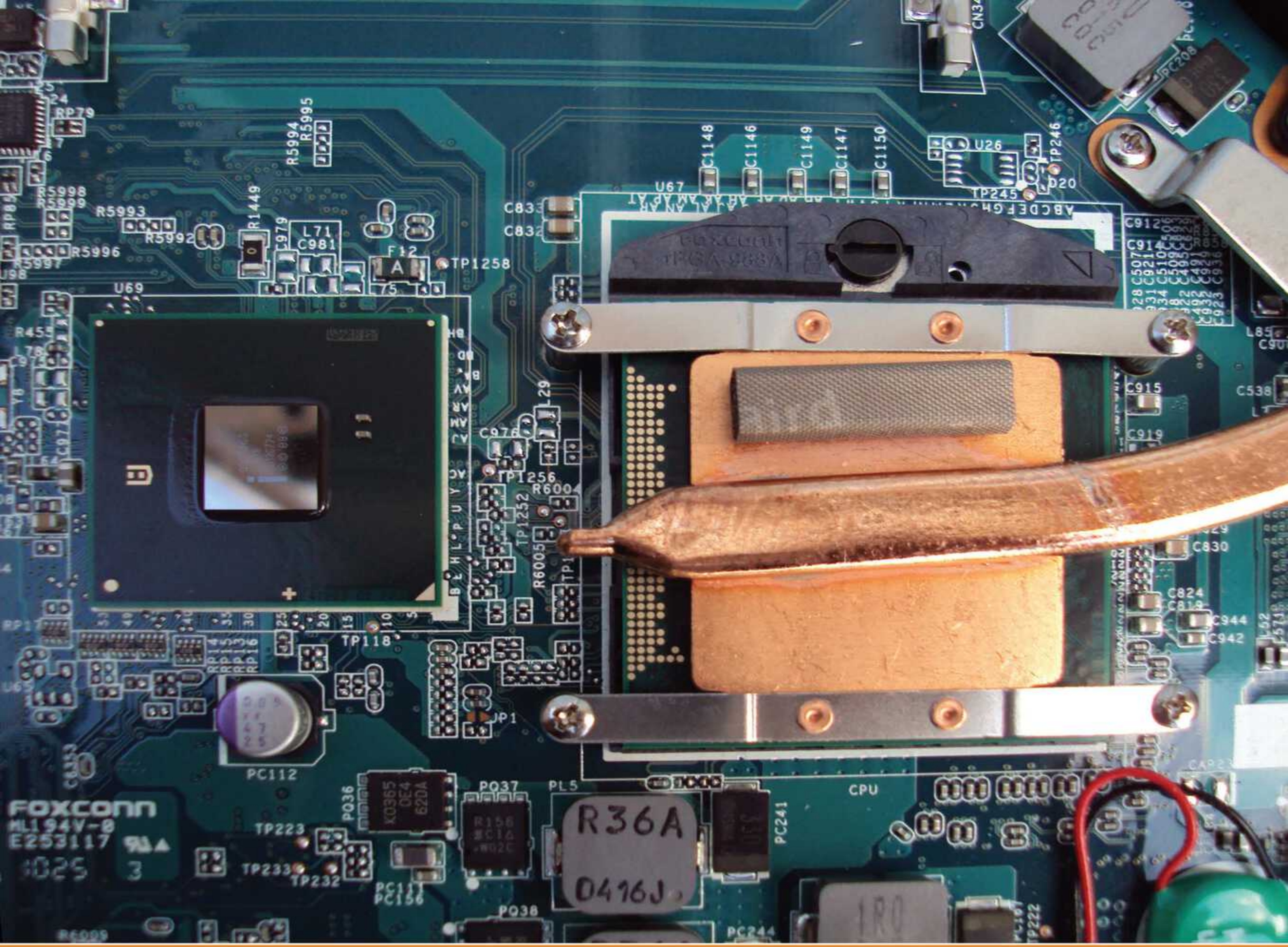


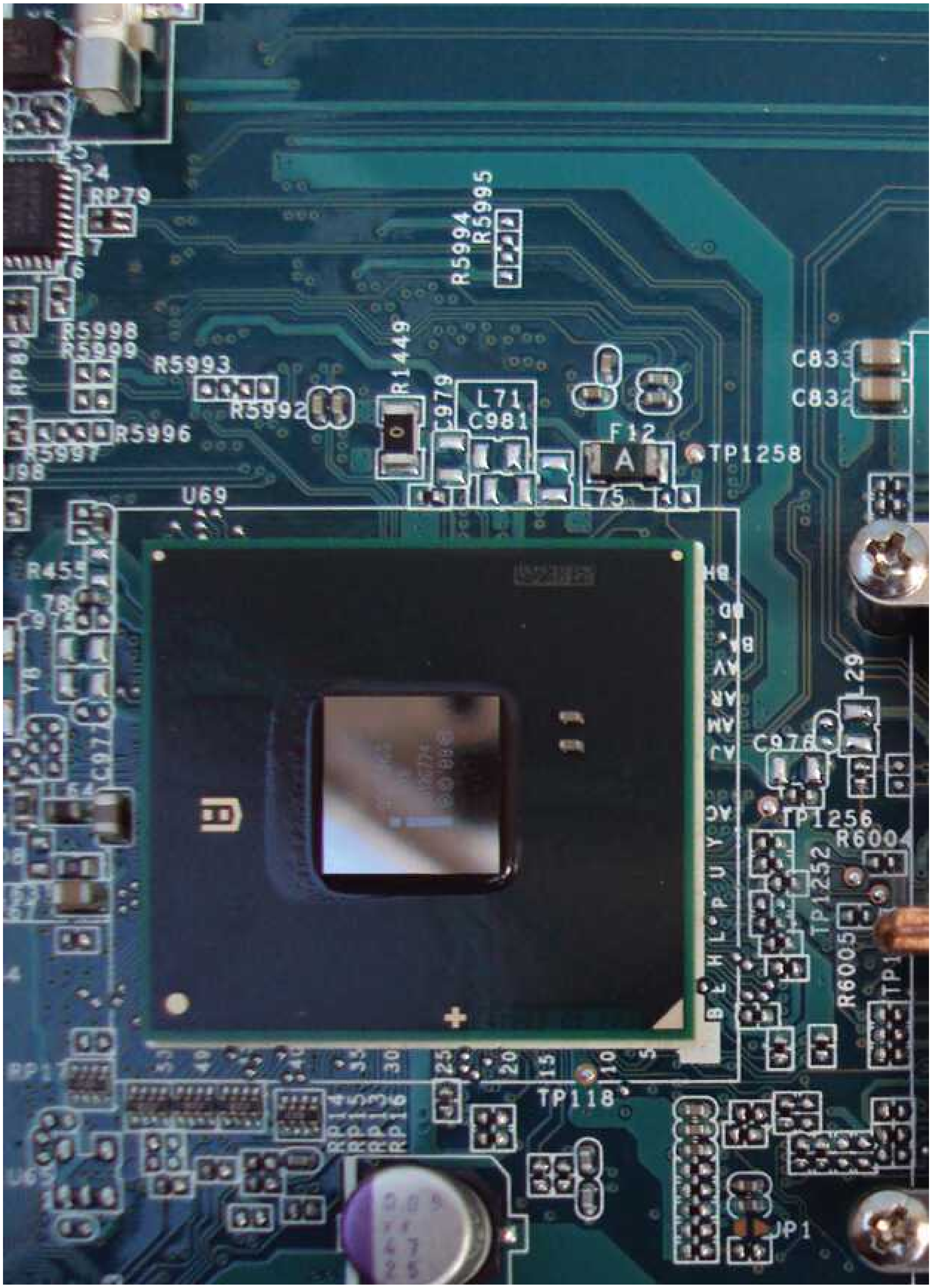
Full Explanation of Central Processing Unit (CPU)

Author: DAYONG LI

Date: September 30, 2020

Version: 1.0

Two heads are better than one.—- Chinese Old Saying


RP79
R5995
R5994
R5998
R5999
R5993
R1449
R5992
L71
C979
C981
F12
TP1258
C833
C832
R5996
R5997
U98
U69
R455
C976
TP1256
R6004
TP1252
R6005
TP118
JP1
U65
RP17

Full Explanation of Central Processing Unit (CPU)

DAYONG LI

September 30, 2020

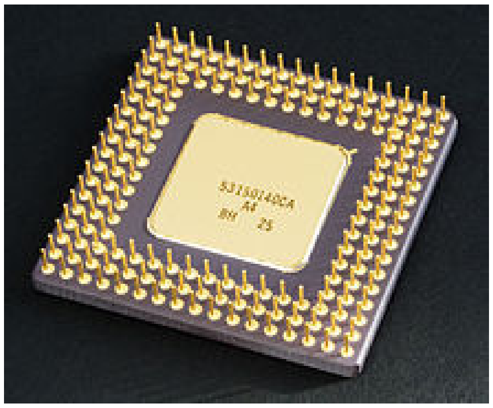
Version: 1.0

Two heads are better than one. — Chinese Old Saying

Copyright

Preface

An Intel 80486DX2 CPU, as seen from above


Bottom side of an Intel 80486DX2, showing its pins

A **central processing unit** (**CPU**) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions. The computer industry has used the term "central processing unit" at least since the early 1960s. Traditionally, the term "CPU" refers to a **processor** , more specifically to its processing unit and control unit (CU), distinguishing these core elements of a computer from external components such as main memory and I/O circuitry.

The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains almost unchanged. Principal components of a CPU include the arithmetic logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations and a control unit that orchestrates the fetching (from memory) and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

Most modern CPUs are microprocessors, meaning they are contained on a single integrated circuit (IC) chip. An IC that contains a CPU may also contain memory, peripheral interfaces, and other components of a computer; such integrated devices are variously called microcontrollers or systems on a chip (SoC). Some computers employ a multi-core processor, which is a single chip containing two or more CPUs called "cores"; in that context, one can speak of such single chips as "sockets".

Array processors or vector processors have multiple processors that operate in parallel, with no unit considered central. There also exists the concept of virtual CPUs which are an abstraction of dynamical aggregated computational resources.

Table of Contents

Chapter 1 Introduction

1.1 History

EDVAC, one of the first stored-program computers

Early computers such as the ENIAC had to be physically rewired to perform different tasks, which caused these machines to be called "fixed-program computers". Since the term "CPU" is generally defined as a device for software (computer program) execution, the earliest devices that could rightly be called CPUs came with the advent of the stored-program computer.

The idea of a stored-program computer had been already present in the design of J. Presper Eckert and John William Mauchly's ENIAC, but was initially omitted so that it could be finished sooner. On June 30, 1945, before ENIAC was made, mathematician John von Neumann distributed the paper entitled *First Draft of a Report on the EDVAC* . It was the outline of a stored-program computer that would eventually be completed in August 1949. EDVAC was designed to perform a certain number of instructions (or operations) of various types. Significantly, the programs written for EDVAC were to be stored in high-speed computer memory rather than specified by the physical wiring of the computer. This overcame a severe limitation of ENIAC, which was the considerable time and effort required to reconfigure the computer to perform a new task. With von Neumann's design, the program that EDVAC ran could be changed simply by changing the contents of the memory. EDVAC, however, was not the first stored-program computer; the Manchester Baby, a small-scale experimental stored-program computer, ran its first program on 21 June 1948 and the Manchester Mark 1 ran its first program during the night of 16–17 June 1949.

Early CPUs were custom designs used as part of a larger and sometimes distinctive computer. However, this method of designing custom CPUs for a particular application has largely given way to the development of multi-purpose processors produced in large quantities. This standardization began in the era of discrete transistor mainframes and minicomputers and has rapidly accelerated with the popularization of the integrated circuit (IC). The IC has allowed increasingly complex CPUs to be designed and manufactured to tolerances on the order of nanometers. Both the miniaturization and standardization of CPUs have increased the presence of digital devices in modern life far beyond the limited application of dedicated computing machines. Modern microprocessors appear in electronic devices ranging from automobiles to cellphones, and sometimes even in toys.

While von Neumann is most often credited with the design of the stored-program computer because of his design of EDVAC, and the design became known as the von Neumann architecture, others before him, such as Konrad Zuse, had suggested and implemented similar ideas. The so-called Harvard architecture of the Harvard Mark I, which was completed before EDVAC, also used a stored-program design using punched paper tape rather than electronic memory. The key difference between the von Neumann and Harvard architectures is that the latter separates the storage and treatment of CPU instructions and data, while the former uses the same memory space for both. Most modern CPUs are primarily von Neumann in design, but CPUs with the Harvard architecture are seen as well, especially in embedded applications; for instance, the Atmel AVR microcontrollers are Harvard architecture processors.

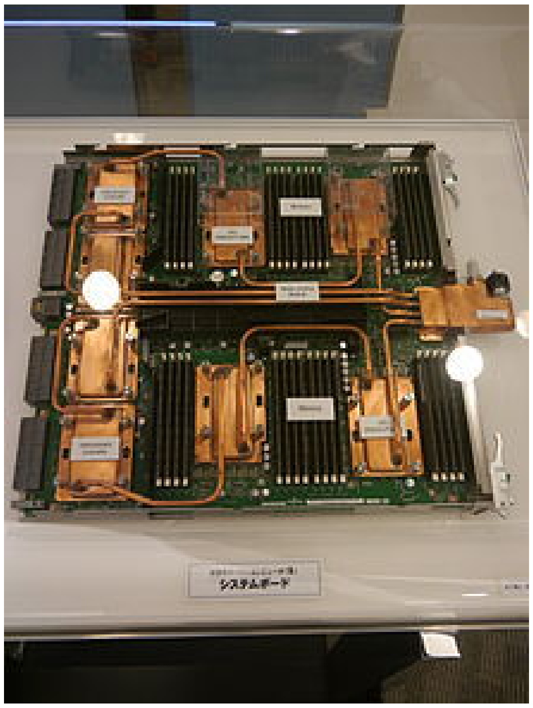
Relays and vacuum tubes (thermionic tubes) were commonly used as switching elements; a useful computer requires thousands or tens of thousands of switching devices. The overall speed of a system is dependent on the speed of the switches. Tube computers like EDVAC tended to average eight hours between failures, whereas relay computers like the (slower, but earlier) Harvard Mark I failed very rarely. In the end, tube-based CPUs became dominant because the significant speed advantages afforded generally outweighed the reliability problems. Most of these early synchronous CPUs ran at low clock rates compared to modern microelectronic designs. Clock signal frequencies ranging from 100 kHz to 4 MHz were very common at this time, limited largely by the speed of the switching devices they were built with.

1.1.1 Transistor CPUs

IBM PowerPC 604e processor

The design complexity of CPUs increased as various technologies facilitated building smaller and more reliable electronic devices. The first such improvement came with the advent of the transistor. Transistorized CPUs during the 1950s and 1960s no longer had to be built out of bulky, unreliable and fragile switching elements like vacuum tubes and relays. With this improvement more complex and reliable CPUs were built onto one or several printed circuit boards containing discrete (individual) components.

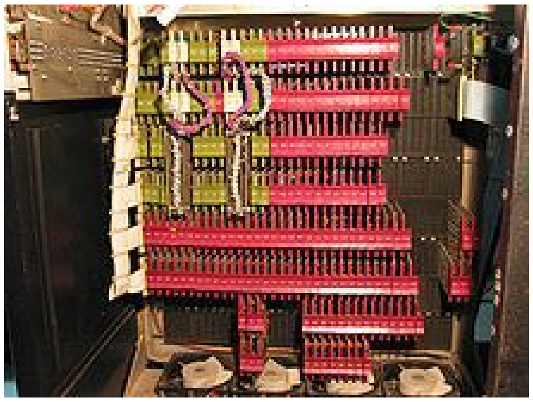
In 1964, IBM introduced its IBM System/360 computer architecture that was used in a series of computers capable of running the same programs with different speed and performance. This was significant at a time when most electronic computers were incompatible with one another, even those made by the same manufacturer. To facilitate this improvement, IBM used the concept of a microprogram (often called "microcode"), which still sees widespread usage in modern CPUs. The System/360 architecture was so popular that it dominated the mainframe computer market for decades and left a legacy that is still continued by similar modern computers like the IBM zSeries. In 1965, Digital Equipment Corporation (DEC) introduced another influential computer aimed at the scientific and research markets, the PDP-8.



Fujitsu board with SPARC64 VIIIfx processors

Transistor-based computers had several distinct advantages over their predecessors. Aside from facilitating increased reliability and lower power consumption, transistors also allowed CPUs to operate at much higher speeds because of the short switching time of a transistor in comparison to a tube or relay. The increased reliability and dramatically increased speed of the switching elements (which were almost exclusively transistors by this time), CPU clock rates in the tens of megahertz were easily obtained during this period. Additionally while discrete transistor and IC CPUs were in heavy usage, new high-performance designs like SIMD (Single Instruction Multiple Data) vector processors began to appear. These early experimental designs later gave rise to the era of specialized supercomputers like those made by Cray Inc and Fujitsu Ltd.

1.1.2 Small-scale integration CPUs



CPU, core memory and external bus interface of a DEC PDP-8/I, made of medium-scale integrated circuits

During this period, a method of manufacturing many interconnected transistors in a compact space was developed. The integrated circuit (IC) allowed a large number of transistors to be manufactured on a single semiconductor-based die, or "chip". At first, only very basic non-specialized digital circuits such as NOR gates were miniaturized into ICs. CPUs based on these "building block" ICs are generally referred to as "small-scale integration" (SSI) devices. SSI ICs, such as the ones used in the Apollo Guidance Computer, usually contained up to a few dozen transistors. To build an entire CPU out of SSI ICs required thousands of individual chips, but still consumed much less space and power than earlier discrete transistor designs.

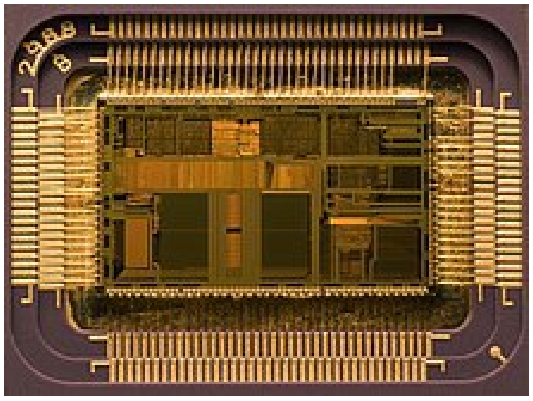
IBM's System/370, follow-on to the System/360, used SSI ICs rather than Solid Logic Technology discrete-transistor modules. DEC's PDP-8/I and KI10 PDP-10 also switched from the individual transistors used by the PDP-8 and PDP-10 to SSI ICs, and their extremely popular PDP-11 line was originally built with SSI ICs but was eventually implemented with LSI components once these became practical.

1.1.3 Large-scale integration CPUs

Lee Boysel published influential articles, including a 1967 "manifesto", which described how to build the equivalent of a 32-bit mainframe computer from a relatively small number of large-scale integration circuits (LSI). At the time, the only way to build LSI chips, which are chips with a hundred or more gates, was to build them using a MOS process (i.e., PMOS logic, NMOS logic, or CMOS logic). However, some companies continued to build processors out of bipolar chips because bipolar junction transistors were so much faster than MOS chips; for example, Datapoint built processors out of transistor–transistor logic (TTL) chips until the early 1980s. At the time, MOS ICs were so slow that they were considered useful only in a few niche applications that required low power.

As the microelectronic technology advanced, an increasing number of transistors were placed on ICs, decreasing the number of individual ICs needed for a complete CPU. MSI and LSI ICs increased transistor counts to hundreds, and then thousands. By 1968, the number of ICs required to build a complete CPU had been reduced to 24 ICs of eight different types, with each IC containing roughly 1000 MOSFETs. In stark contrast with its SSI and MSI predecessors, the first LSI implementation of the PDP-11 contained a CPU composed of only four LSI integrated circuits.

1.1.4 Microprocessors



Die of an Intel 80486DX2 microprocessor (actual size: 12 × 6.75 mm) in its packaging

Intel Core i5 CPU on a Vaio E series laptop motherboard (on the right, beneath the heat pipe)

Since the introduction of the first commercially available microprocessor, the Intel 4004 in 1970, and the first widely used microprocessor, the Intel 8080 in 1974, this class of CPUs has almost completely overtaken all other central processing unit implementation methods. Mainframe and minicomputer manufacturers of the time launched proprietary IC development programs to upgrade their older computer architectures, and eventually produced instruction set compatible microprocessors that were backward-compatible with their older hardware and software. Combined with the advent and eventual success of the ubiquitous personal computer, the term *CPU* is now applied almost exclusively to microprocessors. Several CPUs (denoted *cores*) can be combined in a single processing chip.

Previous generations of CPUs were implemented as discrete components and numerous small integrated circuits (ICs) on one or more circuit boards. Microprocessors, on the other hand, are CPUs manufactured on a very small number of ICs; usually just one. The overall smaller CPU size, as a result of being implemented on a single die, means faster switching time because of physical factors like decreased gate parasitic capacitance. This has allowed synchronous microprocessors to have clock rates ranging from tens of megahertz to several gigahertz. Additionally, the ability to construct exceedingly small transistors on an IC has increased the complexity and number of transistors in a single CPU many fold. This widely observed trend is described by Moore's law, which had proven to be a fairly accurate predictor of the growth of CPU (and other IC) complexity until 2016.

While the complexity, size, construction and general form of CPUs have changed enormously since 1950, the basic design and function has not changed much at all. Almost all common CPUs today can be very accurately described as von Neumann stored-program machines. As Moore's law no longer holds, concerns have arisen about the limits of integrated circuit transistor technology. Extreme miniaturization of electronic gates is causing the effects of phenomena like electromigration and subthreshold leakage to become much more significant. These newer concerns are among the many factors causing researchers to investigate new methods of computing such as the quantum computer, as well as to expand the usage of parallelism and other methods that extend the usefulness of the classical von Neumann model.

1.2 Operation

The fundamental operation of most CPUs, regardless of the physical form they take, is to execute a sequence of stored instructions that is called a program. The instructions to be executed are kept in some kind of computer memory. Nearly all CPUs follow the fetch, decode and execute steps in their operation, which are collectively known as the instruction cycle.

After the execution of an instruction, the entire process repeats, with the next instruction cycle normally fetching the next-in-sequence instruction because of the incremented value in the program counter. If a jump instruction was executed, the program counter will be modified to contain the address of the instruction that was jumped to and program execution continues normally. In more complex CPUs, multiple instructions can be fetched, decoded and executed simultaneously. This section describes what is generally referred to as the "classic RISC pipeline", which is quite common among the simple CPUs used in many electronic devices (often called microcontroller). It largely ignores the important role of CPU cache, and therefore the access stage of the pipeline.

Some instructions manipulate the program counter rather than producing result data directly; such instructions are generally called "jumps" and facilitate program behavior like loops, conditional program execution (through the use of a conditional jump), and existence of functions. In some processors, some other instructions change the state of bits in a "flags" register. These flags can be used to influence how a program behaves, since they often indicate the outcome of various operations. For example, in such processors a "compare" instruction evaluates two values and sets or clears bits in the flags register to indicate which one is greater or whether they are equal; one of these flags could then be used by a later jump instruction to determine program flow.

1.2.1 Fetch

The first step, fetch, involves retrieving an instruction (which is represented by a number or sequence of numbers) from program memory. The instruction's location (address) in program memory is determined by a program counter (PC), which stores a number that identifies the address of the next instruction to be fetched. After an instruction is fetched, the PC is incremented by the length of the instruction so that it will contain the address of the next instruction in the sequence. Often, the instruction to be fetched must be retrieved from relatively slow memory, causing the CPU to stall while waiting for the instruction to be returned. This issue is largely addressed in modern processors by caches and pipeline architectures (see below).

1.2.2 Decode

The instruction that the CPU fetches from memory determines what the CPU will do. In the decode step, performed by the circuitry known as the *instruction decoder* , the instruction is converted into signals that control other parts of the CPU.

The way in which the instruction is interpreted is defined by the CPU's instruction set architecture (ISA). Often, one group of bits (that is, a "field") within the instruction, called the opcode, indicates which operation is to be performed, while the remaining fields usually provide supplemental information required for the operation, such as the operands. Those operands may be specified as a constant value (called an immediate value), or as the location of a value that may be a processor register or a memory address, as determined by some addressing mode.

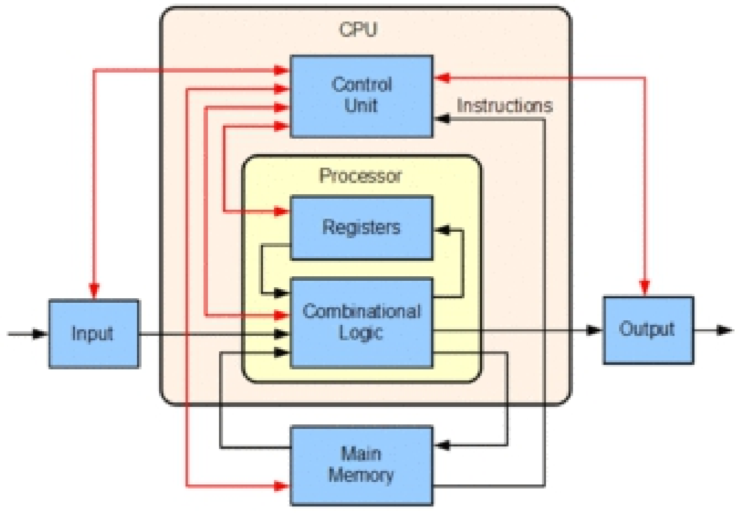
In some CPU designs the instruction decoder is implemented as a hardwired, unchangeable circuit. In others, a microprogram is used to translate instructions into sets of CPU configuration signals that are applied sequentially over multiple clock pulses. In some cases the memory that stores the microprogram is rewritable, making it possible to change the way in which the CPU decodes instructions.

1.2.3 Execute

After the fetch and decode steps, the execute step is performed. Depending on the CPU architecture, this may consist of a single action or a sequence of actions. During each action, various parts of the CPU are electrically connected so they can perform all or part of the desired operation and then the action is completed, typically in response to a clock pulse. Very often the results are written to an internal CPU register for quick access by subsequent instructions. In other cases results may be written to slower, but less expensive and higher capacity main memory.

For example, if an addition instruction is to be executed, the arithmetic logic unit (ALU) inputs are connected to a pair of operand sources (numbers to be summed), the ALU is configured to perform an addition operation so that the sum of its operand inputs will appear at its output, and the ALU output is connected to storage (e.g., a register or memory) that will receive the sum. When the clock pulse occurs, the sum will be transferred to storage and, if the resulting sum is too large (i.e., it is larger than the ALU's output word size), an arithmetic overflow flag will be set.

1.3 Structure and implementation



Block diagram of a basic uniprocessor-CPU computer. Black lines indicate data flow, whereas red lines indicate control flow; arrows indicate flow directions.

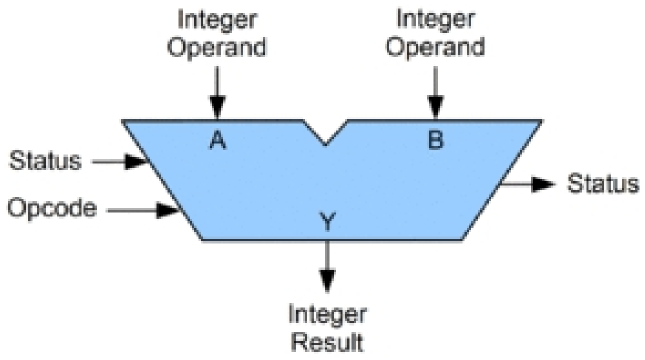
Hardwired into a CPU's circuitry is a set of basic operations it can perform, called an instruction set. Such operations may involve, for example, adding or subtracting two numbers, comparing two numbers, or jumping to a different part of a program. Each basic operation is represented by a particular combination of bits, known as the machine language opcode; while executing instructions in a machine language program, the CPU decides which operation to perform by "decoding" the opcode. A complete machine language instruction consists of an opcode and, in many cases, additional bits that specify arguments for the operation (for example, the numbers to be summed in the case of an addition operation). Going up the complexity scale, a machine language program is a collection of machine language instructions that the CPU executes.

The actual mathematical operation for each instruction is performed by a combinational logic circuit within the CPU's processor known as the arithmetic logic unit or ALU. In general, a CPU executes an instruction by fetching it from memory, using its ALU to perform an operation, and then storing the result to memory. Beside the instructions for integer mathematics and logic operations, various other machine instructions exist, such as those for loading data from memory and storing it back, branching operations, and mathematical operations on floating-point numbers performed by the CPU's floating-point unit (FPU).

1.3.1 Control unit

The control unit of the CPU contains circuitry that uses electrical signals to direct the entire computer system to carry out stored program instructions. The control unit does not execute program instructions; rather, it directs other parts of the system to do so. The control unit communicates with both the ALU and memory.

1.3.2 Arithmetic logic unit



Symbolic representation of an ALU and its input and output signals

The arithmetic logic unit (ALU) is a digital circuit within the processor that performs integer arithmetic and bitwise logic operations. The inputs to the ALU are the data words to be operated on (called operands), status information from previous operations, and a code from the control unit indicating which operation to perform. Depending on the instruction being executed, the operands may come from internal CPU registers or external memory, or they may be constants generated by the ALU itself.

When all input signals have settled and propagated through the ALU circuitry, the result of the performed operation appears at the ALU's outputs. The result consists of both a data word, which may be stored in a register or memory, and status information that is typically stored in a special, internal CPU register reserved for this purpose.

1.3.3 Memory management unit (MMU)

Most high-end microprocessors (in desktop, laptop, server computers) have a memory management unit, translating logical addresses into physical RAM addresses, providing memory protection and paging abilities, useful for virtual memory. Simpler processors, especially microcontrollers, usually don't include an MMU.

1.3.4 Clock rate

Most CPUs are synchronous circuits, which means they employ a clock signal to pace their sequential operations. The clock signal is produced by an external oscillator circuit that generates a consistent number of pulses each second in the form of a periodic square wave. The frequency of the clock pulses determines the rate at which a CPU executes instructions and, consequently, the faster the clock, the more instructions the CPU will execute each second.

To ensure proper operation of the CPU, the clock period is longer than the maximum time needed for all signals to propagate (move) through the CPU. In setting the clock period to a value well above the worst-case propagation delay, it is possible to design the entire CPU and the way it moves data around the "edges" of the rising and falling clock signal. This has the advantage of simplifying the CPU significantly, both from a design perspective and a component-count perspective. However, it also carries the disadvantage that the entire CPU must wait on its slowest elements, even though some portions of it are much faster. This limitation has largely been compensated for by various methods of increasing CPU parallelism (see below).

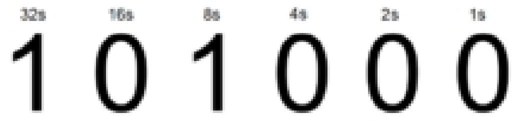
However, architectural improvements alone do not solve all of the drawbacks of globally synchronous CPUs. For example, a clock signal is subject to the delays of any other electrical signal. Higher clock rates in increasingly complex CPUs make it more difficult to keep the clock signal in phase (synchronized) throughout the entire unit. This has led many modern CPUs to require multiple identical clock signals to be provided to avoid delaying a single signal significantly enough to cause the CPU to malfunction. Another major issue, as clock rates increase dramatically, is the amount of heat that is dissipated by the CPU. The constantly changing clock causes many components to switch regardless of whether they are being used at that time. In general, a component that is switching uses more energy than an element in a static state. Therefore, as clock rate increases, so does energy consumption, causing the CPU to require more heat dissipation in the form of CPU cooling solutions.

One method of dealing with the switching of unneeded components is called clock gating, which involves turning off the clock signal to unneeded components (effectively disabling them). However, this is often regarded as difficult to implement and therefore does not see common usage outside of very low-power designs. One notable recent CPU design that uses extensive clock gating is the IBM PowerPC-based Xenon used in the Xbox 360; that way, power requirements of the Xbox 360 are greatly reduced. Another method of addressing some of the problems with a global clock signal is the removal of the clock signal altogether. While removing the global clock signal makes the design process considerably more complex in many ways, asynchronous (or clockless) designs carry marked advantages in power consumption and heat dissipation in comparison with similar synchronous designs. While somewhat uncommon, entire asynchronous CPUs have been built without using a global clock signal. Two notable examples of this are the ARM compliant AMULET and the MIPS R3000 compatible MiniMIPS.

Rather than totally removing the clock signal, some CPU designs allow certain portions of the device to be asynchronous, such as using asynchronous ALUs in conjunction with superscalar pipelining to achieve some arithmetic performance gains. While it is not altogether clear whether totally asynchronous designs can perform at a comparable or better level than their synchronous counterparts, it is evident that they do at least excel in simpler math operations. This, combined with their excellent power consumption and heat dissipation properties, makes them very suitable for embedded computers.

1.3.5 Integer range

Every CPU represents numerical values in a specific way. For example, some early digital computers represented numbers as familiar decimal (base 10) numeral system values, and others have employed more unusual representations such as ternary (base three). Nearly all modern CPUs represent numbers in binary form, with each digit being represented by some two-valued physical quantity such as a "high" or "low" voltage.



A six-bit word containing the binary encoded representation of decimal value 40. Most modern CPUs employ word sizes that are a power of two, for example 8, 16, 32 or 64 bits.

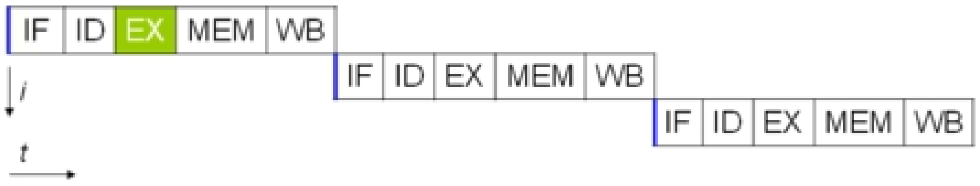
Related to numeric representation is the size and precision of integer numbers that a CPU can represent. In the case of a binary CPU, this is measured by the number of bits (significant digits of a binary encoded integer) that the CPU can process in one operation, which is commonly called *word size* , *bit width* , *data path width* , *integer precision* , or *integer size* . A CPU's integer size determines the range of integer values it can directly operate on. For example, an 8-bit CPU can directly manipulate integers represented by eight bits, which have a range of 256 (2^8) discrete integer values.

Integer range can also affect the number of memory locations the CPU can directly address (an address is an integer value representing a specific memory location). For example, if a binary CPU uses 32 bits to represent a memory address then it can directly address 2^{32} memory locations. To circumvent this limitation and for various other reasons, some CPUs use mechanisms (such as bank switching) that allow additional memory to be addressed.

CPUs with larger word sizes require more circuitry and consequently are physically larger, cost more and consume more power (and therefore generate more heat). As a result, smaller 4- or 8-bit microcontrollers are commonly used in modern applications even though CPUs with much larger word sizes (such as 16, 32, 64, even 128-bit) are available. When higher performance is required, however, the benefits of a larger word size (larger data ranges and address spaces) may outweigh the disadvantages. A CPU can have internal data paths shorter than the word size to reduce size and cost. For example, even though the IBM System/360 instruction set was a 32-bit instruction set, the System/360 Model 30 and Model 40 had 8-bit data paths in the arithmetic logical unit, so that a 32-bit add required four cycles, one for each 8 bits of the operands, and, even though the Motorola 68000 series instruction set was a 32-bit instruction set, the Motorola 68000 and Motorola 68010 had 16-bit data paths in the arithmetic logical unit, so that a 32-bit add required two cycles.

To gain some of the advantages afforded by both lower and higher bit lengths, many instruction sets have different bit widths for integer and floating-point data, allowing CPUs implementing that instruction set to have different bit widths for different portions of the device. For example, the IBM System/360 instruction set was primarily 32 bit, but supported 64-bit floating point values to facilitate greater accuracy and range in floating point numbers. The System/360 Model 65 had an 8-bit adder for decimal and fixed-point binary arithmetic and a 60-bit adder for floating-point arithmetic. Many later CPU designs use similar mixed bit width, especially when the processor is meant for general-purpose usage where a reasonable balance of integer and floating point capability is required.

1.3.6 Parallelism



Model of a subscalar CPU, in which it takes fifteen clock cycles to complete three instructions

The description of the basic operation of a CPU offered in the previous section describes the simplest form that a CPU can take. This type of CPU, usually referred to as *subscalar* , operates on and executes one instruction on one or two pieces of data at a time, that is less than one instruction per clock cycle (IPC < 1).

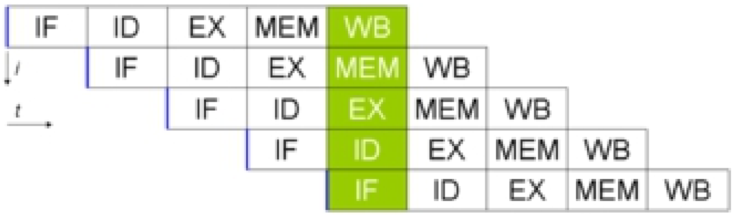
This process gives rise to an inherent inefficiency in subscalar CPUs. Since only one instruction is executed at a time, the entire CPU must wait for that instruction to complete before proceeding to the next instruction. As a result, the subscalar CPU gets "hung up" on instructions which take more than one clock cycle to complete execution. Even adding a second execution unit (see below) does not improve performance much; rather than one pathway being hung up, now two pathways are hung up and the number of unused transistors is increased. This design, wherein the CPU's execution resources can operate on only one instruction at a time, can only possibly reach *scalar* performance (one instruction per clock cycle, IPC = 1). However, the performance is nearly always subscalar (less than one instruction per clock cycle, IPC < 1).

Attempts to achieve scalar and better performance have resulted in a variety of design methodologies that cause the CPU to behave less linearly and more in parallel. When referring to parallelism in CPUs, two terms are generally used to classify these design techniques:

- *instruction-level parallelism* (ILP), which seeks to increase the rate at which instructions are executed within a CPU (that is, to increase the use of on-die execution resources);
- *task-level parallelism* (TLP), which purposes to increase the number of threads or processes that a CPU can execute simultaneously.

Each methodology differs both in the ways in which they are implemented, as well as the relative effectiveness they afford in increasing the CPU's performance for an application.

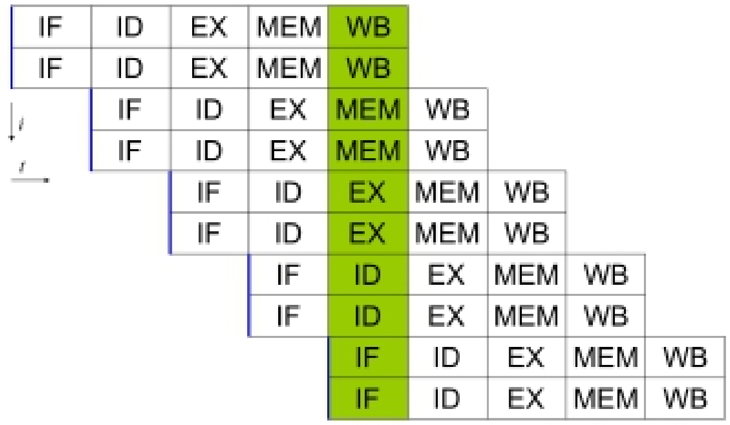
Instruction-level parallelism



Basic five-stage pipeline. In the best case scenario, this pipeline can sustain a completion rate of one instruction per clock cycle.

One of the simplest methods used to accomplish increased parallelism is to begin the first steps of instruction fetching and decoding before the prior instruction finishes executing. This is the simplest form of a technique known as instruction pipelining, and is used in almost all modern general-purpose CPUs. Pipelining allows more than one instruction to be executed at any given time by breaking down the execution pathway into discrete stages. This separation can be compared to an assembly line, in which an instruction is made more complete at each stage until it exits the execution pipeline and is retired.

Pipelining does, however, introduce the possibility for a situation where the result of the previous operation is needed to complete the next operation; a condition often termed data dependency conflict. To cope with this, additional care must be taken to check for these sorts of conditions and delay a portion of the instruction pipeline if this occurs. Naturally, accomplishing this requires additional circuitry, so pipelined processors are more complex than subscalar ones (though not very significantly so). A pipelined processor can become very nearly scalar, inhibited only by pipeline stalls (an instruction spending more than one clock cycle in a stage).



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

A simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per clock cycle can be completed.

Further improvement upon the idea of instruction pipelining led to the development of a method that decreases the idle time of CPU components even further. Designs that are said to be *superscalar* include a long instruction pipeline and multiple identical execution units, such as load-store units, arithmetic-logic units, floating-point units and address generation units. In a superscalar pipeline, multiple instructions are read and passed to a dispatcher, which decides whether or not the instructions can be executed in parallel (simultaneously). If so they are dispatched to available execution units, resulting in the ability for several instructions to be executed simultaneously. In general, the more instructions a superscalar CPU is able to dispatch simultaneously to waiting execution units, the more instructions will be completed in a given cycle.

Most of the difficulty in the design of a superscalar CPU architecture lies in creating an effective dispatcher. The dispatcher needs to be able to quickly and correctly determine whether instructions can be executed in parallel, as well as dispatch them in such a way as to keep as many execution units busy as possible. This requires that the instruction pipeline is filled as often as possible and gives rise to the need in superscalar architectures for significant amounts of CPU cache. It also makes hazard-avoiding techniques like branch prediction, speculative execution, register renaming, out-of-order execution and transactional memory crucial to maintaining high levels of performance. By attempting to predict which branch (or path) a conditional instruction will take, the CPU can minimize the number of times that the entire pipeline must wait until a conditional instruction is completed. Speculative execution often provides modest performance increases by executing portions of code that may not be needed after a conditional operation completes. Out-of-order execution somewhat rearranges the order in which instructions are executed to reduce delays due to data dependencies. Also in case of single instruction stream, multiple data stream—a case when a lot of data from the same type has to be processed—, modern processors can disable parts of the pipeline so that when a single instruction is executed many times, the CPU skips the fetch and decode phases and thus greatly increases performance on certain occasions, especially in highly monotonous program engines such as video creation software and photo processing.

In the case where a portion of the CPU is superscalar and part is not, the part which is not suffers a performance penalty due to scheduling stalls. The Intel P5 Pentium had two superscalar ALUs which could accept one instruction per clock cycle each, but its FPU could not accept one instruction per clock cycle. Thus the P5 was integer superscalar but not floating point superscalar. Intel's successor to the P5 architecture, P6, added superscalar capabilities to its floating point features, and therefore afforded a significant increase in floating point instruction performance.

Both simple pipelining and superscalar design increase a CPU's ILP by allowing a single processor to complete execution of instructions at rates surpassing one instruction per clock cycle. Most modern CPU designs are at least somewhat superscalar, and nearly all general purpose CPUs designed in the last decade are superscalar. In later years some of the emphasis in designing high-ILP computers has been moved out of the CPU's hardware and into its software interface, or ISA. The strategy of the very long instruction word (VLIW) causes some ILP to become implied directly by the software, reducing the amount of work the CPU must perform to boost ILP and thereby reducing the design's complexity.

Task-level parallelism

Another strategy of achieving performance is to execute multiple threads or processes in parallel. This area of research is known as parallel computing. In Flynn's taxonomy, this strategy is known as multiple instruction stream, multiple data stream (MIMD).

One technology used for this purpose was multiprocessing (MP). The initial flavor of this technology is known as symmetric multiprocessing (SMP), where a small number of CPUs share a coherent view of their memory system. In this scheme, each CPU has additional hardware to maintain a constantly up-to-date view of memory. By avoiding stale views of memory, the CPUs can cooperate on the same program and programs can migrate from one CPU to another. To increase the number of cooperating CPUs beyond a handful, schemes such as non-uniform memory access (NUMA) and directory-based coherence protocols were introduced in the 1990s. SMP systems are limited to a small number of CPUs while NUMA systems have been built with thousands of processors. Initially, multiprocessing was built using multiple discrete CPUs and boards to implement the interconnect between the processors. When the processors and their interconnect are all implemented on a single chip, the technology is known as chip-level multiprocessing (CMP) and the single chip as a multi-core processor.

It was later recognized that finer-grain parallelism existed with a single program. A single program might have several threads (or functions) that could be executed separately or in parallel. Some of the earliest examples of this technology implemented input/output processing such as direct memory access as a separate thread from the computation thread. A more general approach to this technology was introduced in the 1970s when systems were designed to run multiple computation threads in parallel. This technology is known as multi-threading (MT). This approach is considered more cost-effective than multiprocessing, as only a small number of components within a CPU is replicated to support MT as opposed to the entire CPU in the case of MP. In MT, the execution units and the memory system including the caches are shared among multiple threads. The downside of MT is that the hardware support for multithreading is more visible to software than that of MP and thus supervisor software like operating systems have to undergo larger changes to support MT. One type of MT that was implemented is known as temporal multithreading, where one thread is executed until it is stalled waiting for data to return from external memory. In this scheme, the CPU would then quickly context switch to another thread which is ready to run, the switch often done in one CPU clock cycle, such as the UltraSPARC T1. Another type of MT is simultaneous multithreading, where instructions from multiple threads are executed in parallel within one CPU clock cycle.

For several decades from the 1970s to early 2000s, the focus in designing high performance general purpose CPUs was largely on achieving high ILP through technologies such as pipelining, caches, superscalar execution, out-of-order execution, etc. This trend culminated in large, power-hungry CPUs such as the Intel Pentium 4. By the early 2000s, CPU designers were thwarted from achieving higher performance from ILP techniques due to the growing disparity between CPU operating frequencies and main memory operating frequencies as well as escalating CPU power dissipation owing to more esoteric ILP techniques.

CPU designers then borrowed ideas from commercial computing markets such as transaction processing, where the aggregate performance of multiple programs, also known as throughput computing, was more important than the performance of a single thread or process.

This reversal of emphasis is evidenced by the proliferation of dual and more core processor designs and notably, Intel's newer designs resembling its less superscalar P6 architecture. Late designs in several processor families exhibit CMP, including the x86-64 Opteron and Athlon 64 X2, the SPARC UltraSPARC T1, IBM POWER4 and POWER5, as well as several video game console CPUs like the Xbox 360's triple-core PowerPC design, and the PlayStation 3's 7-core Cell microprocessor.

Data parallelism

A less common but increasingly important paradigm of processors (and indeed, computing in general) deals with data parallelism. The processors discussed earlier are all referred to as some type of scalar device. As the name implies, vector processors deal with multiple pieces of data in the context of one instruction. This contrasts with scalar processors, which deal with one piece of data for every instruction. Using Flynn's taxonomy, these two schemes of dealing with data are generally referred to as single instruction stream, multiple data stream (SIMD) and single instruction stream, single data stream (SISD), respectively. The great utility in creating processors that deal with vectors of data lies in optimizing tasks that tend to require the same operation (for example, a sum or a dot product) to be performed on a large set of data. Some classic examples of these types of tasks include multimedia applications (images, video and sound), as well as many types of scientific and engineering tasks. Whereas a scalar processor must complete the entire process of fetching, decoding and executing each instruction and value in a set of data, a vector processor can perform a single operation on a comparatively large set of data with one instruction. This is only possible when the application tends to require many steps which apply one operation to a large set of data.

Most early vector processors, such as the Cray-1, were associated almost exclusively with scientific research and cryptography applications. However, as multimedia has largely shifted to digital media, the need for some form of SIMD in general-purpose processors has become significant. Shortly after inclusion of floating-point units started to become commonplace in general-purpose processors, specifications for and implementations of SIMD execution units also began to appear for general-purpose processors. Some of these early SIMD specifications - like HP's Multimedia Acceleration eXtensions (MAX) and Intel's MMX - were integer-only. This proved to be a significant impediment for some software developers, since many of the applications that benefit from SIMD primarily deal with floating-point numbers. Progressively, developers refined and remade these early designs into some of the common modern SIMD specifications, which are usually associated with one ISA. Some notable modern examples include Intel's SSE and the PowerPC-related AltiVec (also known as VMX).

1.3.7 Virtual CPUs

Cloud computing can involve subdividing CPU operation into **virtual central processing units** (**vCPU** s).

A host is the virtual equivalent of a physical machine, on which a virtual system is operating. When there are several physical machines operating in tandem and managed as a whole, the grouped computing and memory resources form a cluster. In some systems, it is possible to dynamically add and remove from a cluster. Resources available at a host and cluster level can be partitioned out into resources pools with fine granularity.

1.4 Performance

The *performance* or *speed* of a processor depends on, among many other factors, the clock rate (generally given in multiples of hertz) and the instructions per clock (IPC), which together are the factors for the instructions per second (IPS) that the CPU can perform. Many reported IPS values have represented "peak" execution rates on artificial instruction sequences with few branches, whereas realistic workloads consist of a mix of instructions and applications, some of which take longer to execute than others. The performance of the memory hierarchy also greatly affects processor performance, an issue barely considered in MIPS calculations. Because of these problems, various standardized tests, often called "benchmarks" for this purpose—such as SPECint—have been developed to attempt to measure the real effective performance in commonly used applications.

Processing performance of computers is increased by using multi-core processors, which essentially is plugging two or more individual processors (called *cores* in this sense) into one integrated circuit. Ideally, a dual core processor would be nearly twice as powerful as a single core processor. In practice, the performance gain is far smaller, only about 50%, due to imperfect software algorithms and implementation. Increasing the number of cores in a processor (i.e. dual-core, quad-core, etc.) increases the workload that can be handled. This means that the processor can now handle numerous asynchronous events, interrupts, etc. which can take a toll on the CPU when overwhelmed. These cores can be thought of as different floors in a processing plant, with each floor handling a different task. Sometimes, these cores will handle the same tasks as cores adjacent to them if a single core is not enough to handle the information.

Due to specific capabilities of modern CPUs, such as hyper-threading and uncore, which involve sharing of actual CPU resources while aiming at increased utilization, monitoring performance levels and hardware use gradually became a more complex task. As a response, some CPUs implement additional hardware logic that monitors actual use of various parts of a CPU and provides various counters accessible to software; an example is Intel's *Performance Counter Monitor* technology.

Chapter 2 History of general-purpose CPUs

The **history of general-purpose CPUs** is a continuation of the earlier history of computing hardware.

2.1 1950s: Early designs

A Vacuum tube module from early 700 series IBM computers

In the early 1950s, each computer design was unique. There were no upward-compatible machines or computer architectures with multiple, differing implementations. Programs written for one machine would run on no other kind, even other kinds from the same company. This was not a major drawback then because no large body of software had been developed to run on computers, so starting programming from scratch was not seen as a large barrier.

The design freedom of the time was very important, for designers were very constrained by the cost of electronics, and only starting to explore how a computer could best be organized. Some of the basic features introduced during this period included index registers (on the Ferranti Mark 1), a return address saving instruction (UNIVAC I), immediate operands (IBM 704), and detecting invalid operations (IBM 650).

By the end of the 1950s, commercial builders had developed factory-constructed, truck-deliverable computers. The most widely installed computer was the IBM 650, which used drum memory onto which programs were loaded using either paper punched tape or punched cards. Some very high-end machines also included core memory which provided higher speeds. Hard disks were also starting to grow popular.

A computer is an automatic abacus. The type of number system affects the way it works. In the early 1950s most computers were built for specific numerical processing tasks, and many machines used decimal numbers as their basic number system; that is, the mathematical functions of the machines worked in base-10 instead of base-2 as is common today. These were not merely binary coded decimal (BCD). Most machines had ten vacuum tubes per digit in each processor register. Some early Soviet computer designers implemented systems based on ternary logic; that is, a bit could have three states: +1, 0, or -1, corresponding to positive, zero, or negative voltage.

An early project for the U.S. Air Force, BINAC attempted to make a lightweight, simple computer by using binary arithmetic. It deeply impressed the industry.

As late as 1970, major computer languages were unable to standardize their numeric behavior because decimal computers had groups of users too large to alienate.

Even when designers used a binary system, they still had many odd ideas. Some used sign-magnitude arithmetic (-1 = 10001), or ones' complement (-1 = 11110), rather than modern two's complement arithmetic (-1 = 11111). Most computers used six-bit character sets, because they adequately encoded Hollerith punched cards. It was a major revelation to designers of this period to realize that the data word should be a multiple of the character size. They began to design computers with 12-, 24- and 36-bit data words (e.g., see the TX-2).

In this era, Grosch's law dominated computer design: computer cost increased as the square of its speed.

2.2 1960s: Computer revolution and CISC

One major problem with early computers was that a program for one would work on no others. Computer companies found that their customers had little reason to remain loyal to a given brand, as the next computer they bought would be incompatible anyway. At that point, the only concerns were usually price and performance.

In 1962, IBM tried a new approach to designing computers. The plan was to make a family of computers that could all run the same software, but with different performances, and at different prices. As users' needs grew, they could move up to larger computers, and still keep all of their investment in programs, data and storage media.

To do this, they designed one *reference computer* named *System/360* (S/360). This was a virtual computer, a reference instruction set, and abilities that all machines in the family would support. To provide different classes of machines, each computer in the family would use more or less hardware emulation, and more or less microprogram emulation, to create a machine able to run the full S/360 instruction set.

For instance, a low-end machine could include a very simple processor for low cost. However this would require the use of a larger microcode emulator to provide the rest of the instruction set, which would slow it down. A high-end machine would use a much more complex processor that could directly process more of the S/360 design, thus running a much simpler and faster emulator.

IBM chose consciously to make the reference instruction set quite complex, and very capable. Even though the computer was complex, its *control store* holding the microprogram would stay relatively small, and could be made with very fast memory. Another important effect was that one instruction could describe quite a complex sequence of operations. Thus the computers would generally have to fetch fewer instructions from the main memory, which could be made slower, smaller and less costly for a given mix of speed and price.

As the S/360 was to be a successor to both scientific machines like the 7090 and data processing machines like the 1401, it needed a design that could reasonably support all forms of processing. Hence the instruction set was designed to manipulate simple binary numbers, and text, scientific floating-point (similar to the numbers used in a calculator), and the binary coded decimal arithmetic needed by accounting systems.

Almost all following computers included these innovations in some form. This basic set of features is now called *complex instruction set computing* (CISC, pronounced "sisk"), a term not invented until many years later, when *reduced instruction set computing* (RISC) began to get market share.

In many CISCs, an instruction could access either registers or memory, usually in several different ways. This made the CISCs easier to program, because a programmer could remember only thirty to a hundred instructions, and a set of three to ten addressing modes rather than thousands of distinct instructions. This was called an *orthogonal instruction set* . The PDP-11 and Motorola 68000 architecture are examples of nearly orthogonal instruction sets.

There was also the *BUNCH* (Burroughs, UNIVAC, NCR, Control Data Corporation, and Honeywell) that competed against IBM at this time; however, IBM dominated the era with S/360.

The Burroughs Corporation (which later merged with Sperry/Univac to form Unisys) offered an alternative to S/360 with their Burroughs large systems B5000 series. In 1961, the B5000 had virtual memory, symmetric multiprocessing, a multiprogramming operating system (Master Control Program (MCP)), written in ALGOL 60, and the industry's first recursive-descent compilers as early as 1963.

2.3 Late 1960s–early 1970s: LSI and microprocessors

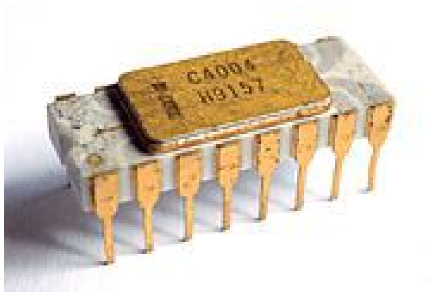
In the 1960s, the development of electronic calculators, electronic clocks, the Apollo guidance computer, and Minuteman missile, helped make integrated circuits economical and practical. In the late 1960s, the first calculator and clock chips began to show that very small computers might be possible with large-scale integration (LSI). This culminated in the invention of the microprocessor, a single-chip CPU. The Intel 4004, released in 1971, was the first commercial microprocessor. The origins of the 4004 date back to the "Busicom Project", which began at Japanese calculator company Busicom in April 1968, when engineer Masatoshi Shima was tasked with designing a special-purpose LSI chipset, along with his supervisor Tadashi Tanba, for use in the Busicom 141-PF desktop calculator with integrated printer. His initial design consisted of seven LSI chips, including a three-chip CPU. His design included arithmetic units (adders), multiplier units, registers, read-only memory, and a macro-instruction set to control a decimal computer system. Busicom then wanted a general-purpose LSI chipset, for not only desktop calculators, but also other equipment such as a teller machine, cash register and billing machine. Shima thus began work on a general-purpose LSI chipset in late 1968. Sharp engineer Tadashi Sasaki, who also became involved with its development, conceived of a single-chip microprocessor in 1968, when he discussed the concept at a brainstorming meeting that was held in Japan. Sasaki attributes the basic invention to break the calculator chipset into four parts with ROM (4001), RAM (4002), shift registers (4003) and CPU (4004) to an unnamed woman, a software engineering researcher from Nara Women's College, who was present at the meeting. Sasaki then had his first meeting with Robert Noyce from Intel in 1968, and presented the woman's four-division chipset concept to Intel and Busicom.

Busicom approached the American company Intel for manufacturing help in 1969. Intel, which was more of a memory company back then, had facilities to manufacture the high density silicon gate MOS chip Busicom required. Shima went to Intel in June 1969 to present his design proposal. Due to Intel lacking logic engineers to understand the logic schematics or circuit engineers to convert them, Intel asked Shima to simplify the logic. Intel wanted a single-chip CPU design, influenced by Sharp's Tadashi Sasaki who presented the concept to Busicom and Intel in 1968. The single-chip microprocessor design was then formulated by Intel's Marcian "Ted" Hoff in 1969, simplifying Shima's initial design down to four chips, including a single-chip CPU. Due to Hoff's formulation lacking key details, Shima came up with his own ideas to find solutions for its implementation. Shima was responsible for adding a 10-bit static shift register to make it useful as a printer's buffer and keyboard interface, many improvements in the instruction set, making the RAM organization suitable for a calculator, the memory address information transfer, the key program in an area of performance and program capacity, the functional specification, decimal computer idea, software, desktop calculator logic, real-time I/O control, and data exchange instruction between the accumulator and general purpose register. Hoff and Shima eventually realized the 4-bit microprocessor concept together, with the help of Intel's Stanley Mazor to interpret the ideas of Shima and Hoff. The specifications of the four chips were developed over a period of a few months in 1969, between an Intel team led by Hoff and a Busicom team led by Shima.

In late 1969, Shima returned to Japan. After that, Intel had done no further work on the project until early 1970. Shima returned to Intel in early 1970, and found that no further work had been done on the 4004 since he left, and that Hoff had moved on to other projects. Only a week before Shima had returned to Intel, Federico Faggin had joined Intel and become the project leader. After Shima explained the project to Faggin, they worked together to design the 4004. Thus, the chief designers of the chip were Faggin who created the design methodology and the silicon-based chip design, Hoff who formulated the architecture before moving on to other projects, and Shima who produced the initial Busicom design and then assisted in the development of the final Intel design. The 4004 was first introduced in Japan, as the microprocessor for the Busicom 141-PF calculator, in March 1971. In North America, the first public mention of the 4004 was an advertisement in the November 15, 1971 edition of *Electronic News* .

NEC released the µPD707 and µPD708, a two-chip 4-bit CPU, in 1971. They were followed by NEC's first single-chip microprocessor, the µPD700, in April 1972. It was a prototype for the µCOM-4 (µPD751), released in April 1973, combining the µPD707 and µPD708 into a single microprocessor. In 1973, Toshiba released the TLCS-12, the first 12-bit microprocessor.

2.4 1970s: Microprocessor revolution



Intel 4004 microprocessor.

The first commercial microprocessor, the binary coded decimal (BCD) based Intel 4004, was released by Busicom and Intel in 1971. In March 1972, Intel introduced a microprocessor with an 8-bit architecture, the 8008, an integrated pMOS logic re-implementation of the transistor–transistor logic (TTL) based Datapoint 2200 CPU. 4004 designers Federico and Masatoshi Shima went on to design its successor, the Intel 8080, released in 1974. The 8080 was the basis for the 8086, which is a direct ancestor to today's ubiquitous x86 family (including Pentium and Core i7). Every instruction of the 8080 has a direct equivalent in the large x86 instruction set, although the opcode values are different in the latter.

By the mid-1970s, the use of integrated circuits in computers was common. The decade was marked by market upheavals caused by the shrinking price of transistors.

It became possible to put an entire CPU on one printed circuit board. The result was that minicomputers, usually with 16-bit words, and 4K to 64K of memory, became common.

CISCs were believed to be the most powerful types of computers, because their microcode was small and could be stored in very high-speed memory. The CISC architecture also addressed the *semantic gap* as it was then perceived. This was a defined distance between the machine language, and the higher level programming languages used to program a machine. It was felt that compilers could do a better job with a richer instruction set.

Custom CISCs were commonly constructed using *bit slice* computer logic such as the AMD 2900 chips, with custom microcode. A bit slice component is a piece of an arithmetic logic unit (ALU), register file or microsequencer. Most bit-slice integrated circuits were 4-bits wide.

By the early 1970s, the PDP-11 was developed, arguably the most advanced small computer of its day. Almost immediately, wider-word CISCs were introduced, the 32-bit VAX and 36-bit PDP-10.

Intel soon developed a slightly more mini computer-like microprocessor, the 8080, largely based on customer feedback on the limited 8008. Much like the 8008, it was used for applications such as terminals, printers, cash registers and industrial robots. However, the more able 8080 also became the original target CPU for an early de facto standard personal computer operating system called CP/M and was used for such demanding control tasks as cruise missiles, and many other uses. The 8080 became one of the first really widespread microprocessors.

IBM continued to make large, fast computers. However, the definition of large and fast now meant more than a megabyte of RAM, clock speeds near one megahertz, and tens of megabytes of disk drives.

IBM's System 370 was a version of the 360 tweaked to run virtual computing environments. The virtual computer was developed to reduce the chances of an unrecoverable software failure.

The Burroughs large systems (B5000, B6000, B7000) series reached its largest market share. It was a stack computer which OS was programmed in a dialect of Algol.

All these different developments competed for market share.

The first single-chip 16-bit microprocessor was introduced in 1975. Panafacom, a conglomerate formed by Japanese companies Fujitsu, Fuji Electric, and Matsushita, introduced the MN1610, a commercial 16-bit microprocessor. According to Fujitsu, it was "the world's first 16-bit microcomputer on a single chip".

2.5 Early 1980s–1990s: Lessons of RISC

In the early 1980s, researchers at UC Berkeley and IBM both discovered that most computer language compilers and interpreters used only a small subset of the instructions of complex instruction set computing (CISC). Much of the power of the CPU was being ignored in real-world use. They realized that by making the computer simpler and less orthogonal, they could make it faster and less costly at the same time.

At the same time, CPU calculation became faster in relation to the time for needed memory accesses. Designers also experimented with using large sets of internal registers. The goal was to cache intermediate results in the registers under the control of the compiler. This also reduced the number of addressing modes and orthogonality.

The computer designs based on this theory were called reduced instruction set computing (RISC). RISCs usually had larger numbers of registers, accessed by simpler instructions, with a few instructions specifically to load and store data to memory. The result was a very simple core CPU running at very high speed, supporting the sorts of operations the compilers were using anyway.

A common variant on the RISC design employs the Harvard architecture, versus Von Neumann architecture or stored program architecture common to most other designs. In a Harvard Architecture machine, the program and data occupy separate memory devices and can be accessed simultaneously. In Von Neumann machines, the data and programs are mixed in one memory device, requiring sequential accessing which produces the so-called *Von Neumann bottleneck* .

One downside to the RISC design was that the programs that run on them tend to be larger. This is because compilers must generate longer sequences of the simpler instructions to perform the same results. Since these instructions must be loaded from memory anyway, the larger code offsets some of the RISC design's fast memory handling.

In the early 1990s, engineers at Japan's Hitachi found ways to compress the reduced instruction sets so they fit in even smaller memory systems than CISCs. Such compression schemes were used for the instruction set of their SuperH series of microprocessors, introduced in 1992. The SuperH instruction set was later adapted for ARM architecture's *Thumb* instruction set. In applications that do not need to run older binary software, compressed RISCs are growing to dominate sales.

Another approach to RISCs was the minimal instruction set computer (MISC), *niladic* , or *zero-operand* instruction set. This approach realized that most space in an instruction was used to identify the operands of the instruction. These machines placed the operands on a push-down (last-in, first out) stack. The instruction set was supplemented with a few instructions to fetch and store memory. Most used simple caching to provide extremely fast RISC machines, with very compact code. Another benefit was that the interrupt latencies were very small, smaller than most CISC machines (a rare trait in RISC machines). The Burroughs large systems architecture used this approach. The B5000 was designed in 1961, long before the term *RISC* was invented. The architecture puts six 8-bit instructions in a 48-bit word, and was a precursor to *very long instruction word* (VLIW) design (see below: 1990 to today).

The Burroughs architecture was one of the inspirations for Charles H. Moore's programming language Forth, which in turn inspired his later MISC chip designs. For example, his f20 cores had 31 5-bit instructions, which fit four to a 20-bit word.

RISC chips now dominate the market for 32-bit embedded systems. Smaller RISC chips are even growing common in the cost-sensitive 8-bit embedded-system market. The main market for RISC CPUs has been systems that need low power or small size.

Even some CISC processors (based on architectures that were created before RISC grew dominant), such as newer x86 processors, translate instructions internally into a RISC-like instruction set.

These numbers may surprise many, because the *market* is perceived as desktop computers. x86 designs dominate desktop and notebook computer sales, but such computers are only a tiny fraction of the computers now sold. Most people in industrialised countries own more computers in embedded systems in their car and house, than on their desks.

2.6 Mid-to-late 1980s: Exploiting instruction level parallelism

In the mid-to-late 1980s, designers began using a technique termed *instruction pipelining* , in which the processor works on multiple instructions in different stages of completion. For example, the processor can retrieve the operands for the next instruction while calculating the result of the current one. Modern CPUs may use over a dozen such stages. (Pipelining was originally developed in the late 1950s by International Business Machines (IBM) on their 7030 (Stretch) mainframe computer.) Minimal instruction set computers (MISC) can execute instructions in one cycle with no need for pipelining.

A similar idea, introduced only a few years later, was to execute multiple instructions in parallel on separate arithmetic logic units (ALUs). Instead of operating on only one instruction at a time, the CPU will look for several similar instructions that do not depend on each other, and execute them in parallel. This approach is called superscalar processor design.

Such methods are limited by the degree of instruction level parallelism (ILP), the number of non-dependent instructions in the program code. Some programs can run very well on superscalar processors due to their inherent high ILP, notably graphics. However, more general problems have far less ILP, thus lowering the possible speedups from these methods.

Branching is one major culprit. For example, a program may add two numbers and branch to a different code segment if the number is bigger than a third number. In this case, even if the branch operation is sent to the second ALU for processing, it still must wait for the results from the addition. It thus runs no faster than if there was only one ALU. The most common solution for this type of problem is to use a type of branch prediction.

To further the efficiency of multiple functional units which are available in superscalar designs, operand register dependencies were found to be another limiting factor. To minimize these dependencies, out-of-order execution of instructions was introduced. In such a scheme, the instruction results which complete out-of-order must be re-ordered in program order by the processor for the program to be restartable after an exception. Out-of-order execution was the main advance of the computer industry during the 1990s.

A similar concept is speculative execution, where instructions from one direction of a branch (the predicted direction) are executed before the branch direction is known. When the branch direction is known, the predicted direction and the actual direction are compared. If the predicted direction was correct, the speculatively executed instructions and their results are kept; if it was incorrect, these instructions and their results are erased. Speculative execution, coupled with an accurate branch predictor, gives a large performance gain.

These advances, which were originally developed from research for RISC-style designs, allow modern CISC processors to execute twelve or more instructions per clock cycle, when traditional CISC designs could take twelve or more cycles to execute one instruction.

The resulting instruction scheduling logic of these processors is large, complex and difficult to verify. Further, higher complexity needs more transistors, raising power consumption and heat. In these, RISC is superior because the instructions are simpler, have less interdependence, and make superscalar implementations easier. However, as Intel has demonstrated, the concepts can be applied to a complex instruction set computing (CISC) design, given enough time and money.

2.7 1990 to today: Looking forward

2.7.1 VLIW and EPIC

The instruction scheduling logic that makes a superscalar processor is boolean logic. In the early 1990s, a significant innovation was to realize that the coordination of a multi-ALU computer could be moved into the compiler, the software that translates a programmer's instructions into machine-level instructions.

This type of computer is called a *very long instruction word* (VLIW) computer.

Scheduling instructions statically in the compiler (versus scheduling dynamically in the processor) can reduce CPU complexity. This can improve performance, and reduce heat and cost.

Unfortunately, the compiler lacks accurate knowledge of runtime scheduling issues. Merely changing the CPU core frequency multiplier will have an effect on scheduling. Operation of the program, as determined by input data, will have major effects on scheduling. To overcome these severe problems, a VLIW system may be enhanced by adding the normal dynamic scheduling, losing some of the VLIW advantages.

Static scheduling in the compiler also assumes that dynamically generated code will be uncommon. Before the creation of Java and the Java virtual machine, this was true. It was reasonable to assume that slow compiles would only affect software developers. Now, with just-in-time compilation (JIT) virtual machines being used for many languages, slow code generation affects users also.

There were several unsuccessful attempts to commercialize VLIW. The basic problem is that a VLIW computer does not scale to different price and performance points, as a dynamically scheduled computer can. Another issue is that compiler design for VLIW computers is very difficult, and compilers, as of 2005, often emit suboptimal code for these platforms.

Also, VLIW computers optimise for throughput, not low latency, so they were unattractive to engineers designing controllers and other computers embedded in machinery. The embedded systems markets had often pioneered other computer improvements by providing a large market unconcerned about compatibility with older software.

In January 2000, Transmeta Corporation took the novel step of placing a compiler in the central processing unit, and making the compiler translate from a reference byte code (in their case, x86 instructions) to an internal VLIW instruction set. This method combines the hardware simplicity, low power and speed of VLIW RISC with the compact main memory system and software reverse-compatibility provided by popular CISC.

Intel's Itanium chip is based on what they call an explicitly parallel instruction computing (EPIC) design. This design supposedly provides the VLIW advantage of increased instruction throughput. However, it avoids some of the issues of scaling and complexity, by explicitly providing in each *bundle* of instructions information concerning their dependencies. This information is calculated by the compiler, as it would be in a VLIW design. The early versions are also backward-compatible with newer x86 software by means of an on-chip emulator mode. Integer performance was disappointing and despite improvements, sales in volume markets continue to be low.

2.7.2 Multi-threading

Current designs work best when the computer is running only one program. However, nearly all modern operating systems allow running multiple programs together. For the CPU to change over and do work on another program needs costly context switching. In contrast, multi-threaded CPUs can handle instructions from multiple programs at once.

To do this, such CPUs include several sets of registers. When a context switch occurs, the contents of the *working registers* are simply copied into one of a set of registers for this purpose.

Such designs often include thousands of registers instead of hundreds as in a typical design. On the downside, registers tend to be somewhat costly in chip space needed to implement them. This chip space might be used otherwise for some other purpose.

Intel calls this technology "hyperthreading" and offers two threads per core in its current Core i3, Core i7 and Core i9 Desktop lineup (as well as in its Core i3, Core i5 and Core i7 Mobile lineup), as well as offering up to four threads per core in high-end Xeon Phi processors.

2.7.3 Multi-core

Multi-core CPUs are typically multiple CPU cores on the same die, connected to each other via a shared L2 or L3 cache, an on-die bus, or an on-die crossbar switch. All the CPU cores on the die share interconnect components with which to interface to other processors and the rest of the system. These components may include a front side bus interface, a memory controller to interface with dynamic random access memory (DRAM), a cache coherent link to other processors, and a non-coherent link to the southbridge and I/O devices. The terms *multi-core* and *microprocessor unit* (MPU) have come into general use for one die having multiple CPU cores.

Intelligent RAM

One way to work around the Von Neumann bottleneck is to mix a processor and DRAM all on one chip.

- The Berkeley IRAM Project
- eDRAM
- Computational RAM
- Memristor

2.7.4 Reconfigurable logic

Another track of development is to combine reconfigurable logic with a general-purpose CPU. In this scheme, a special computer language compiles fast-running subroutines into a bit-mask to configure the logic. Slower, or less-critical parts of the program can be run by sharing their time on the CPU. This process allows creating devices such as software radios, by using digital signal processing to perform functions usually performed by analog electronics.

2.7.5 Open source processors

As the lines between hardware and software increasingly blur due to progress in design methodology and availability of chips such as field-programmable gate arrays (FPGA) and cheaper production processes, even open source hardware has begun to appear. Loosely knit communities like OpenCores and RISC-V have recently announced fully open CPU architectures such as the OpenRISC which can be readily implemented on FPGAs or in custom produced chips, by anyone, with no license fees, and even established processor makers like Sun Microsystems have released processor designs (e.g., OpenSPARC) under open-source licenses.

2.7.6 Asynchronous CPUs

Yet another option is a *clockless* or *asynchronous CPU* . Unlike conventional processors, clockless processors have no central clock to coordinate the progress of data through the pipeline. Instead, stages of the CPU are coordinated using logic devices called *pipe line controls* or *FIFO sequencers* . Basically, the pipeline controller clocks the next stage of logic when the existing stage is complete. Thus, a central clock is unneeded.

Relative to clocked logic, it may be easier to implement high performance devices in asynchronous logic:

- In a clocked CPU, no component can run faster than the clock rate. In a clockless CPU, components can run at different speeds.
- In a clocked CPU, the clock can go no faster than the worst-case performance of the slowest stage. In a clockless CPU, when a stage finishes faster than normal, the next stage can immediately take the results rather than waiting for the next clock tick. A stage might finish faster than normal because of the type of data inputs (e.g., multiplication can be very fast if it occurs by 0 or 1), or because it is running at a higher voltage or lower temperature than normal.

Asynchronous logic proponents believe these abilities would have these benefits:

- lower power dissipation for a given performance
- highest possible execution speeds

The biggest disadvantage of the clockless CPU is that most CPU design tools assume a clocked CPU (a synchronous circuit), so making a clockless CPU (designing an asynchronous circuit) involves modifying the design tools to handle clockless logic and doing extra testing to ensure the design avoids metastability problems.

Even so, several asynchronous CPUs have been built, including

- the ORDVAC and the identical ILLIAC I (1951)
- the ILLIAC II (1962), then the fastest computer on Earth
- The Caltech Asynchronous Microprocessor, the world-first asynchronous microprocessor (1988)
- the ARM-implementing AMULET (1993 and 2000)
- the asynchronous implementation of MIPS Technologies R3000, named MiniMIPS (1998)
- the SEAforth multi-core processor from Charles H. Moore

2.7.7 Optical communication

One promising option is to eliminate the front side bus. Modern vertical laser diodes enable this change. In theory, an optical computer's components could directly connect through a holographic or phased open-air switching system. This would provide a large increase in effective speed and design flexibility, and a large reduction in cost. Since a computer's connectors are also its most likely failure points, a busless system may be more reliable.

Further, as of 2010, modern processors use 64- or 128-bit logic. Optical wavelength superposition could allow data lanes and logic many orders of magnitude higher than electronics, with no added space or copper wires.

2.7.8 Optical processors

Another long-term option is to use light instead of electricity for digital logic. In theory, this could run about 30% faster and use less power, and allow a direct interface with quantum computing devices.

The main problems with this approach are that, for the foreseeable future, electronic computing elements are faster, smaller, cheaper, and more reliable. Such elements are already smaller than some wavelengths of light. Thus, even waveguide-based optical logic may be uneconomic relative to electronic logic. As of 2016, most development effort is for electronic circuitry.

2.7.9 Ionic processors

Early experimental work has been done on using ion based chemical reactions instead of electronic or photonic actions to implement elements of a logic processor.

2.7.10 Belt machine architecture

Relative to conventional register machine or stack machine architecture, yet similar to Intel's Itanium architecture, a temporal register addressing scheme has been proposed by Ivan Godard and company that is intended to greatly reduce the complexity of CPU hardware (specifically the number of internal registers and the resulting huge multiplexer trees). While somewhat harder to read and debug than general-purpose register names, it aids understanding to view the belt as a moving *conveyor belt* where the oldest values *drop off* the belt and vanish. It is implemented in the Mill architecture.

2.8 Timeline of events

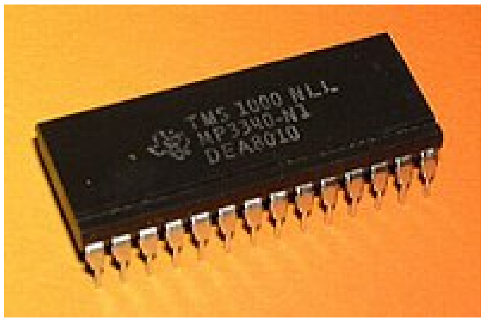
- 1964. IBM release the 32-bit IBM System/360 with memory protection.
- 1968. Busicom's Masatoshi Shima begins designing three-chip CPU that would later evolve into the single-chip Intel 4004 microprocessor.
- 1968. Sharp engineer Tadashi Sasaki conceives single-chip microprocessor, which he discusses with Busicom and Intel.
- 1969. Intel 4004's initial design led by Intel's Ted Hoff and Busicom's Masatoshi Shima.
- 1970. Intel 4004's design completed by Intel's Federico Faggin and Busicom's Masatoshi Shima.

- 1971. Busicom and Intel release the 4-bit Intel 4004, the first commercial microprocessor.
- 1971. NEC release the µPD707 and µPD708, a two-chip 4-bit CPU.
- 1972. NEC release single-chip 4-bit microprocessor, µPD700.
- 1973. NEC release 4-bit µCOM-4 (µPD751), combining the µPD707 and µPD708 into a single microprocessor.
- 1973. Toshiba release TLCS-12, the first 12-bit microprocessor.
- 1974. Intel release the Intel 8080, an 8-bit microprocessor, designed by Federico Faggin and Masatoshi Shima.
- 1975. MOS Technology release the 8-bit MOS Technology 6502, the first integrated processor to have an affordable price of $25 when the 6800 rival was $175.
- 1975. Panafacom introduce the MN1610, the first commercial 16-bit single-chip microprocessor.
- 1976. Zilog introduce the 8-bit Zilog Z80, designed by Federico Faggin and Masatoshi Shima.
- 1977. First 32-bit VAX sold, a VAX-11/780.
- 1978. Intel introduces the Intel 8086 and Intel 8088, the first x86 chips.
- 1978. Fujitsu releases the MB8843 microprocessor.
- 1979. Zilog release the Zilog Z8000, a 16-bit microprocessor, designed by Federico Faggin and Masatoshi Shima.
- 1979. Motorola introduce the Motorola 68000, a 16/32-bit microprocessor.
- 1981. Stanford MIPS introduced, one of the first reduced instruction set computing (RISC) designs.
- 1982. Intel introduces the Intel 80286, which was the first Intel processor that could run all the software written for its predecessors, the 8086 and 8088.

- 1984. Motorola introduces the Motorola 68020+68851, which enabled 32-bit instruction set and virtualization.
- 1985. Intel introduces the Intel 80386, which adds a 32-bit instruction set to the x86 microarchitecture.
- 1985. ARM architecture introduced.
- 1989. Intel introduces the Intel 80486.
- 1992. Hitachi introduces SuperH architecture, which provides the basis for ARM's Thumb instruction set.
- 1993. Intel launches the original Pentium microprocessor, the first processor with a x86 superscalar microarchitecture.
- 1994. ARM's Thumb instruction set introduced, based on Hitachi's SuperH instruction set.
- 1995. Intel introduces the Pentium Pro which becomes the foundation for the Pentium II, Pentium III, Pentium M and Intel Core architectures.
- 2000. AMD announced x86-64 extension to the x86 microarchitecture.
- 2000. AMD hits 1 GHz with its Athlon microprocessor.
- 2000. Analog Devices introduces the Blackfin architecture.
- 2002. Intel released a Pentium 4 with hyper-threading, the first modern desktop processor to implement simultaneous multithreading (SMT).
- 2003. AMD released the Athlon 64, the first 64-bit consumer CPU.
- 2003. Intel introduced the Pentium M, a low power mobile derivative of the Pentium Pro architecture.
- 2005. AMD announced the Athlon 64 X2, their first x86 dual-core processor.
- 2006. Intel introduces the Core line of CPUs based on a modified Pentium M design.
- 2008. About ten billion CPUs were produced.
- 2010. Intel introduced Core i3, i5 i7 processors.

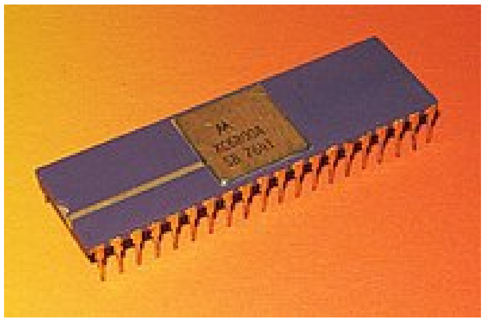
- 2011. AMD announced the world's first 8-core CPU for desktop PCs.
- 2017. AMD announced Ryzen processors based on Zen architecture.
- 2017. Intel introduced Coffee Lake, which increases core counts by two on Core i3, Core i5, and Core i7 processors while removing hyperthreading for Core i3. The Core i7 now has six hyperthreaded cores, which was once only available to high-end desktop computers.

Chapter 3 Microprocessor



Texas Instruments TMS1000

Intel 4004



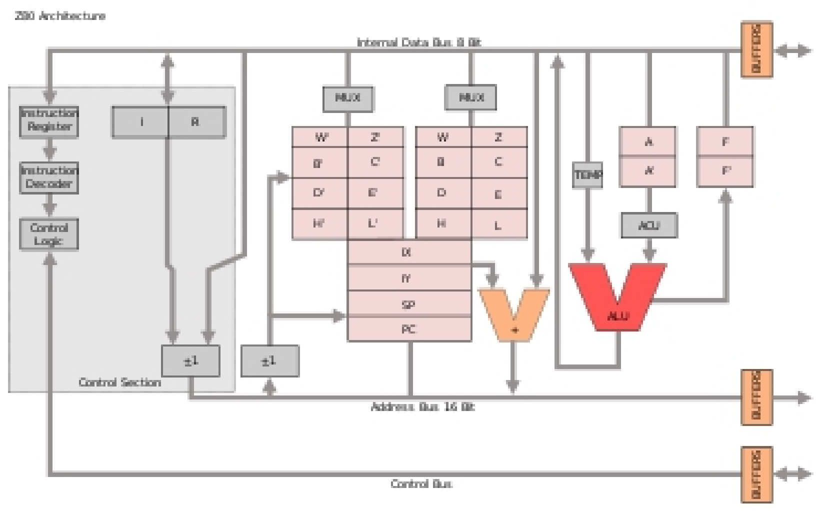
Motorola 6800

A **microprocessor** is a computer processor that incorporates the functions of a central processing unit on a single integrated circuit (IC), or at most a few integrated circuits. The microprocessor is a multipurpose, clock driven, register based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory, and provides results as output. Microprocessors contain both combinational logic and sequential digital logic. Microprocessors operate on numbers and symbols represented in the binary number system.

The integration of a whole CPU onto a single chip or on a few chips greatly reduced the cost of processing power, increasing efficiency. Integrated circuit processors are produced in large numbers by highly automated processes, resulting in a low per-unit cost. Single-chip processors increase reliability because there are many fewer electrical connections that could fail. As microprocessor designs improve, the cost of manufacturing a chip (with smaller components built on a semiconductor chip the same size) generally stays the same according to Rock's law.

Before microprocessors, small computers had been built using racks of circuit boards with many medium- and small-scale integrated circuits. Microprocessors combined this into one or a few large-scale ICs. Continued increases in microprocessor capacity have since rendered other forms of computers almost completely obsolete (see history of computing hardware), with one or more microprocessors used in everything from the smallest embedded systems and handheld devices to the largest mainframes and supercomputers.

3.1 Structure



A block diagram of the architecture of the Z80 microprocessor, showing the arithmetic and logic section, register file, control logic section, and buffers to external address and data lines

The internal arrangement of a **microprocessor** varies depending on the age of the design and the intended purposes of the microprocessor. The complexity of an integrated circuit (IC) is bounded by physical limitations on the number of transistors that can be put onto one chip, the number of package terminations that can connect the processor to other parts of the system, the number of interconnections it is possible to make on the chip, and the heat that the chip can dissipate. Advancing technology makes more complex and powerful chips feasible to manufacture.

A minimal hypothetical microprocessor might include only an arithmetic logic unit (ALU) and a control logic section. The ALU performs operations such as addition, subtraction, and operations such as AND or OR. Each operation of the ALU sets one or more flags in a status register, which indicate the results of the last operation (zero value, negative number, overflow, or others). The control logic retrieves instruction codes from memory and initiates the sequence of operations required for the ALU to carry out the instruction. A single operation code might affect many individual data paths, registers, and other elements of the processor.

As integrated circuit technology advanced, it was feasible to manufacture more and more complex processors on a single chip. The size of data objects became larger; allowing more transistors on a chip allowed word sizes to increase from 4- and 8-bit words up to today's 64-bit words. Additional features were added to the processor architecture; more on-chip registers sped up programs, and complex instructions could be used to make more compact programs. Floating-point arithmetic, for example, was often not available on 8-bit microprocessors, but had to be carried out in software. Integration of the floating point unit first as a separate integrated circuit and then as part of the same microprocessor chip sped up floating point calculations.

Occasionally, physical limitations of integrated circuits made such practices as a bit slice approach necessary. Instead of processing all of a long word on one integrated circuit, multiple circuits in parallel processed subsets of each data word. While this required extra logic to handle, for example, carry and overflow within each slice, the result was a system that could handle, for example, 32-bit words using integrated circuits with a capacity for only four bits each.

The ability to put large numbers of transistors on one chip makes it feasible to integrate memory on the same die as the processor. This CPU cache has the advantage of faster access than off-chip memory and increases the processing speed of the system for many applications. Processor clock frequency has increased more rapidly than external memory speed, except in the recent past, so cache memory is necessary if the processor is not delayed by slower external memory.

3.1.1 Special-purpose designs

A microprocessor is a general-purpose entity. Several specialized processing devices have followed from the technology:

- A digital signal processor (DSP) is specialized for signal processing.
- Graphics processing units (GPUs) are processors designed primarily for realtime rendering of 3D images. They may be fixed function (as was more common in the 1990s), or support programmable shaders. With the continuing rise of GPGPU, GPUs are evolving into increasingly general-purpose stream processors (running compute shaders), while retaining hardware assist for rasterizing, but still differ from CPUs in that they are optimized for throughput over latency, and are not suitable for running application or OS code.
- Other specialized units exist for video processing and machine vision. (See: Hardware acceleration.)
- Microcontrollers integrate a microprocessor with peripheral devices in embedded systems. These tend to have different tradeoffs compared to CPUs.
- Systems on chip (SoCs) often integrate one or more microprocessor or microcontroller cores.

32-bit processors have more digital logic than narrower processors, so 32-bit (and wider) processors produce more digital noise and have higher static consumption than narrower processors. Reducing digital noise improves ADC conversion results. So, 8- or 16-bit processors can be better than 32-bit processors for system on a chip and microcontrollers that require extremely low-power electronics, or are part of a mixed-signal integrated circuit with noise-sensitive on-chip analog electronics such as high-resolution analog to digital converters, or both.

Nevertheless, trade-offs apply: running 32-bit arithmetic on an 8-bit chip could end up using more power, as the chip must execute software with multiple instructions. Modern microprocessors go into low power states when possible, and an 8-bit chip running 32-bit calculations would be active for more cycles. This creates a delicate balance between software, hardware and use patterns, and costs.

When manufactured in a similar process, 8-bit microprocessors use less power when operating and less power when sleeping than 32-bit microprocessors.

However, a 32-bit microprocessor may use less average power than an 8-bit microprocessor when the application requires certain operations such as floating-point math that take many more clock cycles on an 8-bit microprocessor than on a 32-bit microprocessor, so the 8-bit microprocessor spends more time in high-power operating mode.

3.2 Embedded applications

Thousands of items that were traditionally not computer-related include microprocessors. These include large and small household appliances, cars (and their accessory equipment units), car keys, tools and test instruments, toys, light switches/dimmers and electrical circuit breakers, smoke alarms, battery packs, and hi-fi audio/visual components (from DVD players to phonograph turntables). Such products as cellular telephones, DVD video system and HDTV broadcast systems fundamentally require consumer devices with powerful, low-cost, microprocessors. Increasingly stringent pollution control standards effectively require automobile manufacturers to use microprocessor engine management systems to allow optimal control of emissions over the widely varying operating conditions of an automobile. Non-programmable controls would require complex, bulky, or costly implementation to achieve the results possible with a microprocessor.

A microprocessor control program (embedded software) can be easily tailored to different needs of a product line, allowing upgrades in performance with minimal redesign of the product. Different features can be implemented in different models of a product line at negligible production cost.

Microprocessor control of a system can provide control strategies that would be impractical to implement using electromechanical controls or purpose-built electronic controls. For example, an engine control system in an automobile can adjust ignition timing based on engine speed, load on the engine, ambient temperature, and any observed tendency for knocking—allowing an automobile to operate on a range of fuel grades.

3.3 History

The advent of low-cost computers on integrated circuits has transformed modern society. General-purpose microprocessors in personal computers are used for computation, text editing, multimedia display, and communication over the Internet. Many more microprocessors are part of embedded systems, providing digital control over myriad objects from appliances to automobiles to cellular phones and industrial process control.

The first use of the term "microprocessor" is attributed to Viatron Computer Systems describing the custom integrated circuit used in their System 21 small computer system announced in 1968.

By the late 1960s, designers were striving to integrate the central processing unit (CPU) functions of a computer onto a handful of very-large-scale integration metal-oxide semiconductor chips, called microprocessor unit (MPU) chipsets. Building on an earlier Busicom design from 1969, Intel introduced the first commercial microprocessor, the 4-bit Intel 4004, in 1971, followed by its 8-bit microprocessor 8008 in 1972. In 1969, Lee Boysel, based on b8-bit arithmetic logic units (3800/3804) he designed earlier at Fairchild, created the Four-Phase Systems Inc. AL-1, an 8-bit CPU slice that was expandable to 32-bits. In 1970, Steve Geller and Ray Holt of Garrett AiResearch designed the MP944 chipset to implement the F-14A Central Air Data Computer on six metal-gate chips fabricated by AMI.

The first microprocessors emerged in the early 1970s and were used for electronic calculators, using binary-coded decimal (BCD) arithmetic on 4-bit words. Other embedded uses of 4-bit and 8-bit microprocessors, such as terminals, printers, various kinds of automation etc., followed soon after. Affordable 8-bit microprocessors with 16-bit addressing also led to the first general-purpose microcomputers from the mid-1970s on.

Since the early 1970s, the increase in capacity of microprocessors has followed Moore's law; this originally suggested that the number of components that can be fitted onto a chip doubles every year. With present technology, it is actually every two years, and as a result Moore later changed the period to two years.

3.3.1 First projects

Three projects delivered a microprocessor at about the same time: Garrett AiResearch's Central Air Data Computer (CADC), Texas Instruments' TMS 1000 (September 1971) and Intel's 4004 (November 1971, based on an earlier 1969 Busicom design). Arguably, Four-Phase Systems AL1 microprocessor was also delivered in 1969.

CADC

In 1968, Garrett AiResearch (who employed designers Ray Holt and Steve Geller) was invited to produce a digital computer to compete with electromechanical systems then under development for the main flight control computer in the US Navy's new F-14 Tomcat fighter. The design was complete by 1970, and used a MOS-based chipset as the core CPU. The design was significantly (approximately 20 times) smaller and much more reliable than the mechanical systems it competed against, and was used in all of the early Tomcat models. This system contained "a 20-bit, pipelined, parallel multi-microprocessor". The Navy refused to allow publication of the design until 1997. For this reason the CADC, and the MP944 chipset it used, are fairly unknown. Ray Holt's autobiographical story of this design and development is presented in the book: The Accidental Engineer.

Ray Holt graduated from California Polytechnic University in 1968, and began his computer design career with the CADC. From its inception, it was shrouded in secrecy until 1998 when at Holt's request, the US Navy allowed the documents into the public domain. Since then people have debated whether this was the first microprocessor. Holt has stated that no one has compared this microprocessor with those that came later. According to Parab et al. (2007),

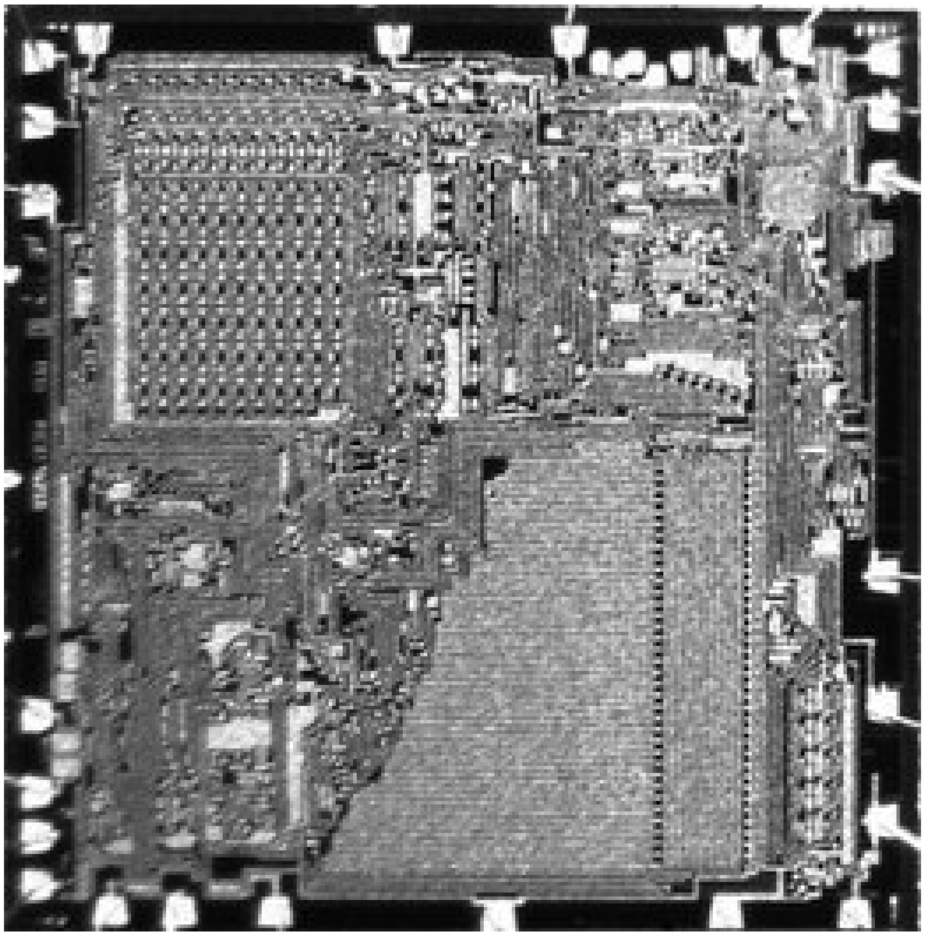
The scientific papers and literature published around 1971 reveal that the MP944 digital processor used for the F-14 Tomcat aircraft of the US Navy qualifies as the first microprocessor. Although interesting, it was not a single-chip processor, as was not the Intel 4004 – they both were more like a set of parallel building blocks you could use to make a general-purpose form. It contains a CPU, RAM, ROM, and two other support chips like the Intel 4004. It was made from the same P-channel technology, operated at military specifications and had larger chips – an excellent computer engineering design by any standards. Its design indicates a major advance over Intel, and two year earlier. It actually worked and was flying in the F-14 when the Intel 4004 was announced. It indicates that today's industry theme of converging DSP-microcontroller architectures was started in 1971.

This convergence of DSP and microcontroller architectures is known as a digital signal controller.

Four-Phase Systems AL1 (1969)

The Four-Phase Systems AL1 was an 8-bit bit slice chip containing eight registers and an ALU. It was designed by Lee Boysel in 1969. At the time, it formed part of a nine-chip, 24-bit CPU with three AL1s, but it was later called a microprocessor when, in response to 1990s litigation by Texas Instruments, a demonstration system was constructed where a single AL1 formed part of a courtroom demonstration computer system, together with RAM, ROM, and an input-output device.

Pico/General Instrument



The PICO1/GI250 chip introduced in 1971: It was designed by Pico Electronics (Glenrothes, Scotland) and manufactured by General Instrument of Hicksville NY.

In 1971, Pico Electronics and General Instrument (GI) introduced their first collaboration in ICs, a complete single chip calculator IC for the Monroe/Litton Royal Digital III calculator. This chip could also arguably lay claim to be one of the first microprocessors or microcontrollers having ROM, RAM and a RISC instruction set on-chip. The layout for the four layers of the PMOS process was hand drawn at x500 scale on mylar film, a significant task at the time given the complexity of the chip.

Pico was a spinout by five GI design engineers whose vision was to create single chip calculator ICs. They had significant previous design experience on multiple calculator chipsets with both GI and Marconi-Elliott. The key team members had originally been tasked by Elliott Automation to create an 8-bit computer in MOS and had helped establish a MOS Research Laboratory in Glenrothes, Scotland in 1967.

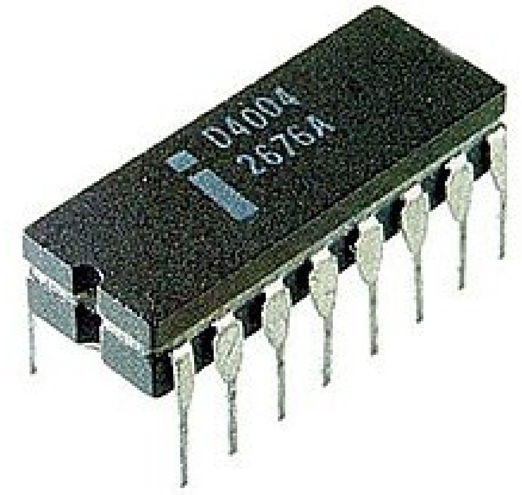
Calculators were becoming the largest single market for semiconductors so Pico and GI went on to have significant success in this burgeoning market. GI continued to innovate in microprocessors and microcontrollers with products including the CP1600, IOB1680 and PIC1650. In 1987, the GI Microelectronics business was spun out into the Microchip PIC microcontroller business.

Intel 4004 (1971)

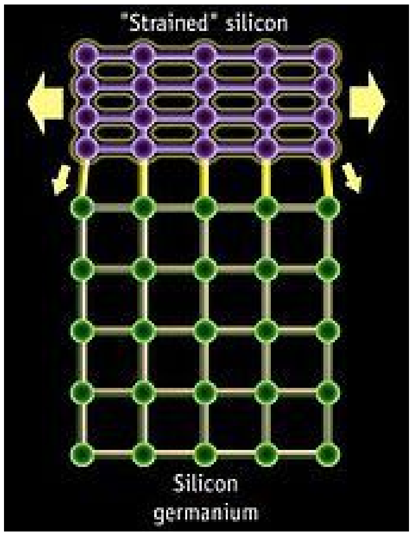
The 4004 with cover removed (left) and as actually used (right)

The Intel 4004 is generally regarded as the first commercially available microprocessor, and cost US$60 (equivalent to $378.78 in 2019). The first known advertisement for the 4004 is dated November 15, 1971 and appeared in *Electronic News* . The microprocessor was designed by a team consisting of Italian engineer Federico Faggin, American engineers Marcian Hoff and Stanley Mazor, and Japanese engineer Masatoshi Shima.

The project that produced the 4004 originated in 1969, when Busicom, a Japanese calculator manufacturer, asked Intel to build a chipset for high-performance desktop calculators. Busicom's original design called for a programmable chip set consisting of seven different chips. Three of the chips were to make a special-purpose CPU with its program stored in ROM and its data stored in shift register read-write memory. Ted Hoff, the Intel engineer assigned to evaluate the project, believed the Busicom design could be simplified by using dynamic RAM storage for data, rather than shift register memory, and a more traditional general-purpose CPU architecture. Hoff came up with a four-chip architectural proposal: a ROM chip for storing the programs, a dynamic RAM chip for storing data, a simple I/O device and a 4-bit central processing unit (CPU). Although not a chip designer, he felt the CPU could be integrated into a single chip, but as he lacked the technical know-how the idea remained just a wish for the time being.



First microprocessor by Intel, the 4004.



Silicon and germanium alloy for microprocessors

While the architecture and specifications of the MCS-4 came from the interaction of Hoff with Stanley Mazor, a software engineer reporting to him, and with Busicom engineer Masatoshi Shima, during 1969, Mazor and Hoff moved on to other projects. In April 1970, Intel hired Italian engineer Federico Faggin as project leader, a move that ultimately made the single-chip CPU final design a reality (Shima meanwhile designed the Busicom calculator firmware and assisted Faggin during the first six months of the implementation).

Faggin, who originally developed the silicon gate technology (SGT) in 1968 at Fairchild Semiconductor and designed the world’s first commercial integrated circuit using SGT, the Fairchild 3708, had the correct background to lead the project into what would become the first commercial general purpose microprocessor. Since SGT was his very own invention, Faggin also used it to create his new methodology for random logic design that made it possible to implement a single-chip CPU with the proper speed, power dissipation and cost. The manager of Intel's MOS Design Department was Leslie L. Vadász at the time of the MCS-4 development but Vadász's attention was completely focused on the mainstream business of semiconductor memories so he left the leadership and the management of the MCS-4 project to Faggin, who was ultimately responsible for leading the 4004 project to its realization. Production units of the 4004 were first delivered to Busicom in March 1971 and shipped to other customers in late 1971.

Gilbert Hyatt

Gilbert Hyatt was awarded a patent claiming an invention pre-dating both TI and Intel, describing a "microcontroller". The patent was later invalidated, but not before substantial royalties were paid out.

3.3.2 8-bit designs

The Intel 4004 was followed in 1972 by the Intel 8008, the world's first 8-bit microprocessor. The 8008 was not, however, an extension of the 4004 design, but instead the culmination of a separate design project at Intel, arising from a contract with Computer Terminals Corporation, of San Antonio TX, for a chip for a terminal they were designing, the Datapoint 2200—fundamental aspects of the design came not from Intel but from CTC. In 1968, CTC's Vic Poor and Harry Pyle developed the original design for the instruction set and operation of the processor. In 1969, CTC contracted two companies, Intel and Texas Instruments, to make a single-chip implementation, known as the CTC 1201. In late 1970 or early 1971, TI dropped out being unable to make a reliable part. In 1970, with Intel yet to deliver the part, CTC opted to use their own implementation in the Datapoint 2200, using traditional TTL logic instead (thus the first machine to run "8008 code" was not in fact a microprocessor at all and was delivered a year earlier). Intel's version of the 1201 microprocessor arrived in late 1971, but was too late, slow, and required a number of additional support chips. CTC had no interest in using it. CTC had originally contracted Intel for the chip, and would have owed them US$50,000 (equivalent to $315,653 in 2019) for their design work. To avoid paying for a chip they did not want (and could not use), CTC released Intel from their contract and allowed them free use of the design. Intel marketed it as the 8008 in April, 1972, as the world's first 8-bit microprocessor. It was the basis for the famous "Mark-8" computer kit advertised in the magazine *Radio-Electronics* in 1974. This processor had an 8-bit data bus and a 14-bit address bus.

The 8008 was the precursor to the successful Intel 8080 (1974), which offered improved performance over the 8008 and required fewer support chips. Federico Faggin conceived and designed it using high voltage N channel MOS. The Zilog Z80 (1976) was also a Faggin design, using low voltage N channel with depletion load and derivative Intel 8-bit processors: all designed with the methodology Faggin created for the 4004. Motorola released the competing 6800 in August 1974, and the similar MOS Technology 6502 in 1975 (both designed largely by the same people). The 6502 family rivaled the Z80 in popularity during the 1980s.

A low overall cost, small packaging, simple computer bus requirements, and sometimes the integration of extra circuitry (e.g. the Z80's built-in memory refresh circuitry) allowed the home computer "revolution" to accelerate sharply in the early 1980s. This delivered such inexpensive machines as the Sinclair ZX81, which sold for US$99 (equivalent to $278.41 in 2019). A variation of the 6502, the MOS Technology 6510 was used in the Commodore 64 and yet another variant, the 8502, powered the Commodore 128.

The Western Design Center, Inc (WDC) introduced the CMOS WDC 65C02 in 1982 and licensed the design to several firms. It was used as the CPU in the Apple IIe and IIc personal computers as well as in medical implantable grade pacemakers and defibrillators, automotive, industrial and consumer devices. WDC pioneered the licensing of microprocessor designs, later followed by ARM (32-bit) and other microprocessor intellectual property (IP) providers in the 1990s.

Motorola introduced the MC6809 in 1978. It was an ambitious and well thought-through 8-bit design that was source compatible with the 6800, and implemented using purely hard-wired logic (subsequent 16-bit microprocessors typically used microcode to some extent, as CISC design requirements were becoming too complex for pure hard-wired logic).

Another early 8-bit microprocessor was the Signetics 2650, which enjoyed a brief surge of interest due to its innovative and powerful instruction set architecture.

A seminal microprocessor in the world of spaceflight was RCA's RCA 1802 (aka CDP1802, RCA COSMAC) (introduced in 1976), which was used on board the *Galileo* probe to Jupiter (launched 1989, arrived 1995). RCA COSMAC was the first to implement CMOS technology. The CDP1802 was used because it could be run at very low power, and because a variant was available fabricated using a special production process, silicon on sapphire (SOS), which provided much better protection against cosmic radiation and electrostatic discharge than that of any other processor of the era. Thus, the SOS version of the 1802 was said to be the first radiation-hardened microprocessor.

The RCA 1802 had a static design, meaning that the clock frequency could be made arbitrarily low, or even stopped. This let the *Galileo* spacecraft use minimum electric power for long uneventful stretches of a voyage. Timers or sensors would awaken the processor in time for important tasks, such as navigation updates, attitude control, data acquisition, and radio communication. Current versions of the Western Design Center 65C02 and 65C816 have static cores, and thus retain data even when the clock is completely halted.

3.3.3 12-bit designs

The Intersil 6100 family consisted of a 12-bit microprocessor (the 6100) and a range of peripheral support and memory ICs. The microprocessor recognised the DEC PDP-8 minicomputer instruction set. As such it was sometimes referred to as the **CMOS-PDP8** .

Since it was also produced by Harris Corporation, it was also known as the **Harris HM-6100** . By virtue of its CMOS technology and associated benefits, the 6100 was being incorporated into some military designs until the early 1980s.

3.3.4 16-bit designs

The first multi-chip 16-bit microprocessor was the National Semiconductor IMP-16, introduced in early 1973. An 8-bit version of the chipset was introduced in 1974 as the IMP-8.

Other early multi-chip 16-bit microprocessors include one that Digital Equipment Corporation (DEC) used in the LSI-11 OEM board set and the packaged PDP 11/03 minicomputer—and the Fairchild Semiconductor MicroFlame 9440, both introduced in 1975–76. In 1975, National introduced the first 16-bit single-chip microprocessor, the National Semiconductor PACE, which was later followed by an NMOS version, the INS8900.

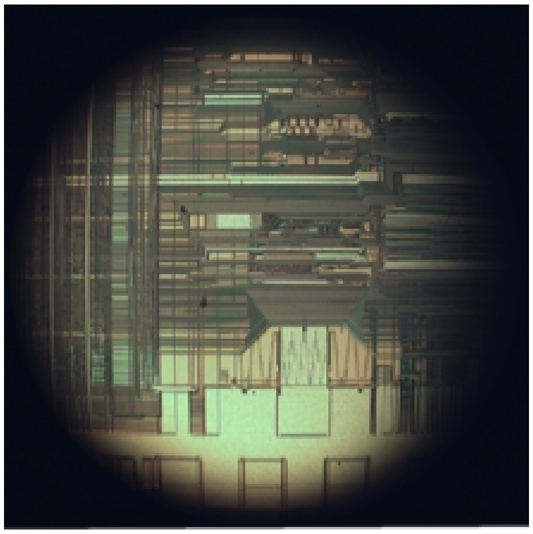
Another early single-chip 16-bit microprocessor was TI's TMS 9900, which was also compatible with their TI-990 line of minicomputers. The 9900 was used in the TI 990/4 minicomputer, the Texas Instruments TI-99/4A home computer, and the TM990 line of OEM microcomputer boards. The chip was packaged in a large ceramic 64-pin DIP package, while most 8-bit microprocessors such as the Intel 8080 used the more common, smaller, and less expensive plastic 40-pin DIP. A follow-on chip, the TMS 9980, was designed to compete with the Intel 8080, had the full TI 990 16-bit instruction set, used a plastic 40-pin package, moved data 8 bits at a time, but could only address 16 KB. A third chip, the TMS 9995, was a new design. The family later expanded to include the 99105 and 99110.

The Western Design Center (WDC) introduced the CMOS 65816 16-bit upgrade of the WDC CMOS 65C02 in 1984. The 65816 16-bit microprocessor was the core of the Apple IIgs and later the Super Nintendo Entertainment System, making it one of the most popular 16-bit designs of all time.

Intel "upsized" their 8080 design into the 16-bit Intel 8086, the first member of the x86 family, which powers most modern PC type computers. Intel introduced the 8086 as a cost-effective way of porting software from the 8080 lines, and succeeded in winning much business on that premise. The 8088, a version of the 8086 that used an 8-bit external data bus, was the microprocessor in the first IBM PC. Intel then released the 80186 and 80188, the 80286 and, in 1985, the 32-bit 80386, cementing their PC market dominance with the processor family's backwards compatibility. The 80186 and 80188 were essentially versions of the 8086 and 8088, enhanced with some onboard peripherals and a few new instructions. Although Intel's 80186 and 80188 were not used in IBM PC type designs, second source versions from NEC, the V20 and V30 frequently were. The 8086 and successors had an innovative but limited method of memory segmentation, while the 80286 introduced a full-featured segmented memory management unit (MMU). The 80386 introduced a flat 32-bit memory model with paged memory management.

The 16-bit Intel x86 processors up to and including the 80386 do not include floating-point units (FPUs). Intel introduced the 8087, 80187, 80287 and 80387 math coprocessors to add hardware floating-point and transcendental function capabilities to the 8086 through 80386 CPUs. The 8087 works with the 8086/8088 and 80186/80188, the 80187 works with the 80186 but not the 80188, the 80287 works with the 80286 and the 80387 works with the 80386. The combination of an x86 CPU and an x87 coprocessor forms a single multi-chip microprocessor; the two chips are programmed as a unit using a single integrated instruction set. The 8087 and 80187 coprocessors are connected in parallel with the data and address buses of their parent processor and directly execute instructions intended for them. The 80287 and 80387 coprocessors are interfaced to the CPU through I/O ports in the CPU's address space, this is transparent to the program, which does not need to know about or access these I/O ports directly; the program accesses the coprocessor and its registers through normal instruction opcodes.

3.3.5 32-bit designs



Upper interconnect layers on an Intel 80486DX2 die

16-bit designs had only been on the market briefly when 32-bit implementations started to appear.

The most significant of the 32-bit designs is the Motorola MC68000, introduced in 1979. The 68k, as it was widely known, had 32-bit registers in its programming model but used 16-bit internal data paths, three 16-bit Arithmetic Logic Units, and a 16-bit external data bus (to reduce pin count), and externally supported only 24-bit addresses (internally it worked with full 32 bit addresses). In PC-based IBM-compatible mainframes the MC68000 internal microcode was modified to emulate the 32-bit System/370 IBM mainframe. Motorola generally described it as a 16-bit processor. The combination of high performance, large (16 megabytes or 2^{24} bytes) memory space and fairly low cost made it the most popular CPU design of its class. The Apple Lisa and Macintosh designs made use of the 68000, as did a host of other designs in the mid-1980s, including the Atari ST and Commodore Amiga.

The world's first single-chip fully 32-bit microprocessor, with 32-bit data paths, 32-bit buses, and 32-bit addresses, was the AT&T Bell Labs BELLMAC-32A, with first samples in 1980, and general production in 1982. After the divestiture of AT&T in 1984, it was renamed the WE 32000 (WE for Western Electric), and had two follow-on generations, the WE 32100 and WE 32200. These microprocessors were used in the AT&T 3B5 and 3B15 minicomputers; in the 3B2, the world's first desktop super microcomputer; in the "Companion", the world's first 32-bit laptop computer; and in "Alexander", the world's first book-sized super microcomputer, featuring ROM-pack memory cartridges similar to today's gaming consoles. All these systems ran the UNIX System V operating system.

The first commercial, single chip, fully 32-bit microprocessor available on the market was the HP FOCUS.

Intel's first 32-bit microprocessor was the iAPX 432, which was introduced in 1981, but was not a commercial success. It had an advanced capability-based object-oriented architecture, but poor performance compared to contemporary architectures such as Intel's own 80286 (introduced 1982), which was almost four times as fast on typical benchmark tests. However, the results for the iAPX432 was partly due to a rushed and therefore suboptimal Ada compiler.

Motorola's success with the 68000 led to the MC68010, which added virtual memory support. The MC68020, introduced in 1984 added full 32-bit data and address buses. The 68020 became hugely popular in the Unix supermicrocomputer market, and many small companies (e.g., Altos, Charles River Data Systems, Cromemco) produced desktop-size systems. The MC68030 was introduced next, improving upon the previous design by integrating the MMU into the chip. The continued success led to the MC68040, which included an FPU for better math performance. The 68050 failed to achieve its performance goals and was not released, and the follow-up MC68060 was released into a market saturated by much faster RISC designs. The 68k family faded from use in the early 1990s.

Other large companies designed the 68020 and follow-ons into embedded equipment. At one point, there were more 68020s in embedded equipment than there were Intel Pentiums in PCs. The ColdFire processor cores are derivatives of the 68020.

During this time (early to mid-1980s), National Semiconductor introduced a very similar 16-bit pinout, 32-bit internal microprocessor called the NS 16032 (later renamed 32016), the full 32-bit version named the NS 32032. Later, National Semiconductor produced the NS 32132, which allowed two CPUs to reside on the same memory bus with built in arbitration. The NS32016/32 outperformed the MC68000/10, but the NS32332—which arrived at approximately the same time as the MC68020—did not have enough performance. The third generation chip, the NS32532, was different. It had about double the performance of the MC68030, which was released around the same time. The appearance of RISC processors like the AM29000 and MC88000 (now both dead) influenced the architecture of the final core, the NS32764. Technically advanced—with a superscalar RISC core, 64-bit bus, and internally overclocked—it could still execute Series 32000 instructions through real-time translation.

When National Semiconductor decided to leave the Unix market, the chip was redesigned into the Swordfish Embedded processor with a set of on chip peripherals. The chip turned out to be too expensive for the laser printer market and was killed. The design team went to Intel and there designed the Pentium processor, which is very similar to the NS32764 core internally. The big success of the Series 32000 was in the laser printer market, where the NS32CG16 with microcoded BitBlt instructions had very good price/performance and was adopted by large companies like Canon. By the mid-1980s, Sequent introduced the first SMP server-class computer using the NS 32032. This was one of the design's few wins, and it disappeared in the late 1980s. The MIPS R2000 (1984) and R3000 (1989) were highly successful 32-bit RISC microprocessors. They were used in high-end workstations and servers by SGI, among others. Other designs included the Zilog Z80000, which arrived too late to market to stand a chance and disappeared quickly.

The ARM first appeared in 1985. This is a RISC processor design, which has since come to dominate the 32-bit embedded systems processor space due in large part to its power efficiency, its licensing model, and its wide selection of system development tools. Semiconductor manufacturers generally license cores and integrate them into their own system on a chip products; only a few such vendors are licensed to modify the ARM cores. Most cell phones include an ARM processor, as do a wide variety of other products. There are microcontroller-oriented ARM cores without virtual memory support, as well as symmetric multiprocessor (SMP) applications processors with virtual memory.

From 1993 to 2003, the 32-bit x86 architectures became increasingly dominant in desktop, laptop, and server markets, and these microprocessors became faster and more capable. Intel had licensed early versions of the architecture to other companies, but declined to license the Pentium, so AMD and Cyrix built later versions of the architecture based on their own designs. During this span, these processors increased in complexity (transistor count) and capability (instructions/second) by at least three orders of magnitude. Intel's Pentium line is probably the most famous and recognizable 32-bit processor model, at least with the public at broad.

3.3.6 64-bit designs in personal computers

While 64-bit microprocessor designs have been in use in several markets since the early 1990s (including the Nintendo 64 gaming console in 1996), the early 2000s saw the introduction of 64-bit microprocessors targeted at the PC market.

With AMD's introduction of a 64-bit architecture backwards-compatible with x86, x86-64 (also called **AMD64**), in September 2003, followed by Intel's near fully compatible 64-bit extensions (first called IA-32e or EM64T, later renamed **Intel 64**), the 64-bit desktop era began. Both versions can run 32-bit legacy applications without any performance penalty as well as new 64-bit software. With operating systems Windows XP x64, Windows Vista x64, Windows 7 x64, Linux, BSD, and macOS that run 64-bit natively, the software is also geared to fully utilize the capabilities of such processors. The move to 64 bits is more than just an increase in register size from the IA-32 as it also doubles the number of general-purpose registers.

The move to 64 bits by PowerPC had been intended since the architecture's design in the early 90s and was not a major cause of incompatibility. Existing integer registers are extended as are all related data pathways, but, as was the case with IA-32, both floating point and vector units had been operating at or above 64 bits for several years. Unlike what happened when IA-32 was extended to x86-64, no new general purpose registers were added in 64-bit

PowerPC, so any performance gained when using the 64-bit mode for applications making no use of the larger address space is minimal.

In 2011, ARM introduced a new 64-bit ARM architecture.

3.3.7 RISC

In the mid-1980s to early 1990s, a crop of new high-performance reduced instruction set computer (RISC) microprocessors appeared, influenced by discrete RISC-like CPU designs such as the IBM 801 and others. RISC microprocessors were initially used in special-purpose machines and Unix workstations, but then gained wide acceptance in other roles.

The first commercial RISC microprocessor design was released in 1984, by MIPS Computer Systems, the 32-bit R2000 (the R1000 was not released). In 1986, HP released its first system with a PA-RISC CPU. In 1987, in the non-Unix Acorn computers' 32-bit, then cache-less, ARM2-based Acorn Archimedes became the first commercial success using the ARM architecture, then known as Acorn RISC Machine (ARM); first silicon ARM1 in 1985. The R3000 made the design truly practical, and the R4000 introduced the world's first commercially available 64-bit RISC microprocessor. Competing projects would result in the IBM POWER and Sun SPARC architectures. Soon every major vendor was releasing a RISC design, including the AT&T CRISP, AMD 29000, Intel i860 and Intel i960, Motorola 88000, DEC Alpha.

In the late 1990s, only two 64-bit RISC architectures were still produced in volume for non-embedded applications: SPARC and Power ISA, but as ARM has become increasingly powerful, in the early 2010s, it became the third RISC architecture in the general computing segment.

3.3.8 Multi-core designs

A different approach to improving a computer's performance is to add extra processors, as in symmetric multiprocessing designs, which have been popular in servers and workstations since the early 1990s. Keeping up with Moore's law is becoming increasingly challenging as chip-making technologies approach their physical limits. In response, microprocessor manufacturers look for other ways to improve performance so they can maintain the momentum of constant upgrades.

A multi-core processor is a single chip that contains more than one microprocessor core. Each core can simultaneously execute processor instructions in parallel. This effectively multiplies the processor's potential performance by the number of cores, if the software is designed to take advantage of more than one processor core. Some components, such as bus interface and cache, may be shared between cores. Because the cores are physically close to each other, they can communicate with each other much faster than separate (off-chip) processors in a multiprocessor system, which improves overall system performance.

In 2001, IBM introduced the first commercial multi-core processor, the monolithic two-core POWER4. Personal computers did not receive multi-core processors until the 2005 introduction, of the two-core Intel Pentium D. The Pentium D, however, was not a monolithic multi-core processor. It was constructed from two dies, each containing a core, packaged on a multi-chip module. The first monolithic multi-core processor in the personal computer market was the AMD Athlon X2, which was introduced a few weeks after the Pentium D. As of 2012, dual- and quad-core processors are widely used in home PCs and laptops, while quad-, six-, eight-, ten-, twelve-, and sixteen-core processors are common in the professional and enterprise markets with workstations and servers.

Sun Microsystems has released the Niagara and Niagara 2 chips, both of which feature an eight-core design. The Niagara 2 supports more threads and operates at 1.6 GHz.

High-end Intel Xeon processors that are on the LGA 775, LGA 1366, and LGA 2011 sockets and high-end AMD Opteron processors that are on the C32 and G34 sockets are DP (dual processor) capable, as well as the older Intel Core 2 Extreme QX9775 also used in an older Mac Pro by Apple and the Intel Skulltrail motherboard. AMD's G34 motherboards can support up to four CPUs and Intel's LGA 1567 motherboards can support up to eight CPUs.

Modern desktop computers support systems with multiple CPUs, but few applications outside of the professional market can make good use of more than four cores. Both Intel and AMD currently offer fast quad, hex and octa-core desktop CPUs, making multi-CPU systems obsolete for many purposes. The desktop market has been in a transition towards quad-core CPUs since Intel's Core 2 Quad was released and are now common, although dual-core CPUs are still more prevalent. Older or mobile computers are less likely to have more than two cores than newer desktops. Not all software is optimised for multi-core CPUs, making fewer, more powerful cores preferable.

AMD offers CPUs with more cores for a given amount of money than similarly priced Intel CPUs—but the AMD cores are somewhat slower, so the two trade blows in different applications depending on how well-threaded the programs running are. For example, Intel's cheapest Sandy Bridge quad-core CPUs often cost almost twice as much as AMD's cheapest Athlon II, Phenom II, and FX quad-core CPUs but Intel has dual-core CPUs in the same price ranges as AMD's cheaper quad-core CPUs. In an application that uses one or two threads, the Intel dual-core CPUs outperform AMD's similarly priced quad-core CPUs—and if a program supports three or four threads the cheap AMD quad-core CPUs outperform the similarly priced Intel dual-core CPUs.

Historically, AMD and Intel have switched places as the company with the fastest CPU several times. Intel currently leads on the desktop side of the computer CPU market, with their Sandy Bridge and Ivy Bridge series. In servers, AMD's new Opterons seem to have superior performance for their price point. This means that AMD are currently more competitive in low- to mid-end servers and workstations that more effectively use fewer cores and threads.

Taken to the extreme, this trend also includes manycore designs, with hundreds of cores, with qualitatively different architectures.

3.4 Market statistics

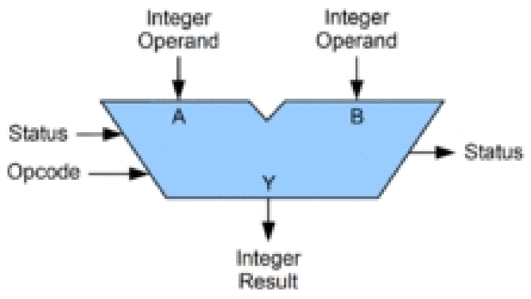
In 1997, about 55% of all CPUs sold in the world were 8-bit microcontrollers, of which over 2 billion were sold.

In 2002, less than 10% of all the CPUs sold in the world were 32-bit or more. Of all the 32-bit CPUs sold, about 2% are used in desktop or laptop personal computers. Most microprocessors are used in embedded control applications such as household appliances, automobiles, and computer peripherals. Taken as a whole, the average price for a microprocessor, microcontroller, or DSP is just over US$6 (equivalent to $8.53 in 2019).

In 2003, about US$44 (equivalent to $61.15 in 2019) billion worth of microprocessors were manufactured and sold. Although about half of that money was spent on CPUs used in desktop or laptop personal computers, those count for only about 2% of all CPUs sold. The quality-adjusted price of laptop microprocessors improved −25% to −35% per year in 2004–2010, and the rate of improvement slowed to −15% to −25% per year in 2010–2013.

About 10 billion CPUs were manufactured in 2008. Most new CPUs produced each year are embedded.

Chapter 4 Arithmetic logic unit



A symbolic representation of an ALU and its input and output signals, indicated by arrows pointing into or out of the ALU, respectively. Each arrow represents one or more signals. Control signals enter from the left and status signals exit on the right; data flows from top to bottom.

An **arithmetic logic unit** (**ALU**) is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. An ALU is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs). A single CPU, FPU or GPU may contain multiple ALUs.

The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed; the ALU's output is the result of the performed operation. In many designs, the ALU also has status inputs or outputs, or both, which convey information about a previous operation or the current operation, respectively, between the ALU and external status registers.

4.1 Signals

An ALU has a variety of input and output nets, which are the electrical conductors used to convey digital signals between the ALU and external circuitry. When an ALU is operating, external circuits apply signals to the ALU inputs and, in response, the ALU produces and conveys signals to external circuitry via its outputs.

4.1.1 Data

A basic ALU has three parallel data buses consisting of two input operands (*A* and *B*) and a result output (*Y*). Each data bus is a group of signals that conveys one binary integer number. Typically, the A, B and Y bus widths (the number of signals comprising each bus) are identical and match the native word size of the external circuitry (e.g., the encapsulating CPU or other processor).

4.1.2 Opcode

The *opcode* input is a parallel bus that conveys to the ALU an operation selection code, which is an enumerated value that specifies the desired arithmetic or logic operation to be performed by the ALU. The opcode size (its bus width) determines the maximum number of different operations the ALU can perform; for example, a four-bit opcode can specify up to sixteen different ALU operations. Generally, an ALU opcode is not the same as a machine language opcode, though in some cases it may be directly encoded as a bit field within a machine language opcode.

4.1.3 Status

Outputs

The status outputs are various individual signals that convey supplemental information about the result of the current ALU operation. General-purpose ALUs commonly have status signals such as:

- *Carry-out* , which conveys the carry resulting from an addition operation, the borrow resulting from a subtraction operation, or the overflow bit resulting from a binary shift operation.
- *Zero* , which indicates all bits of Y are logic zero.
- *Negative* , which indicates the result of an arithmetic operation is negative.
- *Overflow* , which indicates the result of an arithmetic operation has exceeded the numeric range of Y.

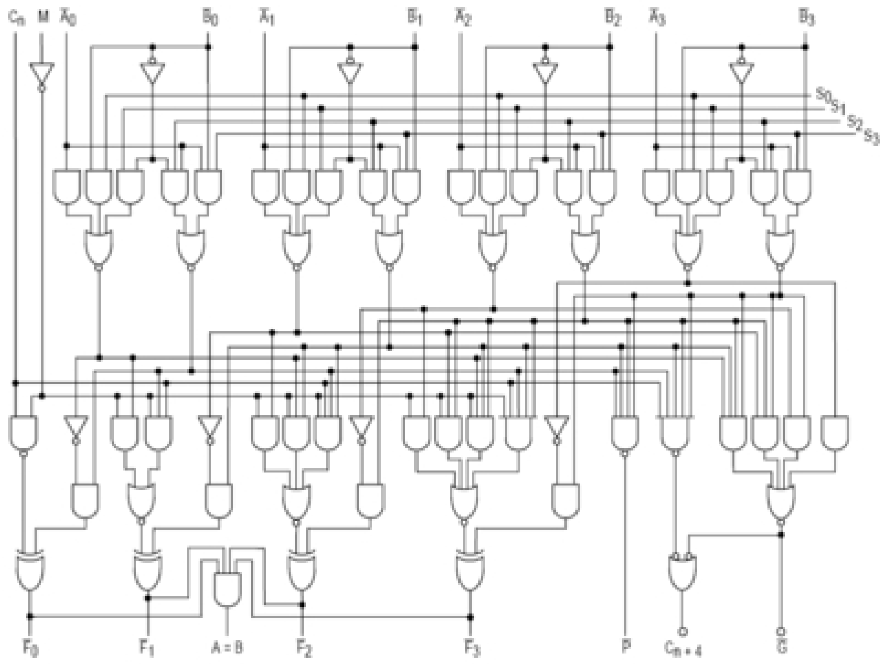
- *Parity* , which indicates whether an even or odd number of bits in Y are logic one.

At the end of each ALU operation, the status output signals are usually stored in external registers to make them available for future ALU operations (e.g., to implement multiple-precision arithmetic) or for controlling conditional branching. The collection of bit registers that store the status outputs are often treated as a single, multi-bit register, which is referred to as the "status register" or "condition code register".

Inputs

The status inputs allow additional information to be made available to the ALU when performing an operation. Typically, this is a single "carry-in" bit that is the stored carry-out from a previous ALU operation.

4.2 Circuit operation



The combinational logic circuitry of the 74181 integrated circuit, which is a simple four-bit ALU

An ALU is a combinational logic circuit, meaning that its outputs will change asynchronously in response to input changes. In normal operation, stable signals are applied to all of the ALU inputs and, when enough time (known as the "propagation delay") has passed for the signals to propagate through the ALU circuitry, the result of the ALU operation appears at the ALU outputs. The external circuitry connected to the ALU is responsible for ensuring the stability of ALU input signals throughout the operation, and for allowing sufficient time for the signals to propagate through the ALU before sampling the ALU result.

In general, external circuitry controls an ALU by applying signals to its inputs. Typically, the external circuitry employs sequential logic to control the ALU operation, which is paced by a clock signal of a sufficiently low frequency to ensure enough time for the ALU outputs to settle under worst-case conditions.

For example, a CPU begins an ALU addition operation by routing operands from their sources (which are usually registers) to the ALU's operand inputs, while the control unit simultaneously applies a value to the ALU's opcode input, configuring it to perform addition. At the same time, the CPU also routes the ALU result output to a destination register that will receive the sum. The ALU's input signals, which are held stable until the next clock, are allowed to propagate through the ALU and to the destination register while the CPU waits for the next clock. When the next clock arrives, the destination register stores the ALU result and, since the ALU operation has completed, the ALU inputs may be set up for the next ALU operation.

4.3 Functions

A number of basic arithmetic and bitwise logic functions are commonly supported by ALUs. Basic, general purpose ALUs typically include these operations in their repertoires:

4.3.1 Arithmetic operations

- *Add* : A and B are summed and the sum appears at Y and carry-out.
- *Add with carry* : A, B and carry-in are summed and the sum appears at Y and carry-out.

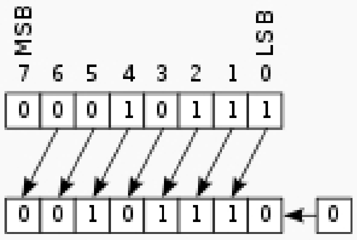
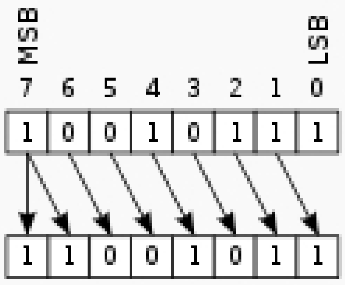
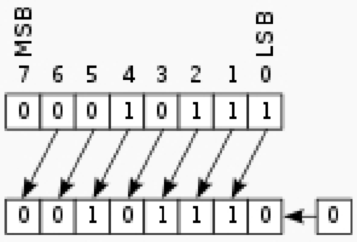
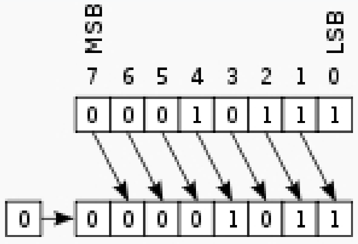
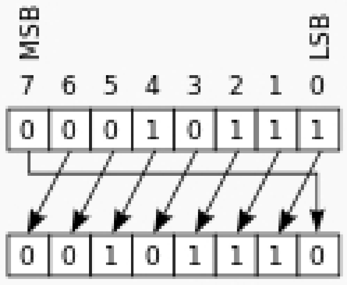
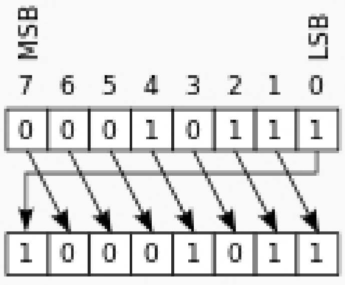
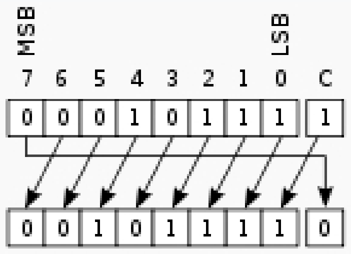
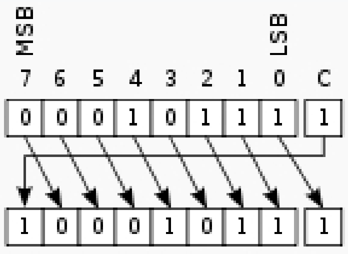
- *Subtract* : B is subtracted from A (or vice versa) and the difference appears at Y and carry-out. For this function, carry-out is effectively a "borrow" indicator. This operation may also be used to compare the magnitudes of A and B; in such cases the Y output may be ignored by the processor, which is only interested in the status bits (particularly zero and negative) that result from the operation.
- *Subtract with borrow* : B is subtracted from A (or vice versa) with borrow (carry-in) and the difference appears at Y and carry-out (borrow out).
- *Two's complement (negate)* : A (or B) is subtracted from zero and the difference appears at Y.
- *Increment* : A (or B) is increased by one and the resulting value appears at Y.
- *Decrement* : A (or B) is decreased by one and the resulting value appears at Y.
- *Pass through* : all bits of A (or B) appear unmodified at Y. This operation is typically used to determine the parity of the operand or whether it is zero or negative, or to load the operand into a processor register.

4.3.2 Bitwise logical operations

- *AND* : the bitwise AND of A and B appears at Y.
- *OR* : the bitwise OR of A and B appears at Y.
- *Exclusive-OR* : the bitwise XOR of A and B appears at Y.
- *Ones' complement* : all bits of A (or B) are inverted and appear at Y.

4.3.3 Bit shift operations

Bit shift examples for an eight-bit ALU		
Type	Left	Right
Arithmetic shift		

		
Logical shift		
Rotate		
Rotate through carry		

ALU shift operations cause operand A (or B) to shift left or right (depending on the opcode) and the shifted operand appears at Y. Simple ALUs typically can shift the operand by only one bit position, whereas more complex ALUs employ barrel shifters that allow them to shift the operand by an arbitrary number of bits in one operation. In all single-bit shift operations, the bit shifted out of the operand appears on carry-out; the value of the bit shifted into the operand depends on the type of shift.

- *Arithmetic shift* : the operand is treated as a two's complement integer, meaning that the most significant bit is a "sign" bit and is preserved.

- *Logical shift* : a logic zero is shifted into the operand. This is used to shift unsigned integers.
- *Rotate* : the operand is treated as a circular buffer of bits so its least and most significant bits are effectively adjacent.
- *Rotate through carry* : the carry bit and operand are collectively treated as a circular buffer of bits.

4.4 Applications

4.4.1 Multiple-precision arithmetic

In integer arithmetic computations, **multiple-precision arithmetic** is an algorithm that operates on integers which are larger than the ALU word size. To do this, the algorithm treats each operand as an ordered collection of ALU-size fragments, arranged from most-significant (MS) to least-significant (LS) or vice versa. For example, in the case of an 8-bit ALU, the 24-bit integer `0x123456` would be treated as a collection of three 8-bit fragments: `0x12` (MS), `0x34` , and `0x56` (LS). Since the size of a fragment exactly matches the ALU word size, the ALU can directly operate on this "piece" of operand.

The algorithm uses the ALU to directly operate on particular operand fragments and thus generate a corresponding fragment (a "partial") of the multi-precision result. Each partial, when generated, is written to an associated region of storage that has been designated for the multiple-precision result. This process is repeated for all operand fragments so as to generate a complete collection of partials, which is the result of the multiple-precision operation.

In arithmetic operations (e.g., addition, subtraction), the algorithm starts by invoking an ALU operation on the operands' LS fragments, thereby producing both a LS partial and a carry out bit. The algorithm writes the partial to designated storage, whereas the processor's state machine typically stores the carry out bit to an ALU status register. The algorithm then advances to the next fragment of each operand's collection and invokes an ALU operation on these fragments along with the stored carry bit from the previous ALU operation, thus producing another (more significant) partial and a carry out bit. As before, the carry bit is stored to the status register and the partial is written to designated storage. This process repeats until all operand fragments have been processed, resulting in a complete collection of partials in storage, which comprise the multi-precision arithmetic result.

In multiple-precision shift operations, the order of operand fragment processing depends on the shift direction. In left-shift operations, fragments are processed LS first because the LS bit of each partial —which is conveyed via the stored carry bit—must be obtained from the MS bit of the previously left-shifted, less-significant operand. Conversely, operands are processed MS first in right-shift operations because the MS bit of each partial must be obtained from the LS bit of the previously right-shifted, more- significant operand.

In bitwise logical operations (e.g., logical AND, logical OR), the operand fragments may be processed in any arbitrary order because each partial depends only on the corresponding operand fragments (the stored carry bit from the previous ALU operation is ignored).

4.4.2 Complex operations

Although an ALU can be designed to perform complex functions, the resulting higher circuit complexity, cost, power consumption and larger size makes this impractical in many cases. Consequently, ALUs are often limited to simple functions that can be executed at very high speeds (i.e., very short propagation delays), and the external processor circuitry is responsible for performing complex functions by orchestrating a sequence of simpler ALU operations.

For example, computing the square root of a number might be implemented in various ways, depending on ALU complexity:

- *Calculation in a single clock* : a very complex ALU that calculates a square root in one operation.
- *Calculation pipeline* : a group of simple ALUs that calculates a square root in stages, with intermediate results passing through ALUs arranged like a factory production line. This circuit can accept new operands before finishing the previous ones and produces results as

fast as the very complex ALU, though the results are delayed by the sum of the propagation delays of the ALU stages. For more information, see the article on instruction pipelining.

- *Iterative calculation* : a simple ALU that calculates the square root through several steps under the direction of a control unit.

The implementations above transition from fastest and most expensive to slowest and least costly. The square root is calculated in all cases, but processors with simple ALUs will take longer to perform the calculation because multiple ALU operations must be performed.

4.5 Implementation

An ALU is usually implemented either as a stand-alone integrated circuit (IC), such as the 74181, or as part of a more complex IC. In the latter case, an ALU is typically instantiated by synthesizing it from a description written in VHDL, Verilog or some other hardware description language. For example, the following VHDL code describes a very simple 8-bit ALU:

```
entity alu is
port (  -- the alu connections to external circuitry:
  A  : in   signed(7 downto 0);   -- operand A
  B  : in   signed(7 downto 0);   -- operand B
  OP : in   unsigned(2 downto 0); -- opcode
  Y  : out signed(7 downto 0));  -- operation result
end alu ;

architecture behavioral of alu is
begin
  case OP is   -- decode the opcode and perform the operation:
    when "000" =>  Y <= A + B;   -- add
    when "001" =>  Y <= A - B;   -- subtract
    when "010" =>  Y <= A - 1;   -- decrement
    when "011" =>  Y <= A + 1;   -- increment
    when "100" =>  Y <= not A;   -- 1's complement
    when "101" =>  Y <= A and B; -- bitwise AND
    when "110" =>  Y <= A or B;  -- bitwise OR
    when "111" =>  Y <= A xor B; -- bitwise XOR
    when others => Y <= (others => 'X');
  end case ;
end behavioral ;
```

4.6 History

Mathematician John von Neumann proposed the ALU concept in 1945 in a report on the foundations for a new computer called the EDVAC.

The cost, size, and power consumption of electronic circuitry was relatively high throughout the infancy of the information age. Consequently, all serial computers and many early computers, such as the PDP-8, had a simple ALU that operated on one data bit at a time, although they often presented a wider word size to programmers. One of the earliest computers to have multiple discrete single-bit ALU circuits was the 1948 Whirlwind I, which employed sixteen of such "math units" to enable it to operate on 16-bit words.

In 1967, Fairchild introduced the first ALU implemented as an integrated circuit, the Fairchild 3800, consisting of an eight-bit ALU with accumulator. Other integrated-circuit ALUs soon emerged, including four-bit ALUs such as the Am2901 and 74181. These devices were typically "bit slice" capable, meaning they had "carry look ahead" signals that facilitated the use of multiple interconnected ALU chips to create an ALU with a wider word size. These devices quickly became popular and were widely used in bit-slice minicomputers.

Microprocessors began to appear in the early 1970s. Even though transistors had become smaller, there was often insufficient die space for a full-word-width ALU and, as a result, some early microprocessors employed a narrow ALU that required multiple cycles per machine language instruction. Examples of this includes the popular Zilog Z80, which performed eight-bit additions with a four-bit ALU. Over time, transistor geometries shrank further, following Moore's law, and it became feasible to build wider ALUs on microprocessors.

Modern integrated circuit (IC) transistors are orders of magnitude smaller than those of the early microprocessors, making it possible to fit highly complex ALUs on ICs. Today, many modern ALUs have wide word widths, and architectural enhancements such as barrel shifters and binary multipliers that allow them to perform, in a single clock cycle, operations that would have required multiple operations on earlier ALUs.

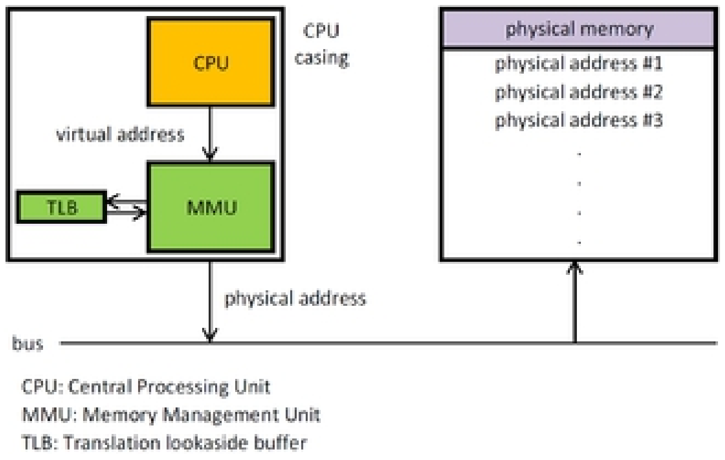
Chapter 5 Memory management unit

This 68451 MMU could be used with the Motorola 68010

A **memory management unit** (**MMU**), sometimes called **paged memory management unit** (**PMMU**), is a computer hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses. It is usually implemented as part of the central processing unit (CPU), but it also can be in the form of a separate integrated circuit.

An MMU effectively performs virtual memory management, handling at the same time memory protection, cache control, bus arbitration and, in simpler computer architectures (especially 8-bit systems), bank switching.

5.1 Overview



Schematic of the operation of an MMU

Modern MMUs typically divide the virtual address space (the range of addresses used by the processor) into pages, each having a size which is a power of 2, usually a few kilobytes, but they may be much larger. The bottom bits of the address (the offset within a page) are left unchanged. The upper address bits are the virtual page numbers.

5.1.1 Page table entries

Most MMUs use an in-memory table of items called a "page table", containing one "page table entry" (PTE) per page, to map virtual page numbers to physical page numbers in main memory. An associative cache of PTEs is called a translation lookaside buffer (TLB) and is used to avoid the necessity of accessing the main memory every time a virtual address is mapped. Other MMUs may have a private array of memory or registers that hold a set of page table entries. The physical page number is combined with the page offset to give the complete physical address.

A PTE may also include information about whether the page has been written to (the "dirty bit"), when it was last used (the "accessed bit," for a least recently used (LRU) page replacement algorithm), what kind of processes (user mode or supervisor mode) may read and write it, and whether it should be cached.

Sometimes, a PTE prohibits access to a virtual page, perhaps because no physical random access memory has been allocated to that virtual page. In this case, the MMU signals a page fault to the

CPU. The operating system (OS) then handles the situation, perhaps by trying to find a spare frame of RAM and set up a new PTE to map it to the requested virtual address. If no RAM is free, it may be necessary to choose an existing page (known as a "victim"), using some replacement algorithm, and save it to disk (a process called "paging"). With some MMUs, there can also be a shortage of PTEs, in which case the OS will have to free one for the new mapping.

The MMU may also generate illegal access error conditions or invalid page faults upon illegal or non-existing memory accesses, respectively, leading to segmentation fault or bus error conditions when handled by the operating system.

5.1.2 Benefits

VLSI VI475 MMU "Apple HMMU" from the Macintosh II used with the Motorola 68020

In some cases, a page fault may indicate a software bug, which can be prevented by using memory protection as one of key benefits of an MMU: an operating system can use it to protect against errant programs by disallowing access to memory that a particular program should not have access to. Typically, an operating system assigns each program its own virtual address space.

An MMU also mitigates the problem of fragmentation of memory. After blocks of memory have been allocated and freed, the free memory may become fragmented (discontinuous) so that the largest contiguous block of free memory may be much smaller than the total amount. With virtual memory, a contiguous range of virtual addresses can be mapped to several non-contiguous blocks of physical memory; this non-contiguous allocation is one of the benefits of paging.

In some early microprocessor designs, memory management was performed by a separate integrated circuit such as the VLSI VI475 (1986), the Motorola 68851 (1984) used with the Motorola 68020 CPU in the Macintosh II, or the Z8015 (1985) used with the Zilog Z8000 family of processors. Later microprocessors (such as the Motorola 68030 and the Zilog Z280) placed the MMU together with the CPU on the same integrated circuit, as did the Intel 80286 and later x86 microprocessors.

While this article concentrates on modern MMUs, commonly based on pages, early systems used a similar concept for base-limit addressing that further developed into segmentation. Those are occasionally also present on modern architectures. The x86 architecture provided segmentation, rather than paging, in the 80286, and provides both paging and segmentation in the 80386 and later processors (although the use of segmentation is not available in 64-bit operation).

5.2 Examples

Most modern systems divide memory into pages that are 4-64 KB in size, often with the capability to use huge pages from 2 MB to 1 GB in size. Page translations are cached in a translation lookaside buffer (TLB). Some systems, mainly older RISC designs, trap into the OS when a page translation is not found in the TLB. Most systems use a hardware-based tree walker. Most systems allow the MMU to be disabled, but some disable the MMU when trapping into OS code.

5.2.1 VAX

VAX pages are 512 bytes, which is very small. An OS may treat multiple pages as if they were a single larger page. For example, Linux on VAX groups eight pages together. Thus, the system is viewed as having 4 KB pages. The VAX divides memory into four fixed-purpose regions, each 1 GB in size. They are:

P0 space

Used for general-purpose per-process memory such as heaps.

P1 space

(Or control space) which is also per-process and is typically used for supervisor, executive, kernel, user stacks and other per-process control structures managed by the operating system.

S0 space

(Or system space) which is global to all processes and stores operating system code and data, whether paged or not, including pagetables.

S1 space

Which is unused and "Reserved to Digital".

Page tables are big linear arrays. Normally, this would be very wasteful when addresses are used at both ends of the possible range, but the page table for applications is itself stored in the kernel's paged memory. Thus, there is effectively a two-level tree, allowing applications to have sparse memory layout without wasting a lot of space on unused page table entries. The VAX MMU is notable for lacking an accessed bit. OSes which implement paging must find some way to emulate the accessed bit if they are to operate efficiently. Typically, the OS will periodically unmap pages so that page-not-present faults can be used to let the OS set an accessed bit.

5.2.2 ARM

ARM architecture-based application processors implement an MMU defined by ARM's virtual memory system architecture. The current architecture defines PTEs for describing 4 KB and 64 KB pages, 1 MB sections and 16 MB super-sections; legacy versions also defined a 1 KB tiny page. ARM uses a two-level page table if using 4 KB and 64 KB pages, or just a one-level page table for 1 MB sections and 16 MB sections.

TLB updates are performed automatically by page table walking hardware. PTEs include read/write access permission based on privilege, cacheability information, an NX bit, and a non-secure bit.

5.2.3 IBM System/360 Model 67, IBM System/370, and successors

The IBM System/360 Model 67, which was introduced Aug. 1965, included an MMU that was called a dynamic address translation (DAT) box. It had the unusual feature of storing accessed and dirty bits outside of the page table. They refer to physical memory rather than virtual memory, and are accessed by special-purpose instructions. This reduces overhead for the OS, which would otherwise need to propagate accessed and dirty bits from the page tables to a more physically oriented data structure. This makes OS-level virtualization easier.

Starting in August, 1972, the IBM System/370 had a similar MMU, although it initially supported only a 24-bit virtual address space rather than the 32-bit virtual address space of the System/360 Model 67. It also stored the accessed and dirty bits outside the page table. In early 1983, the System/370-XA architecture expanded the virtual address space to 31 bits, and in 2000, the 64-bit z/Architecture was introduced, with the address space expanded to 64 bits; those continued to store the accessed and dirty bits outside the page table.

5.2.4 DEC Alpha

The DEC Alpha processor divides memory into 8 KB pages. After a TLB miss, low-level firmware machine code (here called PALcode) walks a three-level tree-structured page table. Addresses are broken down as follows: 21 bits unused, 10 bits to index the root level of the tree, 10 bits to index the middle level of the tree, 10 bits to index the leaf level of the tree, and 13 bits that pass through to the physical address without modification. Full read/write/execute permission bits are supported.

5.2.5 MIPS

The MIPS architecture supports one to 64 entries in the TLB. The number of TLB entries is configurable at CPU configuration before synthesis. TLB entries are dual. Each TLB entry maps a virtual page number (VPN2) to either one of two page frame numbers (PFN0 or PFN1), depending on the least significant bit of the virtual address that is not part of the page mask. This bit and the page mask bits are not stored in the VPN2. Each TLB entry has its own page size, which can be any value from 1 KB to 256 MB in multiples of four. Each

PFN in a TLB entry has a caching attribute, a dirty and a valid status bit. A VPN2 has a global status bit and an OS assigned ID which participates in the virtual address TLB entry match, if the global status bit is set to zero. A PFN stores the physical address without the page mask bits.

A TLB refill exception is generated when there are no entries in the TLB that match the mapped virtual address. A TLB invalid exception is generated when there is a match but the entry is marked invalid. A TLB modified exception is generated when there is a match but the dirty status is not set. If a TLB exception occurs when processing a TLB exception, a double fault TLB exception, it is dispatched to its own exception handler.

MIPS32 and MIPS32r2 support 32 bits of virtual address space and up to 36 bits of physical address space. MIPS64 supports up to 64 bits of virtual address space and up to 59 bits of physical address space.

5.2.6 Sun 1

The original Sun 1 was a single-board computer built around the Motorola 68000 microprocessor and introduced in 1982. It included the original Sun 1 memory management unit that provided address translation, memory protection, memory sharing and memory allocation for multiple processes running on the CPU. All access of the CPU to private on-board RAM, external Multibus memory, on-board I/O and the Multibus I/O ran through the MMU, where they were translated and protected in uniform fashion. The MMU was implemented in hardware on the CPU board.

The MMU consisted of a context register, a segment map and a page map. Virtual addresses from the CPU were translated into intermediate addresses by the segment map, which in turn were translated into physical addresses by the page map. The page size was 2 KB and the segment size was 32 KB which gave 16 pages per segment. Up to 16 contexts could be mapped concurrently. The maximum logical address space for a context was 1024 pages or 2 MB. The maximum physical address that could be mapped simultaneously was also 2 MB.

The context register was important in a multitasking operating system because it allowed the CPU to switch between processes without reloading all the translation state information. The 4-bit context register could switch between 16 sections of the segment map under supervisor control, which allowed 16 contexts to be mapped concurrently. Each context had its own virtual address space. Sharing of virtual address space and inter-context communications could be provided by writing the same values in to the segment or page maps of different contexts. Additional contexts could be handled by treating the segment map as a context cache and replacing out-of-date contexts on a least-recently used basis.

The context register made no distinction between user and supervisor states. Interrupts and traps did not switch contexts which required that all valid interrupt vectors always be mapped in page 0 of context, as well as the valid supervisor stack.

5.2.7 PowerPC

In PowerPC G1, G2, G3, and G4 pages are normally 4 KB. After a TLB miss, the standard PowerPC MMU begins two simultaneous lookups. One lookup attempts to match the address with one of four or eight data block address translation (DBAT) registers, or four or eight instruction block address translation registers (IBAT), as appropriate. The BAT registers can map linear chunks of memory as large as 256 MB, and are normally used by an OS to map large portions of the address space for the OS kernel's own use. If the BAT lookup succeeds, the other lookup is halted and ignored.

The other lookup, not directly supported by all processors in this family, is via a so-called "inverted page table," which acts as a hashed off-chip extension of the TLB. First, the top four bits of the address are used to select one of 16 segment registers. Then 24 bits from the segment register replace those four bits, producing a 52-bit address. The use of segment registers allows multiple processes to share the same hash table.

The 52-bit address is hashed, then used as an index into the off-chip table. There, a group of eight-page table entries is scanned for one that matches. If none match due to excessive hash collisions, the processor tries again with a slightly different hash function. If this, too, fails, the CPU traps into OS (with MMU disabled) so that the problem may be resolved. The OS needs to discard an entry from the hash table to make space for a new entry. The OS may generate the new entry from a more-normal tree-like page table or from per-mapping data structures which are likely to be slower and more space-efficient. Support for no-execute control is in the segment registers, leading to 256 MB granularity.

A major problem with this design is poor cache locality caused by the hash function. Tree-based designs avoid this by placing the page table entries for adjacent pages in adjacent locations. An operating system running on the PowerPC may minimize the size of the hash table to reduce this problem.

It is also somewhat slow to remove the page table entries of a process. The OS may avoid reusing segment values to delay facing this, or it may elect to suffer the waste of memory associated with per-process hash tables. G1 chips do not search for page table entries, but they do generate the hash, with the expectation that an OS will search the standard hash table via software. The OS can write to the TLB. G2, G3, and early G4 chips use hardware to search the hash table. The latest chips allow the OS to choose either method. On chips that make this optional or do not support it at all, the OS may choose to use a tree-based page table exclusively.

5.2.8 IA-32 / x86

The x86 architecture has evolved over a very long time while maintaining full software compatibility, even for OS code. Thus, the MMU is extremely complex, with many different possible operating modes. Normal operation of the traditional 80386 CPU and its successors (IA-32) is described here.

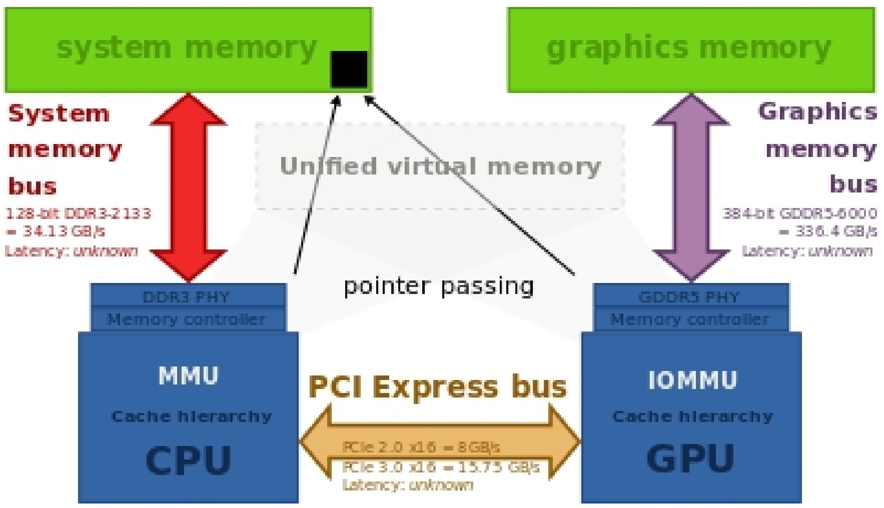
The CPU primarily divides memory into 4 KB pages. Segment registers, fundamental to the older 8088 and 80286 MMU designs, are not used in modern OSes, with one major exception: access to thread-specific data for applications or CPU-specific data for OS kernels, which is done with explicit use of the FS and GS segment registers. All memory access involves a segment register, chosen according to the code being executed. The segment register acts as an index into a table, which provides an offset to be added to the virtual address. Except when using FS or GS, the OS ensures that the offset will be zero.

After the offset is added, the address is masked to be no larger than 32 bits. The result may be looked up via a tree-structured page table, with the bits of the address being split as follows: 10 bits for the branch of the tree, 10 bits for the leaves of the branch, and the 12 lowest bits being directly copied to the result. Some operating systems, such as OpenBSD with its W^X feature, and Linux with the Exec Shield or PaX patches, may also limit the length of the code segment, as specified by the CS register, to disallow execution of code in modifiable regions of the address space.

Minor revisions of the MMU introduced with the Pentium have allowed very large 4 MB pages by skipping the bottom level of the tree (this leaves 10 bits for indexing the first level of page hierarchy with the remaining 10+12 bits being directly copied to the result). Minor revisions of the MMU introduced with the Pentium Pro introduced the physical address extension (PAE) feature, enabling 36-bit physical addresses with 2+9+9 bits for three-level page tables and 12 lowest bits being directly copied to the result. Large pages (2 MB) are also available by skipping the bottom level of the tree (resulting in 2+9 bits for two-level table hierarchy and the remaining 9+12 lowest bits copied directly). In addition, the page attribute table allowed specification of cacheability by looking up a few high bits in a small on-CPU table.

No-execute support was originally only provided on a per-segment basis, making it very awkward to use. More recent x86 chips provide a per-page no-execute bit in the PAE mode. The W^X, Exec Shield, and PaX mechanisms described above emulate per-page non-execute support on machines x86 processors lacking the NX bit by setting the length of the code segment, with a performance loss and a reduction in the available address space.

5.2.9 x86-64



Heterogeneous System Architecture (HSA) creates a unified virtual address space for CPUs, GPUs and DSPs, obsoleting the mapping tricks and data copying.

x86-64 is a 64-bit extension of x86 that almost entirely removes segmentation in favor of the flat memory model used by almost all operating systems for the 386 or newer processors. In long mode, all segment offsets are ignored, except for the FS and GS segments. When used with 4 KB pages, the page table tree has four levels instead of three.

The virtual addresses are divided as follows: 16 bits unused, nine bits each for four tree levels (for a total of 36 bits), and the 12 lowest bits directly copied to the result. With 2 MB pages, there are only three levels of page table, for a total of 27 bits used in paging and 21 bits of offset. Some newer CPUs also support a 1 GB page with two levels of paging and 30 bits of offset.

CPUID can be used to determine if 1 GB pages are supported. In all three cases, the 16 highest bits are required to be equal to the 48th bit, or in other words, the low 48 bits are sign extended to the higher bits. This is done to allow a future expansion of the addressable range, without compromising backwards compatibility. In all levels of the page table, the page table entry includes a no-execute bit.

5.2.10 Unisys MCP Systems (Burroughs B5000)

The Burroughs B5000 from 1961 was the first commercial system to support virtual memory (after the Atlas), even though it has no MMU It provides the two functions of an MMU - virtual memory addresses and memory protection - with a different architectural approach.

First, in the mapping of virtual memory addresses, instead of needing an MMU, the MCP systems are descriptor-based. Each allocated memory block is given a master descriptor with the properties of the block (i.e., the size, address, and whether present in memory). When a request is made to access the block for reading or writing, the hardware checks its presence via the presence bit (pbit) in the descriptor.

A pbit of 1 indicates the presence of the block. In this case, the block can be accessed via the physical address in the descriptor. If the pbit is zero, an interrupt is generated for the MCP (operating system) to make the block present. If the address field is zero, this is the first access to this block, and it is allocated (an init pbit). If the address field is non-zero, it is a disk address of the block, which has previously been rolled out, so the block is fetched from disk and the pbit is set to one and the physical memory address updated to point to the block in memory (another pbit). This makes descriptors equivalent to a page-table entry in an MMU system. System performance can be monitored through the number of pbits. Init pbits indicate initial allocations, but a high level of other pbits indicate that the system may be thrashing.

All memory allocation is therefore completely automatic (one of the features of modern systems) and there is no way to allocate blocks other than this mechanism. There are no such calls as malloc or dealloc, since memory blocks are also automatically discarded. The scheme is also lazy, since a block will not be allocated until it is actually referenced. When memory is nearly full, the MCP examines the working set, trying compaction (since the system is segmented, not paged), deallocating read-only segments (such as code-segments which can be restored from their original copy) and, as a last resort, rolling dirty data segments out to disk.

Another way the B5000 provides a function of a MMU is in protection. Since all accesses are via the descriptor, the hardware can check that all accesses are within bounds and, in the case of a write, that the process has write permission. The MCP system is inherently secure and thus has no need of an MMU to provide this level of memory protection. Descriptors are read only to user processes and may only be updated by the system (hardware or MCP). (Words whose tag is an odd number are read-only; descriptors have a tag of 5 and code words have a tag of 3.)

Blocks can be shared between processes via copy descriptors in the process stack. Thus, some processes may have write permission, whereas others do not. A code segment is read only, thus reentrant and shared between processes. Copy descriptors contain a 20-bit address field giving index of the master descriptor in the master descriptor array. This also implements a very efficient and secure IPC mechanism. Blocks can easily be relocated, since only the master descriptor needs update when a block's status changes.

The only other aspect is performance – do MMU-based or non-MMU-based systems provide better performance? MCP systems may be implemented on top of standard hardware that does have an MMU (for example, a standard PC). Even if the system implementation uses the MMU in some way, this will not be at all visible at the MCP level.

Chapter 6 Parallel computing

IBM's Blue Gene/P massively parallel supercomputer.

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has long been employed in high-performance computing, but it's gaining broader interest due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

Parallel computing is closely related to concurrent computing—they are frequently used together, and often conflated, though the two are distinct: it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU). In parallel computing, a computational task is typically broken down into several, often many, very similar subtasks that can be processed independently and whose results are combined afterwards, upon completion. In contrast, in concurrent computing, the various processes often do not address related tasks; when they do, as is typical in distributed computing, the separate tasks may have a varied nature and often require some inter-process communication during execution.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

In some cases parallelism is transparent to the programmer, such as in bit-level or instruction-level parallelism, but explicitly parallel algorithms, particularly those that use concurrency, are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

A theoretical upper bound on the speed-up of a single program as a result of parallelization is given by Amdahl's law.

6.1 Background

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next one is executed.

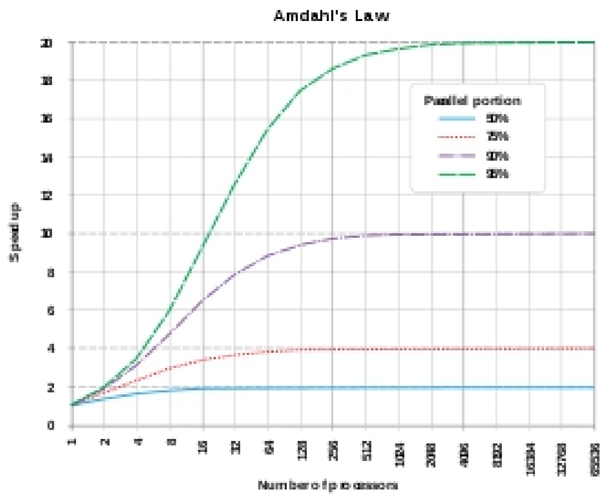
Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above. Historically parallel computing was used for scientific computing and the simulation of scientific problems, particularly in the natural and engineering sciences, such as meteorology. This led to the design of parallel hardware and software, as well as high performance computing.

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all compute-bound programs. However, power consumption P by a chip is given by the equation $P = C \times V^2 \times F$, where C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and F is the processor frequency (cycles per second). Increases in frequency increase the amount of power used in a processor. Increasing processor power consumption led ultimately to Intel's May 8, 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

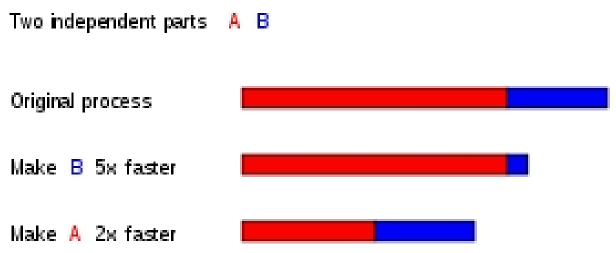
To deal with the problem of power consumption and overheating the major central processing unit (CPU or processor) manufacturers started to produce power efficient processors with multiple cores. The core is the computing unit of the processor and in multi-core processors each core is independent and can access the same memory concurrently. Multi-core processors have brought parallel computing to desktop computers. Thus parallelisation of serial programmes has become a mainstream programming task. In 2012 quad-core processors became standard for desktop computers, while servers have 10 and 12 core processors. From Moore's law it can be predicted that the number of cores per processor will double every 18–24 months. This could mean that after 2020 a typical processor will have dozens or hundreds of cores.

An operating system can ensure that different tasks and user programmes are run in parallel on the available cores. However, for a serial software programme to take full advantage of the multi-core architecture the programmer needs to restructure and parallelise the code. A speed-up of application software runtime will no longer be achieved through frequency scaling, instead programmers will need to parallelise their software code to take advantage of the increasing computing power of multicore architectures.

6.1.1 Amdahl's law and Gustafson's law



A graphical representation of Amdahl's law. The speedup of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 10 times no matter how many processors are used.



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 only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part *A* be twice as fast. This will make the computation much faster than by optimizing part *B* , even though part *B*′ s speedup is greater by ratio, (5 times versus 2 times).

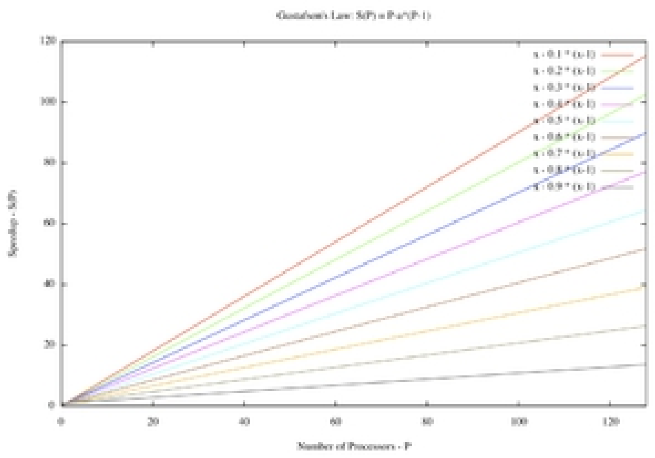
Optimally, the speedup from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speedup. Most of them have a near-linear speedup for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law

where

- S_{latency} is the potential speedup in latency of the execution of the whole task;
- *s* is the speedup in latency of the execution of the parallelizable part of the task;
- *p* is the percentage of the execution time of the whole task concerning the parallelizable part of the task *before parallelization* .

Since $S_{\text{latency}} < 1/(1 - p)$, it shows that a small part of the program which cannot be parallelized will limit the overall speedup available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (serial) parts. If the non-parallelizable part of a program accounts for 10% of the runtime ($p = 0.9$), we can get no more than a 10 times speedup, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."



A graphical representation of Gustafson's law.

Amdahl's law only applies to 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 parallel performance:

Both Amdahl's law and Gustafson's law assume that the running time of the serial part of the program is independent of the number of processors. Amdahl's law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also *independent of the number of processors* , whereas Gustafson's law assumes that the total amount of work to be done in parallel *varies linearly with the number of processors* .

6.1.2 Dependencies

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Let P_i and P_j be two program segments. Bernstein's conditions describe when the two are independent and can be executed in parallel. For P_i, let I_i be all of the input variables and O_i the output variables, and likewise for P_j. P_i and P_j are independent if they satisfy

Violation of the first condition introduces a flow dependency, corresponding to the first segment producing a result used by the second segment. The second condition represents an anti-dependency, when the second segment produces a variable needed by the first segment. The third and final condition represents an output dependency: when two segments write to the same location, the result comes from the logically last executed segment.

Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep(a, b)
2: c := a * b
3: d := 3 * c
4: end function
```

In this example, instruction 3 cannot be executed before (or even in parallel with) instruction 2, because instruction 3 uses a result from instruction 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2: c := a * b
3: d := 3 * b
4: e := a + b
5: end function
```

In this example, there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.

6.1.3 Race conditions, mutual exclusion, synchronization, and parallel slowdown

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, "threads" is generally accepted as a generic term for subtasks. Threads will often need synchronized access to an object or other resource, for example when they must update a variable that is shared between them. Without synchronization, the instructions between the two threads may be interleaved in any order. For example, consider the following program:

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished. Therefore, to guarantee correct program execution, the above program can be rewritten to use locks:

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A: Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks may be necessary to ensure correct program execution when threads must serialize access to resources, but their use can greatly slow a program and may affect its reliability.

Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock locks multiple variables all at once. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a lock or a semaphore. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other or waiting on each other for access to resources. Once the overhead from resource contention or communication dominates the time spent on other computation, further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This problem, known as parallel slowdown, can be improved in some cases by software analysis and redesign.

6.1.4 Fine-grained, coarse-grained, and embarrassing parallelism

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it exhibits embarrassing parallelism if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

6.1.5 Consistency models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

One of the first consistency models was Leslie Lamport's sequential consistency model. Sequential consistency is the property of a parallel program that its parallel execution produces the same results as a sequential program. Specifically, a program is sequentially consistent if "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

Software transactional memory is a common type of consistency model. Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Mathematically, these models can be represented in several ways. Introduced in 1962, Petri nets were an early attempt to codify the rules of consistency models. Dataflow theory later built upon these, and Dataflow architectures were created to physically implement the ideas of dataflow theory. Beginning in the late 1970s, process calculi such as Calculus of Communicating Systems and Communicating Sequential Processes were developed to permit algebraic reasoning about systems composed of interacting components. More recent additions to the process calculus family, such as the π-calculus, have added the capability for reasoning about dynamic topologies. Logics such as Lamport's TLA+, and mathematical models such as traces and Actor event diagrams, have also been developed to describe the behavior of concurrent systems.

6.1.6 Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, and whether or not those instructions were using a single set or multiple sets of data.

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

According to David A. Patterson and John L. Hennessy, "Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme."

6.2 Types of parallelism

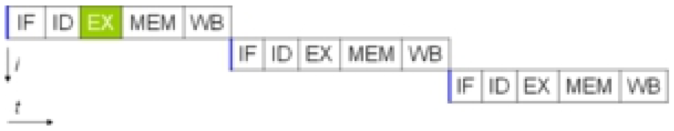
6.2.1 Bit-level parallelism

From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—

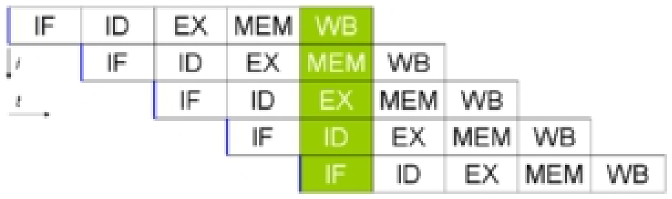
the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.

Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until the early 2000s, with the advent of x86-64 architectures, did 64-bit processors become commonplace.

6.2.2 Instruction-level parallelism



A canonical processor without pipeline. It takes five clock cycles to complete one instruction and thus the processor can issue subscalar performance ($\mathrm{IPC} = 0.2 < 1$).

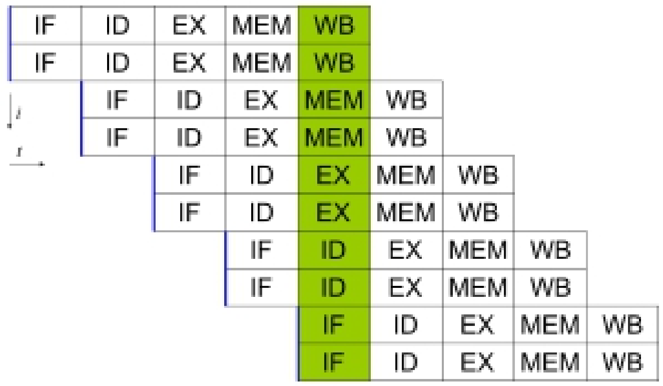


A canonical five-stage pipelined processor. In the best case scenario, it takes one clock cycle to complete one instruction and thus the processor can issue scalar performance ($\mathrm{IPC} = 1$).

A computer program is, in essence, a stream of instructions executed by a processor. Without instruction-level parallelism, a processor can only issue less than one instruction per clock cycle (IPC < 1). These processors are known as *subscalar* processors.

These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

All modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an *N* -stage pipeline can have up to *N* different instructions at different stages of completion and thus can issue one instruction per clock cycle (IPC = 1). These processors are known as *scalar* processors. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and register write back (WB). The Pentium 4 processor had a 35-stage pipeline.



A canonical five-stage pipelined superscalar processor. In the best case scenario, it takes one clock cycle to complete two instructions and thus the processor can issue superscalar performance (IPC = 2 > 1).

Most modern processors also have multiple execution units. They usually combine this feature with pipelining and thus can issue more than one instruction per clock cycle (IPC > 1). These processors are known as *superscalar* processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm (which is similar to scoreboarding but makes use of register renaming) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism.

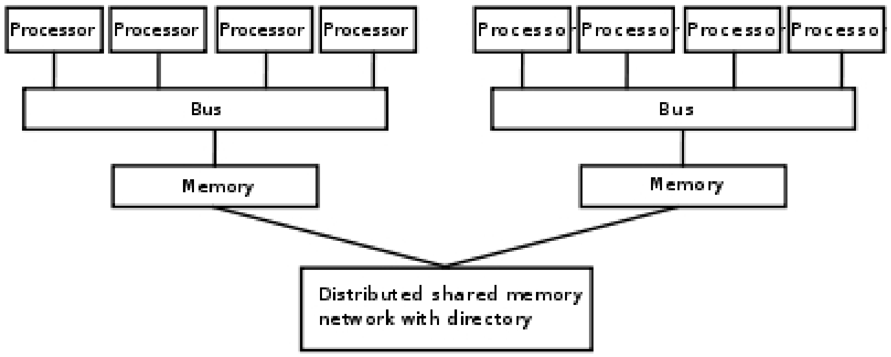
6.2.3 Task parallelism

Task parallelisms is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data". This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism involves the decomposition of a task into sub-tasks and then allocating each sub-task to a processor for execution. The processors would then execute these sub-tasks concurrently and often cooperatively. Task parallelism does not usually scale with the size of a problem.

6.3 Hardware

6.3.1 Memory and communication

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory.



A logical view of a non-uniform memory access (NUMA) architecture. Processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory.

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as uniform memory access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a non-uniform memory access (NUMA) architecture. Distributed memory systems have non-uniform memory access.

Computer systems make use of caches—small and fast memories located close to the processor which store temporary copies of memory values (nearby in both the physical and logical sense). Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Bus snooping is one of the most common methods for keeping track of which values are being accessed (and thus should be purged). Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared memory computer architectures do not scale as well as distributed memory systems do.

Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a hypercube with more than one processor at a node), or n-dimensional mesh.

Parallel computers based on interconnected networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.

6.3.2 Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

Multi-core computing

A multi-core processor is a processor that includes multiple processing units (called "cores") on the same chip. This processor differs from a superscalar processor, which includes multiple execution units and can issue multiple instructions per clock cycle from one instruction stream (thread); in contrast, a multi-core processor can issue multiple instructions per clock cycle from multiple instruction streams. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is a prominent multi-core processor. Each core in a multi-core processor can potentially be superscalar as well—that is, on every clock cycle, each core can issue multiple instructions from one thread.

Instruction pipelining (of which Intel's Hyper-Threading is the best known) was an early form of pseudo-multi-coreism. A processor capable of concurrent multithreading includes multiple execution units in the same processing unit—that is it has a superscalar architecture—and can issue multiple instructions per clock cycle from *multiple* threads. Temporal multithreading on the other hand includes a single execution unit in the same processing unit and can issue one instruction at a time from *multiple* threads.

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists.

Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel.

Cluster computing

A Beowulf cluster.

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. 87% of all Top500 supercomputers are clusters. The remaining are Massively Parallel Processors, explained below.

Because grid computing systems (described below) can easily handle embarrassingly parallel problems, modern clusters are typically designed to handle more difficult problems—problems that require nodes to share intermediate results with each other more often. This requires a high bandwidth and, more importantly, a low-latency interconnection network. Many historic and current supercomputers use customized high-performance network hardware specifically designed for cluster computing, such as the Cray Gemini network. As of 2014, most current supercomputers use some off-the-shelf standard network hardware, often Myrinet, InfiniBand, or Gigabit Ethernet.

Massively parallel computing

A cabinet from IBM's Blue Gene/L massively parallel supercomputer.

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having "far more" than 100 processors. In an MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect."

IBM's Blue Gene/L, the fifth fastest supercomputer in the world according to the June 2009 TOP500 ranking, is an MPP.

Grid computing

Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems. Many distributed computing applications have been created, of which SETI@home and Folding@home are the best-known examples.

Most grid computing applications use middleware (software that sits between the operating system and the application to manage network resources and standardize the software interface). The most common distributed computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, distributed computing software makes use of "spare cycles", performing computations at times when a computer is idling.

Specialized parallel computers

Within parallel computing, there are specialized parallel devices that remain niche areas of interest. While not domain-specific, they tend to be applicable to only a few classes of parallel problems.

Reconfigurable computing with field-programmable gate arrays

Reconfigurable computing is the use of a field-programmable gate array (FPGA) as a co-processor to a general-purpose computer. An FPGA is, in essence, a computer chip that can rewire itself for a given task.

FPGAs can be programmed with hardware description languages such as VHDL or Verilog. However, programming in these languages can be tedious. Several vendors have created C to HDL languages that attempt to emulate the syntax and semantics of the C programming language, with which most programmers are familiar. The best known C to HDL languages are Mitrion-C, Impulse C, DIME-C, and Handel-C. Specific subsets of SystemC based on C++ can also be used for this purpose.

AMD's decision to open its HyperTransport technology to third-party vendors has become the enabling technology for high-performance reconfigurable computing. According to Michael R. D'Amour, Chief Operating Officer of DRC Computer Corporation, "when we first walked into AMD, they called us 'the socket stealers.' Now they call us their partners."

General-purpose computing on graphics processing units (GPGPU)

Nvidia's Tesla GPGPU card

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.

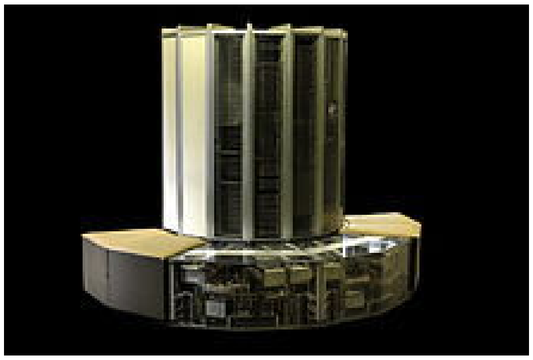
In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively. Other GPU programming languages include BrookGPU, PeakStream, and RapidMind. Nvidia has also released specific products for computation in their Tesla series. The technology consortium Khronos Group has released the OpenCL specification, which is a framework for writing programs that execute across platforms consisting of CPUs and GPUs. AMD, Apple, Intel, Nvidia and others are supporting OpenCL.

Application-specific integrated circuits

Several application-specific integrated circuit (ASIC) approaches have been devised for dealing with parallel applications.

Because an ASIC is (by definition) specific to a given application, it can be fully optimized for that application. As a result, for a given application, an ASIC tends to outperform a general-purpose computer. However, ASICs are created by UV photolithography. This process requires a mask set, which can be extremely expensive. A mask set can cost over a million US dollars. (The smaller the transistors required for the chip, the more expensive the mask will be.) Meanwhile, performance increases in general-purpose computing over time (as described by Moore's law) tend to wipe out these gains in only one or two chip generations. High initial cost, and the tendency to be overtaken by Moore's-law-driven general-purpose computing, has rendered ASICs unfeasible for most parallel computing applications. However, some have been built. One example is the PFLOPS RIKEN MDGRAPE-3 machine which uses custom ASICs for molecular dynamics simulation.

Vector processors



The Cray-1 is a vector processor.

A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. Vector processors have high-level operations that work on linear arrays of numbers or vectors. An example vector operation is $A = B \times C$, where A , B , and C are each 64-element vectors of 64-bit floating-point numbers. They are closely related to Flynn's SIMD classification.

Cray computers became famous for their vector-processing computers in the 1970s and 1980s. However, vector processors—both as CPUs and as full computer systems—have generally disappeared. Modern processor instruction sets do include some vector processing instructions, such as with Freescale Semiconductor's AltiVec and Intel's Streaming SIMD Extensions (SSE).

6.4 Software

6.4.1 Parallel programming languages

Concurrent programming languages, libraries, APIs, and parallel programming models (such as algorithmic skeletons) have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture—shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads and OpenMP are two of the most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API. One concept used in programming parallel programs is the future concept, where one part of a program promises to deliver a required datum to another part of a program at some future time.

CAPS entreprise and Pathscale are also coordinating their effort to make hybrid multi-core parallel programming (HMPP) directives an open standard called OpenHMPP. The OpenHMPP directive-based programming model offers a syntax to efficiently offload computations on hardware accelerators and to optimize data movement to/from the hardware memory. OpenHMPP directives describe remote procedure call (RPC) on an accelerator device (e.g. GPU) or more generally a set of cores. The directives annotate C or Fortran codes to describe two sets of functionalities: the offloading of procedures (denoted codelets) onto a remote device and the optimization of data transfers between the CPU main memory and the accelerator memory.

The rise of consumer GPUs has led to support for compute kernels, either in graphics APIs (referred to as compute shaders), in dedicated APIs (such as OpenCL), or in other language extensions.

6.4.2 Automatic parallelization

Automatic parallelization of a sequential program by a compiler is the "holy grail" of parallel computing, especially with the aforementioned limit of processor frequency. Despite decades of work by compiler researchers, automatic parallelization has had only limited success.

Mainstream parallel programming languages remain either explicitly parallel or (at best) partially implicit, in which a programmer gives the compiler directives for parallelization. A few fully implicit parallel programming languages exist—SISAL, Parallel Haskell, SequenceL, System C (for FPGAs), Mitrion-C, VHDL, and Verilog.

6.4.3 Application checkpointing

As a computer system grows in complexity, the mean time between failures usually decreases. Application checkpointing is a technique whereby the computer system takes a "snapshot" of the application —a record of all current resource allocations and variable states, akin to a core dump—; this information can be used to restore the program if the computer should fail. Application checkpointing means that the program has to restart from only its last checkpoint rather than the beginning. While checkpointing provides benefits in a variety of situations, it is especially useful in highly parallel systems with a large number of processors used in high performance computing.

6.5 Algorithmic methods

As parallel computers become larger and faster, we are now able to solve problems that had previously taken too long to run. Fields as varied as bioinformatics (for protein folding and sequence analysis) and economics (for mathematical finance) have taken advantage of parallel computing. Common types of problems in parallel computing applications include:

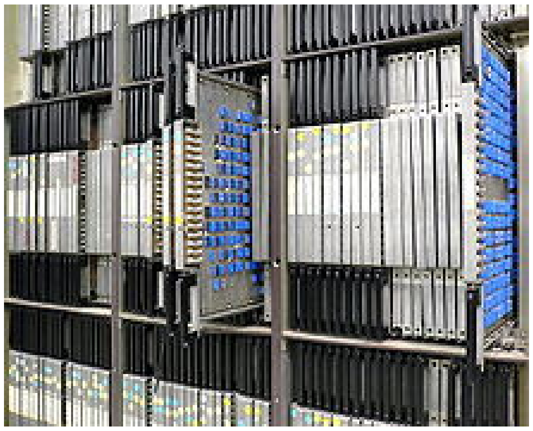
- Dense linear algebra
- Sparse linear algebra
- Spectral methods (such as Cooley–Tukey fast Fourier transform)
- *N*-body problems (such as Barnes–Hut simulation)
- structured grid problems (such as Lattice Boltzmann methods)
- Unstructured grid problems (such as found in finite element analysis)
- Monte Carlo method

- Combinational logic (such as brute-force cryptographic techniques)
- Graph traversal (such as sorting algorithms)
- Dynamic programming
- Branch and bound methods
- Graphical models (such as detecting hidden Markov models and constructing Bayesian networks)
- Finite-state machine simulation

6.6 Fault tolerance

Parallel computing can also be applied to the design of fault-tolerant computer systems, particularly via lockstep systems performing the same operation in parallel. This provides redundancy in case one component fails, and also allows automatic error detection and error correction if the results differ. These methods can be used to help prevent single-event upsets caused by transient errors. Although additional measures may be required in embedded or specialized systems, this method can provide a cost effective approach to achieve n-modular redundancy in commercial off-the-shelf systems.

6.7 History



ILLIAC IV, "the most infamous of supercomputers".

The origins of true (MIMD) parallelism go back to Luigi Federico Menabrea and his *Sketch of the Analytic Engine Invented by Charles Babbage* .

In April 1958, S. Gill (Ferranti) discussed parallel programming and the need for branching and waiting. Also in 1958, IBM researchers John Cocke and Daniel Slotnick discussed the use of parallelism in numerical calculations for the first time. Burroughs Corporation introduced the D825 in 1962, a four-processor computer that accessed up to 16 memory modules through a crossbar switch. In 1967, Amdahl and Slotnick published a debate about the feasibility of parallel processing at American Federation of Information Processing Societies Conference. It was during this debate that Amdahl's law was coined to define the limit of speed-up due to parallelism.

In 1969, Honeywell introduced its first Multics system, a symmetric multiprocessor system capable of running up to eight processors in parallel. C.mmp, a multi-processor project at Carnegie Mellon University in the 1970s, was among the first multiprocessors with more than a few processors. The first bus-connected multiprocessor with snooping caches was the Synapse N+1 in 1984.

SIMD parallel computers can be traced back to the 1970s. The motivation behind early SIMD computers was to amortize the gate delay of the processor's control unit over multiple instructions. In 1964, Slotnick had proposed building a massively parallel computer for the Lawrence Livermore National Laboratory. His design was funded by the US Air Force, which was the earliest SIMD parallel-computing effort, ILLIAC IV. The key to its design was a fairly high parallelism, with up to 256 processors, which allowed the machine to work on large datasets in what would later be known as vector processing. However, ILLIAC IV was called "the most infamous of supercomputers", because the project was only one-fourth completed, but took 11 years and cost almost four times the original estimate. When it was finally ready to run its first real application in 1976, it was outperformed by existing commercial supercomputers such as the Cray-1.

6.8 Biological brain as massively parallel computer

In the early 1970s, at the MIT Computer Science and Artificial Intelligence Laboratory, Marvin Minsky and Seymour Papert started developing the *Society of Mind* theory, which views the biological brain as massively parallel computer. In 1986, Minsky published *The Society of Mind* , which claims that “mind is formed from many little agents, each mindless by itself”. The theory attempts to explain how what we call intelligence could be a product of the interaction of non-intelligent parts. Minsky says that the biggest source of ideas about the theory came from his work in trying to create a machine that uses a robotic arm, a video camera, and a computer to build with children's blocks.

Similar models (which also view biological brain as massively parallel computer, i.e., the brain is made up of a constellation of independent or semi-independent agents) were also described by:

- Thomas R. Blakeslee,
- Michael S. Gazzaniga,
- Robert E. Ornstein,
- Ernest Hilgard,
- Michio Kaku,
- George Ivanovich Gurdjieff,
- Neurocluster Brain Model.

Chapter 7 Instruction pipelining

Basic five-stage pipeline

Clock cycle Instr. No.	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

In computer science, **Instruction pipelining** is a technique for implementing instruction-level parallelism within a single processor. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps (the eponymous "pipeline") performed by different processor units with different parts of instructions processed in parallel. It allows faster CPU throughput than would otherwise be possible at a given clock rate, but may increase latency due to the added overhead of the pipelining process itself.

7.1 Concept and motivation

Central processing units (CPUs) are driven by a clock. Each clock pulse need not do the same thing; rather, logic in the CPU directs successive pulses to different places to perform a useful sequence. There are many reasons that the entire execution of a machine instruction cannot happen at once; in pipelining, effects that cannot happen at the same time are made into dependent steps of the instruction.

For example, if one clock pulse latches a value into a register or begins a calculation, it will take some time for the value to be stable at the outputs of the register or for the calculation to complete. As another example, reading an instruction out of a memory unit cannot be done at the same time that an instruction writes a result to the same memory unit.

7.1.1 Number of steps

The number of dependent steps varies with the machine architecture. For example:

- The 1956-1961 IBM Stretch project proposed the terms Fetch, Decode, and Execute that have become common.
- The classic RISC pipeline comprises:
    1. Instruction fetch
    2. Instruction decode and register fetch
    3. Execute
    4. Memory access
    5. Register write back
- The Atmel AVR and the PIC microcontroller each have a two-stage pipeline.
- Many designs include pipelines as long as 7, 10 and even 20 stages (as in the Intel Pentium 4).
- The later "Prescott" and "Cedar Mill" Netburst cores from Intel, used in the last Pentium 4 models and their Pentium D and Xeon derivatives, have a long 31-stage pipeline.
- The Xelerated X10q Network Processor has a pipeline more than a thousand stages long, although in this case

200 of these stages represent independent CPUs with individually programmed instructions. The remaining stages are used to coordinate accesses to memory and on-chip function units.

As the pipeline is made "deeper" (with a greater number of dependent steps), a given step can be implemented with simpler circuitry, which may let the processor clock run faster. Such pipelines may be called *superpipelines.*

A processor is said to be *fully pipelined* if it can fetch an instruction on every cycle. Thus, if some instructions or conditions require delays that inhibit fetching new instructions, the processor is not fully pipelined.

7.2 History

Seminal uses of pipelining were in the ILLIAC II project and the IBM Stretch project, though a simple version was used earlier in the Z1 in 1939 and the Z3 in 1941.

Pipelining began in earnest in the late 1970s in supercomputers such as vector processors and array processors. One of the early supercomputers was the Cyber series built by Control Data Corporation. Its main architect, Seymour Cray, later headed Cray Research. Cray developed the XMP line of supercomputers, using pipelining for both multiply and add/subtract functions. Later, Star Technologies added parallelism (several pipelined functions working in parallel), developed by Roger Chen. In 1984, Star Technologies added the pipelined divide circuit developed by James Bradley. By the mid 1980s, pipelining was used by many different companies around the world.

Pipelining was not limited to supercomputers. In 1976, the Amdahl Corporation's 470 series general purpose mainframe had a 7-step pipeline, and a patented branch prediction circuit.

Today, pipelining and most of the above innovations are implemented by the instruction unit of most microprocessors.

7.2.1 Hazards

The model of sequential execution assumes that each instruction completes before the next one begins; this assumption is not true on a pipelined processor. A situation where the expected result is problematic is known as a hazard. Imagine the following two register instructions to a hypothetical processor:

1: add 1 to R5
2: copy R5 to R6

If the processor has the 5 steps listed in the initial illustration, instruction 1 would be fetched at time t_1 and its execution would be complete at t_5 . Instruction 2 would be fetched at t_2 and would be complete at t_6 . The first instruction might deposit the incremented number into R5 as its fifth step (register write back) at t_5 . But the second instruction might get the number from R5 (to copy to R6) in its second step (instruction decode and register fetch) at time t_3 . It seems that the first instruction would not have incremented the value by then. The above code invokes a hazard.

Writing computer programs in a compiled language might not raise these concerns, as the compiler could be designed to generate machine code that avoids hazards.

Workarounds

In some early DSP and RISC processors, the documentation advises programmers to avoid such dependencies in adjacent and nearly adjacent instructions (called delay slots), or declares that the second instruction uses an old value rather than the desired value (in the example above, the processor might counter-intuitively copy the unincremented value), or declares that the value it uses is undefined. The programmer may have unrelated work that the processor can do in the meantime; or, to ensure correct results, the programmer may insert NOPs into the code, partly negating the advantages of pipelining.

Solutions

Pipelined processors commonly use three techniques to work as expected when the programmer assumes that each instruction completes before the next one begins:

- Processors that can compute the presence of a hazard may *stall* , delaying processing of the second instruction (and subsequent instructions) until the values it requires as input are ready. This creates a *bubble* in the pipeline, also partly negating the advantages of pipelining.
- Some processors can not only compute the presence of a hazard but can compensate by having additional data paths that provide needed inputs to a computation step before a subsequent instruction would otherwise compute them, an attribute called operand forwarding.
- Some processors can determine that instructions other than the next sequential one are not dependent on the current ones and can be executed without hazards. Such processors may perform out-of-order execution.

7.2.2 Branches

A branch out of the normal instruction sequence often involves a hazard. Unless the processor can give effect to the branch in a single time cycle, the pipeline will continue fetching instructions sequentially. Such instructions cannot be allowed to take effect because the programmer has diverted control to another part of the program.

A conditional branch is even more problematic. The processor may or may not branch, depending on a calculation that has not yet occurred. Various processors may stall, may attempt branch prediction, and may be able to begin to execute two different program sequences (eager execution), both assuming the branch is and is not taken, discarding all work that pertains to the incorrect guess.

A processor with an implementation of branch prediction that usually makes correct predictions can minimize the performance penalty from branching. However, if branches are predicted poorly, it may create more work for the processor, such as flushing from the pipeline the incorrect code path that has begun execution before resuming execution at the correct location.

Programs written for a pipelined processor deliberately avoid branching to minimize possible loss of speed. For example, the programmer can handle the usual case with sequential execution and branch only on detecting unusual cases. Using programs such as gcov to analyze code coverage lets the programmer measure how often particular branches are actually executed and gain insight with which to optimize the code.

7.2.3 Special situations

Self-modifying programs

The technique of self-modifying code can be problematic on a pipelined processor. In this technique, one of the effects of a program is to modify its own upcoming instructions. If the processor has an instruction cache, the original instruction may already have been copied into a prefetch input queue and the modification will not take effect.

Uninterruptible instructions

An instruction may be uninterruptible to ensure its atomicity, such as when it swaps two items. A sequential processor permits interrupts between instructions, but a pipelining processor overlaps instructions, so executing an uninterruptible instruction renders portions of ordinary instructions uninterruptible too. The Cyrix coma bug would hang a single-core system using an infinite loop in which an uninterruptible instruction was always in the pipeline.

7.3 Design considerations

Speed

Pipelining keeps all portions of the processor occupied and increases the amount of useful work the processor can do in a given time. Pipelining typically reduces the processor's cycle time and increases the throughput of instructions. The speed advantage is diminished to the extent that execution encounters hazards that require execution to slow below its ideal rate. A non-pipelined processor executes only a single instruction at a time. The start of the next instruction is delayed not based on hazards but unconditionally.

A pipelined processor's need to organize all its work into modular steps may require the duplication of registers, which increases the latency of some instructions.

Economy

By making each dependent step simpler, pipelining can enable complex operations more economically than adding complex circuitry, such as for numerical calculations. However, a processor that declines to pursue increased speed with pipelining may be simpler and cheaper to manufacture.

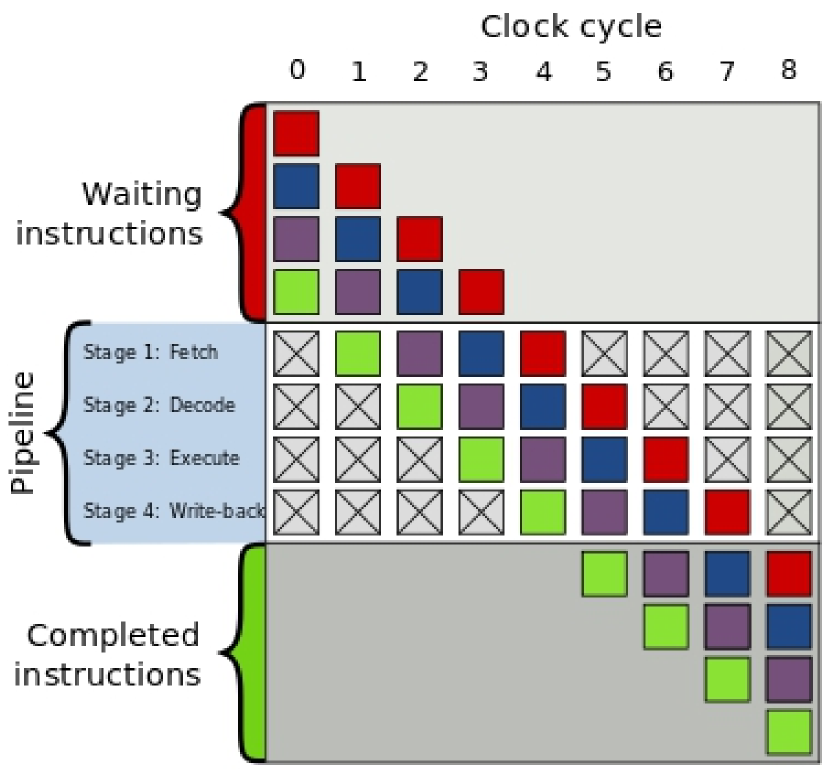
Predictability

Compared to environments where the programmer needs to avoid or work around hazards, use of a non-pipelined processor may make it easier to program and to train programmers. The non-pipelined processor also makes it easier to predict the exact timing of a given sequence of instructions.

7.4 Illustrated example

To the right is a generic pipeline with four stages: fetch, decode, execute and write-back. The top gray box is the list of instructions waiting to be executed, the bottom gray box is the list of instructions that have had their execution completed, and the middle white box is the pipeline.

The execution is as follows:

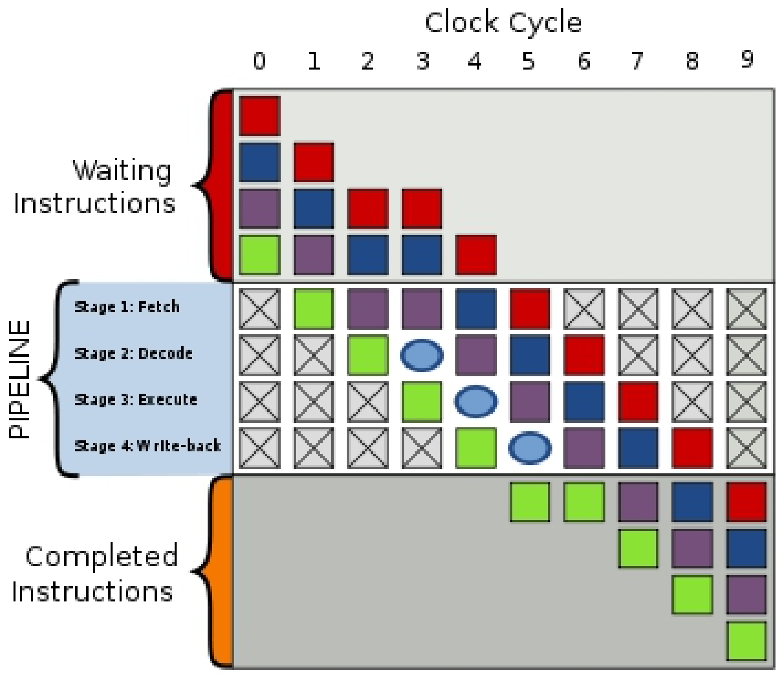


Generic 4-stage pipeline; the colored boxes represent instructions independent of each other

Clock	Execution
0	• Four instructions are waiting to be executed
1	• The green instruction is fetched from memory
2	• The green instruction is decoded • The purple instruction is fetched from memory
3	• The green instruction is executed (actual operation is performed) • The purple instruction is decoded • The blue instruction is fetched

4	• The green instruction's results are written back to the register file or memory • The purple instruction is executed • The blue instruction is decoded • The red instruction is fetched
5	• The execution of green instruction is completed • The purple instruction is written back • The blue instruction is executed • The red instruction is decoded
6	• The execution of purple instruction is completed • The blue instruction is written back • The red instruction is executed
7	• The execution of blue instruction is completed • The red instruction is written back
8	• The execution of red instruction is completed
9	• The execution of all four instructions is completed

7.4.1 Pipeline bubble



A bubble in cycle 3 delays execution.

A pipelined processor may deal with hazards by stalling and creating a bubble in the pipeline, resulting in one or more cycles in which nothing useful happens.

In the illustration at right, in cycle 3, the processor cannot decode the purple instruction, perhaps because the processor determines that decoding depends on results produced by the execution of the green instruction. The green instruction can proceed to the Execute stage and then to the Write-back stage as scheduled, but the purple instruction is stalled for one cycle at the Fetch stage. The blue instruction, which was due to be fetched during cycle 3, is stalled for one cycle, as is the red instruction after it.

Because of the bubble (the blue ovals in the illustration), the processor's Decode circuitry is idle during cycle 3. Its Execute circuitry is idle during cycle 4 and its Write-back circuitry is idle during cycle 5.

When the bubble moves out of the pipeline (at cycle 6), normal execution resumes. But everything now is one cycle late. It will take 8 cycles (cycle 1 through 8) rather than 7 to completely execute the four instructions shown in colors.

Chapter 8 Multi-core processor

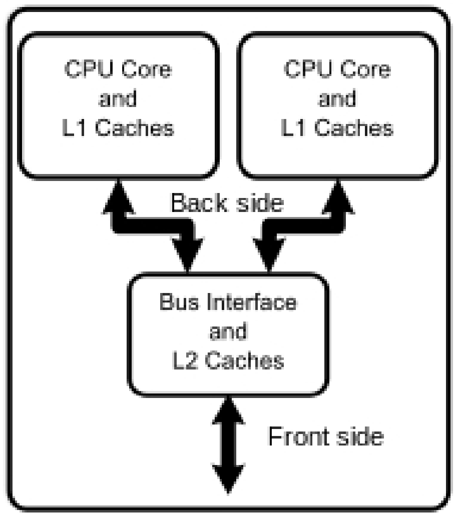
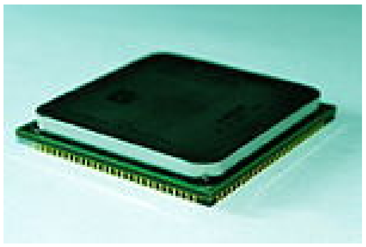


Diagram of a generic dual-core processor with CPU-local level-1 caches and a shared, on-die level-2 cache.

An Intel Core 2 Duo E6750 dual-core processor.



An AMD Athlon X2 6400+ dual-core processor.

A **multi-core processor** is a single computing component with two or more independent processing units called cores, which read and execute program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run multiple instructions on separate cores at the same time, increasing overall speed for programs amenable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP) or onto multiple dies in a single chip package. The microprocessors currently used in almost all personal computers are multi-core.

A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores; heterogeneous multi-core systems have cores that are not identical (e.g. big.LITTLE have heterogeneous cores that share the same instruction set, while AMD Accelerated Processing Units have cores that don't even share the same instruction set). Just as with single-processor systems, cores in multi-core systems may implement architectures such as VLIW, superscalar, vector, or multithreading.

Multi-core processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU).

The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can run in parallel simultaneously on multiple cores; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main-system memory. Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem. The parallelization of software is a significant ongoing topic of research.

8.1 Terminology

The terms *multi-core* and *dual-core* most commonly refer to some sort of central processing unit (CPU), but are sometimes also applied to digital signal processors (DSP) and system on a chip (SoC). The terms are generally used only to refer to multi-core microprocessors that are manufactured on the *same* integrated circuit die; separate microprocessor dies in the same package are generally referred to by another name, such as *multi-chip module* . This article uses the terms "multi-core" and "dual-core" for CPUs manufactured on the *same* integrated circuit, unless otherwise noted.

In contrast to multi-core systems, the term *multi-CPU* refers to multiple physically separate processing-units (which often contain special circuitry to facilitate communication between each other).

The terms *many-core* and *massively multi-core* are sometimes used to describe multi-core architectures with an especially high number of cores (tens to thousands).

Some systems use many soft microprocessor cores placed on a single FPGA. Each "core" can be considered a "semiconductor intellectual property core" as well as a CPU core.

8.2 Development

While manufacturing technology improves, reducing the size of individual gates, physical limits of semiconductor-based microelectronics have become a major design concern. These physical limitations can cause significant heat dissipation and data synchronization problems. Various other methods are used to improve CPU performance. Some *instruction-level parallelism* (ILP) methods such as superscalar pipelining are suitable for many applications, but are inefficient for others that contain difficult-to-predict code. Many applications are better suited to *thread-level parallelism* (TLP) methods, and multiple independent CPUs are commonly used to increase a system's overall TLP. A combination of increased available space (due to refined manufacturing processes) and the demand for increased TLP led to the development of multi-core CPUs.

8.2.1 Commercial incentives

Several business motives drive the development of multi-core architectures. For decades, it was possible to improve performance of a CPU by shrinking the area of the integrated circuit (IC), which reduced the cost per device on the IC. Alternatively, for the same circuit area, more transistors could be used in the design, which increased functionality, especially for complex instruction set computing (CISC) architectures. Clock rates also increased by orders of magnitude in the decades of the late 20th century, from several megahertz in the 1980s to several gigahertz in the early 2000s.

As the rate of clock speed improvements slowed, increased use of parallel computing in the form of multi-core processors has been pursued to improve overall processing performance. Multiple cores were used on the same CPU chip, which could then lead to better sales of CPU chips with two or more cores. For example, Intel has produced a 48-core processor for research in cloud computing; each core has an x86 architecture.

8.2.2 Technical factors

Since computer manufacturers have long implemented symmetric multiprocessing (SMP) designs using discrete CPUs, the issues regarding implementing multi-core processor architecture and supporting it with software are well known.

Additionally:

- Using a proven processing-core design without architectural changes reduces design risk significantly.
- For general-purpose processors, much of the motivation for multi-core processors comes from greatly diminished gains in processor performance from increasing the operating frequency. This is due to three primary factors:
    1. The *memory wall* ; the increasing gap between processor and memory speeds. This, in effect, pushes for cache sizes to be larger in order to mask the latency of memory. This helps only to the extent that memory bandwidth is not the bottleneck in performance.

2. The *ILP wall* ; the increasing difficulty of finding enough parallelism in a single instruction stream to keep a high-performance single-core processor busy.
3. The *power wall* ; the trend of consuming exponentially increasing power (and thus also generating exponentially increasing heat) with each factorial increase of operating frequency. This increase can be mitigated by "shrinking" the processor by using smaller traces for the same logic. The *power wall* poses manufacturing, system design and deployment problems that have not been justified in the face of the diminished gains in performance due to the *memory wall* and *ILP wall* .

In order to continue delivering regular performance improvements for general-purpose processors, manufacturers such as Intel and AMD have turned to multi-core designs, sacrificing lower manufacturing-costs for higher performance in some applications and systems. Multi-core architectures are being developed, but so are the alternatives. An especially strong contender for established markets is the further integration of peripheral functions into the chip.

8.2.3 Advantages

The proximity of multiple CPU cores on the same die allows the cache coherency circuitry to operate at a much higher clock rate than what is possible if the signals have to travel off-chip. Combining equivalent CPUs on a single die significantly improves the performance of cache snoop (alternative: Bus snooping) operations. Put simply, this means that signals between different CPUs travel shorter distances, and therefore those signals degrade less. These higher-quality signals allow more data to be sent in a given time period, since individual signals can be shorter and do not need to be repeated as often.

Assuming that the die can physically fit into the package, multi-core CPU designs require much less printed circuit board (PCB) space than do multi-chip SMP designs. Also, a dual-core processor uses slightly less power than two coupled single-core processors, principally because of the decreased power required to drive signals external to the chip. Furthermore, the cores share some circuitry, like the L2 cache and the interface to the front-side bus (FSB). In terms of competing technologies for the available silicon die area, multi-core design can make use of proven CPU core library designs and produce a product with lower risk of design error than devising a new wider-core design. Also, adding more cache suffers from diminishing returns.

Multi-core chips also allow higher performance at lower energy. This can be a big factor in mobile devices that operate on batteries. Since each core in a multi-core CPU is generally more energy-efficient, the chip becomes more efficient than having a single large monolithic core. This allows higher performance with less energy. A challenge in this, however, is the additional overhead of writing parallel code.

8.2.4 Disadvantages

Maximizing the usage of the computing resources provided by multi-core processors requires adjustments both to the operating system (OS) support and to existing application software. Also, the ability of multi-core processors to increase application performance depends on the use of multiple threads within applications.

Integration of a multi-core chip can lower the chip production yields. They are also more difficult to manage thermally than lower-density single-core designs. Intel has partially countered this first problem by creating its quad-core designs by combining two dual-core ones on a single die with a unified cache, hence any two working dual-core dies can be used, as opposed to producing four cores on a single die and requiring all four to work to produce a quad-core CPU. From an architectural point of view, ultimately, single CPU designs may make better use of the silicon surface area than multiprocessing cores, so a development commitment to this architecture may carry the risk of obsolescence. Finally, raw processing power is not the only constraint on system performance. Two processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage. In a 2009 report, Dr Jun Ni showed that if a single core is close to being memory-bandwidth limited, then going to dual-core might give 30% to 70% improvement; if memory bandwidth is not a problem, then a 90% improvement can be expected; however, Amdahl's law makes this claim dubious. It would be possible for an application that used two CPUs to end up running faster on a single-core one if communication between the CPUs was the limiting factor, which would count as more than 100% improvement.

8.3 Hardware

8.3.1 Trends

The trend in processor development has been towards an ever-increasing number of cores, as processors with hundreds or even thousands of cores become theoretically possible. In addition, multi-core chips mixed with simultaneous multithreading, memory-on-chip, and special-purpose "heterogeneous" (or asymmetric) cores promise further performance and efficiency gains, especially in processing multimedia, recognition and networking applications. For example, a big.LITTLE core includes a high-performance core (called 'big') and a low-power core (called 'LITTLE'). There is also a trend towards improving energy-efficiency by focusing on performance-per-watt with advanced fine-grain or ultra fine-grain power management and dynamic voltage and frequency scaling (i.e. laptop computers and portable media players).

Chips designed from the outset for a large number of cores (rather than having evolved from single core designs) are sometimes referred to as manycore designs, emphasising qualitative differences.

8.3.2 Architecture

The composition and balance of the cores in multi-core architecture show great variety. Some architectures use one core design repeated consistently ("homogeneous"), while others use a mixture of different cores, each optimized for a different, "heterogeneous" role.

The article "CPU designers debate multi-core future" by Rick Merritt, EE Times 2008, includes these comments:

Chuck Moore [...] suggested computers should be like cellphones, using a variety of specialty cores to run modular software scheduled by a high-level applications programming interface.

[...] Atsushi Hasegawa, a senior chief engineer at Renesas, generally agreed. He suggested the cellphone's use of many specialty cores working in concert is a good model for future multi-core designs.

[...] Anant Agarwal, founder and chief executive of startup Tilera, took the opposing view. He said multi-core chips need to be homogeneous collections of general-purpose cores to keep the software model simple.

8.4 Software effects

An outdated version of an anti-virus application may create a new thread for a scan process, while its GUI thread waits for commands from the user (e.g. cancel the scan). In such cases, a multi-core architecture is of little benefit for the application itself due to the single thread doing all the heavy lifting and the inability to balance the work evenly across multiple cores. Programming truly multithreaded code often requires complex co-ordination of threads and can easily introduce subtle and difficult-to-find bugs due to the interweaving of processing on data shared between threads (see thread-safety). Consequently, such code is much more difficult to debug than single-threaded code when it breaks. There has been a perceived lack of motivation for writing consumer-level threaded applications because of the relative rarity of consumer-level demand for maximum use of computer hardware. Although threaded applications incur little additional performance penalty on single-processor machines, the extra overhead of development has been difficult to justify due to the preponderance of single-processor machines. Also, serial tasks like decoding the entropy encoding algorithms used in video codecs are impossible to parallelize because each result generated is used to help create the next result of the entropy decoding algorithm.

Given the increasing emphasis on multi-core chip design, stemming from the grave thermal and power consumption problems posed by any further significant increase in processor clock speeds, the extent to which software can be multithreaded to take advantage of these new chips is likely to be the single greatest constraint on computer performance in the future. If developers are unable to design software to fully exploit the resources provided by multiple cores, then they will ultimately reach an insurmountable performance ceiling.

The telecommunications market had been one of the first that needed a new design of parallel datapath packet processing because there was a very quick adoption of these multiple-core processors for the datapath and the control plane. These MPUs are going to replace the traditional Network Processors that were based on proprietary microcode or picocode.

Parallel programming techniques can benefit from multiple cores directly. Some existing parallel programming models such as Cilk Plus, OpenMP, OpenHMPP, FastFlow, Skandium, MPI, and Erlang can be used on multi-core platforms. Intel introduced a new abstraction for C++ parallelism called TBB. Other research efforts include the Codeplay Sieve System, Cray's Chapel, Sun's Fortress, and IBM's X10.

Multi-core processing has also affected the ability of modern computational software development. Developers programming in newer languages might find that their modern languages do not support multi-core functionality. This then requires the use of numerical libraries to access code written in languages like C and Fortran, which perform math computations faster than newer languages like C#. Intel's MKL and AMD's ACML are written in these native languages and take advantage of multi-core processing. Balancing the application workload across processors can be problematic, especially if they have different performance characteristics. There are different conceptual models to deal with the problem, for example using a coordination language and program building blocks (programming libraries or higher-order functions). Each block can have a different native implementation for each processor type. Users simply program using these abstractions and an intelligent compiler chooses the best implementation based on the context.

Managing concurrency acquires a central role in developing parallel applications. The basic steps in designing parallel applications are:

Partitioning

The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.

Communication

The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the communication phase of a design.

Agglomeration

In the third stage, development moves from the abstract toward the concrete. Developers revisit decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer. In particular, developers consider whether it is useful to combine, or agglomerate, tasks identified by the partitioning phase, so as to provide a smaller number of tasks, each of greater size. They also determine whether it is worthwhile to replicate data and computation.

Mapping

In the fourth and final stage of the design of parallel algorithms, the developers specify where each task is to execute. This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling.

On the other hand, on the server side, multi-core processors are ideal because they allow many users to connect to a site simultaneously and have independent threads of execution. This allows for Web servers and application servers that have much better throughput.

8.4.1 Licensing

Vendors may license some software "per processor". This can give rise to ambiguity, because a "processor" may consist either of a single core or of a combination of cores.

- Initially, for some of its enterprise software, Microsoft continued to use a per-socket licensing system. However, for some software such as BizTalk Server 2013, SQL Server 2014, and Windows Server 2016, Microsoft has shifted to per-core licensing.
- Oracle Corporation counts an AMD X2 or an Intel dual-core CPU as a single processor but uses other metrics for other types, especially for processors with more than two cores.

8.5 Embedded applications

Embedded computing operates in an area of processor technology distinct from that of "mainstream" PCs. The same technological drives towards multi-core apply here too. Indeed, in many cases the application is a "natural" fit for multi-core technologies, if the task can easily be partitioned between the different processors.

In addition, embedded software is typically developed for a specific hardware release, making issues of software portability, legacy code or supporting independent developers less critical than is the case for PC or enterprise computing. As a result, it is easier for developers to adopt new technologies and as a result there is a greater variety of multi-core processing architectures and suppliers.

As of 2010, multi-core network processing devices have become mainstream, with companies such as Freescale Semiconductor, Cavium Networks, Wintegra and Broadcom all manufacturing products with eight processors. For the system developer, a key challenge is how to exploit all the cores in these devices to achieve maximum networking performance at the system level, despite the performance limitations inherent in an SMP operating system. To address this issue, companies such as 6WIND provide portable packet processing software designed so that the networking data plane runs in a fast path environment outside the OS.

In digital signal processing the same trend applies: Texas Instruments has the three-core TMS320C6488 and four-core TMS320C5441, Freescale the four-core MSC8144 and six-core MSC8156 (and both have stated they are working on eight-core successors). Newer entries include the Storm-1 family from Stream Processors, Inc with 40 and 80 general purpose ALUs per chip, all programmable in C as a SIMD engine and Picochip with three-hundred processors on a single die, focused on communication applications.

As of 2016 heterogeneous multi-core solutions are becoming more common: Xilinx Zynq UltraScale+ MPSoC has Quad-core ARM Cortex-A53 and Dual-core ARM Cortex-R5. Software solutions such as OpenAMP are being used to help with inter processor communication.

8.6 Hardware examples

8.6.1 Commercial

- Adapteva Epiphany, a many-core processor architecture which allows up to 4096 processors on-chip, although only a 16 core version has been commercially produced.
- Aeroflex Gaisler LEON3, a multi-core SPARC that also exists in a fault-tolerant version.
- Ageia PhysX, a multi-core physics processing unit.
- Ambric Am2045, a 336-core Massively Parallel Processor Array (MPPA)
- AMD
    - A-Series, dual-, triple-, and quad-core of Accelerated Processor Units (APU).
    - Athlon 64, Athlon 64 FX and Athlon 64 X2 family, dual-core desktop processors.

  - Athlon II, dual-, triple-, and quad-core desktop processors.
  - FX-Series, quad-, 6-, and 8-core desktop processors.
  - Opteron, dual-, quad-, 6-, 8-, 12-, and 16-core server/workstation processors.
  - Phenom, dual-, triple-, and quad-core processors.
  - Phenom II, dual-, triple-, quad-, and 6-core desktop processors.
  - Sempron X2, dual-core entry level processors.
  - Turion 64 X2, dual-core laptop processors.
  - Ryzen, quad-, 6-, 8-, 12-, 16-, 24-, and 32-core desktop processors.
  - Epyc, 8-, 16-, 24-, and 32-core server processors.
  - Radeon and FireStream multi-core GPU/GPGPU (10 cores, 16 5-issue wide superscalar stream processors per core)
- Analog Devices Blackfin BF561, a symmetrical dual-core processor
- ARM MPCore is a fully synthesizable multi-core container for ARM11 MPCore and ARM Cortex-A9 MPCore processor cores, intended for high-performance embedded and entertainment applications.
- ASOCS ModemX, up to 128 cores, wireless applications.
- Azul Systems
  - Vega 1, a 24-core processor, released in 2005.
  - Vega 2, a 48-core processor, released in 2006.
  - Vega 3, a 54-core processor, released in 2008.
- Broadcom SiByte SB1250, SB1255, SB1455; BCM 2836 quad-core ARM SoC (designed for the Raspberry Pi 2)

- Cadence Design Systems Tensilica Xtensa LX6, available in a dual-core configuration in Espressif Systems's ESP32
- ClearSpeed
    - CSX700, 192-core processor, released in 2008 (32/64-bit floating point; Integer ALU)
- Cradle Technologies CT3400 and CT3600, both multi-core DSPs.
- Cavium Networks Octeon, a 32-core MIPS MPU.
- Coherent Logix hx3100 Processor, a 100-core DSP/GPP processor
- Freescale Semiconductor QorIQ series processors, up to 8 cores, Power Architecture MPU.
- Hewlett-Packard PA-8800 and PA-8900, dual core PA-RISC processors.
- IBM
    - POWER4, a dual-core PowerPC processor, released in 2001.
    - POWER5, a dual-core PowerPC processor, released in 2004.
    - POWER6, a dual-core PowerPC processor, released in 2007.
    - POWER7, a 4,6,8-core PowerPC processor, released in 2010.
    - POWER8, a 12-core PowerPC processor, released in 2013.
    - PowerPC 970MP, a dual-core PowerPC processor, used in the Apple Power Mac G5.
    - Xenon, a triple-core, SMT-capable, PowerPC microprocessor used in the Microsoft Xbox 360 game console.
    - z10, a quad-core z/Architecture processor, released in 2008

    - z196, a quad-core z/Architecture processor, released in 2010
    - zEC12, a six-core z/Architecture processor, released in 2012
    - z13, an eight-core z/Architecture processor, released in 2015
- Infineon
    - AURIX
    - Danube, a dual-core, MIPS-based, home gateway processor.
- Intel
    - Atom, single, dual-core and quad-core processors for netbook, tablets and smartphones systems.
    - Celeron Dual-Core, the first dual-core processor for the budget/entry-level market.
    - Core Duo, a dual-core processor.
    - Core 2 Duo, a dual-core processor.
    - Core 2 Quad, 2 dual-core dies packaged in a multi-chip module.
    - Core i3, Core i5, Core i7 and Core i9 a family of dual-, quad-, 6-, 8-, 10-, 12-, 16-, and 18-core processors, the successor of the Core 2 Duo and the Core 2 Quad.
    - Itanium 2, a dual-core processor.
    - Pentium D, 2 single-core dies packaged in a multi-chip module.
    - Pentium Extreme Edition, 2 single-core dies packaged in a multi-chip module.
    - Pentium Dual-Core, a dual-core processor.
    - Teraflops Research Chip (Polaris), a 3.16 GHz, 80-core processor prototype, which the company originally stated would be released by 2011.

    - Xeon dual-, quad-, 6-, 8-, 10-, 12-, 14-, 15-, 16-, 18-, 22-, and 24- core processors.
    - Xeon Phi 57-core, 60-core and 61-core processors.
- IntellaSys
    - SEAforth 40C18, a 40-core processor
    - SEAforth24, a 24-core processor designed by Charles H. Moore
- Kalray
    - MPPA-256, 256-core processor, released 2012 (256 usable VLIW cores, Network-on-Chip (NoC), 32/64-bit IEEE 754 compliant FPU)
- NetLogic Microsystems
    - XLP, a 32-core, quad-threaded MIPS64 processor
    - XLR, an eight-core, quad-threaded MIPS64 processor
    - XLS, an eight-core, quad-threaded MIPS64 processor
- Nvidia
    - GeForce 9 multi-core GPU (8 cores, 16 scalar stream processors per core)
    - GeForce 200 multi-core GPU (10 cores, 24 scalar stream processors per core)
    - Tesla multi-core GPGPU (10 cores, 24 scalar stream processors per core)
- Parallax Propeller P8X32, an eight-core microcontroller.
- picoChip PC200 series 200–300 cores per device for DSP & wireless
- Plurality HAL series tightly coupled 16-256 cores, L1 shared memory, hardware synchronized processor.
- Rapport Kilocore KC256, a 257-core microcontroller with a PowerPC core and 256 8-bit "processing elements".

- SiCortex "SiCortex node" has six MIPS64 cores on a single chip.
- Sony/IBM/Toshiba's Cell processor, a nine-core processor with one general purpose PowerPC core and eight specialized SPUs (Synergystic Processing Unit) optimized for vector operations used in the Sony PlayStation 3
- Sun Microsystems
    - MAJC 5200, two-core VLIW processor
    - UltraSPARC IV and UltraSPARC IV+, dual-core processors.
    - UltraSPARC T1, an eight-core, 32-thread processor.
    - UltraSPARC T2, an eight-core, 64-concurrent-thread processor.
    - UltraSPARC T3, a sixteen-core, 128-concurrent-thread processor.
    - SPARC T4, an eight-core, 64-concurrent-thread processor.
    - SPARC T5, a sixteen-core, 128-concurrent-thread processor.
- Texas Instruments
    - TMS320C80 MVP, a five-core multimedia video processor.
    - TMS320TMS320C66, 2,4,8 core dsp.
- Tilera
    - TILE64, a 64-core 32-bit processor
    - TILE-Gx, a 72-core 64-bit processor
- XMOS Software Defined Silicon quad-core XS1-G4

8.6.2 Free

- OpenSPARC

8.6.3 Academic

- MIT, 16-core RAW processor
- University of California, Davis, Asynchronous array of simple processors (AsAP)
    - 36-core 610 MHz AsAP
    - 167-core 1.2 GHz AsAP2
- University of Washington, Wavescalar processor
- University of Texas, Austin, TRIPS processor
- Linköping University, Sweden, ePUMA processor
- UC Davis, KiloCore, a 1000 core 1.78 GHz processor on a 32 nm IBM process

8.7 Benchmarks

The research and development of multicore processors often compares many options, and benchmarks are developed to help such evaluations. Existing benchmarks include SPLASH-2, PARSEC, and COSMIC for heterogeneous systems.

8.8 Notes

1. **^** Digital signal processors (DSPs) have used multi-core architectures for much longer than high-end general-purpose processors. A typical example of a DSP-specific implementation would be a combination of a RISC CPU and a DSP MPU. This allows for the design of products that require a general-purpose processor for user interfaces and a DSP for real-time data processing; this type of design is common in mobile phones. In other applications, a growing number of companies have developed multi-core DSPs with very large numbers of processors.
2. **^** Two types of operating systems are able to use a dual-CPU multiprocessor: partitioned multiprocessing and symmetric multiprocessing (SMP). In a partitioned architecture, each CPU boots into separate segments of physical memory and operate independently; in an SMP

OS, processors work in a shared space, executing threads within the OS independently.