Central processing unit

A **central processing unit** (**CPU**), also called a **central processor** or **main processor**, is the most important processor in a given computer. Its electronic circuitry executes instructions of a computer program, such as arithmetic, logic, controlling, and input/output (I/O) operations. This role contrasts with that of external components, such as main memory and I/O circuitry,[1] and specialized coprocessors such as graphics processing units (GPUs).

An Intel 80486DX2 CPU, as seen from above

The form, design, and implementation of CPUs have changed over time, 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), decoding and execution (of instructions) by directing the coordinated operations of the ALU, registers and other components.

Bottom side of an Intel 80486DX2, showing its pins

Most modern CPUs are implemented on integrated circuit (IC) microprocessors, with one or more CPUs on a single IC chip. Microprocessor chips with multiple CPUs are multi-core processors. The individual physical CPUs, **processor cores**, can also be multithreaded to create additional virtual or logical CPUs.[2]

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).

Array processors or vector processors have multiple processors that operate in parallel, with no unit considered central. Virtual CPUs are an abstraction of dynamical aggregated computational resources.[3]
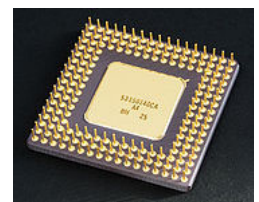
History

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".[4] The "central processing unit" term has been in use since as early as 1955.[5][6] 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.[7] 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.[8] 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.[9] This overcame a severe limitation of ENIAC, which was the considerable time and effort required to reconfigure the computer to perform a new task.[10] With von Neumann's design, the program that EDVAC ran could be changed simply by changing the contents of the memory. EDVAC was not the first stored-program computer; the Manchester Baby, which was a small-scale experimental stored-program computer, ran its first program on 21 June 1948[11] and the Manchester Mark 1 ran its first program during the night of 16–17 June 1949.[12]

Early CPUs were custom designs used as part of a larger and sometimes distinctive computer.[13] 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

EDVAC, one of the first stored-program computers

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.[14] 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[15] to cellphones,[16] and sometimes even in toys.[17][18]

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.[19] The so-called Harvard architecture of the Harvard Mark I, which was completed before EDVAC,[20][21] also used a stored-program design using punched paper tape rather than electronic memory.[22] 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.[23] 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.[24]

Relays and vacuum tubes (thermionic tubes) were commonly used as switching elements;[25][26] 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. Vacuum-tube computers such as EDVAC tended to average eight hours between failures, whereas relay computers like the (slower, but earlier) Harvard Mark I failed very rarely.[6] 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.[27]

Transistor CPUs

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.[28] With this improvement, more complex and reliable CPUs were built onto one or several printed circuit boards containing discrete (individual) components.
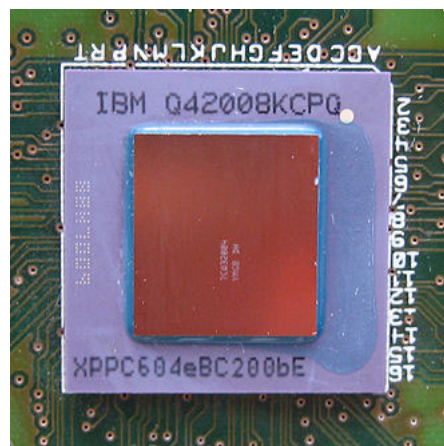

IBM PowerPC 604e processor

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.[29] 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.[30] 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.[31][32] In 1965, Digital Equipment Corporation (DEC) introduced another influential computer aimed at the scientific and research markets, the PDP-8.[33]

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.[34] 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.[35] Additionally, while discrete transistor and IC CPUs were in heavy usage, new high-performance designs like single instruction, multiple data (SIMD) vector processors began to appear.[36] These early experimental designs later gave rise to the era of specialized supercomputers like those made by Cray Inc and Fujitsu Ltd.[36]

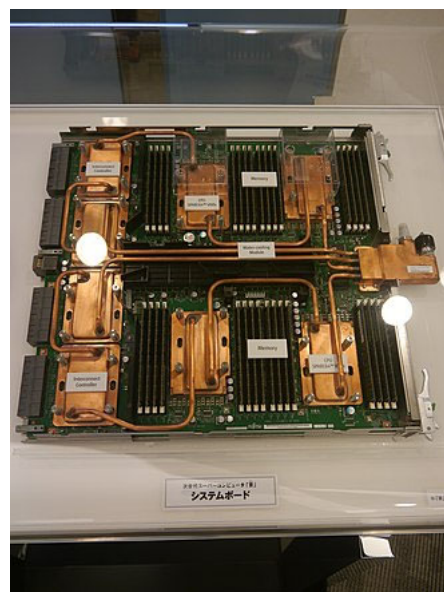Fujitsu board with SPARC64 VIIIfx processors

Small-scale integration CPUs

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.[37] 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.[38]

IBM's System/370, follow-on to the System/360, used SSI ICs rather than Solid Logic

Technology discrete-transistor modules.[39][40] 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,[41] and their extremely popular PDP-11 line was originally built with SSI ICs but was eventually implemented with LSI components once these became practical.

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).[42][43] The

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

only way to build LSI chips, which are chips with a hundred or more gates, was to build them using a metal–oxide–semiconductor (MOS) semiconductor manufacturing process (either PMOS logic, NMOS logic, or CMOS logic). However, some companies continued to build processors out of bipolar transistor–transistor logic (TTL) chips because bipolar junction transistors were faster than MOS chips up until the 1970s (a few companies such as Datapoint continued to build processors out of TTL chips until the early 1980s).[43] In the 1960s, MOS ICs were slower and initially considered useful only in applications that required low power.[44][45] Following the development of silicon-gate MOS technology by Federico Faggin at Fairchild Semiconductor in 1968, MOS ICs largely replaced bipolar TTL as the standard chip technology in the early 1970s.[46]

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.[47] 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.[48]

Microprocessors

Since microprocessors were first introduced they have almost completely overtaken all other central processing unit implementation methods. The first commercially available microprocessor, made in 1971, was the Intel 4004, and the first widely used microprocessor, made in 1974, was the Intel 8080. 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[a] to microprocessors. Several CPUs (denoted *cores*) can be combined in a single processing chip.[49]

Previous generations of CPUs were implemented as discrete components and numerous small integrated circuits (ICs) on one or more circuit boards.[50] Microprocessors, on the other hand, are CPUs manufactured on a very small number of ICs; usually just one.[51] 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.[52][53] This has allowed synchronous microprocessors to have clock rates ranging from tens of megahertz to several
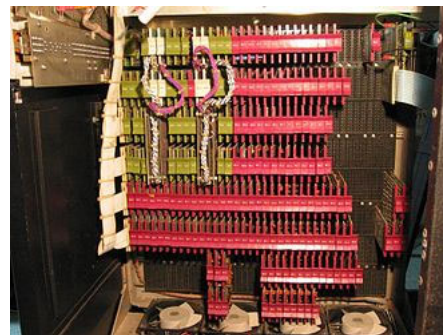
gigahertz. Additionally, the ability to construct exceedingly small transistors on an IC has increased the complexity and number of transistors in a single CPU many fold. This widely observed trend is described by Moore's law, which had proven to be a fairly accurate predictor of the growth of CPU (and other IC) complexity until 2016.[54][55]

While the complexity, size, construction and general form of CPUs have changed enormously since 1950,[56] 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.[57][b] 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.[59][60] 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.

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 microcontrollers). 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.[c] 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
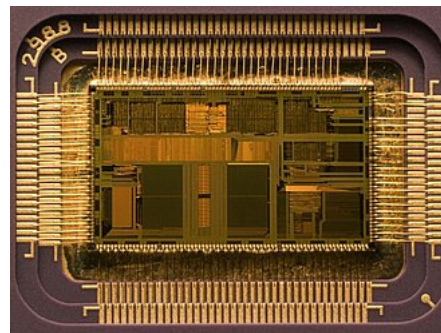
Die of an Intel 80486DX2 microprocessor (actual size: 12 × 6.75 mm) in its packaging
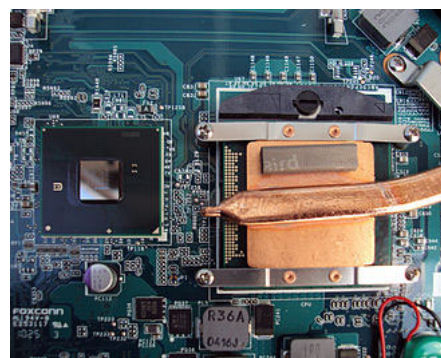
Intel Core i5 CPU on a Vaio E series laptop motherboard (on the right, beneath the heat pipe)

Inside of a laptop, with the CPU removed from socket

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.

Fetch

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 the program counter (PC; called the "instruction pointer" in Intel x86 microprocessors), 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.[d] 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).

Decode

The instruction that the CPU fetches from memory determines what the CPU will do. In the decode step, performed by binary decoder 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).[e] 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 binary decoder 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.
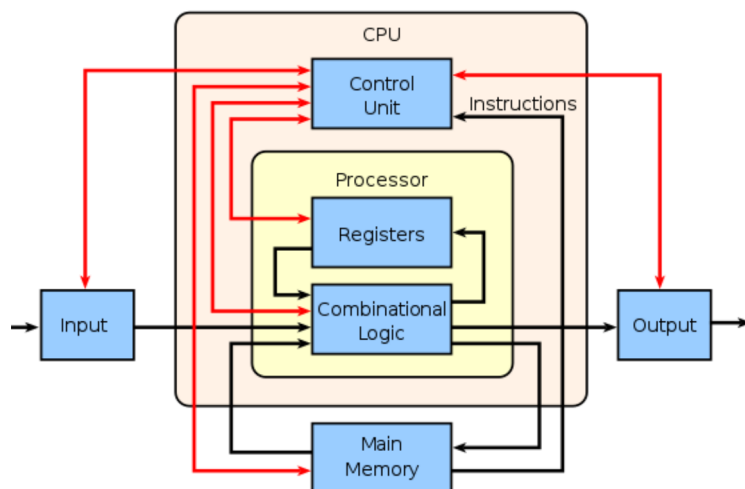
Execute

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, control signals electrically enable or disable various parts of the CPU so they can perform all or part of the desired operation. The action is then 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, registers containing operands (numbers to be summed) are activated, as are the parts of the arithmetic logic unit (ALU) that perform addition. When the clock pulse occurs, the operands flow from the source registers into the ALU, and the sum appears at its output. On subsequent clock pulses, other components are enabled (and disabled) to move the output (the sum of the operation) to storage (e.g., a register or memory). 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, influencing the next operation.

Structure and implementation

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 instruction is represented by a unique combination of bits, known as the machine language opcode. While processing an instruction, the CPU decodes the opcode (via a binary decoder) into control signals, which orchestrate the behavior of the CPU. A complete machine language instruction consists of an opcode and, in many cases, additional bits that specify arguments for the operation (for

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

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).[61]

Control unit

The **control unit** (CU) is a component of the CPU that directs the operation of the processor. It tells the computer's memory, arithmetic and logic unit and input and output devices how to respond to the instructions that have been sent to the processor.

It directs the operation of the other units by providing timing and control signals. Most computer resources are managed by the CU. It directs the flow of data between the CPU and the other devices. John von Neumann included the control unit as part of the von Neumann architecture. In modern computer designs, the control unit is typically an internal part of the CPU with its overall role and operation unchanged since its introduction.[62]

Arithmetic logic unit

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, external memory, or 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.
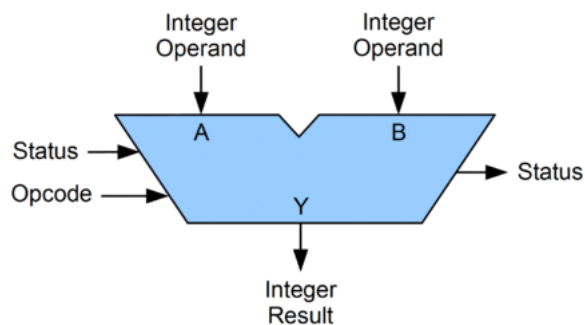
Symbolic representation of an ALU and its input and output signals

Address generation unit

The address generation unit (AGU), sometimes also called the address computation unit (ACU),[63] is an execution unit inside the CPU that calculates addresses used by the CPU to access main memory. By having address calculations handled by separate circuitry that operates in parallel with the rest of the CPU, the number of CPU cycles required for executing various machine instructions can be reduced, bringing performance improvements.

While performing various operations, CPUs need to calculate memory addresses required for fetching data from the memory; for example, in-memory positions of array elements must be calculated before the CPU can fetch the data from actual memory locations. Those address-generation calculations involve different integer arithmetic operations, such as addition, subtraction, modulo operations, or bit shifts. Often, calculating a memory address involves more than one general-purpose machine instruction, which do not necessarily decode and execute quickly. By incorporating an AGU into a CPU design, together with introducing specialized instructions that use the AGU, various address-generation calculations can be offloaded from the rest of the CPU, and can often be executed quickly in a single CPU cycle.

Capabilities of an AGU depend on a particular CPU and its architecture. Thus, some AGUs implement and expose more address-calculation operations, while some also include more advanced specialized instructions that can operate on multiple operands at a time. Some CPU architectures include multiple AGUs so more than one address-calculation operation can be executed simultaneously, which brings further performance improvements due to the superscalar nature of advanced CPU designs. For example, Intel incorporates multiple AGUs into its Sandy Bridge and Haswell microarchitectures, which increase bandwidth of the CPU memory subsystem by allowing multiple memory-access instructions to be executed in parallel.

Memory management unit (MMU)

Many microprocessors (in smartphones and 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.

Cache

A CPU cache[64] is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3, L4, etc.).

All modern (fast) CPUs (with few specialized exceptions[f]) have multiple levels of CPU caches. The first CPUs that used a cache had only one level of cache; unlike later level 1 caches, it was not split into L1d (for data) and L1i (for instructions). Almost all current CPUs with caches have a split L1 cache. They also have L2 caches and, for larger processors, L3 caches as well. The L2 cache is usually not split and acts as a common repository for the already split L1 cache. Every core of a multi-core processor has a dedicated L2 cache and is usually not shared between the cores. The L3 cache, and higher-level caches, are shared between the cores and are not split. An L4 cache is currently uncommon, and is generally on dynamic random-access memory (DRAM), rather than on static random-access memory (SRAM), on a separate die or chip. That was also the case historically with L1, while bigger chips have allowed integration of it and generally all cache levels, with the possible exception of the last level. Each extra level of cache tends to be bigger and be optimized differently.

Other types of caches exist (that are not counted towards the "cache size" of the most important caches mentioned above), such as the translation lookaside buffer (TLB) that is part of the memory management unit (MMU) that most CPUs have.

Caches are generally sized in powers of two: 2, 8, 16 etc. KiB or MiB (for larger non-L1) sizes, although the IBM z13 has a 96 KiB L1 instruction cache.[65]

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.[66]

Clockless CPUs

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.[67]

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.[68]

Voltage regulator module

Many modern CPUs have a die-integrated power managing module which regulates on-demand voltage supply to the CPU circuitry allowing it to keep balance between performance and power consumption.

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.[g]

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.[h] 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.

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.

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.[30] 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.[69] 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.

Parallelism

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).

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

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.[i]

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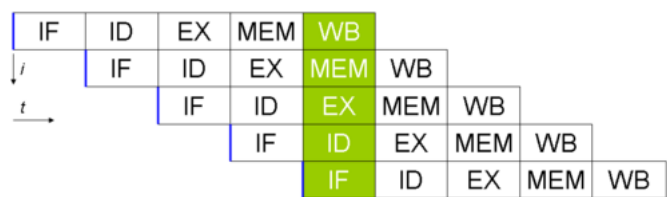
Basic five-stage pipeline. In the best case scenario, this pipeline can sustain a completion rate of one instruction per clock cycle.

One of the simplest methods for increased parallelism is to begin the first steps of instruction fetching and decoding before the prior instruction finishes executing. This is a technique known as instruction pipelining, and is used in almost all modern general-purpose CPUs. Pipelining allows multiple instruction to be executed at a time by breaking 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. Therefore pipelined processors must check for these sorts of conditions and delay a portion of the pipeline if necessary. A pipelined processor can become very nearly scalar, inhibited only by pipeline stalls (an instruction spending more than one clock cycle in a stage).

Improvements in instruction pipelining led to further decreases in the idle time of CPU components. 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.[70] In a superscalar pipeline, 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 execution units, resulting in their simultaneous execution. In general, the number of instructions that a superscalar CPU will complete in a cycle is dependent on
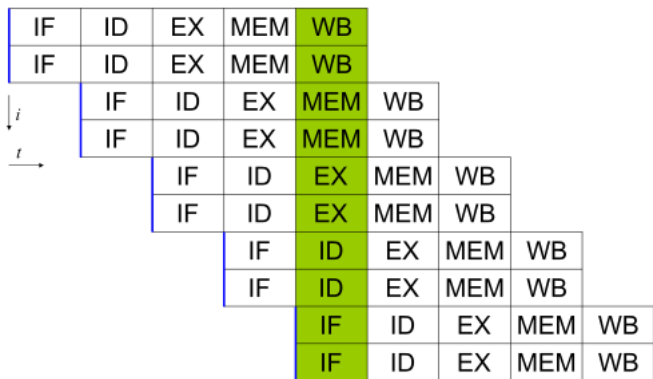
A simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per clock cycle can be completed.

the number of instructions it is able to dispatch simultaneously to execution units.

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 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 requires 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.

When just a fraction of the CPU is superscalar, the part that 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. Thus the P5 was integer superscalar but not floating point superscalar. Intel's successor to the P5 architecture, P6, added superscalar abilities to its floating-point features.

Simple pipelining and superscalar design increase a CPU's ILP by allowing it to execute 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 instruction set architecture (ISA). The strategy of the very long instruction word (VLIW) causes some ILP to become implied directly by the software, reducing the CPU's work in boosting ILP and thereby reducing design 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.[71] In Flynn's taxonomy, this strategy is known as multiple instruction stream, multiple data stream (MIMD).[72]

One technology used for this purpose was multiprocessing (MP).[73] 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.[j] 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 instruction set architecture (ISA). Some notable modern examples include Intel's Streaming SIMD Extensions (SSE) and the PowerPC-related AltiVec (also known as VMX).[k]

Hardware performance counter

Many modern architectures (including embedded ones) often include hardware performance counters (HPC), which enables low-level (instruction-level) collection, benchmarking, debugging or analysis of running software metrics.[74][75] HPC may also be used to discover and analyze unusual or suspicious activity of the software, such as return-oriented programming (ROP) or sigreturn-oriented programming (SROP) exploits etc.[76] This is usually done by software-security teams to assess and find malicious binary programs.

Many major vendors (such as IBM, Intel, AMD, and Arm etc.) provide software interfaces (usually written in C/C++) that can be used to collected data from CPUs registers in order to get metrics.[77] Operating system vendors also provide software like `perf` (Linux) to record, benchmark, or trace CPU events running kernels and applications.

Virtual CPUs

Cloud computing can involve subdividing CPU operation into **virtual central processing units**[78] (**vCPU**s[79]).

A host is the virtual equivalent of a physical machine, on which a virtual system is operating.[80] 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.

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.[81] 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.[82] 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.[83] 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 simultaneous multithreading 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.[84] 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.[2]

See also

> ⧉ **Technology portal**

- Addressing mode
- AMD Accelerated Processing Unit
- Complex instruction set computer
- Computer bus
- Computer engineering
- CPU core voltage
- CPU socket
- Data processing unit
- Digital signal processor

- Graphics processing unit
- Comparison of instruction set architectures
- Protection ring
- Reduced instruction set computer
- Stream processing
- True Performance Index
- Tensor Processing Unit
- Wait state

Notes

a. Integrated circuits are now used to implement all CPUs, except for a few machines designed to withstand large electromagnetic pulses, say from a nuclear weapon.

b. The so-called "von Neumann" memo expounded the idea of stored programs,[58] which for example may be stored on punched cards, paper tape, or magnetic tape.

c. Some early computers, like the Harvard Mark I, did not support any kind of "jump" instruction, effectively limiting the complexity of the programs they could run. It is largely for this reason that these computers are often not considered to contain a proper CPU, despite their close similarity to stored-program computers.

d. Since the program counter counts *memory addresses* and not *instructions*, it is incremented by the number of memory units that the instruction word contains. In the case of simple fixed-length instruction word ISAs, this is always the same number. For example, a fixed-length 32-bit instruction word ISA that uses 8-bit memory words would always increment the PC by four (except in the case of jumps). ISAs that use variable-length instruction words increment the PC by the number of memory words corresponding to the last instruction's length.

e. Because the instruction set architecture of a CPU is fundamental to its interface and usage, it is often used as a classification of the "type" of CPU. For example, a "PowerPC CPU" uses some variant of the PowerPC ISA. A system can execute a different ISA by running an emulator.

f. A few specialized CPUs, accelerators or microcontrollers do not have a cache. To be fast, if needed/wanted, they still have an on-chip scratchpad memory that has a similar function, while software managed. In e.g. microcontrollers it can be better for hard real-time use, to have that or at least no cache, as with one level of memory latencies of loads are predictable.

g. The physical concept of voltage is an analog one by nature, practically having an infinite range of possible values. For the purpose of physical representation of binary numbers, two specific ranges of voltages are defined, one for logic '0' and another for logic '1'. These ranges are dictated by design considerations such as noise margins and characteristics of the devices used to create the CPU.

h. While a CPU's integer size sets a limit on integer ranges, this can (and often is) overcome using a combination of software and hardware techniques. By using additional memory, software can represent integers many magnitudes larger than the CPU can. Sometimes the CPU's instruction set will even facilitate operations on integers larger than it can natively represent by providing instructions to make large integer arithmetic relatively quick. This method of dealing with large integers is slower than utilizing a CPU with higher integer size, but is a reasonable trade-off in cases where natively supporting the full integer range needed would be cost-prohibitive. See Arbitrary-precision arithmetic for more details on purely software-supported arbitrary-sized integers.

i. Neither ILP nor TLP is inherently superior over the other; they are simply different means by which to increase CPU parallelism. As such, they both have advantages and disadvantages, which are often determined by the type of software that the processor is intended to run. High-TLP CPUs are often used in applications that lend themselves well to being split up into numerous smaller applications, so-called "embarrassingly parallel problems". Frequently, a computational problem that can be solved quickly with high TLP design strategies like symmetric multiprocessing takes significantly more time on high ILP devices like superscalar CPUs, and vice versa.

j. Earlier the term *scalar* was used to compare the IPC count afforded by various ILP methods. Here the term is used in the strictly mathematical sense to contrast with vectors. See scalar (mathematics) and vector (geometric).

k. Although SSE/SSE2/SSE3 have superseded MMX in Intel's general-purpose processors, later IA-32 designs still support MMX. This is usually done by providing most of the MMX functionality with the same hardware that supports the much more expansive SSE instruction sets.