, Inspur with 13.6 percent, Cray with 11.2 percent, and Sugon with 11 percent.

On 18 March 2019, the United States Department of Energy and Intel announced the first exaFLOP supercomputer would be operational at Argonne National Laboratory by the end of 2021. The computer, named "Aurora", is to be delivered to Argonne by Intel and Cray.

On 7 May 2019, The U.S. Department of Energy announced a contract with Cray to build the "Frontier" supercomputer at Oak Ridge National Laboratory. Frontier is anticipated to be operational in 2021 and, with a performance of greater than 1.5 exaflops, should then be the world's most powerful computer.

As of June 2019, all TOP500 systems deliver a petaflop or more on the High Performance Linpack (HPL) benchmark, with the entry level to the list now at 1.022 petaflops.

Large machines not on the list

Some major systems are not on the list. The largest example is the NCSA's Blue Waters which publicly announced the decision not to participate in the list because they do not feel it accurately indicates the ability of any system to do useful work. Other organizations decide not to list systems for security and/or commercial competitiveness reasons. Additional purpose-built machines that are not capable or do not run the benchmark were not included, such as RIKEN MDGRAPE-3 and MDGRAPE-4.

Computers and architectures that have dropped off the list

IBM Roadrunner is no longer on the list (nor is any other using the Cell coprocessor, or PowerXCell).

Although Itanium-based systems reached second rank in 2004, none now remain.

Similarly (non-SIMD-style) vector processors (NEC-based such as the Earth simulator that was fastest in 2002) have also fallen off the list. Also the Sun Starfire computers that occupied many spots in the past now no longer appear.

The last non-Linux computers on the list – the two AIX ones – running on POWER7 (in July 2017 ranked 494th and 495th originally 86th and 85th), dropped off the list in November 2017.