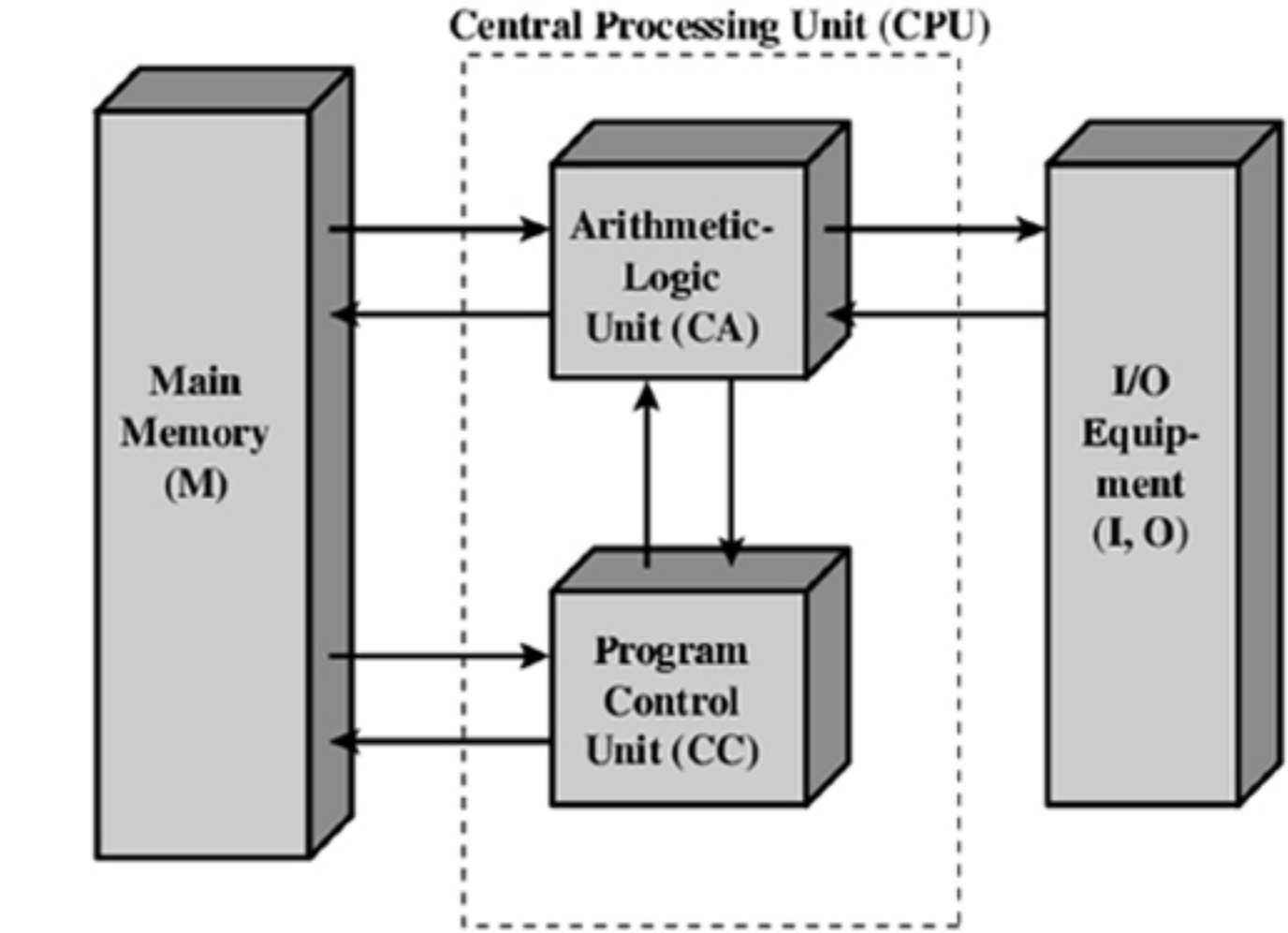
Computer Architecture and Engineering



Vivien Manning
Garth Faulk

First Edition, 2012

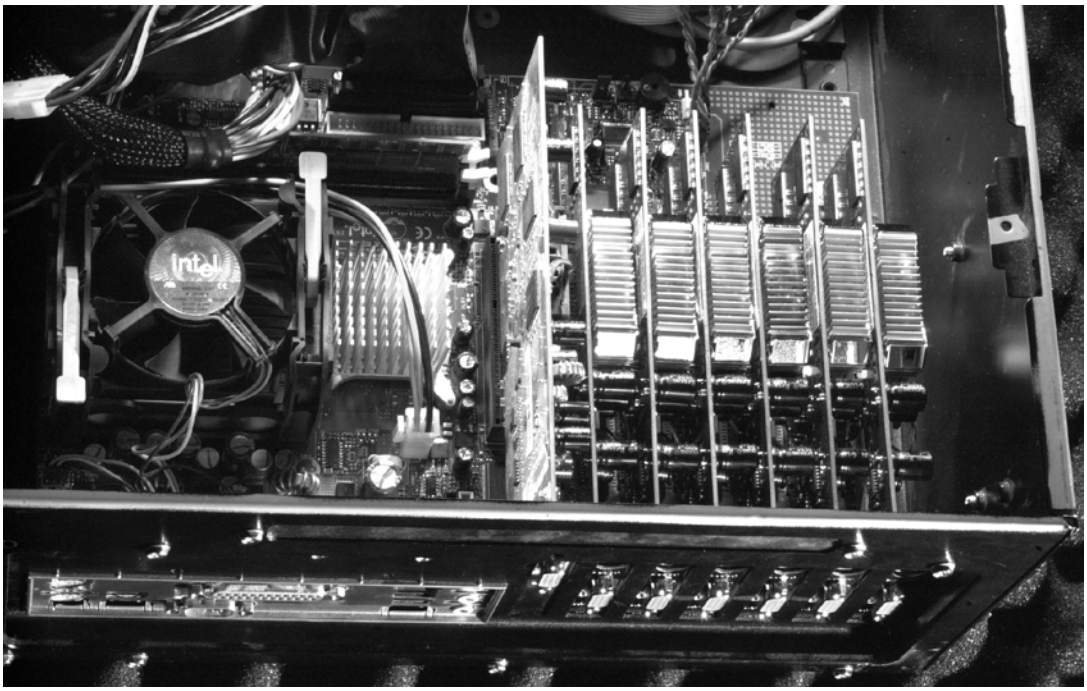
ISBN 978-81-323-1267-3

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Computer Architecture



In computer science and computer engineering, **computer architecture** or **digital computer organization** is the conceptual design and fundamental operational structure of a computer system. It's a blueprint and functional description of requirements and design implementations for the various parts of a computer, focusing largely on the way by which the central processing unit (CPU) performs internally and accesses addresses in memory.

It may also be defined as the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

Computer architecture comprises at least three main subcategories:

- *Instruction set architecture*, or ISA, is the abstract image of a computing system that is seen by a machine language (or assembly language) programmer, including

the instruction set, word size, memory address modes, processor registers, and address and data formats.

- *Microarchitecture*, also known as *Computer organization* is a lower level, more concrete and detailed, description of the system that involves how the constituent parts of the system are interconnected and how they interoperate in order to implement the ISA. The size of a computer's cache for instance, is an organizational issue that generally has nothing to do with the ISA.

- *System Design* which includes all of the other hardware components within a computing system such as:

1. System interconnects such as computer buses and switches
2. Memory controllers and hierarchies
3. CPU off-load mechanisms such as direct memory access (DMA)
4. Issues like multiprocessing.

Once both ISA and microarchitecture have been specified, the actual device needs to be designed into hardware. This design process is called the *implementation*. Implementation is usually not considered architectural definition, but rather hardware design engineering.

Implementation can be further broken down into three (not fully distinct) pieces:

- **Logic Implementation** — design of blocks defined in the microarchitecture at (primarily) the register-transfer and gate levels.
- **Circuit Implementation** — transistor-level design of basic elements (gates, multiplexers, latches etc.) as well as of some larger blocks (ALUs, caches etc.) that may be implemented at this level, or even (partly) at the physical level, for performance reasons.
- **Physical Implementation** — physical circuits are drawn out, the different circuit components are placed in a chip floorplan or on a board and the wires connecting them are routed.

For CPUs, the entire implementation process is often called CPU design.

More specific usages of the term include more general wider-scale hardware architectures, such as cluster computing and Non-Uniform Memory Access (NUMA) architectures.

History

The term “architecture” in computer literature can be traced to the work of Lyle R. Johnson, Muhammad Usman Khan and Frederick P. Brooks, Jr., members in 1959 of the Machine Organization department in IBM’s main research center. Johnson had the opportunity to write a proprietary research communication about Stretch, an IBM-

developed supercomputer for Los Alamos Scientific Laboratory. In attempting to characterize his chosen level of detail for discussing the luxuriously embellished computer, he noted that his description of formats, instruction types, hardware parameters, and speed enhancements was at the level of "system architecture" – a term that seemed more useful than "machine organization". Subsequently, Brooks, one of the Stretch designers, started Chapter 2 of a book (Planning a Computer System: Project Stretch, ed. W. Buchholz, 1962) by writing, "Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints". Brooks went on to play a major role in the development of the IBM System/360 line of computers, where "architecture" gained currency as a noun with the definition "what the user needs to know". Later the computer world would employ the term in many less-explicit ways.

he way by which the CPU performs internally and accesses addresses in memory," mentioned above, slip into the definition of computer architecture.

Computer architectures

There are many types of computer architectures:

- Quantum computer vs Chemical computer
- Scalar processor vs Vector processor
- Non-Uniform Memory Access (NUMA) computers
- Register machine vs Stack machine
- Harvard architecture vs von Neumann architecture
- Cellular architecture

The quantum computer architecture holds the most promise to revolutionize computing.

Computer architecture topics

Sub-definitions

Some practitioners of computer architecture at companies such as Intel and AMD use more fine distinctions:

- Macroarchitecture — architectural layers that are more abstract than microarchitecture, e.g. ISA
- Instruction Set Architecture (ISA) — as defined above
- Assembly ISA — a smart assembler may convert an abstract assembly language common to a group of machines into slightly different machine language for different implementations
- Programmer Visible Macroarchitecture — higher level language tools such as compilers may define a consistent interface or contract to programmers using

them, abstracting differences between underlying ISA, UISA, and microarchitectures. E.g. the C, C++, or Java standards define different Programmer Visible Macroarchitecture — although in practice the C microarchitecture for a particular computer includes

- UISA (Microcode Instruction Set Architecture) — a family of machines with different hardware level microarchitectures may share a common microcode architecture, and hence a UISA.
- Pin Architecture — the set of functions that a microprocessor is expected to provide, from the point of view of a hardware platform. E.g. the x86 A20M, FERR/IGNNE or FLUSH pins, and the messages that the processor is expected to emit after completing a cache invalidation so that external caches can be invalidated. Pin architecture functions are more flexible than ISA functions - external hardware can adapt to changing encodings, or changing from a pin to a message - but the functions are expected to be provided in successive implementations even if the manner of encoding them changes.

The Role Of Computer Architecture

Computer Architecture: The Definition

The coordination of abstract levels of a processor under changing forces, involving design, measurement and evaluation. It also includes the overall fundamental working principle of the internal logical structure of a computer system.

It can also be defined as the design of the task-performing part of computers, i.e. how various gates and transistors are interconnected and are caused to function per the instructions given by an assembly language programmer.

Instruction Set Architecture

1. The ISA is the interface between the software and hardware.
2. It is the set of instructions that bridges the gap between high level languages and the hardware.
3. For a processor to understand a command, it should be in binary and not in High Level Language. The ISA encodes these values.
4. The ISA also defines the items in the computer that are available to a programmer. For example, it defines data types, registers, addressing modes, memory organization etc.
5. Register are high Addressing modes are the ways in which the instructions locate their operands.

Memory organization defines how instructions interact with the memory.

Computer Organization

Computer organization helps optimize performance-based products. For example, software engineers need to know the processing ability of processors. They may need to optimize software in order to gain the most performance at the least expense. This can require quite detailed analysis of the computer organization. For example, in a multimedia decoder, the designers might need to arrange for most data to be processed in the fastest data path and the various components are assumed to be in place and task is to investigate the organisational structure to verify the computer parts operates.

Computer organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access, while supervisory software may need fast interrupts.

Sometimes certain tasks need additional components as well. For example, a computer capable of virtualization needs virtual memory hardware so that the memory of different simulated computers can be kept separated.

The computer organization and features also affect the power consumption and the cost of the processor.

Design goals

Performance

Computer performance is often described in terms of clock speed (usually in MHz or GHz). This refers to the cycles per second of the main clock of the CPU. However, this metric is somewhat misleading, as a machine with a higher clock rate may not necessarily have higher performance. As a result manufacturers have moved away from clock speed as a measure of performance.

Computer performance can also be measured with the amount of cache a processor has. If the speed, MHz or GHz, were to be a car then the cache is like the gas tank. No matter how fast the car goes, it will still need to get gas. The higher the speed, and the greater the cache, the faster a processor runs.

Modern CPUs can execute multiple instructions per clock cycle, which dramatically speeds up a program. Other factors influence speed, such as the mix of functional units, bus speeds, available memory, and the type and order of instructions in the programs being run.

There are two main types of speed, latency and throughput. Latency is the time between the start of a process and its completion. Throughput is the amount of work done per unit time. Interrupt latency is the guaranteed maximum response time of the system to an electronic event (*e.g.* when the disk drive finishes moving some data). Performance is affected by a very wide range of design choices — for example, pipelining a processor

usually makes latency worse (slower) but makes throughput better. Computers that control machinery usually need low interrupt latencies. These computers operate in a real-time environment and fail if an operation is not completed in a specified amount of time. For example, computer-controlled anti-lock brakes must begin braking almost immediately after they have been instructed to brake.

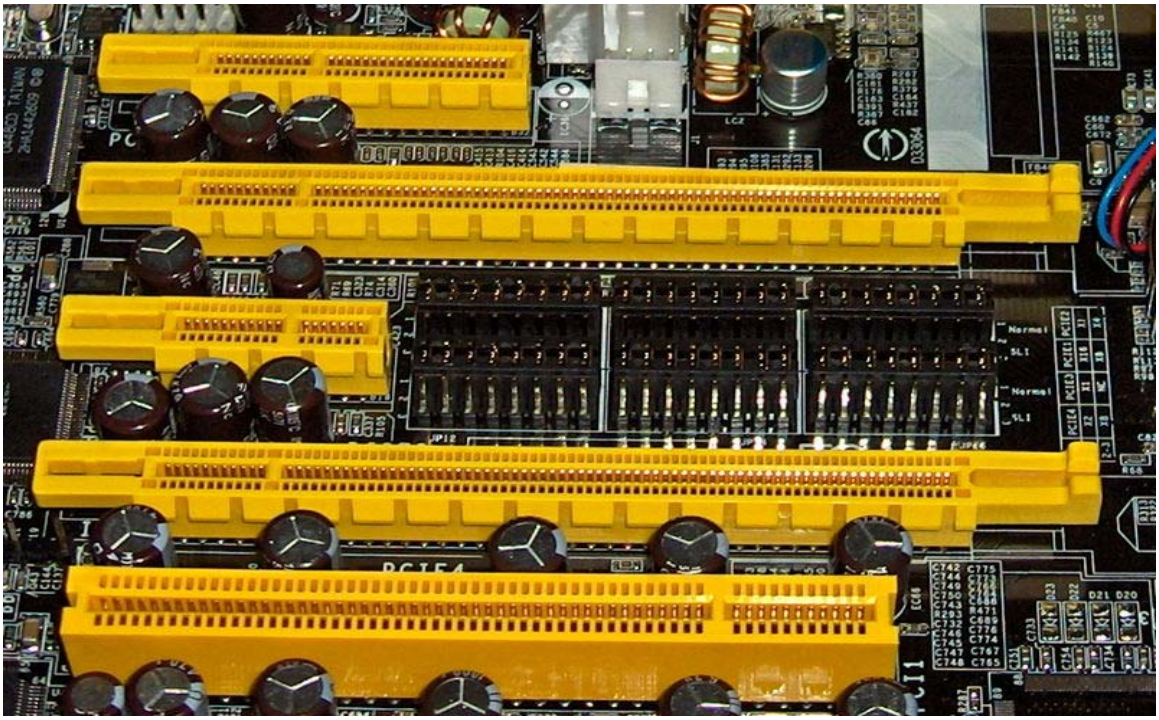
The performance of a computer can be measured using other metrics, depending upon its application domain. A system may be CPU bound (as in numerical calculation), I/O bound (as in a webserving application) or memory bound (as in video editing). Power consumption has become important in servers and portable devices like laptops.

Benchmarking tries to take all these factors into account by measuring the time a computer takes to run through a series of test programs. Although benchmarking shows strengths, it may not help one to choose a computer. Often the measured machines split on different measures. For example, one system might handle scientific applications quickly, while another might play popular video games more smoothly. Furthermore, designers have been known to add special features to their products, whether in hardware or software, which permit a specific benchmark to execute quickly but which do not offer similar advantages to other, more general tasks.

Chapter 2

Bus (Computing) and Instruction Set

Bus (computing)



4 PCI Express bus card slots (from top to bottom: x4, x16, x1 and x16), compared to a 32-bit conventional PCI bus card slot (very bottom).

In computer architecture, a **bus** is a subsystem that transfers data between computer components inside a computer or between computers.

Early computer buses were literally parallel electrical buses with multiple connections, but the term is now used for any physical arrangement that provides the same logical functionality as a parallel electrical bus. Modern computer buses can use both parallel and bit-serial connections, and can be wired in either a multidrop (electrical parallel) or daisy chain topology, or connected by switched hubs, as in the case of USB.

History

First generation

Early computer buses were bundles of wire that attached memory and peripherals. They were named after electrical buses, or busbars. Almost always, there was one bus for memory, and another for peripherals, and these were accessed by separate instructions, with completely different timings and protocols.

One of the first complications was the use of interrupts. Early computer programs performed I/O by waiting in a loop for the peripheral to become ready. This was a waste of time for programs that had other tasks to do. Also, if the program attempted to perform those other tasks, it might take too long for the program to check again, resulting in loss of data. Engineers thus arranged for the peripherals to interrupt the CPU. The interrupts had to be prioritized, because the CPU can only execute code for one peripheral at a time, and some devices are more time-critical than others.

Later computer programs began to share memory common to several CPUs. Access to this memory bus had to be prioritized, as well.

The classic, simple way to prioritize interrupts or bus access was with a daisy chain.

DEC noted that having two buses seemed wasteful and expensive for mass-produced minicomputers, and mapped peripherals into the memory bus, so that the devices appeared to be memory locations.

Early microcomputer bus systems were essentially a passive backplane connected directly or through buffer amplifiers to the pins of the CPU. Memory and other devices would be added to the bus using the same address and data pins as the CPU itself used, connected in parallel. Communication was controlled by the CPU, which had read and written data from the devices as if they are blocks of memory, using the same instructions, all timed by a central clock controlling the speed of the CPU. Still, devices interrupted the CPU by signaling on separate CPU pins. For instance, a disk drive controller would signal the CPU that new data was ready to be read, at which point the CPU would move the data by reading the "memory location" that corresponded to the disk drive. Almost all early microcomputers were built in this fashion, starting with the S-100 bus in the Altair.

In some instances, most notably in the IBM PC, although similar physical architecture is employed, instructions to access peripherals (`in` and `out`) and memory (`mov` and others) have not been made uniform at all, and still generate distinct CPU signals, that could be used to implement a separate I/O bus.

These simple bus systems had a serious drawback when used for general-purpose computers. All the equipment on the bus has to talk at the same speed, as it shares a single clock.

Increasing the speed of the CPU becomes harder, because the speed of all the devices must increase as well. When it is not practical or economical to have all devices as fast as the CPU, the CPU must either enter a wait state, or work at a slower clock frequency temporarily, to talk to other devices in the computer. While acceptable in embedded systems, this problem was not tolerated for long in general-purpose, user-expandable computers.

Such bus systems are also difficult to configure when constructed from common off-the-shelf equipment. Typically each added expansion card requires many jumpers in order to set memory addresses, I/O addresses, interrupt priorities, and interrupt numbers.

A *bus controller* accepted data from the CPU side to be moved to the peripherals side, thus shifting the communications protocol burden from the CPU itself. This allowed the CPU and memory side to evolve separately from the device bus, or just "bus". Devices on the bus could talk to each other with no CPU intervention. This led to much better "real world" performance, but also required the cards to be much more complex. These buses also often addressed speed issues by being "bigger" in terms of the size of the data path, moving from 8-bit parallel buses in the first generation, to 16 or 32-bit in the second, as well as adding software setup (now standardised as Plug-n-play) to supplant or replace the jumpers.

However these newer systems shared one quality with their earlier cousins, in that everyone on the bus had to talk at the same speed. While the CPU was now isolated and could increase speed without fear, CPUs and memory continued to increase in speed much faster than the buses they talked to. The result was that the bus speeds were now very much slower than what a modern system needed, and the machines were left starved for data. A particularly common example of this problem was that video cards quickly outran even the newer bus systems like PCI, and computers began to include AGP just to drive the video card. By 2004 AGP was outgrown again by high-end video cards and other peripherals and has been replaced by the new PCI Express bus.

An increasing number of external devices started employing their own bus systems as well. When disk drives were first introduced, they would be added to the machine with a card plugged into the bus, which is why computers have so many slots on the bus. But through the 1980s and 1990s, new systems like SCSI and IDE were introduced to serve this need, leaving most slots in modern systems empty. Today there are likely to be about five different buses in the typical machine, supporting various devices.

Third generation

"Third generation" buses have been emerging into the market since about 2001, including HyperTransport and InfiniBand. They also tend to be very flexible in terms of their physical connections, allowing them to be used both as internal buses, as well as connecting different machines together. This can lead to complex problems when trying to service different requests, so much of the work on these systems concerns software design, as opposed to the hardware itself. In general, these third generation buses tend to

look more like a network than the original concept of a bus, with a higher protocol overhead needed than early systems, while also allowing multiple devices to use the bus at once.

Buses such as Wishbone have been developed by the open source hardware movement in an attempt to further remove legal and patent constraints from computer design.

Description of a bus

At one time, "bus" meant an electrically parallel system, with electrical conductors similar or identical to the pins on the CPU. This is no longer the case, and modern systems are blurring the lines between buses and networks.

Buses can be parallel buses, which carry data words in parallel on multiple wires, or serial buses, which carry data in bit-serial form. The addition of extra power and control connections, differential drivers, and data connections in each direction usually means that most serial buses have more conductors than the minimum of one used in the 1-Wire and UNI/O serial buses. As data rates increase, the problems of timing skew, power consumption, electromagnetic interference and crosstalk across parallel buses become more and more difficult to circumvent. One partial solution to this problem has been to double pump the bus. Often, a serial bus can actually be operated at higher overall data rates than a parallel bus, despite having fewer electrical connections, because a serial bus inherently has no timing skew or crosstalk. USB, FireWire, and Serial ATA are examples of this. Multidrop connections do not work well for fast serial buses, so most modern serial buses use daisy-chain or hub designs.

Most computers have both internal and external buses. An *internal bus* connects all the internal components of a computer to the motherboard (and thus, the CPU and internal memory). These types of buses are also referred to as a local bus, because they are intended to connect to local devices, not to those in other machines or external to the computer. An *external bus* connects external peripherals to the motherboard.

Network connections such as Ethernet are not generally regarded as buses, although the difference is largely conceptual rather than practical. The arrival of technologies such as InfiniBand and HyperTransport is further blurring the boundaries between networks and buses. Even the lines between internal and external are sometimes fuzzy, I²C can be used as both an internal bus, or an external bus (where it is known as ACCESS.bus), and InfiniBand is intended to replace both internal buses like PCI as well as external ones like Fibre Channel. In the typical desktop application, USB serves as a peripheral bus, but it also sees some use as a networking utility and for connectivity between different computers, again blurring the conceptual distinction.

Bus topology

In a network, the master scheduler controls the data traffic. If data is to be transferred, the requesting computer sends a message to the scheduler, which puts the request into a queue. The message contains an identification code which is broadcast to all nodes of the network. The scheduler works out priorities and notifies the receiver as soon as the bus is available.

The identified node takes the message and performs the data transfer between the two computers. Having completed the data transfer the bus becomes free for the next request in the scheduler's queue.

- Advantage: Any computer can be accessed directly and messages can be sent in a relatively simple and fast way.
- Disadvantage: A scheduler is required to organize the traffic by assigning frequencies and priorities to each signal.

Examples of internal computer buses

Parallel

- ASUS Media Bus proprietary, used on some ASUS Socket 7 motherboards
- Computer Automated Measurement and Control (CAMAC) for instrumentation systems
- Extended ISA or EISA
- Industry Standard Architecture or ISA
- Low Pin Count or LPC
- MBus
- MicroChannel or MCA
- Multibus for industrial systems
- NuBus or IEEE 1196
- OPTi local bus used on early Intel 80486 motherboards.
- Conventional PCI
- Parallel ATA (aka Advanced Technology Attachment, ATA, PATA, IDE, EIDE, ATAPI, etc.) disk/tape peripheral attachment bus
- Q-Bus, a proprietary bus developed by Digital Equipment Corporation for their PDP and later VAX computers.
- S-100 bus or IEEE 696, used in the Altair and similar microcomputers
- SBus or IEEE 1496
- SS-50 Bus
- STEbus
- STD Bus (for STD-80 [8-bit] and STD32 [16-/32-bit]), FAQ
- Unibus, a proprietary bus developed by Digital Equipment Corporation for their PDP-11 and early VAX computers.
- VESA Local Bus or VLB or VL-bus

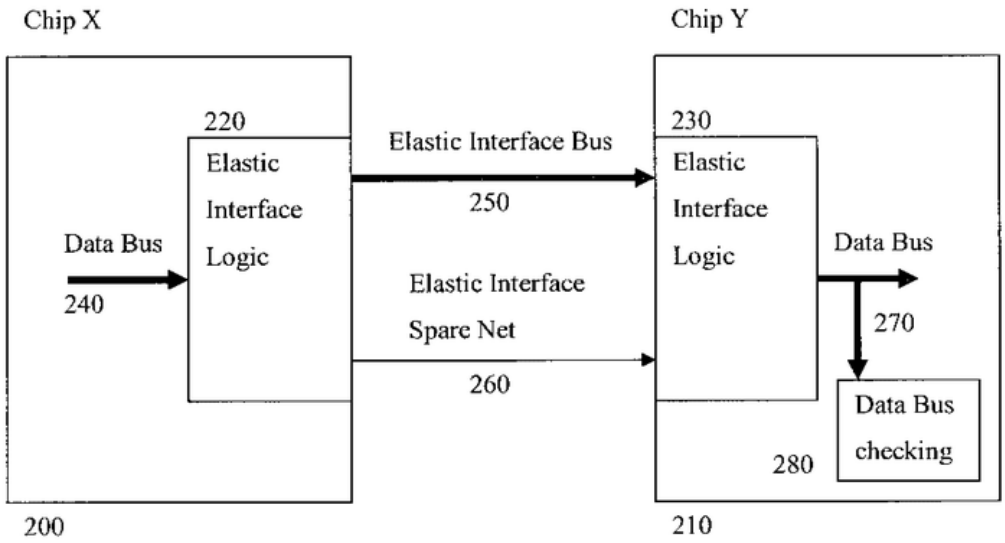
- VMEbus, the VERSAmodule Eurocard bus

Serial

- 1-Wire
- HyperTransport
- I²C
- PCI Express or PCIe
- Serial ATA
- Serial Peripheral Interface Bus or SPI bus
- UNI/O
- SMBus

Self Repairable

FIGURE 2 An Elastic Interface Bus Topology with the Spare Net



Spare Net for Elastic Interface Bus from US patent application 20080082878

Self repairable elastic interface buses have recently been invented by IBM. IBM has filed a patent application on these buses which is undergoing peer review on Peer to Patent. The public commentary period closed on July 24, 2008. The IBM invention provides a spare net which the system switches to in the event that an alternate net doesn't function.

Examples of external computer buses

Parallel

- HIPPI HIgh Performance Parallel Interface
- IEEE-488 (aka GPIB, General-Purpose Interface Bus, and HPIB, Hewlett-Packard Instrumentation Bus)
- PC card, previously known as *PCMCIA*, much used in laptop computers and other portables, but fading with the introduction of USB and built-in network and modem connections

Serial

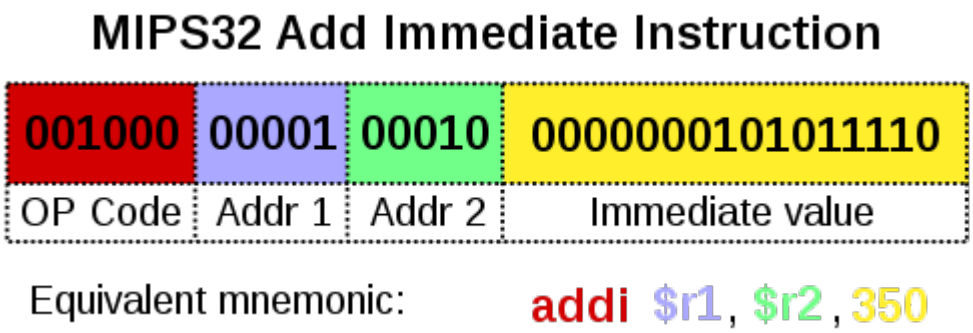
- USB Universal Serial Bus, used for a variety of external devices
- Controller Area Network ("CAN bus")
- EIA-485
- eSATA
- FireWire

Examples of internal/external computer buses

- Futurebus
- InfiniBand
- QuickRing
- SCI
- SCSI Small Computer System Interface, disk/tape peripheral attachment bus
- Serial Attached SCSI and other serial SCSI buses

Instruction set

An **instruction set**, or **instruction set architecture** (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.



One instruction may have several fields, which identify the logical operation to be done, and may also include source and destination addresses and constant values. This is the MIPS "Add" instruction which allows selection of source and destination registers and inclusion of a small constant.

Instruction set architecture is distinguished from the microarchitecture, which is the set of processor design techniques used to implement the instruction set. Computers with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.

This concept can be extended to unique ISAs like TIMI (Technology-Independent Machine Interface) present in the IBM System/38 and IBM AS/400. TIMI is an ISA that is implemented by low-level software translating TIMI code into "native" machine code, and functionally resembles what is now referred to as a virtual machine. It was designed to increase the longevity of the platform and applications written for it, allowing the entire platform to be moved to very different hardware without having to modify any software except that which translates TIMI into native machine code, and the code that implements services used by the resulting native code. This allowed IBM to move the AS/400 platform from an older CISC architecture to the newer POWER architecture without having to rewrite or recompile any parts of the OS or software associated with it other than the aforementioned low-level code. Some virtual machines that support bytecode for Smalltalk, the Java virtual machine, and Microsoft's Common Language Runtime virtual machine as their ISA implement it by translating the bytecode for commonly-used code paths into native machine code, and executing less-frequently-used code paths by interpretation; Transmeta implemented the x86 instruction set atop VLIW processors in the same fashion.

Machine language

Machine language is built up from discrete *statements* or *instructions*. On the processing architecture, a given instruction may specify:

- Particular registers for arithmetic, addressing, or control functions
- Particular memory locations or offsets

- Particular addressing modes used to interpret the operands

More complex operations are built up by combining these simple instructions, which (in a von Neumann architecture) are executed sequentially, or as otherwise directed by control flow instructions.

Some operations available in most instruction sets include:

- Data handling and Memory operations
  - **set** a register (a temporary "scratchpad" location in the CPU itself) to a fixed constant value
  - **move** data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation.
  - **read** and **write** data from hardware devices
- Arithmetic and Logic
  - **add**, **subtract**, **multiply**, or **divide** the values of two registers, placing the result in a register
  - perform bitwise operations, taking the **conjunction** and **disjunction** of corresponding bits in a pair of registers, or the **negation** of each bit in a register
  - **compare** two values in registers (for example, to see if one is less, or if they are equal)
- Control flow
  - **branch** to another location in the program and execute instructions there
  - **conditionally branch** to another location if a certain condition holds
  - **indirectly branch** to another location, but save the location of the next instruction as a point to return to (a *call*)

Some computers include "complex" instructions in their instruction set. A single "complex" instruction does something that may take many instructions on other computers. Such instructions are typified by instructions that take multiple steps, control multiple functional units, or otherwise appear on a larger scale than the bulk of simple instructions implemented by the given processor. Some examples of "complex" instructions include:

- saving many registers on the stack at once
- moving large blocks of memory
- complex and/or floating-point arithmetic (sine, cosine, square root, etc.)
- performing an atomic test-and-set instruction
- instructions that combine ALU with an operand from memory rather than a register

A complex instruction type that has become particularly popular recently is the SIMD or Single-Instruction Stream Multiple-Data Stream operation or vector instruction, an operation that performs the same arithmetic operation on multiple pieces of data at the

same time. SIMD have the ability of manipulating large vectors and matrices in minimal time. SIMD instructions allow easy parallelization of algorithms commonly involved in sound, image, and video processing. Various SIMD implementations have been brought to market under trade names such as MMX, 3DNow! and AltiVec.

The design of instruction sets is a complex issue. There were two stages in history for the microprocessor. The first was the CISC (Complex Instruction Set Computer) which had many different instructions. In the 1970s, however, places like IBM did research and found that many instructions in the set could be eliminated. The result was the RISC (Reduced Instruction Set Computer), an architecture which uses a smaller set of instructions. A simpler instruction set may offer the potential for higher speeds, reduced processor size, and reduced power consumption. However, a more complex set may optimize common operations, improve memory/cache efficiency, or simplify programming.

Instruction set implementation

Any given instruction set can be implemented in a variety of ways. All ways of implementing an instruction set give the same programming model, and they all are able to run the same binary executables. The various ways of implementing an instruction set give different tradeoffs between cost, performance, power consumption, size, etc.

When designing microarchitectures, engineers use blocks of "hard-wired" electronic circuitry (often designed separately) such as adders, multiplexers, counters, registers, ALUs etc. Some kind of register transfer language is then often used to describe the decoding and sequencing of each instruction of an ISA using this physical microarchitecture. There are two basic ways to build a control unit to implement this description (although many designs use middle ways or compromises):

1. Early computer designs and some of the simpler RISC computers "hard-wired" the complete instruction set decoding and sequencing (just like the rest of the microarchitecture).
2. Other designs employ microcode routines and/or tables to do this—typically as on chip ROMs and/or PLAs (although separate RAMs have been used historically).

There are also some new CPU designs which compile the instruction set to a writable RAM or FLASH inside the CPU (such as the Rekursiv processor and the Imsys Cjip), or an FPGA (reconfigurable computing). The Western Digital MCP-1600 is an older example, using a dedicated, separate ROM for microcode.

An ISA can also be emulated in software by an interpreter. Naturally, due to the interpretation overhead, this is slower than directly running programs on the emulated hardware, unless the hardware running the emulator is an order of magnitude faster. Today, it is common practice for vendors of new ISAs or microarchitectures to make software emulators available to software developers before the hardware implementation is ready.

Often the details of the implementation have a strong influence on the particular instructions selected for the instruction set. For example, many implementations of the instruction pipeline only allow a single memory load or memory store per instruction, leading to a load-store architecture (RISC). For another example, some early ways of implementing the instruction pipeline led to a delay slot.

The demands of high-speed digital signal processing have pushed in the opposite direction—forcing instructions to be implemented in a particular way. For example, in order to perform digital filters fast enough, the MAC instruction in a typical digital signal processor (DSP) must be implemented using a kind of Harvard architecture that can fetch an instruction and two data words simultaneously, and it requires a single-cycle multiply-accumulate multiplier.

Instruction set design

Some instruction set designers reserve one or more opcodes for some kind of software interrupt. For example, MOS Technology 6502 uses 00_H, Zilog Z80 uses the eight codes C7,CF,D7,DF,E7,EF,F7,FF_H while Motorola 68000 use codes in the range A000..$AFFF_H$.

Fast virtual machines are much easier to implement if an instruction set meets the Popek and Goldberg virtualization requirements.

The NOP slide used in Immunity Aware Programming is much easier to implement if the "unprogrammed" state of the memory is interpreted as a NOP.

On systems with multiple processors, non-blocking synchronization algorithms are much easier to implement if the instruction set includes support for something like "fetch-and-increment" or "load linked/store conditional (LL/SC)" or "atomic compare and swap".

Code density

In early computers, program memory was expensive, so minimizing the size of a program to make sure it would fit in the limited memory was often central. Thus the combined size of all the instructions needed to perform a particular task, the *code density*, was an important characteristic of any instruction set. Computers with high code density also often had (and have still) complex instructions for procedure entry, parameterized returns, loops etc. (therefore retroactively named *Complex Instruction Set Computers*, CISC). However, more typical, or frequent, "CISC" instructions merely combine a basic ALU operation, such as "add", with the access of one or more operands in memory (using addressing modes such as direct, indirect, indexed etc.). Certain architectures may allow two or three operands (including the result) directly in memory or may be able to perform functions such as automatic pointer increment etc. Software-implemented instruction sets may have even more complex and powerful instructions.

Reduced instruction-set computers, RISC, were first widely implemented during a period of rapidly-growing memory subsystems and sacrifice code density in order to simplify

implementation circuitry and thereby try to increase performance via higher clock frequencies and more registers. RISC instructions typically perform only a single operation, such as an "add" of registers or a "load" from a memory location into a register; they also normally use a fixed instruction width, whereas a typical CISC instruction set has many instructions shorter than this fixed length. Fixed-width instructions are less complicated to handle than variable-width instructions for several reasons (not having to check whether an instruction straddles a cache line or virtual memory page boundary for instance), and are therefore somewhat easier to optimize for speed. However, as RISC computers normally require more and often longer instructions to implement a given task, they inherently make less optimal use of bus bandwidth and cache memories.

Minimal instruction set computers (MISC) are a form of stack machine, where there are few separate instructions (16-64), so that multiple instructions can be fit into a single machine word. These type of cores often take little silicon to implement, so they can be easily realized in an FPGA or in a multi-core form. Code density is similar to RISC; the increased instruction density is offset by requiring more of the primitive instructions to do a task.

There has been research into executable compression as a mechanism for improving code density. The mathematics of Kolmogorov complexity describes the challenges and limits of this.

Number of operands

Instruction sets may be categorized by the maximum number of operands *explicitly* specified in instructions.

(In the examples that follow, *a*, *b*, and *c* are (direct or calculated) addresses referring to memory cells, while *reg1* and so on refer to machine registers.)

- 0-operand (*zero address machines*), so called stack machines: All arithmetic operations take place using the top one or two positions on the stack; 1-operand push and pop instructions are used to access memory: **push** a, **push** b, **add**, **pop** c.
- 1-operand (*one address machines*), so called accumulator machines, include most early computers and many small microcontrollers: Most instructions specify a single explicit right operand (a register, a memory location, or a constant) with the implicit accumulator as both destination and left (or only) operand: **load** *a*, **add** *b*, **store** *c*. A related class is practical stack machines which often allow a single explicit operand in arithmetic instructions: **push** a, **add** b, **pop** c.
- 2-operand — many CISC and RISC machines fall under this category:
  - CISC — **load** *a,reg1*; **add** *reg1,b*; **store** *reg1,c*
  - RISC — Requiring explicit memory loads, the instructions would be: **load** *a,reg1*; **load** *b,reg2*; **add** *reg1,reg2*; **store** *reg2,c*
- 3-operand, allowing better reuse of data:.

    - CISC — It becomes either a single instruction: **add** *a,b,c*, or more typically: **move** *a,reg1*; **add** *reg1,b,c* as most machines are limited to two memory operands.
    - RISC — Due to the large number of bits needed to encode three registers, this scheme is typically not available in RISC processors using small 16-bit instructions; arithmetic instructions use registers only, so explicit 2-operand load/store instructions are needed: **load** *a,reg1*; **load** *b,reg2*; **add** *reg1+reg2->reg3*; **store** *reg3,c*;
- more operands—some CISC machines permit a variety of addressing modes that allow more than 3 operands (registers or memory accesses), such as the VAX "POLY" polynomial evaluation instruction.

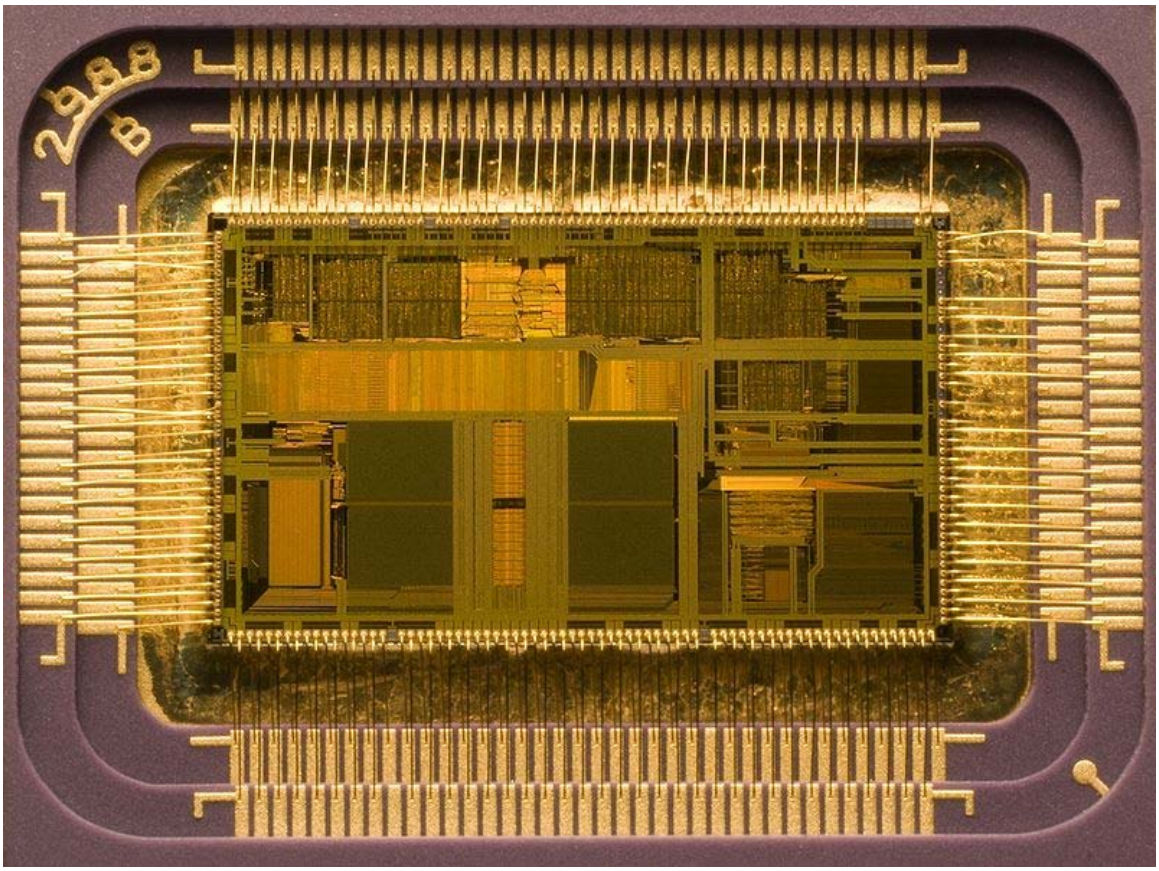
Each instruction specifies some number of operands (registers, memory locations, or immediate values) *explicitly*. Some instructions give one or both operands implicitly, such as by being stored on top of the stack or in an implicit register. When some of the operands are given implicitly, the number of specified operands in an instruction is smaller than the arity of the operation. When a "destination operand" explicitly specifies the destination, the number of operand specifiers in an instruction is larger than the arity of the operation. Some instruction sets have different numbers of operands for different instructions.

List of ISAs

This list is far from comprehensive as old architectures are developed and new ones invented. There are many commercially available microprocessors and microcontrollers implementing ISAs. Customized ISAs are also quite common in some applications, e.g. ASIC, FPGA, and reconfigurable computing.

Chapter 3

Central Processing Unit

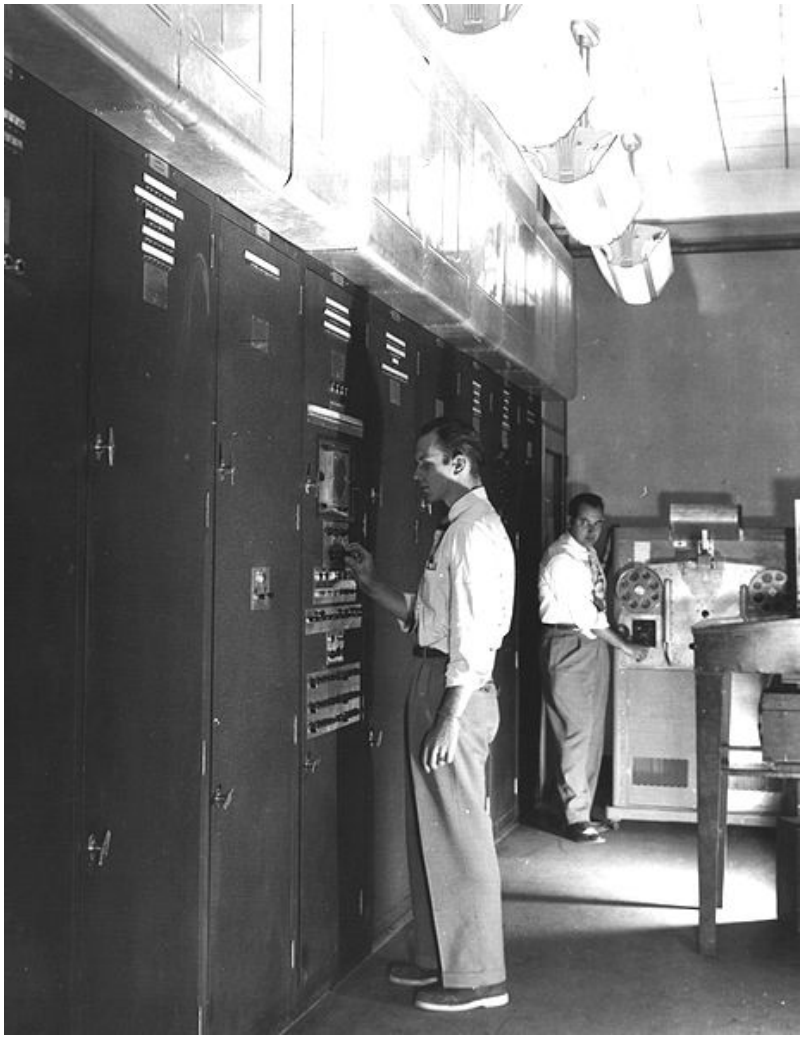


Die of an Intel 80486DX2 microprocessor (actual size: 12×6.75 mm) in its packaging.

The **central processing unit** (**CPU**) is the portion of a computer system that carries out the instructions of a computer program, and is the primary element carrying out the computer's functions. The central processing unit carries out each instruction of the program in sequence, to perform the basic arithmetical, logical, and input/output operations of the system. This term has been in use in the computer industry at least since the early 1960s. The form, design and implementation of CPUs have changed dramatically since the earliest examples, but their fundamental operation remains much the same.

Early CPUs were custom-designed as a part of a larger, sometimes one-of-a-kind, computer. However, this costly method of designing custom CPUs for a particular application has largely given way to the development of mass-produced processors that are made for one or many purposes. This standardization trend generally 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 these digital devices in modern life far beyond the limited application of dedicated computing machines. Modern microprocessors appear in everything from automobiles to cell phones and children's toys.

History



EDVAC, one of the first electronic stored program computers

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

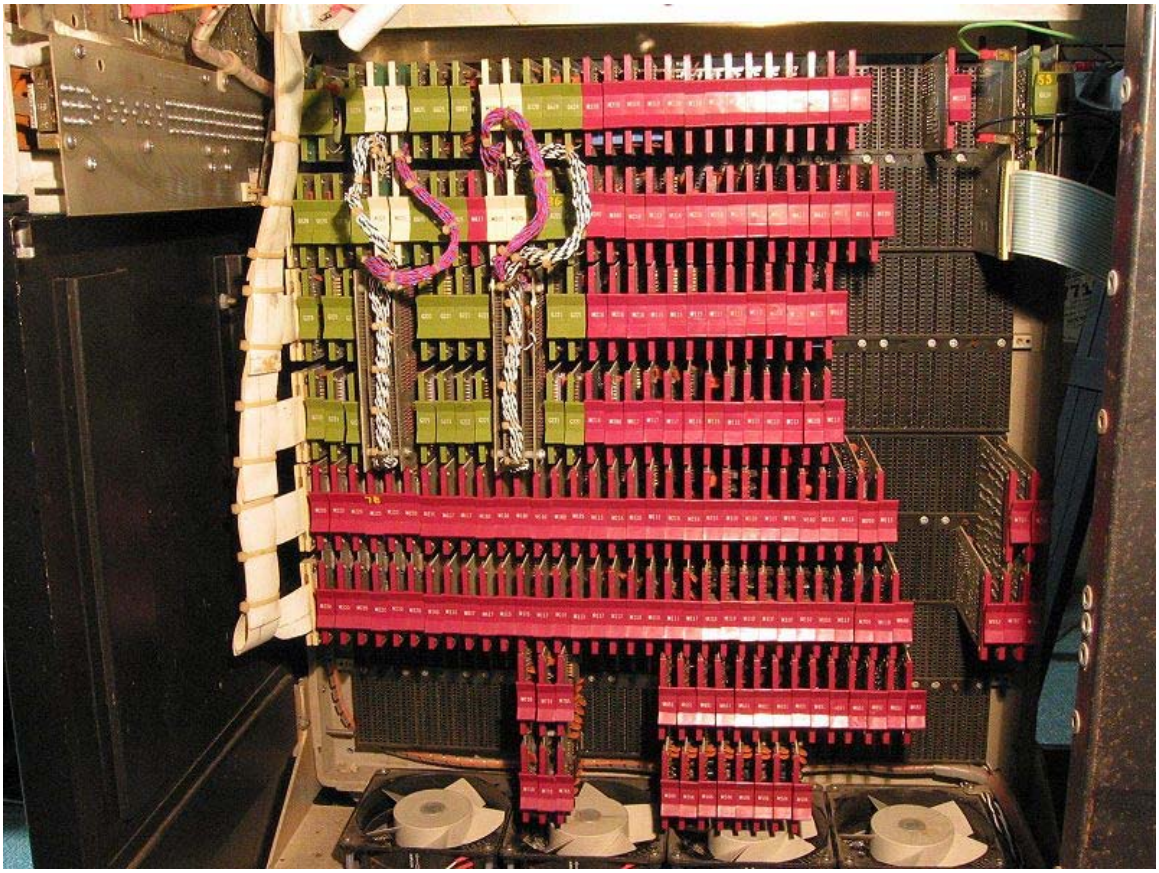
The idea of a stored-program computer was already present in the design of J. Presper Eckert and John William Mauchly's ENIAC, but was initially omitted so the machine could be finished sooner. On June 30, 1945, before ENIAC was even completed, mathematician John von Neumann distributed the paper entitled *First Draft of a Report on the EDVAC*. It outlined the design 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. These instructions could be combined to create useful programs for the EDVAC to run. Significantly, the programs written for EDVAC were 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, or software, that EDVAC ran could be changed simply by changing the contents of the computer's memory.

While von Neumann is most often credited with the design of the stored-program computer because of his design of EDVAC, 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 utilized 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 elements of the Harvard architecture are commonly seen as well.

As a digital device, a CPU is limited to a set of discrete states, and requires some kind of switching elements to differentiate between and change states. Prior to commercial development of the transistor, electrical relays and vacuum tubes (thermionic valves) were commonly used as switching elements. Although these had distinct speed advantages over earlier, purely mechanical designs, they were unreliable for various reasons. For example, building direct current sequential logic circuits out of relays requires additional hardware to cope with the problem of contact bounce. While vacuum tubes do not suffer from contact bounce, they must heat up before becoming fully operational, and they eventually cease to function due to slow contamination of their cathodes that occurs in the course of normal operation. If a tube's vacuum seal leaks, as sometimes happens, cathode contamination is accelerated. Usually, when a tube failed, the CPU would have to be diagnosed to locate the failed component so it could be replaced. Therefore, early electronic (vacuum tube based) computers were generally faster but less reliable than electromechanical (relay based) computers.

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.

Discrete transistor and integrated circuit CPUs



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

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

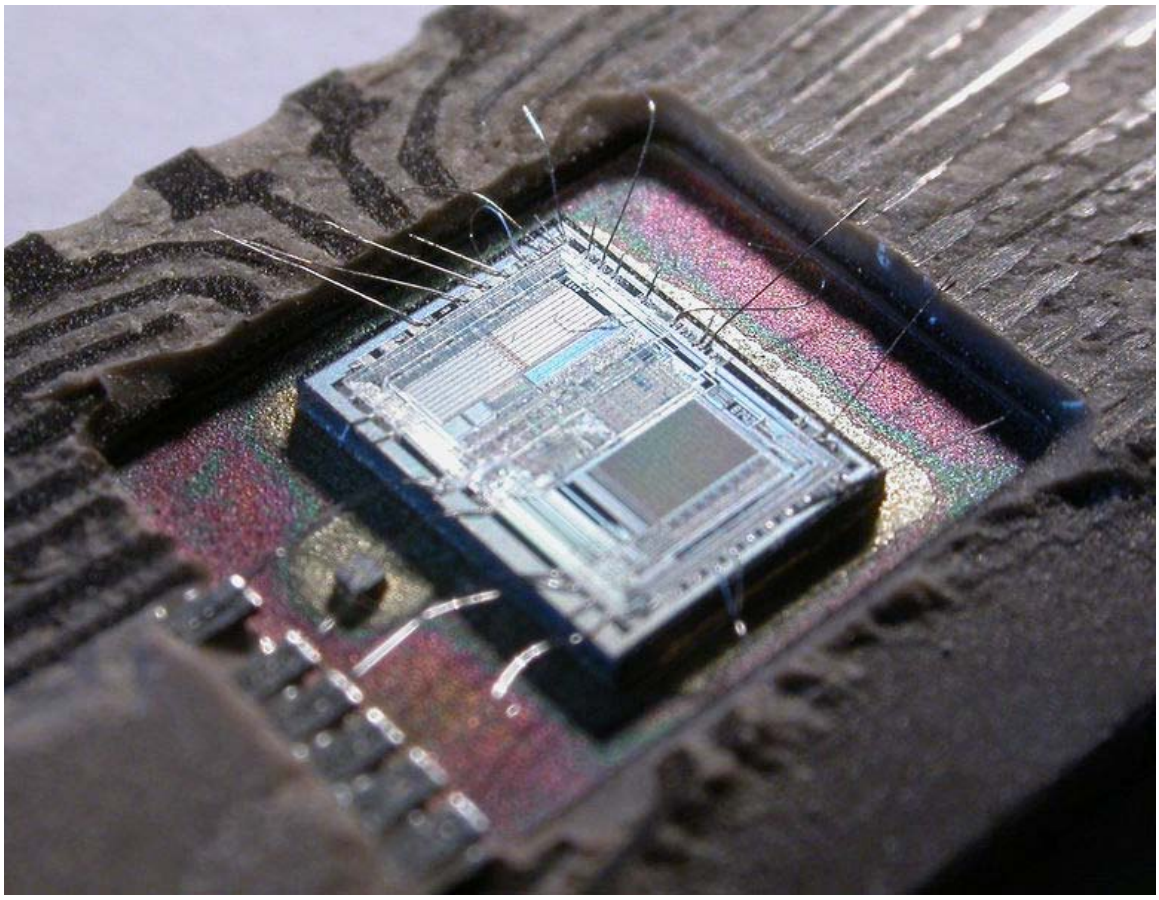
During this period, a method of manufacturing many transistors in a compact space gained popularity. The integrated circuit (**IC**) allowed a large number of transistors to be manufactured on a single semiconductor-based die, or "chip." At first only very basic

non-specialized digital circuits such as NOR gates were miniaturized into ICs. CPUs based upon 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 transistor counts numbering in multiples of ten. 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. As microelectronic technology advanced, an increasing number of transistors were placed on ICs, thus decreasing the quantity of individual ICs needed for a complete CPU. **MSI** and **LSI** (medium- and large-scale integration) ICs increased transistor counts to hundreds, and then thousands.

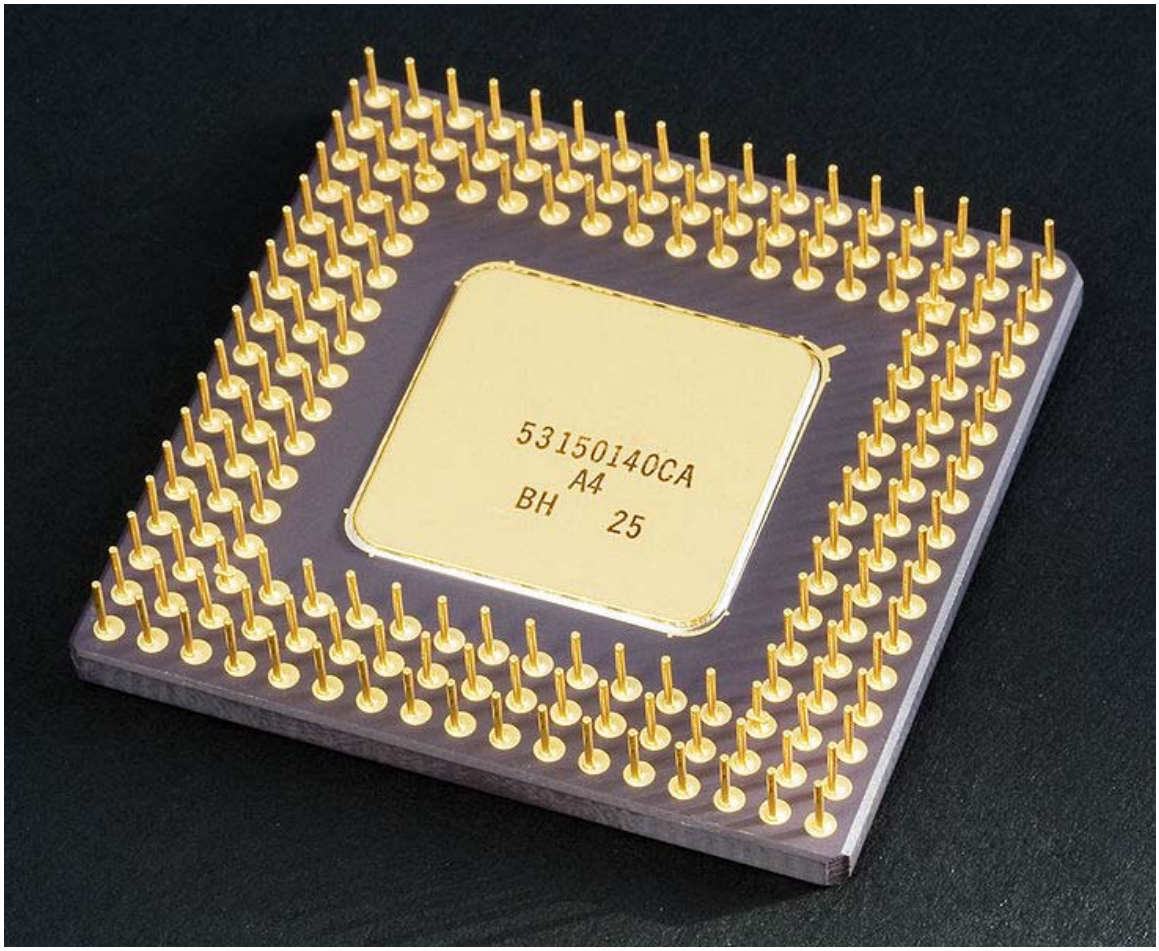
In 1964 IBM introduced its System/360 computer architecture which was used in a series of computers that could run the same programs with different speed and performance. This was significant at a time when most electronic computers were incompatible with one another, even those made by the same manufacturer. To facilitate this improvement, IBM utilized 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 the same year (1964), Digital Equipment Corporation (DEC) introduced another influential computer aimed at the scientific and research markets, the PDP-8. DEC would later introduce the extremely popular PDP-11 line that originally was built with SSI ICs but was eventually implemented with LSI components once these became practical. 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.

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. Thanks to both the increased reliability as well as the dramatically increased speed of the switching elements (which were almost exclusively transistors by this time), CPU clock rates in the tens of megahertz were 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.

Microprocessors



The die from an Intel 8742



Intel 80486DX2 microprocessor in a ceramic PGA package

The introduction of the microprocessor in the 1970s significantly affected the design and implementation of CPUs. 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 vast success of the now ubiquitous personal computer, the term "CPU" is now applied almost exclusively to microprocessors.

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, as the ability to construct exceedingly small

transistors on an IC has increased, the complexity and number of transistors in a single CPU has increased dramatically. This widely observed trend is described by Moore's law, which has proven to be a fairly accurate predictor of the growth of CPU (and other IC) complexity to date.

While the complexity, size, construction, and general form of CPUs have changed drastically over the past sixty years, it is notable that 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 the aforementioned Moore's law continues to hold true, 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.

Operation

The fundamental operation of most CPUs, regardless of the physical form they take, is to execute a sequence of stored instructions called a program. The program is represented by a series of numbers that are kept in some kind of computer memory. There are four steps that nearly all CPUs use in their operation: **fetch**, **decode**, **execute**, and **writeback**.

The first step, **fetch**, involves retrieving an instruction (which is represented by a number or sequence of numbers) from program memory. The location in program memory is determined by a program counter (PC), which stores a number that identifies the current position in the program. In other words, the program counter keeps track of the CPU's place in the program. After an instruction is fetched, the PC is incremented by the length of the instruction word in terms of memory units. 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).

The instruction that the CPU fetches from memory is used to determine what the CPU is to do. In the **decode** step, the instruction is broken up into parts that have significance to other portions of the CPU. The way in which the numerical instruction value is interpreted is defined by the CPU's instruction set architecture (**ISA**). Often, one group of numbers in the instruction, called the opcode, indicates which operation to perform. The remaining parts of the number usually provide information required for that instruction, such as operands for an addition operation. Such operands may be given as a constant value (called an immediate value), or as a place to locate a value: a register or a memory address, as determined by some addressing mode. In older designs the portions of the CPU responsible for instruction decoding were unchangeable hardware devices. However, in more abstract and complicated CPUs and ISAs, a microprogram is often used to assist in translating instructions into various configuration signals for the CPU.

This microprogram is sometimes rewritable so that it can be modified to change the way the CPU decodes instructions even after it has been manufactured.

After the fetch and decode steps, the **execute** step is performed. During this step, various portions of the CPU are connected so they can perform the desired operation. If, for instance, an addition operation was requested, an arithmetic logic unit (**ALU**) will be connected to a set of inputs and a set of outputs. The inputs provide the numbers to be added, and the outputs will contain the final sum. The ALU contains the circuitry to perform simple arithmetic and logical operations on the inputs (like addition and bitwise operations). If the addition operation produces a result too large for the CPU to handle, an arithmetic overflow flag in a flags register may also be set.

The final step, **writeback**, simply "writes back" the results of the execute step to some form of memory. Very often the results are written to some internal CPU register for quick access by subsequent instructions. In other cases results may be written to slower, but cheaper and larger, main memory. Some types of instructions manipulate the program counter rather than directly produce result data. These are generally called "jumps" and facilitate behavior like loops, conditional program execution (through the use of a conditional jump), and functions in programs. Many instructions will also change the state of digits 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, one type of "compare" instruction considers two values and sets a number in the flags register according to which one is greater. This flag could then be used by a later jump instruction to determine program flow.

After the execution of the instruction and writeback of the resulting data, 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 the completed instruction was a jump, 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 than the one described here, multiple instructions can be fetched, decoded, and executed simultaneously. This section describes what is generally referred to as the "Classic RISC pipeline", which in fact 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.

Design and implementation

Integer range

The way a **CPU** represents numbers is a design choice that affects the most basic ways in which the device functions. Some early digital computers used an electrical model of the common decimal (base ten) numeral system to represent numbers internally. A few other computers have used more exotic numeral systems like 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.

MOS 6502 microprocessor in a dual in-line package, an extremely popular 8-bit design

Related to number representation is the size and precision of numbers that a CPU can represent. In the case of a binary CPU, a **bit** refers to one significant place in the numbers a CPU deals with. The number of bits (or numeral places) a CPU uses to represent numbers is often called "word size", "bit width", "data path width", or "integer precision" when dealing with strictly integer numbers (as opposed to floating point). This number differs between architectures, and often within different parts of the very same CPU. For example, an 8-bit CPU deals with a range of numbers that can be represented by eight binary digits (each digit having two possible values), that is, 2^8 or 256 discrete numbers. In effect, integer size sets a hardware limit on the range of integers the software run by the CPU can utilize.

Integer range can also affect the number of locations in memory the CPU can **address** (locate). For example, if a binary CPU uses 32 bits to represent a memory address, and each memory address represents one octet (8 bits), the maximum quantity of memory that CPU can address is 2^{32} octets, or 4 GiB. This is a very simple view of CPU address space, and many designs use more complex addressing methods like paging in order to locate more memory than their integer range would allow with a flat address space.

Higher levels of integer range require more structures to deal with the additional digits, and therefore more complexity, size, power usage, and general expense. It is not at all uncommon, therefore, to see 4- or 8-bit microcontrollers used in modern applications, even though CPUs with much higher range (such as 16, 32, 64, even 128-bit) are available. The simpler microcontrollers are usually cheaper, use less power, and therefore dissipate less heat, all of which can be major design considerations for electronic devices. However, in higher-end applications, the benefits afforded by the extra range (most often the additional address space) are more significant and often affect design choices. To gain some of the advantages afforded by both lower and higher bit lengths, many CPUs are designed with different bit widths for different portions of the device. For example, the IBM System/370 used a CPU that was primarily 32 bit, but it used 128-bit precision inside its floating point units to facilitate greater accuracy and range in floating point numbers. 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.

Clock rate

The clock rate is the speed at which a microprocessor executes instructions. Every computer contains an internal clock that regulates the rate at which instructions are executed and synchronizes all the various computer components. The CPU requires a fixed number of clock ticks (or clock cycles) to execute each instruction. The faster the clock, the more instructions the CPU can execute per second.

Most CPUs, and indeed most sequential logic devices, are synchronous in nature. That is, they are designed and operate on assumptions about a synchronization signal. This signal, known as a **clock signal**, usually takes the form of a periodic square wave. By calculating the maximum time that electrical signals can move in various branches of a CPU's many circuits, the designers can select an appropriate period for the clock signal.

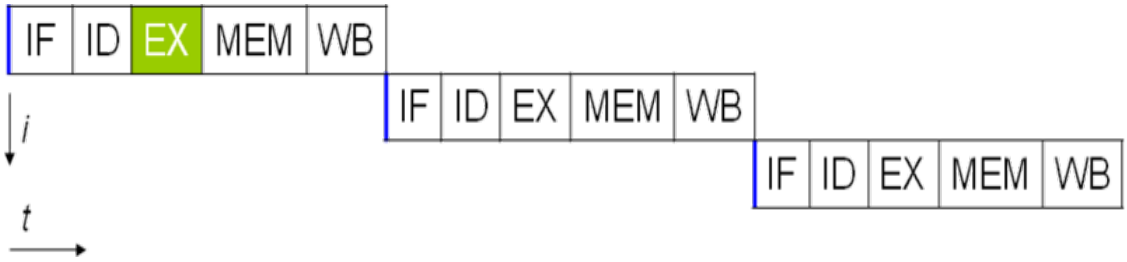
This period must be longer than the amount of time it takes for a signal to move, or propagate, in the worst-case scenario. In setting the clock period to a value well above the worst-case propagation delay, it is possible to design the entire CPU and the way it moves data around the "edges" of the rising 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 in order 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 heat dissipation, causing the CPU to require more effective 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 late CPU design that uses clock gating is that of the IBM PowerPC-based Xbox 360. It utilizes extensive clock gating in order to reduce the power requirements of the aforementioned

videogame console in which it is used. 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 utilizing 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.

Parallelism



Model of a subscalar CPU. Notice that it takes fifteen cycles to complete three instructions.

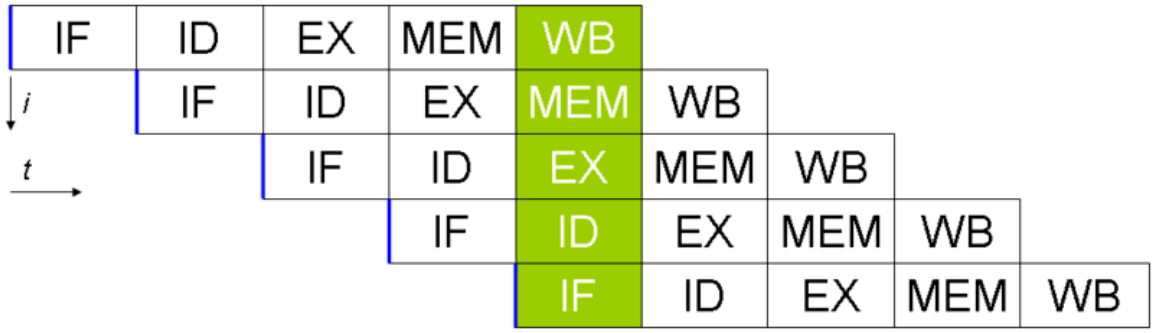
The description of the basic operation of a CPU offered in the previous section describes the simplest form that a CPU can take. This type of CPU, usually referred to as **subscalar**, operates on and executes one instruction on one or two pieces of data at a time.

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). However, the performance is nearly always subscalar (less than one instruction per cycle).

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) seeks to increase the rate at which instructions are executed within a CPU (that is, to increase the utilization of on-die execution resources), and thread level parallelism (TLP) purposes to increase the number of threads (effectively individual programs) 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.

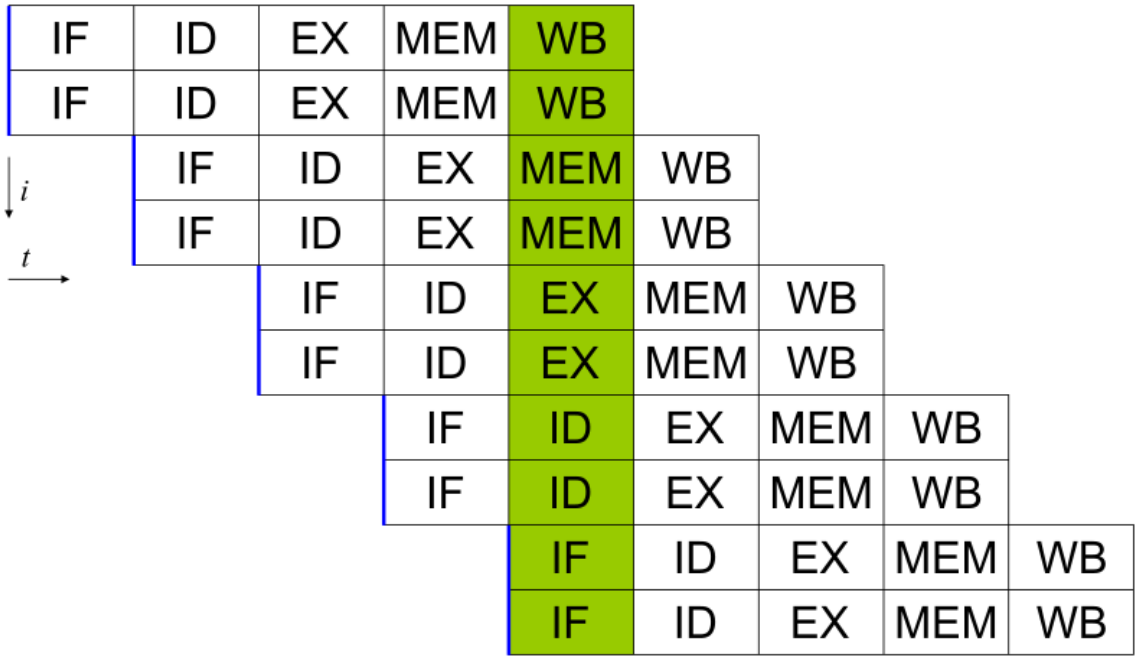
Instruction level parallelism



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



Simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per 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. 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, and out-of-order execution 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 Instructions Multiple Data — 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 increasing 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 each, but its FPU could not accept one instruction per clock. 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 cycle (**IPC**). 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.

Thread level parallelism

Another strategy of achieving performance is to execute multiple programs or threads in parallel. This area of research is known as parallel computing. In Flynn's taxonomy, this strategy is known as Multiple Instructions-Multiple Data or 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 silicon chip, the technology is known as a multi-core microprocessor.

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 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 in order 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 block 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 switch to another thread which is ready to run, the switch often done in one CPU clock cycle, such as the UltraSPARC Technology. Another type of MT is known as simultaneous multithreading, where instructions of 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 program.

This reversal of emphasis is evidenced by the proliferation of dual and multiple core CMP (chip-level multiprocessing) 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 PS3's 7-core Cell microprocessor.

Data parallelism

A less common but increasingly important paradigm of CPUs (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 SISD (single instruction, single data) and SIMD (single instruction, multiple data), respectively. The great utility in creating CPUs 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 are multimedia applications (images, video, and sound), as well as many types of scientific and engineering tasks. Whereas a scalar CPU must complete the entire process of fetching, decoding, and executing each instruction and value in a set of data, a vector CPU can perform a single operation on a comparatively large set of data with one instruction. Of course, 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 CPUs, 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 CPUs has become significant. Shortly after floating point execution units started to become commonplace to include in general-purpose processors, specifications for and implementations of SIMD execution units also began to appear for general-purpose CPUs. 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, these early designs were refined and remade into some of the common, modern SIMD specifications, which are usually associated with one ISA. Some notable modern examples are Intel's SSE and the PowerPC-related AltiVec (also known as VMX).

Performance

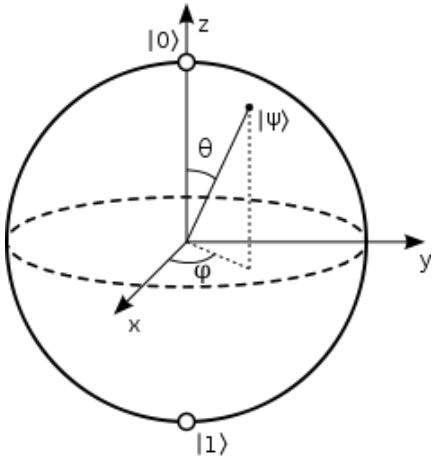
The *performance* or *speed* of a processor depends on the clock rate and the instructions per clock (IPC), which together are the factors for the instructions per second (IPS) that the CPU can perform. Many reported IPS values have represented "peak" execution rates on artificial instruction sequences 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 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, however, the performance gain is far less, only about fifty percent, due to imperfect software algorithms and implementation.

Chapter 4

Quantum Computer and Chemical Computer

Quantum computer



The Bloch sphere is a representation of a qubit, the fundamental building block of quantum computers.

A **quantum computer** is a device for computation that makes direct use of quantum mechanical phenomena, such as superposition and entanglement, to perform operations on data. Quantum computers are different from traditional computers based on transistors. The basic principle behind quantum computation is that quantum properties can be used to represent data and perform operations on these data. A theoretical model is the quantum Turing machine, also known as the universal quantum computer.

Although quantum computing is still in its infancy, experiments have been carried out in which quantum computational operations were executed on a very small number of qubits (quantum bit). Both practical and theoretical research continues, and many national government and military funding agencies support quantum computing research to develop quantum computers for both civilian and national security purposes, such as cryptanalysis.

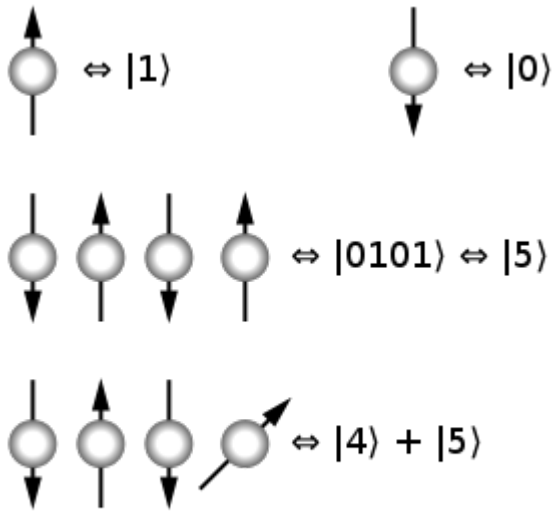
If large-scale quantum computers can be built, they will be able to solve certain problems much faster than any current classical computers (for example Shor's algorithm). Quantum computers do not allow the computation of functions that are not theoretically computable by classical computers, i.e. they do not alter the Church–Turing thesis. The gain is only in efficiency.

Basis

A classical computer has a memory made up of bits, where each bit represents either a one or a zero. A quantum computer maintains a sequence of qubits. A single qubit can represent a one, a zero, or, crucially, any quantum superposition of these; moreover, a pair of qubits can be in any quantum superposition of 4 states, and three qubits in any superposition of 8. In general a quantum computer with *n* qubits can be in an arbitrary superposition of up to 2^n different states simultaneously (this compares to a normal computer that can only be in *one* of these 2^n states at any one time). A quantum computer operates by manipulating those qubits with a fixed sequence of quantum logic gates. The sequence of gates to be applied is called a quantum algorithm.

An example of an implementation of qubits for a quantum computer could start with the use of particles with two spin states: "down" and "up" (typically written $|\downarrow\rangle$ and $|\uparrow\rangle$, or $|0\rangle$ and $|1\rangle$). But in fact any system possessing an observable quantity *A* which is *conserved* under time evolution and such that *A* has at least two discrete and sufficiently spaced consecutive eigenvalues, is a suitable candidate for implementing a qubit. This is true because any such system can be mapped onto an effective spin-1/2 system.

Bits vs. qubits



qubits can be in a superposition of all the clasically allowed states

Qubits are made up of controlled particles and the means of control (e.g. devices that trap particles and switch them from one state to another).

Consider first a classical computer that operates on a three-bit register. The state of the computer at any time is a probability distribution over the $2^3 = 8$ different three-bit strings `000, 001, 010, 011, 100, 101, 110, 111`. If it is a deterministic computer, then it is in exactly one of these states with probability 1. However, if it is a probabilistic computer, then there is a possibility of it being in any *one* of a number of different states. We can describe this probabilistic state by eight nonnegative numbers *a*,*b*,*c*,*d*,*e*,*f*,*g*,*h* (where *a* = probability computer is in state `000`, *b* = probability computer is in state `001`, etc.). There is a restriction that these probabilities sum to 1.

The state of a three-qubit quantum computer is similarly described by an eight-dimensional vector (*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*), called a ket. However, instead of adding to one, the sum of the *squares* of the coefficient magnitudes, $|a|^2 + |b|^2 + ... + |h|^2$, must equal one. Moreover, the coefficients are complex numbers. Since states are represented by complex wavefunctions, two states being added together will undergo interference. This is a key difference between quantum computing and probabilistic classical computing.

If you measure the three qubits, then you will observe a three-bit string. The probability of measuring a string will equal the squared magnitude of that string's coefficients (using our example, probability that we read state as `000` = $|a|^2$, probability that we read state as `001` = $|b|^2$, etc..). Thus a measurement of the quantum state with coefficients (*a*,*b*,...,*h*) gives the classical probability distribution ($|a|^2$, $|b|^2$,..., $|h|^2$). We say that the quantum state "collapses" to a classical state.

Note that an eight-dimensional vector can be specified in many different ways, depending on what basis you choose for the space. The basis of three-bit strings 000, 001, ..., 111 is known as the computational basis, and is often convenient, but other bases of unit-length, orthogonal vectors can also be used. Ket notation is often used to make explicit the choice of basis. For example, the state (*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*) in the computational basis can be written as

$$a\,|000\rangle + b\,|001\rangle + c\,|010\rangle + d\,|011\rangle + e\,|100\rangle + f\,|101\rangle + g\,|110\rangle + h\,|111\rangle,$$

where, e.g., $|010\rangle$= (0,0,1,0,0,0,0,0).

The computational basis for a single qubit (two dimensions) is $|0\rangle$= (1,0), $|1\rangle$= (0,1), but another common basis are the eigenvectors of the Pauli-x operator: $|+\rangle = \frac{1}{\sqrt{2}}(1,1)$ and $|-\rangle = \frac{1}{\sqrt{2}}(1,-1)$.

Note that although recording a classical state of *n* bits, a 2^n-dimensional probability distribution, requires an exponential number of real numbers, practically we can always think of the system as being exactly one of the *n*-bit strings—we just don't know which one. Quantum mechanically, this is not the case, and all 2^n complex coefficients need to be kept track of to see how the quantum system evolves. For example, a 300-qubit quantum computer has a state described by 2^{300} (approximately 10^{90}) complex numbers, more than the number of atoms in the observable universe.

Operation

While a classical three-bit state and a quantum three-qubit state are both eight-dimensional vectors, they are manipulated quite differently for classical or quantum computation. For computing in either case, the system must be initialized, for example into the all-zeros string, $|000\rangle$, corresponding to the vector (1,0,0,0,0,0,0,0). In classical randomized computation, the system evolves according to the application of stochastic matrices, which preserve that the probabilities add up to one (i.e., preserve the L1 norm). In quantum computation, on the other hand, allowed operations are unitary matrices, which are effectively rotations (they preserve that the sum of the squares add up to one, the Euclidean or L2 norm). (Exactly what unitaries can be applied depend on the physics of the quantum device.) Consequently, since rotations can be undone by rotating backward, quantum computations are reversible. (Technically, quantum operations can be probabilistic combinations of unitaries, so quantum computation really does generalize classical computation.)

Finally, upon termination of the algorithm, the result needs to be read off. In the case of a classical computer, we *sample* from the probability distribution on the three-bit register to obtain one definite three-bit string, say 000. Quantum mechanically, we *measure* the three-qubit state, which is equivalent to collapsing the quantum state down to a classical distribution (with the coefficients in the classical state being the squared magnitudes of the coefficients for the quantum state, as described above) followed by sampling from

that distribution. Note that this destroys the original quantum state. Many algorithms will only give the correct answer with a certain probability, however by repeatedly initializing, running and measuring the quantum computer, the probability of getting the correct answer can be increased.

Potential

Integer factorization is believed to be computationally infeasible with an ordinary computer for large integers if they are the product of few prime numbers (e.g., products of two 300-digit primes). By comparison, a quantum computer could efficiently solve this problem using Shor's algorithm to find its factors. This ability would allow a quantum computer to decrypt many of the cryptographic systems in use today, in the sense that there would be a polynomial time (in the number of digits of the integer) algorithm for solving the problem. In particular, most of the popular public key ciphers are based on the difficulty of factoring integers (or the related discrete logarithm problem which can also be solved by Shor's algorithm), including forms of RSA. These are used to protect secure Web pages, encrypted email, and many other types of data. Breaking these would have significant ramifications for electronic privacy and security.

However, other existing cryptographic algorithms don't appear to be broken by these algorithms. Some public-key algorithms are based on problems other than the integer factorization and discrete logarithm problems to which Shor's algorithm applies, like the McEliece cryptosystem based on a problem in coding theory. Lattice based cryptosystems are also not known to be broken by quantum computers, and finding a polynomial time algorithm for solving the dihedral hidden subgroup problem, which would break many lattice based cryptosystems, is a well-studied open problem. It has been proven that applying Grover's algorithm to break a symmetric (secret key) algorithm by brute force requires roughly $2^{n/2}$ invocations of the underlying cryptographic algorithm, compared with roughly 2^n in the classical case, meaning that symmetric key lengths are effectively halved: AES-256 would have the same security against an attack using Grover's algorithm that AES-128 has against classical brute-force search. Quantum cryptography could potentially fulfill some of the functions of public key cryptography.

Besides factorization and discrete logarithms, quantum algorithms offering a more than polynomial speedup over the best known classical algorithm have been found for several problems, including the simulation of quantum physical processes from chemistry and solid state physics, the approximation of Jones polynomials, and solving Pell's equation. No mathematical proof has been found that shows that an equally fast classical algorithm cannot be discovered, although this is considered unlikely. For some problems, quantum computers offer a polynomial speedup. The most well-known example of this is *quantum database search*, which can be solved by Grover's algorithm using quadratically fewer queries to the database than are required by classical algorithms. In this case the advantage is provable. Several other examples of provable quantum speedups for query problems have subsequently been discovered, such as for finding collisions in two-to-one functions and evaluating NAND trees.

Consider a problem that has these four properties:

1. The only way to solve it is to guess answers repeatedly and check them,
2. There are *n* possible answers to check,
3. Every possible answer takes the same amount of time to check, and
4. There are no clues about which answers might be better: generating possibilities randomly is just as good as checking them in some special order.

An example of this is a password cracker that attempts to guess the password for an encrypted file (assuming that the password has a maximum possible length).

For problems with all four properties, the time for a quantum computer to solve this will be proportional to the square root of *n*. That can be a very large speedup, reducing some problems from years to seconds. It can be used to attack symmetric ciphers such as Triple DES and AES by attempting to guess the secret key.

Grover's algorithm can also be used to obtain a quadratic speed-up [over a brute-force search] for a class of problems known as NP-complete.

Since chemistry and nanotechnology rely on understanding quantum systems, and such systems are impossible to simulate in an efficient manner classically, many believe quantum simulation will be one of the most important applications of quantum computing.

There are a number of practical difficulties in building a quantum computer, and thus far quantum computers have only solved trivial problems. David DiVincenzo, of IBM, listed the following requirements for a practical quantum computer:

- scalable physically to increase the number of qubits;
- qubits can be initialized to arbitrary values;
- quantum gates faster than decoherence time;
- universal gate set;
- qubits can be read easily.

Quantum decoherence

One of the greatest challenges is controlling or removing quantum decoherence. This usually means isolating the system from its environment as the slightest interaction with the external world would cause the system to decohere. This effect is irreversible, as it is non-unitary, and is usually something that should be highly controlled, if not avoided. Decoherence times for candidate systems, in particular the transverse relaxation time T_2 (for NMR and MRI technology, also called the *dephasing time*), typically range between nanoseconds and seconds at low temperature.

These issues are more difficult for optical approaches as the timescales are orders of magnitude shorter and an often-cited approach to overcoming them is optical pulse

shaping. Error rates are typically proportional to the ratio of operating time to decoherence time, hence any operation must be completed much more quickly than the decoherence time.

If the error rate is small enough, it is thought to be possible to use quantum error correction, which corrects errors due to decoherence, thereby allowing the total calculation time to be longer than the decoherence time. An often cited figure for required error rate in each gate is 10^{-4}. This implies that each gate must be able to perform its task in one 10,000th of the decoherence time of the system.

Meeting this scalability condition is possible for a wide range of systems. However, the use of error correction brings with it the cost of a greatly increased number of required qubits. The number required to factor integers using Shor's algorithm is still polynomial, and thought to be between L and L^2, where L is the number of bits in the number to be factored; error correction algorithms would inflate this figure by an additional factor of L. For a 1000-bit number, this implies a need for about 10^4 qubits without error correction. With error correction, the figure would rise to about 10^7 qubits. Note that computation time is about L^2 or about 10^7 steps and on 1 MHz, about 10 seconds.

A very different approach to the stability-decoherence problem is to create a topological quantum computer with anyons, quasi-particles used as threads and relying on braid theory to form stable logic gates.

Developments

There are a number of quantum computing candidates, among those:

- Superconductor-based quantum computers (including SQUID-based quantum computers)
- Trapped ion quantum computer
- Optical lattices
- Topological quantum computer
- Quantum dot on surface (e.g. the Loss-DiVincenzo quantum computer)
- Nuclear magnetic resonance on molecules in solution (liquid NMR)
- Solid state NMR Kane quantum computers
- Electrons on helium quantum computers
- Cavity quantum electrodynamics (CQED)
- Molecular magnet
- Fullerene-based ESR quantum computer
- Optic-based quantum computers (Quantum optics)
- Diamond-based quantum computer
- Bose–Einstein condensate-based quantum computer
- Transistor-based quantum computer - string quantum computers with entrainment of positive holes using an electrostatic trap
- Spin-based quantum computer
- Adiabatic quantum computation

- Rare-earth-metal-ion-doped inorganic crystal based quantum computers

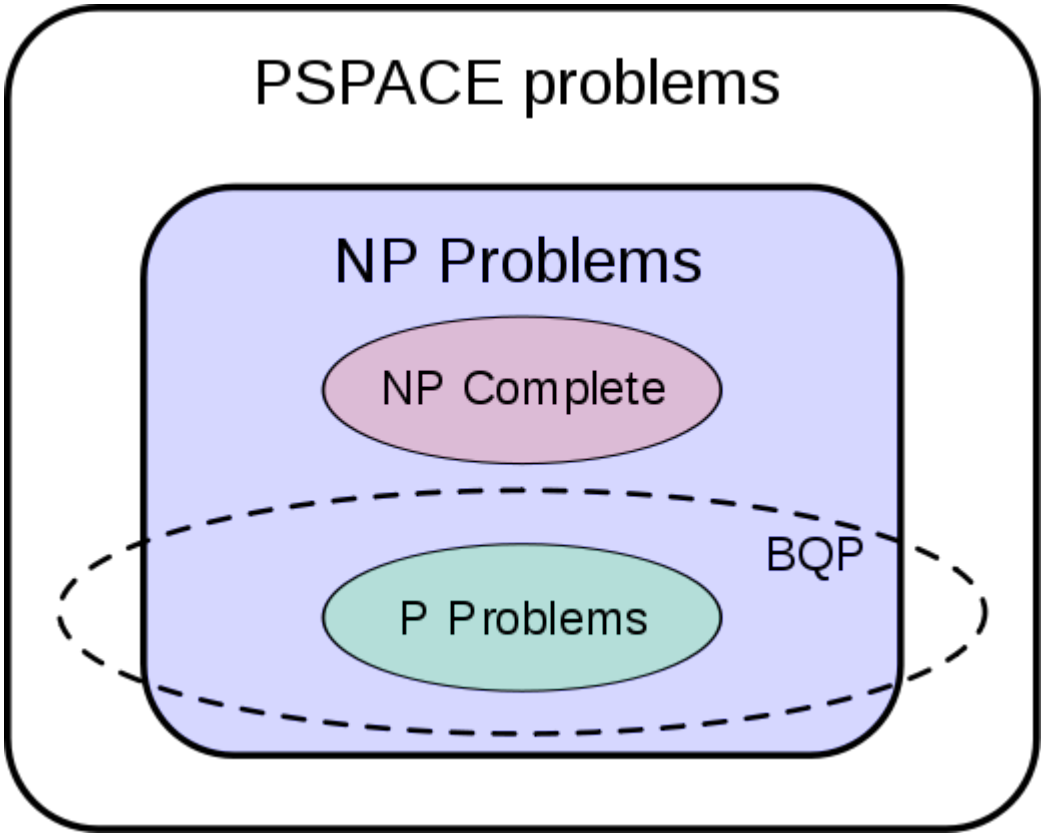
The large number of candidates demonstrates that the topic, in spite of rapid progress, is still in its infancy. But at the same time there is also a vast amount of flexibility.

In 2005, researchers at the University of Michigan built a semiconductor chip which functioned as an ion trap. Such devices, produced by standard lithography techniques, may point the way to scalable quantum computing tools. An improved version was made in 2006.

In 2009, researchers at Yale University created the first rudimentary solid-state quantum processor. The two-qubit superconducting chip was able to run elementary algorithms. Each of the two artificial atoms (or qubits) were made up of a billion aluminum atoms but they acted like a single one that could occupy two different energy states.

Another team, working at the University of Bristol, also created a silicon-based quantum computing chip, based on quantum optics. The team was able to run Shor's algorithm on the chip. The latest developments [for 2010] can be found in .

Relation to computational complexity theory



The suspected relationship of BQP to other problem spaces

The class of problems that can be efficiently solved by quantum computers is called BQP, for "bounded error, quantum, polynomial time". Quantum computers only run probabilistic algorithms, so BQP on quantum computers is the counterpart of BPP ("bounded error, probabilistic, polynomial time") on classical computers. It is defined as the set of problems solvable with a polynomial-time algorithm, whose probability of error is bounded away from one half. A quantum computer is said to "solve" a problem if, for every instance, its answer will be right with high probability. If that solution runs in polynomial time, then that problem is in BQP.

BQP is contained in the complexity class $\#P$ (or more precisely in the associated class of decision problems $P^{\#P}$), which is a subclass of PSPACE.

BQP is suspected to be disjoint from NP-complete and a strict superset of P, but that is not known. Both integer factorization and discrete log are in BQP. Both of these problems are NP problems suspected to be outside BPP, and hence outside P. Both are suspected to not be NP-complete. There is a common misconception that quantum computers can solve NP-complete problems in polynomial time. That is not known to be true, and is generally suspected to be false.

Although quantum computers may be faster than classical computers, those described above can't solve any problems that classical computers can't solve, given enough time and memory (however, those amounts might be practically infeasible). A Turing machine can simulate these quantum computers, so such a quantum computer could never solve an undecidable problem like the halting problem. The existence of "standard" quantum computers does not disprove the Church–Turing thesis. It has been speculated that theories of quantum gravity, such as M-theory or loop quantum gravity, may allow even faster computers to be built. Currently, it's an open problem to even *define* computation in such theories due to the problem of time, i.e. there's no obvious way to describe what it means for an observer to submit input to a computer and later receive output.

Chemical computer

A **chemical computer**, also called reaction-diffusion computer, BZ computer or gooware computer is an unconventional computer based on a semi-solid chemical "soup" where data is represented by varying concentrations of chemicals. The computations are performed by naturally occurring chemical reactions. So far it is still in a very early experimental stage, but may have great potential for the computer industry.

Rationale

The simplicity of this technology is one of the main reasons why it in the future could turn into a serious competitor to machines based on conventional hardware. A modern

microprocessor is an incredibly complicated device that can be destroyed during production by no more than a single airborne microscopic particle. In contrast a cup of chemicals is a simple and stable component that is cheap to produce.

In a conventional microprocessor the bits behave much like cars in city traffic; they can only use certain roads, they have to slow down and wait for each other in crossing traffic, and only one driving field at once can be used. In a BZ solution the waves are moving in all thinkable directions in all dimensions, across, away and against each other. These properties might make a chemical computer able to handle billions of times more data than a traditional computer. An analogy would be the brain; even if a microprocessor can transfer information much faster than a neuron, the brain is still much more effective for some tasks because it can work with a much higher amount of data at the same time.

Historical background

Originally chemical reactions were seen as a simple move towards a stable equilibrium which was not very promising for computation. This was changed by a discovery made by Boris Belousov, a Soviet scientist, in the 1950s. He created a chemical reaction between different salts and acids that swing back and forth between being yellow and clear because the concentration of the different components changes up and down in a cyclic way. At the time this was considered impossible because it seemed to go against the second law of thermodynamics, which says that in a closed system the entropy will only increase over time, causing the components in the mixture to distribute themselves till equilibrium is gained and making any changes in the concentration impossible. But modern theoretical analyses shows sufficiently complicated reactions can indeed comprise wave phenomena without breaking the laws of nature. (A convincing directly visible demonstration was achieved by Anatol Zhabotinsky with the Belousov-Zhabotinsky reaction showing spiraling colored waves.)

Basic principles

The wave properties of the BZ reaction means it can move information in the same way as all other waves. This still leaves the need for computation, performed by conventional microchips using the binary code transmitting and changing ones and zeros through a complicated system of logic gates. To perform any conceivable computation it is sufficient to have NAND gates. (A NAND gate has two bits input. Its output is 0 if both bits are 1, otherwise it's 1). In the chemical computer version logic gates are implemented by concentration waves blocking or amplifying each other in different ways.

Current research

In 1989 it was demonstrated how light-sensitive chemical reactions could perform image processing. This led to an upsurge in the field of chemical computing. Andrew Adamatzky at the University of the West of England has demonstrated simple logic gates

using reaction-diffusion processes. Furthermore he has theoretically shown how a hypothetical "2^+ medium" modelled as a cellular automaton can perform computation.

The breakthrough came when he read a theoretical article of two scientists who illustrated how to make logic gates to a computer by using the balls on a billiard table as an example. Like in the case with the AND-gate, two balls represents two different bits. If a single ball shoots towards a common colliding point, the bit is 1. If not, it is 0. A collision will only occur if both balls are sent toward the point, which then is registered in the same way as when two electronic 1's gives a new and single 1. In this way the balls work together like an AND-gate. Adamatzkys' great achievement was to transfer this principle to the BZ-chemicale and replace the billiard balls with waves. If it occurs two waves in the solution, they will meet and create as a third wave which is registered as a 1. He has tested the theory in practice and has already documented that it works. For the moment he is cooperating with some other scientists in producing some thousand chemical versions of logic gates that is going to become a form of chemical pocket calculator. One of the problems with the present version of this technology is the speed of the waves; they only spread at a rate of a few millimeters per minute. According to Adamatzky, this problem can be eliminated by placing the gates very close to each other, to make sure the signals are transferred quickly. Another possibility could be new chemical reactions where waves propagate much faster. If these teething problems are overcome, a chemical computer will offer clear advantages over an electronic computer.

An increasing number of individuals in the computer industry are starting to realise the potential of this technology. IBM is at the moment testing out new ideas in the field of microprocessing with many similarities to the basic principles of a chemical computer.

Chapter 5

Vector Processor and Non-Uniform Memory Access

Vector processor

A **vector processor**, or **array processor**, is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called *vectors*. This is in contrast to a scalar processor, whose instructions operate on single data items. The vast majority of CPUs are scalar.

Vector processors first appeared in the 1970s, and formed the basis of most supercomputers through the 1980s and into the 1990s. Improvements in scalar processors, particularly microprocessors, resulted in the decline of traditional vector processors in supercomputers, and the appearance of vector processing techniques in mass market CPUs around the early 1990s. Today, most commodity CPUs implement architectures that feature instructions for some vector processing on multiple (vectorized) data sets, typically known as SIMD (**S**ingle **I**nstruction, **M**ultiple **D**ata). Common examples include MMX, SSE, and AltiVec. Vector processing techniques are also found in video game console hardware and graphics accelerators. In 2000, IBM, Toshiba and Sony collaborated to create the Cell processor, consisting of one scalar processor and eight vector processors, which found use in the Sony PlayStation 3 among other applications.

Other CPU designs may include some multiple instructions for vector processing on multiple (vectorised) data sets, typically known as MIMD (**M**ultiple **I**nstruction, **M**ultiple **D**ata). Such designs are usually dedicated to a particular application and not commonly marketed for general purpose computing.

History

Vector processing was first worked on in the early 1960s at Westinghouse in their **Solomon** project. Solomon's goal was to dramatically increase math performance by using a large number of simple math co-processors under the control of a single master CPU. The CPU fed a single common instruction to all of the arithmetic logic units

(ALUs), one per "cycle", but with a different data point for each one to work on. This allowed the Solomon machine to apply a single algorithm to a large data set, fed in the form of an array. In 1962, Westinghouse cancelled the project, but the effort was re-started at the University of Illinois as the ILLIAC IV. Their version of the design originally called for a 1 GFLOPS machine with 256 ALUs, but, when it was finally delivered in 1972, it had only 64 ALUs and could reach only 100 to 150 MFLOPS. Nevertheless it showed that the basic concept was sound, and, when used on data-intensive applications, such as computational fluid dynamics, the "failed" ILLIAC was the fastest machine in the world. The ILLIAC approach of using separate ALUs for each data element is not common to later designs, and is often referred to under a separate category, massively parallel computing.

The first *successful* implementation of vector processing appears to be the Control Data Corporation STAR-100 and the Texas Instruments Advanced Scientific Computer (ASC). The basic ASC (i.e., "one pipe") ALU used a pipeline architecture that supported both scalar and vector computations, with peak performance reaching approximately 20 MFLOPS, readily achieved when processing long vectors. Expanded ALU configurations supported "two pipes" or "four pipes" with a corresponding 2X or 4X performance gain. Memory bandwidth was sufficient to support these expanded modes. The STAR was otherwise slower than CDC's own supercomputers like the CDC 7600, but at data related tasks they could keep up while being much smaller and less expensive. However the machine also took considerable time decoding the vector instructions and getting ready to run the process, so it required very specific data sets to work on before it actually sped anything up.

The vector technique was first fully exploited in the famous Cray-1. Instead of leaving the data in memory like the STAR and ASC, the Cray design had eight "vector registers," which held sixty-four 64-bit words each. The vector instructions were applied between registers, which is much faster than talking to main memory. The Cray design used pipeline parallelism to implement vector instructions rather than multiple ALUs. In addition the design had completely separate pipelines for different instructions, for example, addition/subtraction was implemented in different hardware than multiplication. This allowed a batch of vector instructions themselves to be pipelined, a technique they called *vector chaining*. The Cray-1 normally had a performance of about 80 MFLOPS, but with up to three chains running it could peak at 240 MFLOPS – a respectable number even as of 2002.

Other examples followed. Control Data Corporation tried to re-enter the high-end market again with its ETA-10 machine, but it sold poorly and they took that as an opportunity to leave the supercomputing field entirely. In the early and mid-1980s Japanese companies (Fujitsu, Hitachi and Nippon Electric Corporation (NEC) introduced register-based vector machines similar to the Cray-1, typically being slightly faster and much smaller. Oregon-based Floating Point Systems (FPS) built add-on array processors for minicomputers, later building their own minisupercomputers. However Cray continued to be the performance leader, continually beating the competition with a series of machines that led to the Cray-2, Cray X-MP and Cray Y-MP. Since then, the supercomputer market has

focused much more on massively parallel processing rather than better implementations of vector processors. However, recognising the benefits of vector processing IBM developed Virtual Vector Architecture for use in supercomputers coupling several scalar processors to act as a vector processor.

Vector processing techniques have since been added to almost all modern CPU designs, although they are typically referred to as SIMD. In these implementations, the vector unit runs beside the main scalar CPU, and is fed data from programs that know it is there.

Description

In general terms, CPUs are able to manipulate one or two pieces of data at a time. For instance, many CPUs have an instruction that essentially says "add A to B and put the result in C". The data for A, B and C could be—in theory at least—encoded directly into the instruction. However things are rarely that simple. In general the data is rarely sent in raw form, and is instead "pointed to" by passing in an address to a memory location that holds the data. Decoding this address and getting the data out of the memory takes some time. As CPU speeds have increased, this *memory latency* has historically become a large impediment to performance.

In order to reduce the amount of time this takes, most modern CPUs use a technique known as instruction pipelining in which the instructions pass through several sub-units in turn. The first sub-unit reads the address and decodes it, the next "fetches" the values at those addresses, and the next does the math itself. With pipelining the "trick" is to start decoding the next instruction even before the first has left the CPU, in the fashion of an assembly line, so the address decoder is constantly in use. Any particular instruction takes the same amount of time to complete, a time known as the *latency*, but the CPU can process an entire batch of operations much faster than if it did so one at a time.

Vector processors take this concept one step further. Instead of pipelining just the instructions, they also pipeline the data itself. They are fed instructions that say not just to add A to B, but to add all of the numbers "from here to here" to all of the numbers "from there to there". Instead of constantly having to decode instructions and then fetch the data needed to complete them, it reads a single instruction from memory, and "knows" that the next address will be one larger than the last. This allows for significant savings in decoding time.

To illustrate what a difference this can make, consider the simple task of adding two groups of 10 numbers together. In a normal programming language you would write a "loop" that picked up each of the pairs of numbers in turn, and then added them. To the CPU, this would look something like this:

```
execute this loop 10 times
  read the next instruction and decode it
  fetch this number
  fetch that number
  add them
```

```
  put the result here
end loop
```

But to a vector processor, this task looks considerably different:

```
read instruction and decode it
fetch these 10 numbers
fetch those 10 numbers
add them
put the results here
```

There are several savings inherent in this approach. For one, only two address translations are needed. Depending on the architecture, this can represent a significant savings by itself. Another saving is fetching and decoding the instruction itself, which has to be done only one time instead of ten. The code itself is also smaller, which can lead to more efficient memory use.

But more than that, a vector processor may have multiple functional units adding those numbers in parallel. The checking of dependencies between those numbers is not required as a vector instruction specifies multiple independent operations. This simplifies the control logic required, and can improve performance by avoiding stalls.

As mentioned earlier, the Cray implementations took this a step further, allowing several different types of operations to be carried out at the same time. Consider code that adds two numbers and then multiplies by a third; in the Cray, these would all be fetched at once, and both added and multiplied in a single operation. Using the pseudocode above, the Cray did:

```
read instruction and decode it
fetch these 10 numbers
fetch those 10 numbers
fetch another 10 numbers
add and multiply them
put the results here
```

The math operations thus completed far faster overall, the limiting factor being the time required to fetch the data from memory.

Not all problems can be attacked with this sort of solution. Adding these sorts of instructions necessarily adds complexity to the core CPU. That complexity typically makes *other* instructions run slower—i.e., whenever it is **not** adding up many numbers in a row. The more complex instructions also add to the complexity of the decoders, which might slow down the decoding of the more common instructions such as normal adding.

In fact, vector processors work best only when there are large amounts of data to be worked on. For this reason, these sorts of CPUs were found primarily in supercomputers, as the supercomputers themselves were, in general, found in places such as weather prediction centres and physics labs, where huge amounts of data are "crunched".

Real world example: vector instructions usage with the x86 architecture

Shown below is a actual x86 architecture example for vector instruction usage with the SSE instruction set. The example multiplies two arrays of single precision floating point numbers. It's written in the C language with inline assembly code parts for compilation with GCC (32bit).

//SSE simd function for vectorized multiplication of 2 arrays with single-precision floatingpoint numbers
//1st param pointer on source/destination array, 2nd param 2. source array, 3th param number of floats per array

```
 void mul_asm(float* out, float* in, unsigned int leng)
 {    unsigned int count, rest;

       //compute if array is big enough for vector operation
       rest  = (leng*4)%16;
       count = (leng*4)-rest;

      // vectorized part; 4 floats per loop iteration
       if (count>0){
       __asm __volatile__  (".intel_syntax noprefix\n\t"
       "loop:                  \n\t"
       "movups xmm0,[ebx+ecx] ;loads 4 floats in first
register (xmm0)\n\t"
       "movups xmm1,[eax+ecx] ;loads 4 floats in second
register (xmm1)\n\t"
       "mulps xmm0,xmm1       ;multiplies both vector
registers\n\t"
       "movups [eax+ecx],xmm0 ;write back the result to
memory\n\t"
       "sub ecx,16            ;increase address pointer by 4
floats\n\t"
       "jnz loop              \n\t"
       ".att_syntax prefix    \n\t"
         : : "a" (out), "b" (in), "c"(count), "d"(rest):
"xmm0","xmm1");
       }

       // scalar part; 1 float per loop iteration
       if (rest!=0)
       {
         __asm __volatile__  (".intel_syntax noprefix\n\t"
       "add eax,ecx           \n\t"
       "add ebx,ecx           \n\t"
```

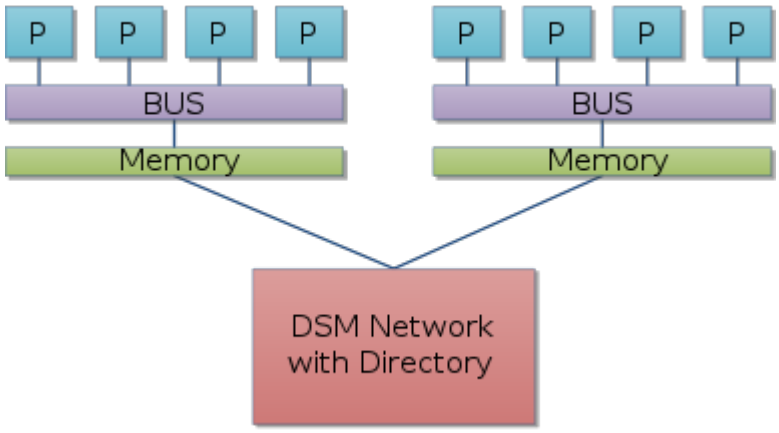
```
      "rest:                  \n\t"
      "movss xmm0,[ebx+edx]  ;load 1 float in first
register (xmm0)\n\t"
      "movss xmm1,[eax+edx]  ;load 1 float in second
register (xmm1)\n\t"
      "mulss xmm0,xmm1       ;multiplies both scalar parts
of registers\n\t"
      "movss [eax+edx],xmm0  ;write back the result\n\t"
      "sub edx,4             \n\t"
      "jnz rest              \n\t"
      ".att_syntax prefix    \n\t"
        : : "a" (out), "b" (in), "c"(count), "d"(rest):
"xmm0","xmm1");
      }
      return;
 }
```

Non-Uniform Memory Access

Non-Uniform Memory Access or **Non-Uniform Memory Architecture** (**NUMA**) is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors.

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. Their commercial development came in work by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Silicon Graphics, Sequent Computer Systems, Data General and Digital during the 1990s. Techniques developed by these companies later featured in a variety of Unix-like operating systems, and somewhat in Windows NT.

Basic concept



One possible architecture of a NUMA system. Notice that the processors are connected to the bus or crossbar by connections of varying thickness/number. This shows that different CPUs have different priorities to memory access based on their location.

Modern CPUs operate considerably faster than the main memory to which they are attached. In the early days of computing and data processing the CPU generally ran slower than its memory. The performance lines crossed in the 1960s with the advent of the first supercomputers and high-speed computing. Since then, CPUs, increasingly *starved for data*, have had to stall while they wait for memory accesses to complete. Many supercomputer designs of the 1980s and 90s focused on providing high-speed memory access as opposed to faster processors, allowing them to work on large data sets at speeds other systems could not approach.

Limiting the number of memory accesses provided the key to extracting high performance from a modern computer. For commodity processors, this means installing an ever-increasing amount of high-speed cache memory and using increasingly sophisticated algorithms to avoid "cache misses". But the dramatic increase in size of the operating systems and of the applications run on them has generally overwhelmed these cache-processing improvements. Multi-processor systems make the problem considerably worse. Now a system can starve several processors at the same time, notably because only one processor can access memory at a time.

NUMA attempts to address this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory. For problems involving spread data (common for servers and similar applications), NUMA can improve the performance over a single shared memory by a factor of roughly the number of processors (or separate memory banks).

Of course, not all data ends up confined to a single task, which means that more than one processor may require the same data. To handle these cases, NUMA systems include

additional hardware or software to move data between banks. This operation has the effect of slowing down the processors attached to those banks, so the overall speed increase due to NUMA will depend heavily on the exact nature of the tasks run on the system at any given time.

Cache coherent NUMA (ccNUMA)

Nearly all CPU architectures use a small amount of very fast non-shared memory known as cache to exploit locality of reference in memory accesses. With NUMA, maintaining cache coherence across shared memory has a significant overhead.

Although simpler to design and build, non-cache-coherent NUMA systems become prohibitively complex to program in the standard von Neumann architecture programming model. As a result, all NUMA computers sold to the market use special-purpose hardware to maintain cache coherence, and thus class as "cache-coherent NUMA", or **ccNUMA**.

Typically, this takes place by using inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, ccNUMA may perform poorly when multiple processors attempt to access the same memory area in rapid succession. Operating-system support for NUMA attempts to reduce the frequency of this kind of access by allocating processors and memory in NUMA-friendly ways and by avoiding scheduling and locking algorithms that make NUMA-unfriendly accesses necessary. Alternatively, cache coherency protocols such as the MESIF protocol attempt to reduce the communication required to maintain cache coherency. Scalable Coherent Interface (SCI) is an IEEE standard defining a directory based cache coherency protocol to avoid scalability limitations found in earlier multiprocessor systems. SCI is used as basis for the Numascale NumaConnect technology.

Current ccNUMA systems are multiprocessor systems based on the AMD Opteron, which can be implemented without external logic, and Intel Itanium, which requires the chipset to support NUMA. Examples of ccNUMA enabled chipsets are the SGI Shub (Super hub), the Intel E8870, the HP sx2000 (used in the Integrity and Superdome servers), and those found in recent NEC Itanium-based systems. Earlier ccNUMA systems such as those from Silicon Graphics were based on MIPS processors and the DEC Alpha 21364 (EV7) processor.

Intel announced NUMA Template:Unclear - ccNUMA? introduction to its x86 and Itanium servers in late 2007 with Nehalem and Tukwila CPUs . Both CPU families will share a common chipset; the interconnection is called Intel Quick Path Interconnect (QPI).

NUMA vs. cluster computing

One can view NUMA as a very tightly coupled form of cluster computing. The addition of virtual memory paging to a cluster architecture can allow the implementation of NUMA entirely in software where no NUMA hardware exists. However, the inter-node latency of software-based NUMA remains several orders of magnitude greater than that of hardware-based NUMA.

Chapter 6

Register Machine and Harvard Architecture

Register machine

In mathematical logic and theoretical computer science a **register machine** is a generic class of abstract machines used in a manner similar to a Turing machine. All the models are Turing equivalent.

Overview

The register machine gets its name from its one or more "registers" – in place of a Turing machine's tape and head (or tapes and heads) the model uses **multiple, uniquely-addressed registers**, each of which holds a single positive integer.

There are at least 4 sub-classes found in the literature, here listed from most primitive to the most like a computer:

- counter machine – the most primitive and reduced model. Lacks indirect addressing. Instructions are in the finite state machine in the manner of the Harvard architecture.
- Pointer machine – a blend of counter machine and RAM models. Less common and more abstract than either model. Instructions are in the finite state machine in the manner of the Harvard architecture.
- Random access machine (RAM) – a counter machine with indirect addressing and, usually, an augmented instruction set. Instructions are in the finite state machine in the manner of the Harvard architecture.
- Random access stored program machine model (RASP) – a RAM with instructions in its registers analogous to the Universal Turing machine; thus it is an example of the von Neumann architecture. But unlike a computer the model is *idealized* with effectively-infinite registers (and if used, effectively-infinite special registers such as an accumulator). Unlike a computer or even a RISC, the instruction set is much reduced in the number of instructions.

Any properly-defined register machine model is Turing equivalent. Computational speed is very dependent on the model specifics.

In practical computer science, a similar concept known as a virtual machine is sometimes used to minimise dependencies on underlying machine architectures. Such machines are also used for teaching. The term "register machine" is sometimes used to refer to a virtual machine in textbooks.

Formal definition

No standard terminology exists; each author is responsible for defining in prose the meanings of their mnemonics or symbols. Many authors use a "register-transfer"-like symbolism to explain the actions of their models, but again they are responsible for defining its syntax.

A register machine consists of:

1. **An unbounded number of labeled, discrete, unbounded registers unbounded in extent (capacity)**: a finite (or infinite in some models) set of registers $r_0 \ldots r_n$ each considered to be of infinite extent and each of which holds a single non-negative integer (0, 1, 2, ...). The registers may do their own arithmetic, or there may be one or more special registers that do the arithmetic e.g. an "accumulator" and/or "address register"..
2. **Tally counters or marks**: discrete, indistinguishable objects or marks of only one sort suitable for the model. In the most-reduced counter machine model, per each arithmetic operation only one object/mark is either added to or removed from its location/tape. In some counter machine models (e.g. Melzak (1961), Minsky (1961)) and most RAM and RASP models more than one object/mark can be added or removed in one operation with "addition" and usually "subtraction"; sometimes with "multiplication" and/or "division". Some models have control operations such as "copy" (variously: "move", "load", "store") that move "clumps" of objects/marks from register to register in one action.
3. **A (very) limited set of instructions**: the instructions tend to divide into two classes: arithmetic and control. The instructions are drawn from the two classes to form "instruction-sets", such that an instruction set must allow the model to be Turing equivalent (it must be able to compute any partial recursive function).
    1. **Arithmetic**: arithmetic instructions may operate on all registers or on just a special register (e.g. accumulator). They are *usually* chosen from the following sets (but exceptions abound):
        - Counter machine: { Increment (r), Decrement (r), Clear-to-zero (r) }
        - Reduced RAM, RASP: { Increment (r), Decrement (r), Clear-to-zero (r), Load-immediate-constant k, Add (r_1,r_2), proper-Subtract (r_1,r_2), Increment accumulator, Decrement accumulator, Clear accumulator, Add to accumulator contents of register r, proper-Subtract from accumulator contents of register r, }

        - Augmented RAM, RASP: All of the reduced instructions plus: { Multiply, Divide, various Boolean bit-wise (left-shift, bit test, etc.)}
    2. **Control**:
        - Counter machine models: optional { Copy (r_1,r_2) }
        - RAM and RASP models: most have { Copy (r_1,r_2) }, or { Load Accumulator from r, Store accumulator into r, Load Accumulator with immediate constant }
        - All models: at least one *conditional "jump"* (branch, goto) following test of a register e.g. { Jump-if-zero, Jump-if-not-zero (i.e. Jump-if-positive), Jump-if-equal, Jump-if-not equal }
        - All models optional: { unconditional program jump (goto) }
    3. **Register-addressing method**:
        - Counter machine: no indirect addressing, immediate operands possible in highly atomized models
        - RAM and RASP: indirect addressing available, immediate operands typical
    4. **Input-output**: optional in all models
4. **State register**: A special Instruction Register "IR", finite and separate from the registers above, stores the current instruction to be executed and its address in the TABLE of instructions; this register and its TABLE is located in the finite state machine.
    - The IR is off-limits to all models. In the case of the RAM and RASP, for purposes of determining the "address" of a register, the model can select either (i) in the case of direct addressing—the address specified by the TABLE and temporarily located in the IR or (ii) in the case of indirect addressing—the contents of the register specified by the IR's instruction.
    - The IR is *not* the "program counter" (PC) of the RASP (or conventional computer). The PC is just another register similar to an accumulator, but dedicated to holding the number of the RASP's current register-based instruction. Thus a RASP has *two* "instruction/program" registers -- (i) the IR (finite state machine's Instruction Register), and (ii) a PC (Program Counter) for the program located in the registers. (As well as a register dedicated to "the PC", a RASP may dedicate another register to "the Program-Instruction Register" (going by any number of names such as "PIR, "IR", "PR", etc.)
5. **List of labeled instructions, usually in sequential order**: A finite list of instructions $I_1 \ldots I_m$. In the case of the counter machine, random access machine (RAM) and pointer machine the instruction store is in the "TABLE" of the finite state machine; thus these models are example of the Harvard architecture. In the case of the RASP the program store is in the registers; thus this is an example of the von Neumann architecture.
   Usually, like computer programs, the instructions are listed in sequential order; unless a jump is successful the default sequence continues in numerical order. An exception to this is the abacus (Lambek (1961), Minsky (1961)) counter machine

models—every instruction has at least one "next" instruction identifier "z", and the conditional branch has two.

- Observe also that the abacus model combines two instructions, JZ then DEC: e.g. { INC (r, z), JZDEC (r, z_{true}, z_{false}) }.

Historical development of the register machine model

Two trends appeared in the early 1950s—the first to characterize the computer as a Turing machine, the second to define computer-like models—models with sequential instruction sequences and conditional jumps—with the power of a Turing machine, i.e. a so-called Turing equivalence. Need for this work was carried out in context of two "hard" problems: the unsolvable word problem posed by Emil Post -- his problem of "tag" -- and the very "hard" problem of Hilbert's problems -- the 10th question around Diophantine equations. Researchers were questing for Turing-equivalent models that were less "logical" in nature and more "arithmetic" (cf Melzak (1961) p. 281, Shepherdson-Sturgis (1963) p. 218).

The first trend—toward characterizing computers—seems to have originated with Hans Hermes (1954) and Heinz Kaphengst (1959), the second trend with Hao Wang (1954, 1957) and, as noted above, furthered along by Z. A.Melzak (1961), Joachim Lambek (1961), Marvin Minsky (1961, 1967), and John Shepherdson and H. E. Sturgis (1963).

The last five names are listed explicitly in that order by Yuri Matiyasevich. He follows up with:

"Register machines [some authors use "register machine" synonymous with "counter-machine"] are particularly suitable for constructing Diophantine equations. Like Turing machines, they have very primitive instructions and, in addition, they deal with numbers"**.**

It appears that Lambek, Melzak, Minsky and Shepherdson and Sturgis independently anticipated the same idea at the same time.

The history begins with Wang's model.

(1954, 1957) Wang's model: Post-Turing machine

Wang's work followed from Emil Post's (1936) paper and led Wang to his definition of his Wang B-machine—a two-symbol Post-Turing machine computation model with only four atomic instructions:

{ LEFT, RIGHT, PRINT, JUMP_if_marked_to_instruction_z }

To these four both Wang (1954, 1957) and then C.Y. Lee (1961) added another other instruction from the Post set { ERASE }, and then a Post's unconditional jump {

JUMP_to_ instruction_z } (or to make things easier, the conditional jump JUMP_IF_blank_to_instruction_z, or both. Lee named this a "W-machine" model:

> { LEFT, RIGHT, PRINT, ERASE, JUMP_if_marked, [maybe JUMP or JUMP_IF_blank] }

Wang expressed hope that his model would be "a rapprochement" (p. 63) between the theory of Turing machines and the practical world of the computer.

Wang's work was highly influential. We find him referenced by Minsky (1961) and (1967), Melzak (1961), Shepherdson and Sturgis (1963). Indeed, Shepherdson and Sturgis (1963) remark that:

"...we have tried to carry a step further the 'rapprochement' between the practical and theoretical aspects of computation suggested by Wang" (p. 218)

Martin Davis eventually evolved this model into the (2-symbol) Post-Turing machine.

Difficulties with the Wang/Post-Turing model:

Except there was a problem: the Wang model (the six instructions of the 7-instruction Post-Turing machine) was still a single-tape Turing-like device, however nice its *sequential program instruction-flow* might be. Both Melzak (1961) and Shepherdson and Sturgis (1963) observed this (in the context of certain proofs and investigations):

"...a Turing machine has a certain opacity... a Turing machine is slow in (hypothetical) operation and, usually, complicated. This makes it rather hard to design it, and even harder to investigate such matters as time or storage optimization or a comparison between efficiency of two algorithms. (Melzak (1961) p. 281)
"...although not difficult ... proofs are complicated and tedious to follow for two reasons: (1) A Turing machine has only head so that one is obliged to break down the computation into very small steps of operations on a single digit. (2) It has only one tape so that one has to go to some trouble to find the under one wishing to work on and keep it separate from other numbers" (Shepherdson and Sturgis (1963) p. 218).

Indeed as examples at Turing machine examples, Post-Turing machine and partial function show, the work can be "complicated".

Minsky, Melzak-Lambek and Shepherdson-Sturgis models "cut the tape" into many

So why not 'cut the tape' so each is infinitely long (to accommodate any size integer) but left-ended, and call these three tapes "Post-Turing (ie. Wang-like) tapes"? The individual heads will move left (for decrement) and right (for increment). In one sense the heads indicate "the tops of the stack" of concatenated marks. Or in Minsky (1961) and Hopcroft

and Ullman (1979, p. 171ff) the tape is always blank except for a mark at the left end—at no time does a head ever print or erase.

We just have to be careful to write our instructions so that a test-for-zero and jump occurs *before* we decrement otherwise our machine will "fall off the end" or "bump against the end" -- we will have an instance of a partial function. Before a decrement our machine must always ask the question: "Is the tape/counter empty? If so then I can't decrement, otherwise I can."

For example of the addition algorithm written for a counter machine see Algorithm examples, and for an example of (im-) proper subtraction see Partial function.

Minsky (1961) and Shepherdson-Sturgis (1963) prove that only a few tapes—as few as one—still allow the machine to be Turing equivalent *IF* the data on the tape is represented as a Gödel number (or some other uniquely encodable-decodable number); this number will evolve as the computation proceeds. In the one tape version with Gödel number encoding the counter machine must be able to (i) multiply the Gödel number by a constant (numbers "2" or "3"), and (ii) divide by a constant (numbers "2" or "3") and jump if the remainder is zero. Minsky (1967) shows that the need for this bizarre instruction set can be relaxed to { INC (r), JZDEC (r, z) } and the convenience instructions { CLR (r), J (r) } if two tapes are available. A simple Gödelization is still required, however. A similar result appears in Elgot-Robinson (1964) with respect to their RASP model.

(1961) Melzak's model is different: clumps of pebbles go into and out of holes

Melzak's (1961) model is significantly different. He took his own model, flipped the tapes vertically, called them "holes in the ground" to be filled with "pebble counters". Unlike Minsky's "increment" and "decrement", Melzak allowed for proper subtraction of any count of pebbles and "adds" of any count of pebbles.

He defines indirect addressing for his model (p. 288) and provides two examples of its use (p. 89); his "proof" (p. 290-292) that his model is Turing equivalent is so sketchy that the reader cannot tell whether or not he intended the indirect addressing to be a requirement for the proof.

Legacy of Melzak's model is Lambek's simplification and the reappearance of his mnemonic conventions in Cook and Reckhow 1973.

Lambek (1961) atomizes Melzak's model into the Minsky (1961) model: INC and DEC-with-test

Lambek (1961) took Melzak's ternary model and atomized it down to the two unary instructions—X+, X- if possible else jump—exactly the same two that Minsky (1961) had come up with.

However, like the Minsky (1961) model, the Lambek model does execute its instructions in a default-sequential manner—both X+ and X- carry the identifier of the next instruction, and X- also carries the jump-to instruction of the zero-test is successful.

Elgot-Robinson (1964) and the problem of the RASP without indirect addressing

A RASP or Random access stored program machine begins as a counter machine with its "program of instruction" placed in its "registers". Analogous to, but independent of, the finite state machine's "Instruction Register", at least one of the registers (nicknamed the "program counter" (PC)) and one or more "temporary" registers maintain a record of, and operate on, the current instruction's number. The finite state machine's TABLE of instructions is responsible for (i) fetching the current *program* instruction from the proper register, (ii) parsing the *program* instruction, (ii) fetching operands specified by the *program* instruction, and (iv) executing the *program* instruction.

Except there is a problem: If based on the *counter machine* chassis this computer-like, von Neumann machine will not be Turing equivalent. It cannot compute everything that is computable. Intrinsically the model is bounded by the size of its (very-) *finite* state machine's instructions. The counter machine based RASP can compute any primitive recursive function (e.g. multiplication) but not all mu recursive functions (e.g. the Ackermann function).

Elgot-Robinson investigate the possibility of allowing their RASP model to "self modify" its program instructions. The idea was an old one, proposed by Burks-Goldstine-von Neumann (1946-7), and sometimes called "the computed goto." Melzak (1961) specifically mentions the "computed goto" by name but instead provides his model with indirect addressing.

Computed goto: A RASP *program* of instructions that modifies the "goto address" in a conditional- or unconditional-jump *program* instruction.

But this does not solve the problem (unless one resorts to Gödel numbers). What is necessary is a method to fetch the address of a program instruction that lies (far) "beyond/above" the upper bound of the *finite* state machine's instruction register and TABLE.

Example: A counter machine equipped with only four unbounded registers can e.g. multiply any two numbers (m, n) together to yield p -- and thus be a primitive recursive function -- no matter how large the numbers m and n; moreover, less than 20 instructions are required to do this! e.g. { 1: CLR (p), 2: JZ (m, done), 3 outer_loop: JZ (n, done), 4: CPY (m, temp), 5: inner_loop: JZ (m, outer_loop), 6: DEC (m), 7: INC (p), 8: J (inner_loop), 9: outer_loop: DEC (n), 10 J (outer_loop), HALT }
However, with only 4 registers, this machine has not nearly big enough to build a RASP that can execute the multiply algorithm as a *program*. No matter how big we build our finite state machine there will always be a *program* (including its parameters) which is

larger. So by definition the bounded program machine that does not use unbounded encoding tricks such as Gödel numbers cannot not *universal*.

Minsky (1967) hints at the issue in his investigation of a counter machine (he calls them "program computer models") equipped with the instructions { CLR (r), INC (r), and RPT ("a" times the instructions m to n) }. He doesn't tell us how to fix the problem, but he does observe that:

"... the program computer has to have some way to keep track of how many RPT's remain to be done, and this might exhaust any particular amount of storage allowed in the finite part of the computer. RPT operations require infinite registers of their own, in general, and they must be treated differently from the other kinds of operations we have considered." (p. 214)

But Elgot and Robinson solve the problem: They augment their P_0 RASP with an indexed set of instructions—a somewhat more complicated (but more flexible) form of indirect addressing. Their P'_0 model addresses the registers by adding the contents of the "base" register (specified in the instruction) to the "index" specified explicitly in the instruction (or vice versa, swapping "base" and "index"). Thus the indexing P'_0 instructions have one more parameter than the non-indexing P_0 instructions:

Example: INC (r_{base}, index) ; effective address will be $[r_{base}]$ + index, where the natural number "index" is derived from the finite-state machine instruction itself.

Hartmanis (1971)

By 1971 Hartmanis has simplified the indexing to indirection for use in his RASP model.

Indirect addressing: A pointer-register supplies the finite state machine with the address of the target register required for the instruction. Said another way: The *contents* of the pointer-register is the *address* of the "target" register to be used by the instruction. If the pointer-register is unbounded, the RAM, and a suitable RASP built on its chassis, will be Turing equivalent. The target register can serve either as a source or destination register, as specified by the instruction.

Note that the finite state machine does not have to explicitly specify this target register's address. It just says to the rest of the machine: Get me the contents of the register pointed to by my pointer-register and then do xyz with it. It must specify explicitly by name, via its instruction, this pointer-register (e.g. "N", or "72" or "PC", etc.) but it doesn't have to know what number the pointer-register actually contains (perhaps 279,431).

Cook and Reckhow (1973) describe the RAM

Cook and Reckhow (1973) cite Hartmanis (1971) and simplify his model to what they call a Random access machine (RAM—i.e. a machine with indirection and the Harvard

architecture). In a sense we are back to Melzak (1961) but with a much simpler model than Melzak's.

Precedence

Minsky was working at the M.I.T. Lincoln Labs and published his work there; his paper was received for publishing in the *Annals of Mathematics* on August 15, 1960 but not published until November 1961. While receipt occurred a full year before the work of Melzak and Lambek was received and published (received, respectively, May and June 15, 1961 and published side-by-side September 1961). That (i) both were Canadians and published in the Canadian Mathematical Bulletin, (ii) neither would have had reference to Minsky's work because it was not yet published in a peer-reviewed journal, but (iii) Melzak references Wang, and Lambek references Melzak, leads one to hypothesize that their work occurred simultaneously and independently.

Almost exactly the same thing happened to Shepherdson and Sturgis. Their paper was received in December 1961—just a few months after Melzak and Lambek's work was received. Again, they had little (at most 1 month) or no benefit of reviewing the work of Minsky. They were careful to observe in footnotes that papers by Ershov, Kaphengst and Peter had "recently appeared" (p. 219). These were published much earlier but appeared in the German language in German journals so issues of accessibility present themselves.

The final paper of Shepherdson and Sturgis did not appear in a peer-reviewed journal until 1963. And as they fairly and honestly note in their Appendix A, the 'systems' of Kaphengst (1959), Ershov (1958), Peter (1958) are all so similar to what results were obtained later as to be indistinguishable to a set of the following:

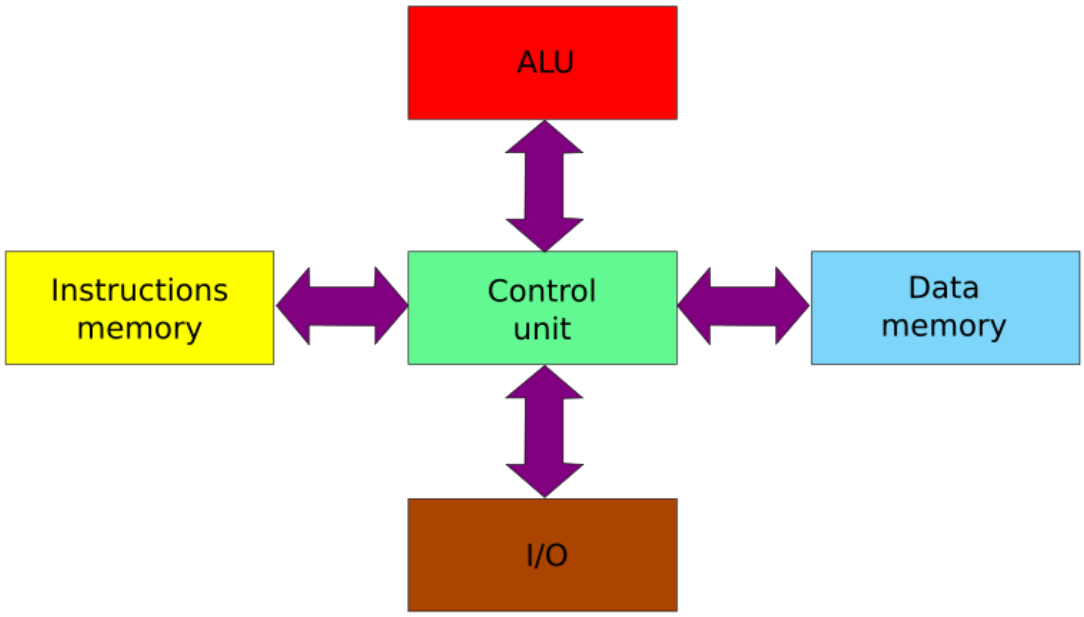
produce 0 i.e. 0 --> n
increment a number i.e. n+1 --> n
"i.e. of performing the operations which generate the natural numbers" (p. 246)
copy a number i.e. n --> m
to "change the course of a computation", either comparing two numbers or decrementing until 0

Indeed, Shepherson and Sturgis conclude

"The various minimal systems are very similar"(p. 246)

By order of *publishing* date the work of Kaphengst (1959), Ershov (1958), Peter (1958) were first. Does context matter? An answer would require close examination of the papers. Conclusions and opinions about this will be left to the reader.

Harvard architecture



Harvard architecture

The **Harvard architecture** is a computer architecture with physically separate storage and signal pathways for instructions and data. The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters. These early machines had limited data storage, entirely contained within the central processing unit, and provided no access to the instruction storage as data. Programs needed to be loaded by an operator, the processor could not boot itself.

Today, most processors implement such separate signal pathways for performance reasons but actually implement a Modified Harvard architecture, so they can support tasks like loading a program from disk storage as data and then executing it.

Memory details

In a Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions can be stored in read-only memory while data memory generally requires read-write memory. In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.

Contrast with von Neumann architectures

Under pure von Neumann architecture the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same bus system. In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache. A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway .

Also, a Harvard architecture machine has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four bit value, while data address zero might indicate an eight bit byte that isn't part of that twenty-four bit value.

Contrast with Modified Harvard architecture

A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The most common modification includes separate instruction and data caches backed by a common address space. While the CPU executes from cache, it acts as a pure Harvard machine. When accessing backing memory, it acts like a von Neumann machine (where code can be moved around like data, a powerful technique). This modification is widespread in modern processors such as the ARM architecture and X86 processors. It is sometimes loosely called a Harvard architecture, overlooking the fact that it is actually "modified".

Another modification provides a pathway between the instruction memory (such as ROM or flash) and the CPU to allow words from the instruction memory to be treated as read-only data. This technique is used in some microcontrollers, including the Atmel AVR. This allows constant data, such as text strings or function tables, to be accessed without first having to be copied into data memory, preserving scarce (and power-hungry) data memory for read/write variables. Special machine language instructions are provided to read data from the instruction memory. (This is distinct from instructions which themselves embed constant data, although for individual constants the two mechanisms can substitute for each other.)

Speed

In recent years, the speed of the CPU has grown many times in comparison to the access speed of the main memory. Care needs to be taken to reduce the number of times main memory is accessed in order to maintain performance. If, for instance, every instruction run in the CPU requires an access to memory, the computer gains nothing for increased CPU speed—a problem referred to as being "memory bound".

It is possible to make extremely fast memory but this is only practical for small amounts of memory for cost, power and signal routing reasons. The solution is to provide a small amount of very fast memory known as a CPU cache which holds recently accessed data. As long as the memory that the CPU needs is in the cache, the performance hit is much smaller than it is when the cache has to turn around and get the data from the main memory.

Internal vs. external design

Modern high performance CPU chip designs incorporate aspects of both Harvard and von Neumann architecture. In particular, the Modified Harvard architecture is very common. CPU cache memory is divided into an instruction cache and a data cache. Harvard architecture is used as the CPU accesses the cache. In the case of a cache miss, however, the data is retrieved from the main memory, which is not formally divided into separate instruction and data sections, although it may well have separate memory controllers used for concurrent access to RAM, ROM and (NOR) flash memory.

Thus, while a von Neumann architecture is visible in some contexts, such as when data and code come through the same memory controller, the hardware implementation gains the efficiencies of the Harvard architecture for cache accesses and at least some main memory accesses.

In addition, CPUs often have write buffers which let CPUs proceed after writes to non-cached regions. The von Neumann nature of memory is then visible when instructions are written as data by the CPU and software must ensure that the caches (data and instruction) and write buffer are synchronized before trying to execute those just-written instructions.

Modern uses of the Harvard architecture

The principal advantage of the pure Harvard architecture—simultaneous access to more than one memory system—has been reduced by modified Harvard processors using modern CPU cache systems. Relatively pure Harvard architecture machines are used mostly in applications where tradeoffs, such as the cost and power savings from omitting caches, outweigh the programming penalties from having distinct code and data address spaces.

- Digital signal processors (DSPs) generally execute small, highly-optimized audio or video processing algorithms. They avoid caches because their behavior must be extremely reproducible. The difficulties of coping with multiple address spaces are of secondary concern to speed of execution. As a result, some DSPs have multiple data memories in distinct address spaces to facilitate SIMD and VLIW processing. Texas Instruments TMS320 C55x processors, as one example, have multiple parallel data buses (two write, three read) and one instruction bus.

- Microcontrollers are characterized by having small amounts of program (flash memory) and data (SRAM) memory, with no cache, and take advantage of the Harvard architecture to speed processing by concurrent instruction and data access. The separate storage means the program and data memories can have different bit depths, for example using 16-bit wide instructions and 8-bit wide data. They also mean that instruction prefetch can be performed in parallel with other activities. Examples include, the AVR by Atmel Corp, the PIC by Microchip Technology, Inc. and the ARM Cortex-M3 processor (not all ARM chips have Harvard architecture).

Even in these cases, it is common to have special instructions to access program memory as data for read-only tables, or for reprogramming.

Chapter 7

CPU Design and Out-of-Order Execution

CPU design

CPU design is the design engineering task of creating a central processing unit (CPU), a component of computer hardware. It is a subfield of electronics engineering and computer engineering.

Overview

CPU design focuses on these areas:

1. datapaths (such as ALUs and pipelines)
2. control unit: logic which controls the datapaths
3. Memory components such as register files, caches
4. Clock circuitry such as clock drivers, PLLs, clock distribution networks
5. Pad transceiver circuitry
6. Logic gate cell library which is used to implement the logic

CPUs designed for high-performance markets might require custom designs for each of these items to achieve frequency, power-dissipation, and chip-area goals.

CPUs designed for lower performance markets might lessen the implementation burden by:

- Acquiring some of these items by purchasing them as intellectual property
- Use control logic implementation techniques (logic synthesis using CAD tools) to implement the other components - datapaths, register files, clocks

Common logic styles used in CPU design include:

- Unstructured random logic

- Finite-state machines
- Microprogramming (common from 1965 to 1985)
- Programmable logic array (common in the 1980s, no longer common)

Device types used to implement the logic include:

- Transistor-transistor logic Small Scale Integration logic chips - no longer used for CPUs
- Programmable Array Logic and Programmable logic devices - no longer used for CPUs
- Emitter-coupled logic (ECL) gate arrays - no longer common
- CMOS gate arrays - no longer used for CPUs
- CMOS ASICs - what's commonly used today, they're so common that the term ASIC is not used for CPUs
- Field-programmable gate arrays (FPGA) - common for soft microprocessors, and more or less required for reconfigurable computing

A CPU design project generally has these major tasks:

- Programmer-visible instruction set architecture, which can be implemented by a variety of microarchitectures
- Architectural study and performance modeling in ANSI C/C++ or SystemC
- High-level synthesis (HLS) or RTL (eg. logic) implementation
- RTL Verification
- Circuit design of speed critical components (caches, registers, ALUs)
- Logic synthesis or logic-gate-level design
- Timing analysis to confirm that all logic and circuits will run at the specified operating frequency
- Physical design including floorplanning, place and route of logic gates
- Checking that RTL, gate-level, transistor-level and physical-level representations are equivalent
- Checks for signal integrity, chip manufacturability

As with most complex electronic designs, the logic verification effort (proving that the design does not have bugs) now dominates the project schedule of a CPU.

Key CPU architectural innovations include index register, cache, virtual memory, instruction pipelining, superscalar, CISC, RISC, virtual machine, emulators, microprogram, and stack.

Goals

The first CPUs were designed to do mathematical calculations faster and more reliably than human computers.

Each successive generation of CPU might be designed to achieve some of these goals:

- higher performance levels of a single program or thread
- higher throughput levels of multiple programs/threads
- less power consumption for the same performance level
- lower cost for the same performance level
- greater connectivity to build larger, more parallel systems
- more specialization to aid in specific targeted markets

Re-designing a CPU core to a smaller die-area helps achieve several of these goals.

- Shrinking everything (a "photomask shrink"), resulting in the same number of transistors on a smaller die, improves performance (smaller transistors switch faster), reduces power (smaller wires have less parasitic capacitance) and reduces cost (more CPUs fit on the same wafer of silicon).
- Releasing a CPU on the same size die, but with a smaller CPU core, keeps the cost about the same but allows higher levels of integration within one VLSI chip (additional cache, multiple CPUs, or other components), improving performance and reducing overall system cost.

Performance analysis and benchmarking

Because there are too many programs to test a CPU's speed on all of them, benchmarks were developed. The most famous benchmarks are the SPECint and SPECfp benchmarks developed by Standard Performance Evaluation Corporation and the ConsumerMark benchmark developed by the Embedded Microprocessor Benchmark Consortium EEMBC.

Some important measurements include:

- Instructions per second - Most consumers pick a computer architecture (normally Intel IA32 architecture) to be able to run a large base of pre-existing pre-compiled software. Being relatively uninformed on computer benchmarks, some of them pick a particular CPU based on operating frequency.
- FLOPS - The number of floating point operations per second is often important in selecting computers for scientific computations.
- Performance per watt - System designers building parallel computers, such as Google, pick CPUs based on their speed per watt of power, because the cost of powering the CPU outweighs the cost of the CPU itself.
- Some system designers building parallel computers pick CPUs based on the speed per dollar.
- System designers building real-time computing systems want to guarantee worst-case response. That is easier to do when the CPU has low interrupt latency and when it has deterministic response. (DSP)
- Computer programmers who program directly in assembly language want a CPU to support a full featured instruction set.
- Low power - For systems with limited power sources (e.g. solar, batteries, human power).

- Small size or low weight - for portable embedded systems, systems for spacecraft.
- Environmental impact - Minimizing environmental impact of computers during manufacturing and recycling as well during use. Reducing waste, reducing hazardous materials.

Some of these measures conflict. In particular, many design techniques that make a CPU run faster make the "performance per watt", "performance per dollar", and "deterministic response" much worse, and vice versa.

Markets

There are several different markets in which CPUs are used. Since each of these markets differ in their requirements for CPUs, the devices designed for one market are in most cases inappropriate for the other markets.

General purpose computing

The vast majority of revenues generated from CPU sales is for general purpose computing. That is, desktop, laptop and server computers commonly used in businesses and homes. In this market, the Intel IA-32 architecture dominates, with its rivals PowerPC and SPARC maintaining much smaller customer bases. Yearly, hundreds of millions of IA-32 architecture CPUs are used by this market.

Since these devices are used to run countless different types of programs, these CPU designs are not specifically targeted at one type of application or one function. The demands of being able to run a wide range of programs efficiently has made these CPU designs among the more advanced technically, along with some disadvantages of being relatively costly, and having high power consumption.

High-end processor economics

In 1984, most high-performance CPUs required four to five years to develop.

Developing new, high-end CPUs is a very costly proposition. Both the logical complexity (needing very large logic design and logic verification teams and simulation farms with perhaps thousands of computers) and the high operating frequencies (needing large circuit design teams and access to the state-of-the-art fabrication process) account for the high cost of design for this type of chip. The design cost of a high-end CPU will be on the order of US $100 million. Since the design of such high-end chips nominally takes about five years to complete, to stay competitive a company has to fund at least two of these large design teams to release products at the rate of 2.5 years per product generation.

As an example, the typical loaded cost for one computer engineer is often quoted to be $250,000 US dollars/year. This includes salary, benefits, CAD tools, computers, office space rent, etc. Assuming that 100 engineers are needed to design a CPU and the project takes 4 years.

Total cost = $250,000 / Engineer-Man/Year x 100 engineers x 4 years = $100,000,000 USD.

The above amount is just an example. The design teams for modern day general purpose CPUs have several hundred team members.

Scientific computing

A much smaller niche market (in revenue and units shipped) is scientific computing, used in government research labs and universities. Previously much CPU design was done for this market, but the cost-effectiveness of using mass markets CPUs has curtailed almost all specialized designs for this market. The main remaining area of active hardware design and research for scientific computing is for high-speed system interconnects.

Embedded design

As measured by units shipped, most CPUs are embedded in other machinery, such as telephones, clocks, appliances, vehicles, and infrastructure. Embedded processors sell in the volume of many billions of units per year, however, mostly at much lower price points than that of the general purpose processors.

These single-function devices differ from the more familiar general-purpose CPUs in several ways:

- Low cost is of utmost importance.
- It is important to maintain a low power dissipation as embedded devices often have a limited battery life and it is often impractical to include cooling fans.
- To give lower system cost, peripherals are integrated with the processor on the same silicon chip.
- Keeping peripherals on-chip also reduces power consumption as external GPIO ports typically require buffering so that they can source or sink the relatively high current loads that are required to maintain a strong signal outside of the chip.
    - Many embedded applications have a limited amount of physical space for circuitry; keeping peripherals on-chip will reduce the space required for the circuit board.
    - The program and data memories are often integrated on the same chip. When the only allowed program memory is ROM, the device is known as a microcontroller.
- For many embedded applications, interrupt latency will be more critical than in some general-purpose processors.

Embedded processor economics

As of 2009, more CPUs are produced using the ARM architecture instruction set than any other 32-bit instruction set. The ARM architecture and the first ARM chip were designed in about one and a half years and 5 man years of work time.

The 32-bit Parallax Propeller microcontroller architecture and the first chip were designed by two people in about 10 man years of work time.

It is believed that the 8-bit AVR architecture and first AVR microcontroller was conceived and designed by two students at the Norwegian Institute of Technology.

The 8-bit 6502 architecture and the first MOS Technology 6502 chip were designed in 13 months by a group of about 9 people.

Research and educational CPU design

The 32 bit Berkeley RISC I and RISC II architecture and the first chips were mostly designed by a series of students as part of a four quarter sequence of graduate courses. This design became the basis of the commercial SPARC processor design.

For about a decade, every student taking the 6.004 class at MIT was part of a team—each team had one semester to design and build a simple 8 bit CPU out of 7400 series integrated circuits. One team of 4 students designed and built a simple 32 bit CPU during that semester.

Some undergraduate courses require a team of 2 to 5 students to design, implement, and test a simple CPU in a FPGA in a single 15 week semester.

Soft microprocessor cores

For embedded systems, the highest performance levels are often not needed or desired due to the power consumption requirements. This allows for the use of processors which can be totally implemented by logic synthesis techniques. These synthesized processors can be implemented in a much shorter amount of time, giving quicker time-to-market.

Out-of-order execution

In computer engineering, **out-of-order execution** (**OoOE** or **OOE**) is a paradigm used in most high-performance microprocessors to make use of instruction cycles that would otherwise be wasted by a certain type of costly delay.

History

Out-of-order execution is a restricted form of data flow computation, which was a major research area in computer architecture in the 1970s and early 1980s. Important academic research in this subject was led by Yale Patt and his HPSm simulator. A paper by James

E. Smith and A.R. Pleszkun, published in 1985 completed the scheme by describing how the precise behavior of exceptions could be maintained in out-of-order machines.

Arguably the first machine to use out-of-order execution was probably the CDC 6600 (1964), which used a scoreboard to resolve conflicts. In modern usage, such scoreboarding is considered to be in-order execution, not out-of-order execution, since such machines stall on the first RAW (Read After Write) conflict. Strictly speaking, such machines initiate execution in-order, although they may complete execution out-of-order.

About three years later, the IBM 360/91 (1966) introduced Tomasulo's algorithm, supporting full out-of-order execution.

In 1990, IBM introduced the first out-of-order microprocessor, the POWER1, although out-of-order execution was limited to floating point instructions only.

Throughout the 1990s out-of-order execution became more common, and was featured in the IBM/Motorola PowerPC 601 (1993), Fujitsu/HAL SPARC64 (1995), Intel Pentium Pro (1995), MIPS R10000 (1996), HP PA-8000 (1996), AMD K5 (1996) and DEC Alpha 21264 (1998). Notable exceptions to this trend include the Sun UltraSPARC, HP/Intel Itanium, Transmeta Crusoe, Intel Atom, and the IBM POWER6.

The logical complexity of the out-of-order schemes was the reason that this technique did not reach mainstream machines until the mid-1990s. Many low-end processors meant for cost-sensitive markets still do not use this paradigm due to large silicon area that is required to build this class of machine. Low power usage is another design goal that's harder to achieve with an OoOE design.

Basic concept

In-order processors

In earlier processors, the processing of instructions is normally done in these steps:

1. Instruction fetch.
2. If input operands are available (in registers for instance), the instruction is dispatched to the appropriate functional unit. If one or more operand is unavailable during the current clock cycle (generally because they are being fetched from memory), the processor stalls until they are available.
3. The instruction is executed by the appropriate functional unit.
4. The functional unit writes the results back to the register file.

Out-of-order processors

This new paradigm breaks up the processing of instructions into these steps:

1. Instruction fetch.
2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
4. The instruction is issued to the appropriate functional unit and executed by that unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage.

The key concept of OoO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable. In the outline above, the OoO processor avoids the stall that occurs in step (2) of the in-order processor when the instruction is not completely ready to be processed due to missing data.

OoO processors fill these "slots" in time with other instructions that *are* ready, then re-order the results at the end to make it appear that the instructions were processed as normal. The way the instructions are ordered in the original computer code is known as *program order*, in the processor they are handled in *data order*, the order in which the data, operands, become available in the processor's registers. Fairly complex circuitry is needed to convert from one ordering to the other and maintain a logical ordering of the output; the processor itself runs the instructions in seemingly random order.

The benefit of OoO processing grows as the instruction pipeline deepens and the speed difference between main memory (or cache memory) and the processor widens. On modern machines, the processor runs many times faster than the memory, so during the time an in-order processor spends waiting for data to arrive, it could have processed a large number of instructions.

Dispatch and issue decoupling allows out-of-order issue

One of the differences created by the new paradigm is the creation of queues which allows the dispatch step to be decoupled from the issue step and the graduation stage to be decoupled from the execute stage. An early name for the paradigm was *decoupled architecture*. In the earlier *in-order* processors, these stages operated in a fairly lock-step, pipelined fashion.

To avoid false operand dependencies, which would decrease the frequency when instructions could be issued out of order, a technique called register renaming is used. In this scheme, there are more physical registers than defined by the architecture. The

physical registers are tagged so that multiple versions of the same architectural register can exist at the same time.

Execute and writeback decoupling allows program restart

The queue for results is necessary to resolve issues such as branch mispredictions and exceptions/traps. The results queue allows programs to be restarted after an exception, which requires the instructions to be completed in program order. The queue allows results to be discarded due to mispredictions on older branch instructions and exceptions taken on older instructions.

The ability to issue instructions past branches which have yet to resolve is known as speculative execution.

Micro-architectural choices

- Are the instructions dispatched to a centralized queue or to multiple distributed queues?

  IBM PowerPC processors use queues which are distributed among the different functional units while other Out-of-Order processors use a centralized queue. IBM uses the term *reservation stations* for their distributed queues.

- Is there an actual results queue or are the results written directly into a register file? For the latter, the queueing function is handled by register maps which hold the register renaming information for each instruction in flight.

  Early Intel out-of-order processors use a results queue called a *re-order buffer*, while most later Out-of-Order processors use register maps.
  More precisely: Intel P6 family microprocessors have both a ROB *re-order buffer* and a RAT register map mechanism. The ROB was motivated mainly by branch misprediction recovery.
  The Intel P6 family was among the earliest OoO processors, was supplanted by the Intel Pentium 4 Williamette microarchitecture, but which returned after the right hand turn and, at the time of writing (2009) is still Intel's flagship microprocessor family.

Chapter 8

Memory Disambiguation

Memory disambiguation is a set of techniques employed by high-performance out-of-order execution microprocessors that execute memory access instructions (loads and stores) out of program order. The mechanisms for performing memory disambiguation, implemented using digital logic inside the microprocessor core, detect true dependencies between memory operations at execution time and allow the processor to recover when a dependence has been violated. They also eliminate spurious memory dependencies and allow for greater instruction-level parallelism by allowing safe out-of-order execution of loads and stores.

Background

Dependencies

When attempting to execute instructions out of order, a microprocessor must respect true dependencies between instructions. For example, consider a simple true dependence:

```
1: add $1, $2, $3      # R1 <= R2 + R3
2: add $5, $1, $4      # R5 <= R1 + R4 (dependent on 1)
```

In this example, the `add` instruction on line 2 is dependent on the `add` instruction on line 1 because the register R1 is a source operand of the addition operation on line 2. The `add` on line 2 cannot execute until the `add` on line 1 completes. In this case, the dependence is **static** and easily determined by a microprocessor, because the sources and destinations are registers. The destination register of the `add` instruction on line 1 (`R1`) is part of the instruction encoding, and so can be determined by the microprocessor early on, during the decode stage of the pipeline. Similarly, the source registers of the `add` instruction on line 2 (`R1` and `R4`) are also encoded into the instruction itself and are determined in decode. To respect this true dependence, the microprocessor's scheduler logic will issue these instructions in the correct order (instruction 1 first, followed by instruction 2) so that the results of 1 are available when instruction 2 needs them.

Complications arise when the dependence is not statically determinable. Such non-static dependencies arise with memory instructions (loads and stores) because the location of

the operand may be indirectly specified as a register operand rather than directly specified in the instruction encoding itself.

```
1: store ($2), $1       # Mem[R2] <= R1
2: load  $3, ($4)       # R3 <= Mem[R4] (possibly dependent on 1)
```

Here, the store instruction writes a value to the memory location specified by the value in R2, and the load instruction reads the value at the memory location specified by the value in R4. The microprocessor cannot statically determine, prior to execution, if the memory locations specified in these two instructions are different, or are the same location, because the locations depend on the values in R2 and R4. If the locations are different, the instructions are independent and can be successfully executed out of order. However, if the locations are the same, then the load instruction is dependent on the store to produce its value. This is known as an **ambiguous dependence**.

Out-of-order execution and memory access operations

Executing loads and stores out of order can produce incorrect results if a dependent load/store pair was executed out of order. Consider the following code snippet, given in MIPS assembly:

```
1: div $27, $20
2: sw  ($30), $27
3: lw  $08, ($31)
4: sw  ($30), $26
5: lw  $09, ($31)
```

Assume that the scheduling logic will issue an instruction to the execution unit when all of its register operands are ready. Further assume that registers `$30` and `$31` are ready: the values in `$30` and `$31` were computed a long time ago and have not changed. However, assume `$27` is not ready: its value is still in the process of being computed by the `div` (integer divide) instruction. Finally, assume that registers `$30` and `$31` hold the same value, and thus all the loads and stores in the snippet access the same memory word.

In this situation, the `sw ($30), $27` instruction on line 2 is not ready to execute, but the `lw $08, ($31)` instruction on line 3 is ready. If the processor allows the `lw` instruction to execute before the `sw`, the load will read an old value from the memory system; however, it should have read the value that was just written there by the `sw`. The load and store executed out of program order, but there was a memory dependence between them that was violated.

Similarly, assume that register `$26` *is* ready. The `sw ($30), $26` instruction on line 4 is also ready to execute, and it may execute before the preceding `lw $08, ($31)` on line 3. If this occurs, the `lw $08, ($31)` instruction will read the *wrong* value from the memory system, since a later store instruction wrote its value there before the load executed.

Characterization of memory dependencies

Memory dependencies come in three flavors:

- **Read-After-Write** (RAW) dependencies: Also known as true dependencies, RAW dependencies arise when a load operation reads a value from memory that was produced by the most recent preceding store operation to that same address.
- **Write-After-Read** (WAR) dependencies: Also known as anti dependencies, WAR dependencies arise when a store operation writes a value to memory that a preceding load reads.
- **Write-After-Write** (WAW) dependencies: Also known as output dependencies, WAW dependencies arise when two store operations write values to the same memory address.

The three dependencies are shown in the preceding code segment (reproduced for clarity):

```
1: div $27, $20
2: sw  ($30), $27
3: lw  $08, ($31)
4: sw  ($30), $26
5: lw  $09, ($31)
```

- The `lw $08, ($31)` instruction on line 3 has a RAW dependence on the `sw ($30), $27` instruction on line 2, and the `lw $09, ($31)` instruction on line 5 has a RAW dependence on the `sw ($30), $26` instruction on line 4. Both load instructions read the memory address that the preceding stores wrote. The stores were the most recent producers to that memory address, and the loads are reading that memory address's value.
- The `sw ($30), $26` instruction on line 4 has a WAR dependence on the `lw $08, ($31)` instruction on line 3 since it writes the memory address that the preceding load reads from.
- The `sw ($30), $26` instruction on line 4 has a WAW dependence on the `sw ($30), $27` instruction on line 2 since both stores write to the same memory address.

Memory disambiguation mechanisms

Modern microprocessors use the following mechanisms, implemented in hardware, to resolve ambiguous dependences and recover when a dependence was violated.

Avoiding WAR and WAW dependencies

Values from store instructions are not committed to the memory system (in modern microprocessors, CPU cache) when they execute. Instead, the store instructions, including the memory address and store data, are buffered in a **store queue** until they

reach the retirement point. When a store retires, it *then* writes its value to the memory system. This avoids the WAR and WAW dependence problems shown in the code snippet above where an earlier load receives an incorrect value from the memory system because a later store was allowed to execute before the earlier load.

Additionally, buffering stores until retirement allows processors to speculatively execute store instructions that follow an instruction that may produce an exception (such as a load of a bad address, divide by zero, etc.) or a conditional branch instruction whose direction (taken or not taken) is not yet known. If the exception-producing instruction has not executed or the branch direction was predicted incorrectly, the processor will have fetched and executed instructions on a "wrong path." These instructions should not have been executed at all; the exception condition should have occurred before any of the speculative instructions executed, or the branch should have gone the other direction and caused different instructions to be fetched and executed. The processor must "throw away" any results from the bad-path, speculatively-executed instructions when it discovers the exception or branch misprediction. The complication for stores is that any stores on the bad or mispredicted path should not have committed their values to the memory system; if the stores had committed their values, it would be impossible to "throw away" the commit, and the memory state of the machine would be corrupted by data from a store instruction that should not have executed.

Thus, without store buffering, stores cannot execute until all previous possibly-exception-causing instructions have executed (and not caused an exception) and all previous branch directions are known. Forcing stores to wait until branch directions and exceptions are known significantly reduces the out-of-order aggressiveness and limits ILP (Instruction level parallelism) and performance. With store buffering, stores can execute ahead of exception-causing or unresolved branch instructions, buffering their data in the store queue but not committing their values until retirement. This prevents stores on mispredicted or bad paths from committing their values to the memory system while still offering the increased ILP and performance from full out-of-order execution of stores.

Store to load forwarding

Buffering stores until retirement avoids WAW and WAR dependencies but introduces a new issue. Consider the following scenario: a store executes and buffers its address and data in the store queue. A few instructions later, a load executes that reads from the same memory address to which the store just wrote. If the load reads its data from the memory system, it will read an old value that would have been overwritten by the preceding store. The data obtained by the load will be incorrect.

To solve this problem, processors employ a technique called **store-to-load forwarding** using the store queue. In addition to buffering stores until retirement, the store queue serves a second purpose: forwarding data from completed but not-yet-retired ("in-flight") stores to later loads. Rather than a simple FIFO queue, the store queue is really a Content-Addressable Memory (CAM) searched using the memory address. When a load executes, it searches the store queue for in-flight stores to the same address that are logically earlier

in program order. If a matching store exists, the load obtains its data value from that store instead of the memory system. If there is no matching store, the load accesses the memory system as usual; any preceding, matching stores must have already retired and committed their values. This technique allows loads to obtain correct data if their producer store has completed but not yet retired.

Multiple stores to the load's memory address may be present in the store queue. To handle this case, the store queue is priority encoded to select the *latest* store that is logically earlier than the load in program order. The determination of which store is "latest" can be achieved by attaching some sort of timestamp to the instructions as they are fetched and decoded, or alternatively by knowing the relative position (slot) of the load with respect to the oldest and newest stores within the store queue.

RAW dependence violations

Detecting RAW dependence violations

Modern out-of-order CPUs can use a number of techniques to detect a RAW dependence violation, but all techniques require tracking in-flight loads from execution until retirement. When a load executes, it accesses the memory system and/or store queue to obtain its data value, and then its address and data are buffered in a **load queue** until retirement. The load queue is similar in structure and function to the store queue, and in fact in some processors may be combined with the store queue in a single structure called a **load-store queue**, or **LSQ**. The following techniques are used or have been proposed to detect RAW dependence violations:

Load queue CAM search

With this technique, the load queue, like the store queue, is a CAM searched using the memory access address, and keeps track of all in-flight loads. When a store executes, it searches the load queue for completed loads from the same address that are logically later in program order. If such a matching load exists, it must have executed before the store and thus read an incorrect, old value from the memory system/store queue. Any instructions that used the load's value have also used bad data. To recover if such a violation is detected, the load is marked as "violated" in the retirement buffer. The store remains in the store queue and retirement buffer and retires normally, committing its value to the memory system when it retires. However, when the violated load reaches the retirement point, the processor flushes the pipeline and restarts execution from the load instruction. At this point, all previous stores have committed their values to the memory system. The load instruction will now read the correct value from the memory system, and any dependent instructions will re-execute using the correct value.

This technique requires an associative search of the load queue on every store execution, which consumes circuit power and can prove to be a difficult timing path for large load queues. However, it does not require any additional memory (cache) ports or create resource conflicts with other loads or stores that are executing.

Disambiguation At Retirement

With this technique, load instructions that have executed out-of-order are re-executed (they access the memory system and read the value from their address a second time) when they reach the retirement point. Since the load is now the retiring instruction, it has no dependencies on any instruction still in-flight; all stores ahead of it have committed their values to the memory system, and so any value read from the memory system is guaranteed to be correct. The value read from memory at re-execution time is compared to the value obtained when the load first executed. If the values are the same, the original value was correct and no violation has occurred. If the re-execution value differs from the original value, a RAW violation has occurred and the pipeline must be flushed because instructions dependent on the load have used an incorrect value.

This technique is conceptually simpler than the load queue search, and it eliminates a second CAM and its power-hungry search (the load queue can now be a simple FIFO queue). Since the load must re-access the memory system just before retirement, the access must be very fast, so this scheme relies on a fast cache. No matter how fast the cache is, however, the second memory system access for every out-of-order load instruction does increase instruction retirement latency and increases the total number of cache accesses that must be performed by the processor. The additional retire-time cache access can be satisfied by re-using an existing cache port; however, this creates port resource contention with other loads and stores in the processor trying to execute, and thus may cause a decrease in performance. Alternatively, an additional cache port can be added just for load disambiguation, but this increases the complexity, power, and area of the cache. Some recent work (Roth 2005) has shown ways to filter many loads from re-executing if it is known that no RAW dependence violation could have occurred; such a technique would help or eliminate such latency and resource contention.

A minor benefit of this scheme (compared to a load-queue search) is that it will not flag a RAW dependence violation and trigger a pipeline flush if a store that would have caused a RAW dependence violation (the store's address matches an in-flight load's address) has a data value that matches the data value already in the cache. In the load-queue search scheme, an additional data comparison would need to be added to the load-queue search hardware to prevent such a pipeline flush.

Avoiding RAW dependence violations

CPUs that fully support out-of-order execution of loads and stores must be able to detect RAW dependence violations when they occur. However, many CPUs avoid this problem by forcing all loads and stores to execute in-order, or by supporting only a limited form of out-of-order load/store execution. This approach offers lower performance compared to supporting full out-of-order load/store execution, but it can significantly reduce the complexity of the execution core and caches.

The first option, making loads and stores go in-order, avoids RAW dependences because there is no possibility of a load executing before its producer store and obtaining incorrect

data. Another possibility is to effectively break loads and stores into two operations: address generation and cache access. With these two separate but linked operations, the CPU can allow loads and stores to access the memory system only once all previous loads and stores have had their address generated and buffered in the LSQ. After address generation, there are no longer any ambiguous dependencies since all addresses are known, and so dependent loads will not be executed until their corresponding stores complete. This scheme still allows for some "out-of-orderness"—the address generation operations for any in-flight loads and stores can execute out-of-order, and once addresses have been generated, the cache accesses for each load or store can happen in any order that respects the (now known) true dependences.

Additional Issues

Memory dependence prediction

Processors that fully support out-of-order load/store execution can use an additional, related technique, called memory dependence prediction, to attempt to predict true dependences between loads and stores *before* their addresses are known. Using this technique, the processor can prevent loads that are predicted to be dependent on an in-flight store from executing before that store completes, avoiding a RAW dependence violation and thus avoiding the pipeline flush and the performance penalty that is incurred.

Chapter 9

Orthogonal Instruction Set and Microarchitecture

Orthogonal instruction set

Orthogonal instruction set is a term used in computer engineering. A computer's instruction set is said to be orthogonal if any instruction can use data of any type via any addressing mode. In this context, the word orthogonal, which means *right angle*, implies that it is possible to move along one axis (the operations) independently of the other axis (the addressing modes) and vice versa; this meaning is similar – but not identical – to the meaning of the word in pure mathematics.

Orthogonality in practice

In many CISC computers, an instruction could access either registers or memory, usually in several different ways. This made the CISC machines easier to program, because rather than being required to remember thousands of individual instruction opcodes, an orthogonal instruction set allowed a programmer to instead remember just thirty to a hundred operation codes ("ADD", "SUBTRACT", "MULTIPLY", "DIVIDE", etc.) and a set of three to ten addressing modes ("FROM REGISTER 0", "FROM REGISTER 1", "FROM MEMORY", etc.). The DEC PDP-11 and Motorola 68000 computer architectures are examples of nearly orthogonal instruction sets, while the ARM11 and VAX are examples of CPUs with fully orthogonal instruction sets.

The PDP-11

With the exception of its floating point instructions, the PDP-11 was very strongly orthogonal. Every integer instruction could operate on either 1-byte or 2-byte integers and could access data stored in registers, stored as part of the instruction, stored in memory, or stored in memory and pointed to by addresses in registers. Even the PC and the stack pointer could be affected by the ordinary instructions using all of the ordinary data modes. In fact, "immediate" mode (hardcoded numbers within an instruction, such as ADD #4, R1 (R1 = R1 + 4) was implemented as the mode "register indirect,

autoincrement" and specifying the program counter (R7) as the register to use reference for indirection and to autoincrement.

Since the PDP-11 was an octal-oriented (3-bit sub-byte) machine (addressing modes 0 - 7, registers R0 - R7), there were (electronically) 8 addressing modes. Through the use of the Stack Pointer (R6) and Program Counter (R7) as referenceable registers, there were 10 conceptual addressing modes available.

The VAX-11

The VAX-11 extended the PDP-11's orthogonality to all data types, including floating point numbers (although instructions such as 'ADD' was divided into data-size dependant variants such as ADDB, ADDW, ADDL, ADDP, ADDF for add byte, word, longword, packed BCD and floating point, respectively). Like the PDP-11, the Stack Pointer and Program Counter were in the general register file (R14 and R15).

The general form of a VAX 11 instruction would be:

opcode [operand] [operand] ...

Each component being one byte, the opcode a value in the range 0 - 255, and each operand consisting of two nibbles, the upper 4 bits specifying an addressing mode, and the lower 4 bits (usually) specifying a register number (R0 - R15).

Unlike the octal-oriented PDP-11, the VAX-11 was a hexadecimal-oriented machine (4-bit sub-byte). This resulted in 16 logical addressing modes (0-15), however, addressing modes 0-3 were "short immediate" for immediate data of 6 bits or less (the 2 low-order bits of the addressing mode being the 2 high-order bits of the immediate data, when prepended to the remaining 4 bits in that data-addressing byte). Since addressing modes 0-3 were identical, this made 13 (electronic) addressing modes, but as in the PDP-11, the use of the Stack Pointer (R14) and Program Counter (R15) created a total of over 15 conceptual addressing modes (with the assembler program translating the source code into the actual stack-pointer or program-counter based addressing mode needed).

The MC68000

By comparison, Motorola's designers attempted to make the assembly language orthogonal while the underlying machine language was somewhat less so. Compared to the PDP-11, the MC68000 used separate registers to store instructions and the addresses of data in memory; the assembly language "hid" some of this separation from the programmer. Many programmers disliked the "near" orthogonality, while others were grateful for the attempt.

At the bit level, the person writing the assembler (or debugging machine code) would clearly see that these "instructions" could become any of several different op-codes. It was quite a good compromise because it gave almost the same convenience as a truly

orthogonal machine, and yet also gave the CPU designers freedom to use the bits in the instructions more efficiently than a purely-orthogonal approach might have.

The 8080 and follow on designs

The 8-bit Intel 8080 (as well as the 8085 and 8051) microprocessor was basically a slightly extended accumulator-based design and therefore not orthogonal. An assembly-language programmer or compiler writer had to be mindful of which operations were possible on each register: Most 8-bit operations could be performed only on the 8-bit accumulator (the A-register), while 16-bit operations could be performed only on the 16-bit pointer/accumulator (the HL-register pair), whereas simple operations, such as increment, was possible on all seven 8-bit registers. This was largely due to a desire to keep all opcodes one byte long and to maintain source code compatibility with the original Intel 8008 (an LSI-implementation of the Datapoint 2200's CPU).

The binary-compatible Z80 later added prefix-codes to escape from this 1-byte limit and allow for a more powerful instruction set. The same basic idea was employed for the Intel 8086, although, to allow for more radical extensions, *binary*-compatibility with the 8080 was not attempted here; instead the 8086 was designed as a more regular and fully 16-bit processor that was *source*-compatible with the 8008, 8080, and 8085. It maintained some degree of non-orthogonality for the sake of high code density (even though this was derided as being "baroque" by some computer scientists at the time). The 32-bit extension of this architecture that was introduced with the 80386, was, by all practical means, fully orthogonal, despite keeping all the 8086 instructions and their extended counterparts. However, the *encoding-strategy* used still shows many traces from the 8008 and 8080 (and Z80); for instance, single-byte encodings remain for certain frequent operations such as **push** and **pop** of registers and constants, and the primary accumulator, **eax**, employ shorter encodings than the other registers on certain types of operations; observations like this are sometimes exploited for code optimization in both compilers and hand written code.

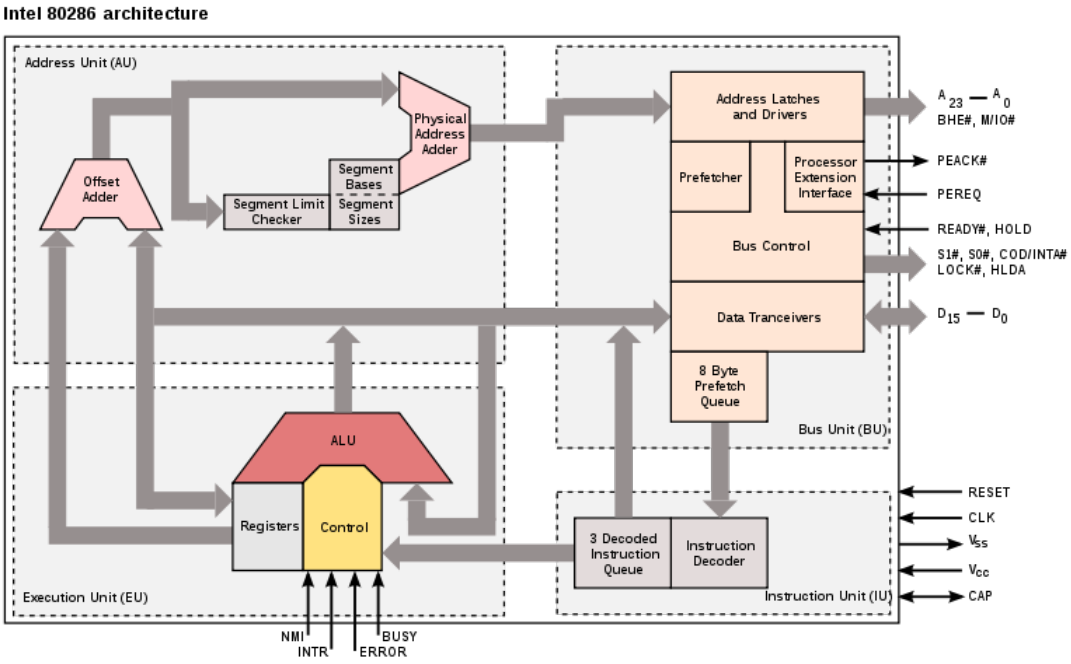
Into the RISC age

A fully orthogonal architecture may not be the most "bit efficient" architecture. In the late 1970s research at IBM (and similar projects elsewhere) demonstrated that the majority of these "orthogonal" addressing modes were ignored by most programs. Perhaps some of the bits that were used to express the fully orthogonal instruction set could instead be used to express more virtual address bits or select from among more registers.

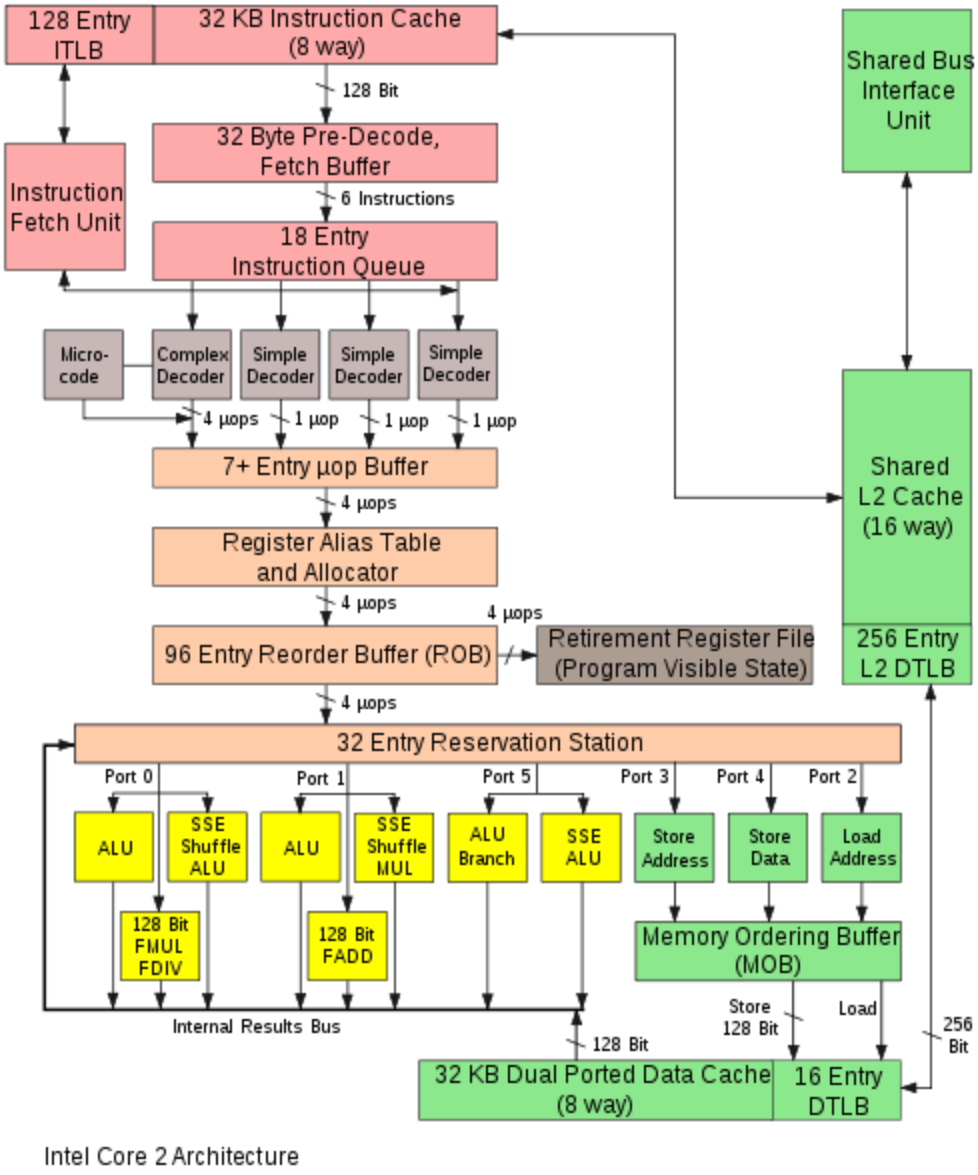
In the RISC age, computer designers strove to achieve a balance that they thought better. In particular, most RISC computers, while still being highly orthogonal with regard to which instructions can process which data types, now have reverted to "load/store" architectures. In these architectures, only a very few *memory reference instructions* can access main memory and only for the purpose of loading data into registers or storing register data back into main memory; only a few addressing modes may be available, and these modes may vary depending on whether the instruction refers to data or involves a

transfer of control (jump). Conversely, data must be in registers before it can be operated upon by the other instructions in the computer's instruction set. This trade off is made explicitly to enable the use of much larger register sets, extended virtual addresses, and longer *immediate data* (data stored directly within the computer instruction).

Microarchitecture



Intel 80286 microarchitecture



Intel Core microarchitecture

In computer engineering, **microarchitecture** (sometimes abbreviated to µarch or uarch), also called **computer organization**, is the way a given instruction set architecture (ISA) is implemented on a processor. A given ISA may be implemented with different microarchitectures. Implementations might vary due to different goals of a given design or due to shifts in technology. Computer architecture is the combination of microarchitecture and instruction set design.

Relation to instruction set architecture

The ISA is roughly the same as the programming model of a processor as seen by an assembly language programmer or compiler writer. The ISA includes the execution model, processor registers, address and data formats among other things. The microarchitecture includes the constituent parts of the processor and how these interconnect and interoperate to implement the ISA.

The microarchitecture of a machine is usually represented as (more or less detailed) diagrams that describe the interconnections of the various microarchitectual elements of the machine, which may be everything from single gates and registers, to complete arithmetic logic units (ALU)s and even larger elements. These diagrams generally separate the data path (where data is placed) and the control path (which can be said to steer the data).

Each microarchitectural element is in turn represented by a schematic describing the interconnections of logic gates used to implement it. Each logic gate is in turn represented by a circuit diagram describing the connections of the transistors used to implement it in some particular logic family. Machines with different microarchitectures may have the same instruction set architecture, and thus be capable of executing the same programs. New microarchitectures and/or circuitry solutions, along with advances in semiconductor manufacturing, are what allows newer generations of processors to achieve higher performance while using the same ISA.

In principle, a single microarchitecture could execute several different ISAs with only minor changes to the microcode.

Aspects of microarchitecture

The pipelined datapath is the most commonly used datapath design in microarchitecture today. This technique is used in most modern microprocessors, microcontrollers, and DSPs. The pipelined architecture allows multiple instructions to overlap in execution, much like an assembly line. The pipeline includes several different stages which are fundamental in microarchitecture designs. Some of these stages include instruction fetch, instruction decode, execute, and write back. Some architectures include other stages such as memory access. The design of pipelines is one of the central microarchitectural tasks.

Execution units are also essential to microarchitecture. Execution units include arithmetic logic units (ALU), floating point units (FPU), load/store units, branch prediction, and SIMD. These units perform the operations or calculations of the processor. The choice of the number of execution units, their latency and throughput is a central microarchitectural design task. The size, latency, throughput and connectivity of memories within the system are also microarchitectural decisions.

System-level design decisions such as whether or not to include peripherals, such as memory controllers, can be considered part of the microarchitectural design process. This includes decisions on the performance-level and connectivity of these peripherals.

Unlike architectural design, where achieving a specific performance level is the main goal, microarchitectural design pays closer attention to other constraints. Since microarchitecture design decisions directly affect what goes into a system, attention must be paid to such issues as:

- Chip area/cost
- Power consumption
- Logic complexity
- Ease of connectivity
- Manufacturability
- Ease of debugging
- Testability

Microarchitectural concepts

In general, all CPUs, single-chip microprocessors or multi-chip implementations run programs by performing the following steps:

1. Read an instruction and decode it
2. Find any associated data that is needed to process the instruction
3. Process the instruction
4. Write the results out

Complicating this simple-looking series of steps is the fact that the memory hierarchy, which includes caching, main memory and non-volatile storage like hard disks, (where the program instructions and data reside) has always been slower than the processor itself. Step (2) often introduces a lengthy (in CPU terms) delay while the data arrives over the computer bus. A considerable amount of research has been put into designs that avoid these delays as much as possible. Over the years, a central goal was to execute more instructions in parallel, thus increasing the effective execution speed of a program. These efforts introduced complicated logic and circuit structures. Initially these techniques could only be implemented on expensive mainframes or supercomputers due to the amount of circuitry needed for these techniques. As semiconductor manufacturing progressed, more and more of these techniques could be implemented on a single semiconductor chip.

What follows is a survey of micro-architectural techniques that are common in modern CPUs.

Instruction set choice

Instruction sets have shifted over the years, from originally very simple to sometimes very complex (in various respects). In recent years, load-store architectures, VLIW and EPIC types have been in fashion. Architectures that are dealing with data parallelism include SIMD and Vectors. Some labels used to denote classes of CPU architectures are not particularly descriptive, especially so the CISC label; many early designs retroactively denoted "CISC" are in fact significantly simpler than modern RISC processors (in several respects).

However, the choice of instruction set architecture may greatly affect the complexity of implementing high performance devices. The prominent strategy, used to develop the first RISC processors, was to simplify instructions to a minimum of individual semantic complexity combined with high encoding regularity and simplicity. Such uniform instructions were easily fetched, decoded and executed in a pipelined fashion and a simple strategy to reduce the number of logic levels in order to reach high operating frequencies; instruction cache-memories compensated for the higher operating frequency and inherently low code density while large register sets were used to factor out as much of the (slow) memory accesses as possible.

Instruction pipelining

One of the first, and most powerful, techniques to improve performance is the use of the instruction pipeline. Early processor designs would carry out all of the steps above for one instruction before moving onto the next. Large portions of the circuitry were left idle at any one step; for instance, the instruction decoding circuitry would be idle during execution and so on.

Pipelines improve performance by allowing a number of instructions to work their way through the processor at the same time. In the same basic example, the processor would start to decode (step 1) a new instruction while the last one was waiting for results. This would allow up to four instructions to be "in flight" at one time, making the processor look four times as fast. Although any one instruction takes just as long to complete (there are still four steps) the CPU as a whole "retires" instructions much faster and can be run at a much higher clock speed.

RISC make pipelines smaller and much easier to construct by cleanly separating each stage of the instruction process and making them take the same amount of time — one cycle. The processor as a whole operates in an assembly line fashion, with instructions coming in one side and results out the other. Due to the reduced complexity of the Classic RISC pipeline, the pipelined core and an instruction cache could be placed on the same size die that would otherwise fit the core alone on a CISC design. This was the real

reason that RISC was faster. Early designs like the SPARC and MIPS often ran over 10 times as fast as Intel and Motorola CISC solutions at the same clock speed and price.

Pipelines are by no means limited to RISC designs. By 1986 the top-of-the-line VAX implementation (VAX 8800) was a heavily pipelined design, slightly predating the first commercial MIPS and SPARC designs. Most modern CPUs (even embedded CPUs) are now pipelined, and microcoded CPUs with no pipelining are seen only in the most area-constrained embedded processors. Large CISC machines, from the VAX 8800 to the modern Pentium 4 and Athlon, are implemented with both microcode and pipelines. Improvements in pipelining and caching are the two major microarchitectural advances that have enabled processor performance to keep pace with the circuit technology on which they are based.

Cache

It was not long before improvements in chip manufacturing allowed for even more circuitry to be placed on the die, and designers started looking for ways to use it. One of the most common was to add an ever-increasing amount of cache memory on-die. Cache is simply very fast memory, memory that can be accessed in a few cycles as opposed to "many" needed to talk to main memory. The CPU includes a cache controller which automates reading and writing from the cache, if the data is already in the cache it simply "appears," whereas if it is not the processor is "stalled" while the cache controller reads it in.

RISC designs started adding cache in the mid-to-late 1980s, often only 4 KB in total. This number grew over time, and typical CPUs now have at least 512 KB, while more powerful CPUs come with 1 or 2 or even 4, 6, 8 or 12 MB, organized in multiple levels of a memory hierarchy. Generally speaking, more cache means more performance, due to reduced stalling.

Caches and pipelines were a perfect match for each other. Previously, it didn't make much sense to build a pipeline that could run faster than the access latency of off-chip memory. Using on-chip cache memory instead, meant that a pipeline could run at the speed of the cache access latency, a much smaller length of time. This allowed the operating frequencies of processors to increase at a much faster rate than that of off-chip memory.

Branch prediction

One barrier to achieving higher performance through instruction-level parallelism stems from pipeline stalls and flushes due to branches. Normally, whether a conditional branch will be taken isn't known until late in the pipeline as conditional branches depend on results coming from a register. From the time that the processor's instruction decoder has figured out that it has encountered a conditional branch instruction to the time that the deciding register value can be read out, the pipeline needs to be stalled for several cycles, or if it's not and the branch is taken, the pipeline needs to be flushed. As clock speeds

increase the depth of the pipeline increases with it, and some modern processors may have 20 stages or more. On average, every fifth instruction executed is a branch, so without any intervention, that's a high amount of stalling.

Techniques such as branch prediction and speculative execution are used to lessen these branch penalties. Branch prediction is where the hardware makes educated guesses on whether a particular branch will be taken. In reality one side or the other of the branch will be called much more often than the other. Modern designs have rather complex statistical prediction systems, which watch the results of past branches to predict the future with greater accuracy. The guess allows the hardware to prefetch instructions without waiting for the register read. Speculative execution is a further enhancement in which the code along the predicted path is not just prefetched but also executed before it is known whether the branch should be taken or not. This can yield better performance when the guess is good, with the risk of a huge penalty when the guess is bad because instructions need to be undone.

Superscalar

Even with all of the added complexity and gates needed to support the concepts outlined above, improvements in semiconductor manufacturing soon allowed even more logic gates to be used.

In the outline above the processor processes parts of a single instruction at a time. Computer programs could be executed faster if multiple instructions were processed simultaneously. This is what superscalar processors achieve, by replicating functional units such as ALUs. The replication of functional units was only made possible when the die area of a single-issue processor no longer stretched the limits of what could be reliably manufactured. By the late 1980s, superscalar designs started to enter the market place.

In modern designs it is common to find two load units, one store (many instructions have no results to store), two or more integer math units, two or more floating point units, and often a SIMD unit of some sort. The instruction issue logic grows in complexity by reading in a huge list of instructions from memory and handing them off to the different execution units that are idle at that point. The results are then collected and re-ordered at the end.

Out-of-order execution

The addition of caches reduces the frequency or duration of stalls due to waiting for data to be fetched from the memory hierarchy, but does not get rid of these stalls entirely. In early designs a *cache miss* would force the cache controller to stall the processor and wait. Of course there may be some other instruction in the program whose data *is* available in the cache at that point. Out-of-order execution allows that ready instruction to be processed while an older instruction waits on the cache, then re-orders the results to make it appear that everything happened in the programmed order. This technique is also

used to avoid other operand dependency stalls, such as an instruction awaiting a result from a long latency floating-point operation or other multi-cycle operations.

Register renaming

Register renaming refers to a technique used to avoid unnecessary serialized execution of program instructions because of the reuse of the same registers by those instructions. Suppose we have two groups of instruction that will use the same register, one set of instruction is executed first to leave the register to the other set, but if the other set is assigned to a different similar register both sets of instructions can be executed in parallel.

Multiprocessing and multithreading

Computer architects have become stymied by the growing mismatch in CPU operating frequencies and DRAM access times. None of the techniques that exploited instruction-level parallelism within one program could make up for the long stalls that occurred when data had to be fetched from main memory. Additionally, the large transistor counts and high operating frequencies needed for the more advanced ILP techniques required power dissipation levels that could no longer be cheaply cooled. For these reasons, newer generations of computers have started to exploit higher levels of parallelism that exist outside of a single program or program thread.

This trend is sometimes known as *throughput computing*. This idea originated in the mainframe market where online transaction processing emphasized not just the execution speed of one transaction, but the capacity to deal with massive numbers of transactions. With transaction-based applications such as network routing and web-site serving greatly increasing in the last decade, the computer industry has re-emphasized capacity and throughput issues.

One technique of how this parallelism is achieved is through multiprocessing systems, computer systems with multiple CPUs. Once reserved for high-end mainframes and supercomputers, small scale (2-8) multiprocessors servers have become commonplace for the small business market. For large corporations, large scale (16-256) multiprocessors are common. Even personal computers with multiple CPUs have appeared since the 1990s.

With further transistor size reductions made available with semiconductor technology advances, multicore CPUs have appeared where multiple CPUs are implemented on the same silicon chip. Initially used in chips targeting embedded markets, where simpler and smaller CPUs would allow multiple instantiations to fit on one piece of silicon. By 2005, semiconductor technology allowed dual high-end desktop CPUs *CMP* chips to be manufactured in volume. Some designs, such as Sun Microsystems' UltraSPARC T1 have reverted back to simpler (scalar, in-order) designs in order to fit more processors on one piece of silicon.

Another technique that has become more popular recently is multithreading. In multithreading, when the processor has to fetch data from slow system memory, instead of stalling for the data to arrive, the processor switches to another program or program thread which is ready to execute. Though this does not speed up a particular program/thread, it increases the overall system throughput by reducing the time the CPU is idle.

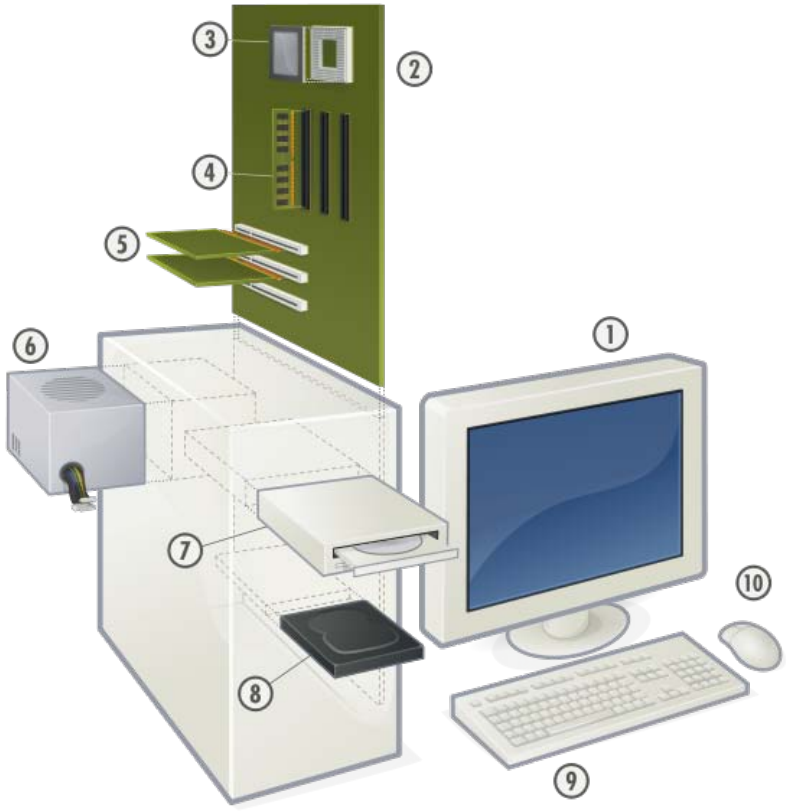
Conceptually, multithreading is equivalent to a context switch at the operating system level. The difference is that a multithreaded CPU can do a thread switch in one CPU cycle instead of the hundreds or thousands of CPU cycles a context switch normally requires. This is achieved by replicating the state hardware (such as the register file and program counter) for each active thread.

A further enhancement is simultaneous multithreading. This technique allows superscalar CPUs to execute instructions from different programs/threads simultaneously in the same cycle.

Chapter 10

Personal Computer Hardware and Computer Software

Personal computer hardware



Hardware of a modern Personal Computer.
1. Monitor
2. Motherboard
3. CPU
4. RAM
5. Expansion cards

6. Power supply
7. Optical disc drive
8. Hard disk drive
9. Keyboard
10. Mouse

A personal computer is made up of multiple physical components of **computer hardware**, upon which can be installed a system software called operating system and a multitude of software applications to perform the operator's desired functions.

Though a PC comes in many different forms, a typical personal computer consists of a case or chassis in a tower shape (desktop), containing components such as a motherboard.

Motherboard

The motherboard is the main component inside the case. It is a large rectangular board with integrated circuitry that connects the rest of the parts of the computer including the CPU, the RAM, the disk drives (CD, DVD, hard disk, or any others) as well as any peripherals connected via the ports or the expansion slots.

Components directly attached to the motherboard include:

- The **central processing unit (**CPU**)** performs most of the calculations which enable a computer to function, and is sometimes referred to as the "brain" of the computer. It is usually cooled by a heat sink and fan.
- The **chip set** mediates communication between the CPU and the other components of the system, including main memory.
- **RAM** (Random Access Memory) stores all running processes (applications) and the current running OS.
- The **BIOS** includes boot firmware and power management. The **B**asic **I**nput **O**utput **S**ystem tasks are handled by operating system drivers.
- **Internal Buses** connect the CPU to various internal components and to expansion cards for graphics and sound.
    - **Current**
        - The north bridge memory controller, for RAM and PCI Express
            - PCI Express, for expansion cards such as graphics and physics processors, and high-end network interfaces
        - PCI, for other expansion cards
        - SATA, for disk drives
    - **Obsolete**
        - ATA (superseded by SATA)
        - AGP (superseded by PCI Express)
        - VLB VESA Local Bus (superseded by AGP)
        - ISA (expansion card slot format obsolete in PCs, but still used in industrial computers)

- **External Bus Controllers** support ports for external peripherals. These ports may be controlled directly by the south bridge I/O controller or based on expansion cards attached to the motherboard through the PCI bus.
    - USB
    - FireWire
    - eSATA
    - SCSI

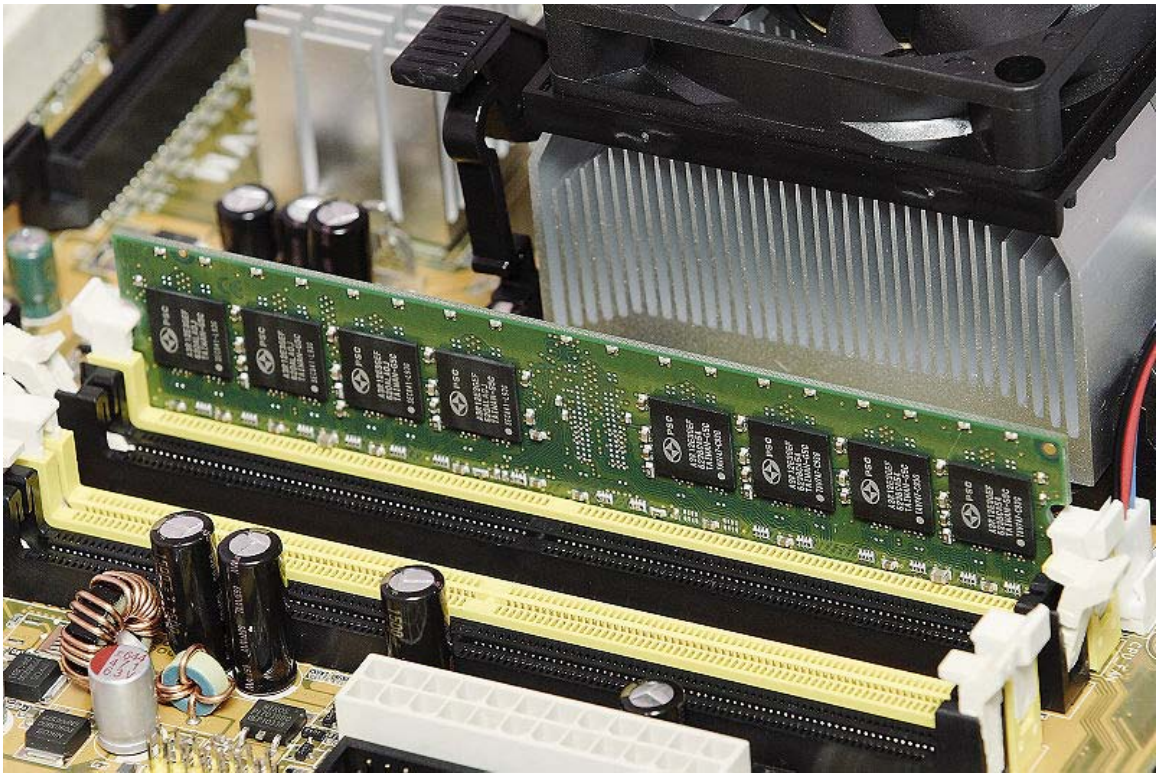
Power supply

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

A power supply unit (PSU) converts alternating current (AC) electric power to low-voltage DC power for the internal components of the computer. Some power supplies have a switch to change between 230 V and 115 V. Other models have automatic sensors

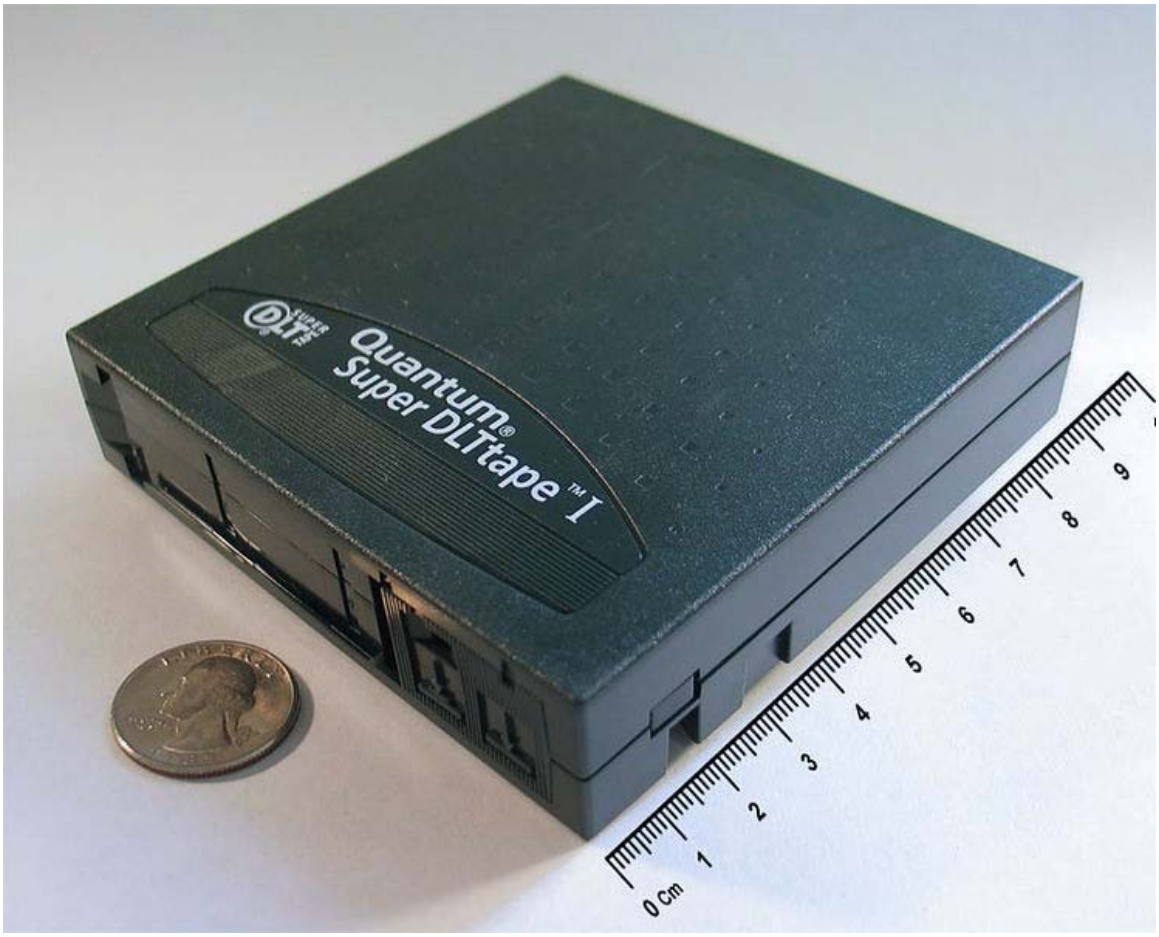
that switch input voltage automatically, or are able to accept any voltage between those limits. Power supply units used in computers are nearly always switch mode power supplies (SMPS). The SMPS provides regulated direct current power at the several voltages required by the motherboard and accessories such as disk drives and cooling fans.

Computer data storage



1 GB of SDRAM mounted in a personal computer. An example of *primary* storage.

40 GB PATA hard disk drive (HDD); when connected to a computer it serves as *secondary* storage.



160 GB SDLT tape cartridge, an example of *off-line* storage. When used within a robotic tape library, it is classified as *tertiary* storage instead.

Computer data storage, often called **storage** or **memory**, refers to computer components and recording media that retain digital data used for computing for some interval of time. Computer data storage provides one of the core functions of the modern computer, that of information retention. It is one of the fundamental components of all modern computers, and coupled with a central processing unit (CPU, a processor), implements the basic computer model used since the 1940s.

In contemporary usage, *memory* usually refers to a form of semiconductor storage known as random-access memory, typically DRAM (Dynamic-RAM) but *memory* can refer to other forms of fast but temporary storage. Similarly, *storage* today more commonly refers to storage devices and their media not directly accessible by the CPU (secondary or tertiary storage) — typically hard disk drives, optical disc drives, and other devices slower than RAM but more permanent. Historically, *memory* has been called *main memory*, *real storage* or *internal memory* while storage devices have been referred to as *secondary storage*, *external memory* or *auxiliary/peripheral storage*.

The contemporary distinctions are helpful, because they are also fundamental to the architecture of computers in general. The distinctions also reflect an important and significant technical difference between memory and mass storage devices, which has been blurred by the historical usage of the term *storage*.

Purpose of storage

Many different forms of storage, based on various natural phenomena, have been invented. So far, no practical universal storage medium exists, and all forms of storage have some drawbacks. Therefore a computer system usually contains several kinds of storage, each with an individual purpose.

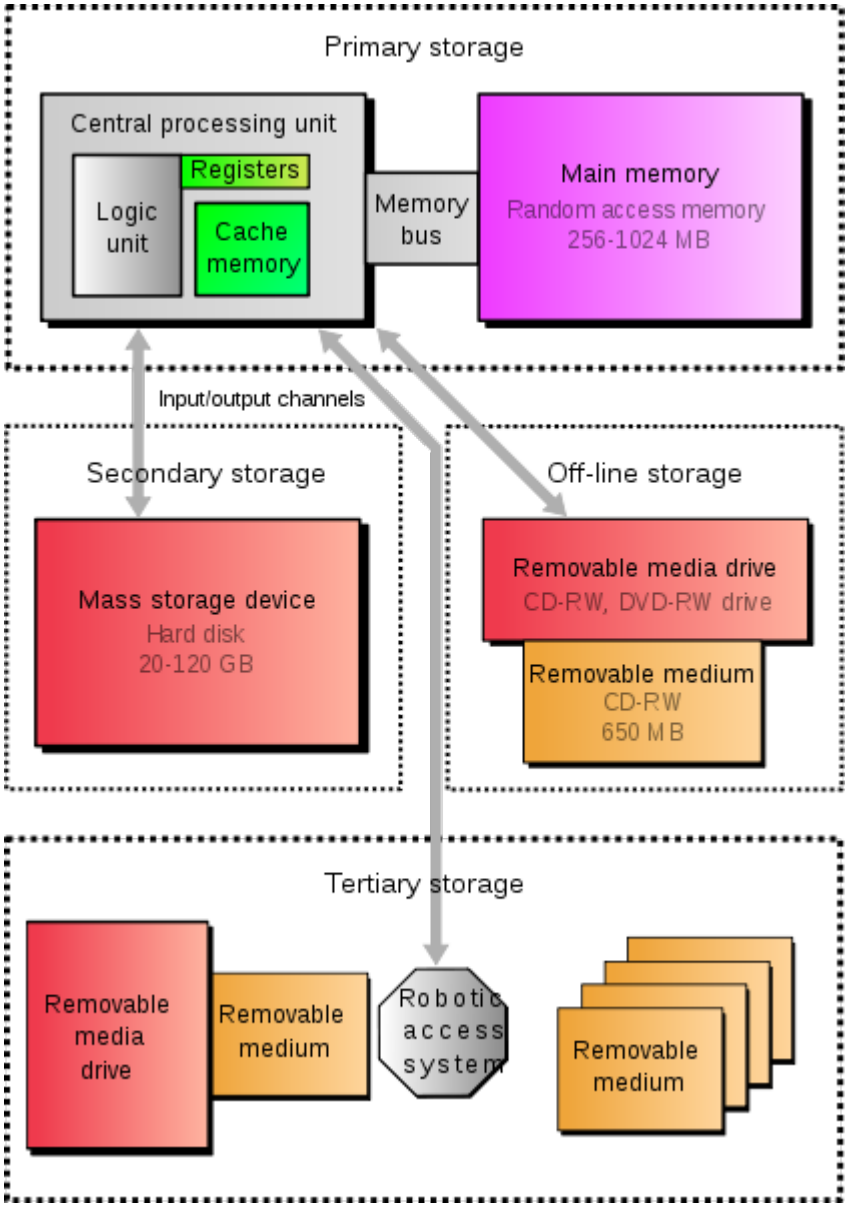
A digital computer represents data using the binary numeral system. Text, numbers, pictures, audio, and nearly any other form of information can be converted into a string of bits, or binary digits, each of which has a value of 1 or 0. The most common unit of storage is the byte, equal to 8 bits. A piece of information can be handled by any computer whose storage space is large enough to accommodate *the binary representation of the piece of information*, or simply data. For example, using eight million bits, or about one megabyte, a typical computer could store a short novel.

Traditionally the most important part of every computer is the central processing unit (CPU, or simply a processor), because it actually operates on data, performs any calculations, and controls all the other components.

Without a significant amount of memory, a computer would merely be able to perform fixed operations and immediately output the result. It would have to be reconfigured to change its behavior. This is acceptable for devices such as desk calculators or simple digital signal processors. Von Neumann machines differ in that they have a memory in which they store their operating instructions and data. Such computers are more versatile in that they do not need to have their hardware reconfigured for each new program, but can simply be reprogrammed with new in-memory instructions; they also tend to be simpler to design, in that a relatively simple processor may keep state between successive computations to build up complex procedural results. Most modern computers are von Neumann machines.

In practice, almost all computers use a variety of memory types, organized in a storage hierarchy around the CPU, as a trade-off between performance and cost. Generally, the lower a storage is in the hierarchy, the lesser its bandwidth and the greater its access latency is from the CPU. This traditional division of storage to primary, secondary, tertiary and off-line storage is also guided by cost per bit.

Hierarchy of storage



Various forms of storage, divided according to their distance from the central processing unit. The fundamental components of a general-purpose computer are arithmetic and logic unit, control circuitry, storage space, and input/output devices. Technology and capacity as in common home computers around 2005.

Primary storage

Primary storage (or **main memory** or **internal memory**), often referred to simply as **memory**, is the only one directly accessible to the CPU. The CPU continuously reads

instructions stored there and executes them as required. Any data actively operated on is also stored there in uniform manner.

Historically, early computers used delay lines, Williams tubes, or rotating magnetic drums as primary storage. By 1954, those unreliable methods were mostly replaced by magnetic core memory. Core memory remained dominant until the 1970s, when advances in integrated circuit technology allowed semiconductor memory to become economically competitive.

This led to modern random-access memory (RAM). It is small-sized, light, but quite expensive at the same time. (The particular types of RAM used for primary storage are also volatile, i.e. they lose the information when not powered).

As shown in the diagram, traditionally there are two more sub-layers of the primary storage, besides main large-capacity RAM:

- Processor registers are located inside the processor. Each register typically holds a word of data (often 32 or 64 bits). CPU instructions instruct the arithmetic and logic unit to perform various calculations or other operations on this data (or with the help of it). Registers are the fastest of all forms of computer data storage.
- Processor cache is an intermediate stage between ultra-fast registers and much slower main memory. It's introduced solely to increase performance of the computer. Most actively used information in the main memory is just duplicated in the cache memory, which is faster, but of much lesser capacity. On the other hand it is much slower, but much larger than processor registers. Multi-level hierarchical cache setup is also commonly used—*primary cache* being smallest, fastest and located inside the processor; *secondary cache* being somewhat larger and slower.

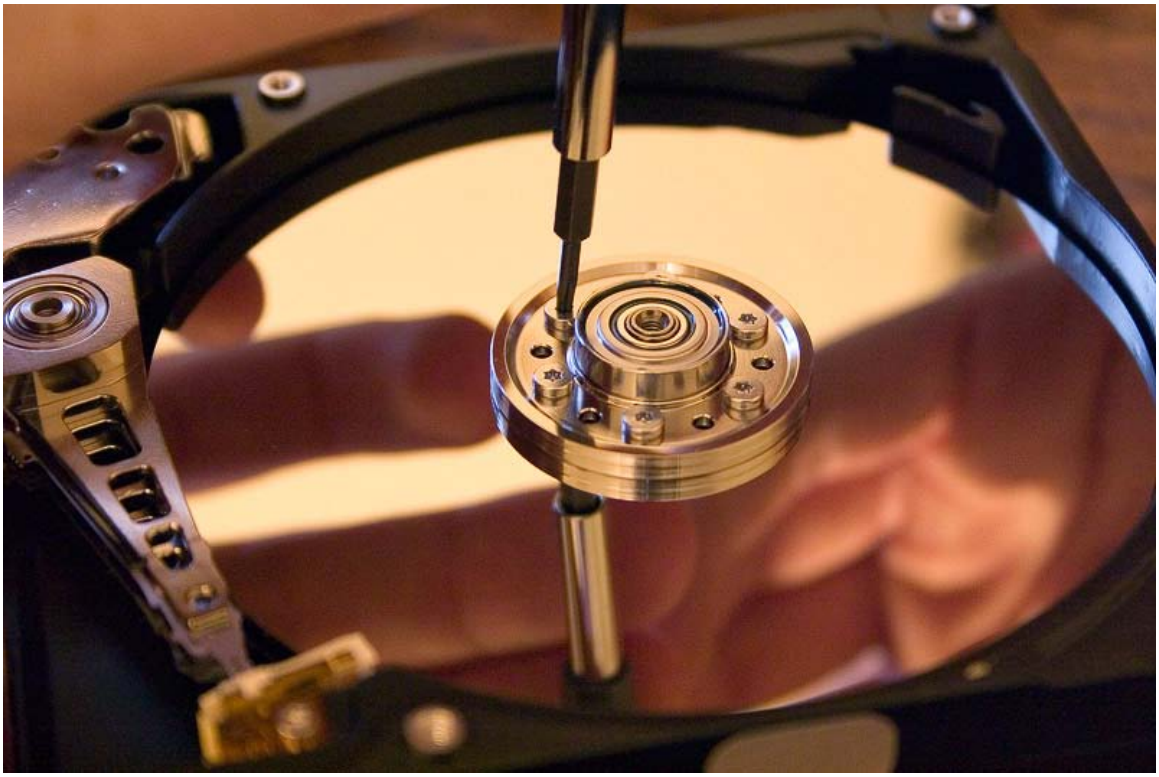
Main memory is directly or indirectly connected to the central processing unit via a *memory bus*. It is actually two buses (not on the diagram): an address bus and a data bus. The CPU firstly sends a number through an address bus, a number called memory address, that indicates the desired location of data. Then it reads or writes the data itself using the data bus. Additionally, a memory management unit (MMU) is a small device between CPU and RAM recalculating the actual memory address, for example to provide an abstraction of virtual memory or other tasks.

As the RAM types used for primary storage are volatile (cleared at start up), a computer containing only such storage would not have a source to read instructions from, in order to start the computer. Hence, non-volatile primary storage containing a small startup program (BIOS) is used to bootstrap the computer, that is, to read a larger program from non-volatile *secondary* storage to RAM and start to execute it. A non-volatile technology used for this purpose is called ROM, for read-only memory (the terminology may be somewhat confusing as most ROM types are also capable of *random access*).

Many types of "ROM" are not literally *read only*, as updates are possible; however it is slow and memory must be erased in large portions before it can be re-written. Some embedded systems run programs directly from ROM (or similar), because such programs are rarely changed. Standard computers do not store non-rudimentary programs in ROM, rather use large capacities of secondary storage, which is non-volatile as well, and not as costly.

Recently, *primary storage* and *secondary storage* in some uses refer to what was historically called, respectively, *secondary storage* and *tertiary storage*.

Secondary storage



A hard disk drive with protective cover removed.

Secondary storage (also known as external memory or auxiliary storage), differs from primary storage in that it is not directly accessible by the CPU. The computer usually uses its input/output channels to access secondary storage and transfers the desired data using intermediate area in primary storage. Secondary storage does not lose the data when the device is powered down—it is non-volatile. Per unit, it is typically also two orders of magnitude less expensive than primary storage. Consequently, modern computer systems typically have two orders of magnitude more secondary storage than primary storage and data is kept for a longer time there.

In modern computers, hard disk drives are usually used as secondary storage. The time taken to access a given byte of information stored on a hard disk is typically a few

thousandths of a second, or milliseconds. By contrast, the time taken to access a given byte of information stored in random access memory is measured in billionths of a second, or nanoseconds. This illustrates the significant access-time difference which distinguishes solid-state memory from rotating magnetic storage devices: hard disks are typically about a million times slower than memory. Rotating optical storage devices, such as CD and DVD drives, have even longer access times. With disk drives, once the disk read/write head reaches the proper placement and the data of interest rotates under it, subsequent data on the track are very fast to access. As a result, in order to hide the initial seek time and rotational latency, data are transferred to and from disks in large contiguous blocks.

When data reside on disk, block access to hide latency offers a ray of hope in designing efficient external memory algorithms. Sequential or block access on disks is orders of magnitude faster than random access, and many sophisticated paradigms have been developed to design efficient algorithms based upon sequential and block access . Another way to reduce the I/O bottleneck is to use multiple disks in parallel in order to increase the bandwidth between primary and secondary memory.

Some other examples of secondary storage technologies are: flash memory (e.g. USB flash drives or keys), floppy disks, magnetic tape, paper tape, punched cards, standalone RAM disks, and Iomega Zip drives.

The secondary storage is often formatted according to a file system format, which provides the abstraction necessary to organize data into files and directories, providing also additional information (called metadata) describing the owner of a certain file, the access time, the access permissions, and other information.

Most computer operating systems use the concept of virtual memory, allowing utilization of more primary storage capacity than is physically available in the system. As the primary memory fills up, the system moves the least-used chunks (*pages*) to secondary storage devices (to a swap file or page file), retrieving them later when they are needed. As more of these retrievals from slower secondary storage are necessary, the more the overall system performance is degraded.

Tertiary storage

Large tape library. Tape cartridges placed on shelves in the front, robotic arm moving in the back. Visible height of the library is about 180 cm.

Tertiary storage or **tertiary memory**, provides a third level of storage. Typically it involves a robotic mechanism which will *mount* (insert) and *dismount* removable mass storage media into a storage device according to the system's demands; this data is often copied to secondary storage before use. It is primarily used for archival of rarely accessed information since it is much slower than secondary storage (e.g. 5–60 seconds vs. 1-10 milliseconds). This is primarily useful for extraordinarily large data stores, accessed without human operators. Typical examples include tape libraries and optical jukeboxes.

When a computer needs to read information from the tertiary storage, it will first consult a catalog database to determine which tape or disc contains the information. Next, the computer will instruct a robotic arm to fetch the medium and place it in a drive. When the computer has finished reading the information, the robotic arm will return the medium to its place in the library.

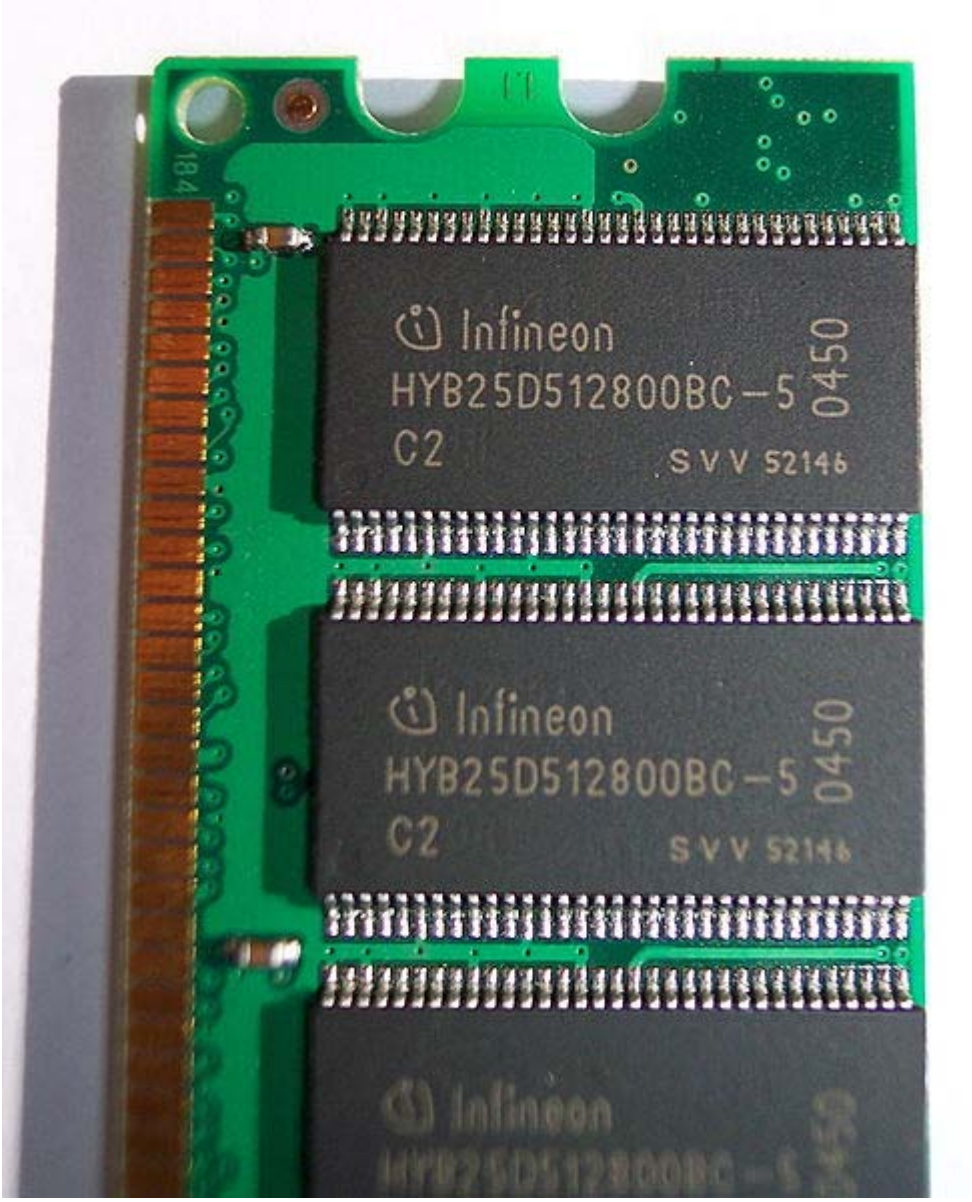
Off-line storage

Off-line storage is a computer data storage on a medium or a device that is not under the control of a processing unit. The medium is recorded, usually in a secondary or tertiary storage device, and then physically removed or disconnected. It must be inserted or connected by a human operator before a computer can access it again. Unlike tertiary storage, it cannot be accessed without human interaction.

Off-line storage is used to transfer information, since the detached medium can be easily physically transported. Additionally, in case a disaster, for example a fire, destroys the original data, a medium in a remote location will probably be unaffected, enabling disaster recovery. Off-line storage increases general information security, since it is physically inaccessible from a computer, and data confidentiality or integrity cannot be affected by computer-based attack techniques. Also, if the information stored for archival purposes is accessed seldom or never, off-line storage is less expensive than tertiary storage.

In modern personal computers, most secondary and tertiary storage media are also used for off-line storage. Optical discs and flash memory devices are most popular, and to much lesser extent removable hard disk drives. In enterprise uses, magnetic tape is predominant. Older examples are floppy disks, Zip disks, or punched cards.

Characteristics of storage



A 1GB DDR RAM memory module (detail)

Storage technologies at all levels of the storage hierarchy can be differentiated by evaluating certain core characteristics as well as measuring characteristics specific to a particular implementation. These core characteristics are volatility, mutability, accessibility, and addressibility. For any particular implementation of any storage technology, the characteristics worth measuring are capacity and performance.

Volatility

Non-volatile memory
: Will retain the stored information even if it is not constantly supplied with electric power. It is suitable for long-term storage of information. Nowadays used for most of secondary, tertiary, and off-line storage. In 1950s and 1960s, it was also used for primary storage, in the form of magnetic core memory.

Volatile memory
: Requires constant power to maintain the stored information. The fastest memory technologies of today are volatile ones (not a universal rule). Since primary storage is required to be very fast, it predominantly uses volatile memory.

Differentiation

Dynamic random access memory
: A form of volatile memory which also requires the stored information to be periodically re-read and re-written, or refreshed, otherwise it would vanish.

Static memory
: A form of volatile memory similar to DRAM with the exception that it never needs to be refreshed as long as power is applied. (It loses its content if power is removed).

Mutability

Read/write storage or mutable storage
: Allows information to be overwritten at any time. A computer without some amount of read/write storage for primary storage purposes would be useless for many tasks. Modern computers typically use read/write storage also for secondary storage.

Read only storage
: Retains the information stored at the time of manufacture, and **write once storage** (Write Once Read Many) allows the information to be written only once at some point after manufacture. These are called **immutable storage**. Immutable storage is used for tertiary and off-line storage. Examples include CD-ROM and CD-R.

Slow write, fast read storage
: Read/write storage which allows information to be overwritten multiple times, but with the write operation being much slower than the read operation. Examples include CD-RW and flash memory.

Accessibility

Random access
: Any location in storage can be accessed at any moment in approximately the same amount of time. Such characteristic is well suited for primary and secondary storage.

Sequential access
: The accessing of pieces of information will be in a serial order, one after the other; therefore the time to access a particular piece of information depends upon

which piece of information was last accessed. Such characteristic is typical of off-line storage.

Addressability

Location-addressable
: Each individually accessible unit of information in storage is selected with its numerical memory address. In modern computers, location-addressable storage usually limits to primary storage, accessed internally by computer programs, since location-addressability is very efficient, but burdensome for humans.

File addressable
: Information is divided into *files* of variable length, and a particular file is selected with human-readable directory and file names. The underlying device is still location-addressable, but the operating system of a computer provides the file system abstraction to make the operation more understandable. In modern computers, secondary, tertiary and off-line storage use file systems.

Content-addressable
: Each individually accessible unit of information is selected based on the basis of (part of) the contents stored there. Content-addressable storage can be implemented using software (computer program) or hardware (computer device), with hardware being faster but more expensive option. Hardware content addressable memory is often used in a computer's CPU cache.

Capacity

Raw capacity
: The total amount of stored information that a storage device or medium can hold. It is expressed as a quantity of bits or bytes (e.g. 10.4 megabytes).

Memory storage density
: The compactness of stored information. It is the storage capacity of a medium divided with a unit of length, area or volume (e.g. 1.2 megabytes per square inch).

Performance

Latency
: The time it takes to access a particular location in storage. The relevant unit of measurement is typically nanosecond for primary storage, millisecond for secondary storage, and second for tertiary storage. It may make sense to separate read latency and write latency, and in case of sequential access storage, minimum, maximum and average latency.

Throughput
: The rate at which information can be read from or written to the storage. In computer data storage, throughput is usually expressed in terms of megabytes per second or MB/s, though bit rate may also be used. As with latency, read rate and write rate may need to be differentiated. Also accessing media sequentially, as opposed to randomly, typically yields maximum throughput.

Energy use

- Storage devices that reduce fan usage, automatically shut-down during inactivity, and low power hard drives can reduce energy consumption 90 percent.
- 2.5 inch hard disk drives often consume less power than larger ones. Low capacity solid-state drives have no moving parts and consume less power than hard disks. Also, memory may use more power than hard disks.

Fundamental storage technologies

As of 2008, the most commonly used data storage technologies are semiconductor, magnetic, and optical, while paper still sees some limited usage. Some other fundamental storage technologies have also been used in the past or are proposed for development.

Semiconductor

Semiconductor memory uses semiconductor-based integrated circuits to store information. A semiconductor memory chip may contain millions of tiny transistors or capacitors. Both *volatile* and *non-volatile* forms of semiconductor memory exist. In modern computers, primary storage almost exclusively consists of dynamic volatile semiconductor memory or dynamic random access memory. Since the turn of the century, a type of non-volatile semiconductor memory known as flash memory has steadily gained share as off-line storage for home computers. Non-volatile semiconductor memory is also used for secondary storage in various advanced electronic devices and specialized computers.

Magnetic

Magnetic storage uses different patterns of magnetization on a magnetically coated surface to store information. Magnetic storage is *non-volatile*. The information is accessed using one or more read/write heads which may contain one or more recording transducers. A read/write head only covers a part of the surface so that the head or medium or both must be moved relative to another in order to access data. In modern computers, magnetic storage will take these forms:

- Magnetic disk
    - Floppy disk, used for off-line storage
    - Hard disk drive, used for secondary storage
- Magnetic tape data storage, used for tertiary and off-line storage

In early computers, magnetic storage was also used for primary storage in a form of magnetic drum, or core memory, core rope memory, thin-film memory, twistor memory or bubble memory. Also unlike today, magnetic tape was often used for secondary storage.

Optical

Optical storage, the typical optical disc, stores information in deformities on the surface of a circular disc and reads this information by illuminating the surface with a laser diode and observing the reflection. Optical disc storage is *non-volatile*. The deformities may be permanent (read only media), formed once (write once media) or reversible (recordable or read/write media). The following forms are currently in common use:

- CD, CD-ROM, DVD, BD-ROM: Read only storage, used for mass distribution of digital information (music, video, computer programs)
- CD-R, DVD-R, DVD+R, BD-R: Write once storage, used for tertiary and off-line storage
- CD-RW, DVD-RW, DVD+RW, DVD-RAM, BD-RE: Slow write, fast read storage, used for tertiary and off-line storage
- Ultra Density Optical or UDO is similar in capacity to BD-R or BD-RE and is slow write, fast read storage used for tertiary and off-line storage.

Magneto-optical disc storage is optical disc storage where the magnetic state on a ferromagnetic surface stores information. The information is read optically and written by combining magnetic and optical methods. Magneto-optical disc storage is *non-volatile*, *sequential access*, slow write, fast read storage used for tertiary and off-line storage.

3D optical data storage has also been proposed.

Paper

Paper data storage, typically in the form of paper tape or punched cards, has long been used to store information for automatic processing, particularly before general-purpose computers existed. Information was recorded by punching holes into the paper or cardboard medium and was read mechanically (or later optically) to determine whether a particular location on the medium was solid or contained a hole. A few technologies allow people to make marks on paper that are easily read by machine—these are widely used for tabulating votes and grading standardized tests. Barcodes made it possible for any object that was to be sold or transported to have some computer readable information securely attached to it.

Uncommon

Vacuum tube memory

A Williams tube used a cathode ray tube, and a Selectron tube used a large vacuum tube to store information. These primary storage devices were short-lived in the market, since Williams tube was unreliable and Selectron tube was expensive.

Electro-acoustic memory

Delay line memory used sound waves in a substance such as mercury to store information. Delay line memory was dynamic volatile, cycle sequential read/write storage, and was used for primary storage.

Optical tape
: is a medium for optical storage generally consisting of a long and narrow strip of plastic onto which patterns can be written and from which the patterns can be read back. It shares some technologies with cinema film stock and optical discs, but is compatible with neither. The motivation behind developing this technology was the possibility of far greater storage capacities than either magnetic tape or optical discs.

Phase-change memory
: uses different mechanical phases of Phase Change Material to store information in an X-Y addressable matrix, and reads the information by observing the varying electrical resistance of the material. Phase-change memory would be non-volatile, random access read/write storage, and might be used for primary, secondary and off-line storage. Most rewritable and many write once optical disks already use phase change material to store information.

Holographic data storage
: stores information optically inside crystals or photopolymers. Holographic storage can utilize the whole volume of the storage medium, unlike optical disc storage which is limited to a small number of surface layers. Holographic storage would be non-volatile, sequential access, and either write once or read/write storage. It might be used for secondary and off-line storage.

Molecular memory
: stores information in polymer that can store electric charge. Molecular memory might be especially suited for primary storage. The theoretical storage capacity of molecular memory is 10 terabits per square inch.

Related technologies

Network connectivity

A secondary or tertiary storage may connect to a computer utilizing computer networks. This concept does not pertain to the primary storage, which is shared between multiple processors in a much lesser degree.

- **Direct-attached storage** (DAS) is a traditional mass storage, that does not use any network. This is still a most popular approach. This term was coined lately, together with NAS and SAN.
- **Network-attached storage** (NAS) is mass storage attached to a computer which another computer can access at file level over a local area network, a private wide area network, or in the case of online file storage, over the Internet. NAS is commonly associated with the NFS and CIFS/SMB protocols.
- **Storage area network** (SAN) is a specialized network, that provides other computers with storage capacity. The crucial difference between NAS and SAN is the former presents and manages file systems to client computers, whilst the latter provides access at block-addressing (raw) level, leaving it to attaching systems to manage data or file systems within the provided capacity. SAN is commonly associated with Fibre Channel networks.

Robotic storage

Large quantities of individual magnetic tapes, and optical or magneto-optical discs may be stored in robotic tertiary storage devices. In tape storage field they are known as tape libraries, and in optical storage field optical jukeboxes, or optical disk libraries per analogy. Smallest forms of either technology containing just one drive device are referred to as autoloaders or autochangers.

Robotic-access storage devices may have a number of slots, each holding individual media, and usually one or more picking robots that traverse the slots and load media to built-in drives. The arrangement of the slots and picking devices affects performance. Important characteristics of such storage are possible expansion options: adding slots, modules, drives, robots. Tape libraries may have from 10 to more than 100,000 slots, and provide terabytes or petabytes of near-line information. Optical jukeboxes are somewhat smaller solutions, up to 1,000 slots.

Robotic storage is used for backups, and for high-capacity archives in imaging, medical, and video industries. Hierarchical storage management is a most known archiving strategy of automatically *migrating* long-unused files from fast hard disk storage to libraries or jukeboxes. If the files are needed, they are *retrieved* back to disk.

Secondary storage

Hardware that keeps data inside the computer for later use and remains persistent even when the computer has no power.

- Hard disk - for medium-term storage of data.
- Solid-state drive - a device similar to hard disk, but containing no moving parts and stores data in a digital format.
- RAID array controller - a device to manage several internal or external hard disks and optionally some peripherals in order to achieve performance or reliability improvement in what is called a RAID array.

Sound card

Enables the computer to output sound to audio devices, as well as accept input from a microphone. Most modern computers have sound cards built-in to the motherboard, though it is common for a user to install a separate sound card as an upgrade. Most sound cards, either built-in or added, have surround sound capabilities.

Input and output peripherals

Input device

An **input device** is any peripheral (piece of computer hardware equipment) used to provide data and control signals to an information processing system (such as a computer). Input and output devices make up the hardware interface between a computer as a scanner or 6DOF controller.

Many input devices can be classified according to:

- modality of input (e.g. mechanical motion, audio, visual, etc.)
- the input is discrete (e.g. keypresses) or continuous (e.g. a mouse's position, though digitized into a discrete quantity, is fast enough to be considered continuous)
- the number of degrees of freedom involved (e.g. two-dimensional traditional mice, or three-dimensional navigators designed for CAD applications)

Pointing devices, which are input devices used to specify a position in space, can further be classified according to:

- Whether the input is direct or indirect. With direct input, the input space coincides with the display space, i.e. pointing is done in the space where visual feedback or the cursor appears. Touchscreens and light pens involve direct input. Examples involving indirect input include the mouse and trackball.
- Whether the positional information is absolute (e.g. on a touch screen) or relative (e.g. with a mouse that can be lifted and repositioned)

Note that direct input is almost necessarily absolute, but indirect input may be either absolute or relative. For example, digitizing Graphics tablets that do not have an embedded screen involve indirect input and sense absolute positions and are often run in an absolute input mode, but they may also be setup to simulate a relative input mode where the stylus or puck can be lifted and repositioned.

Keyboards

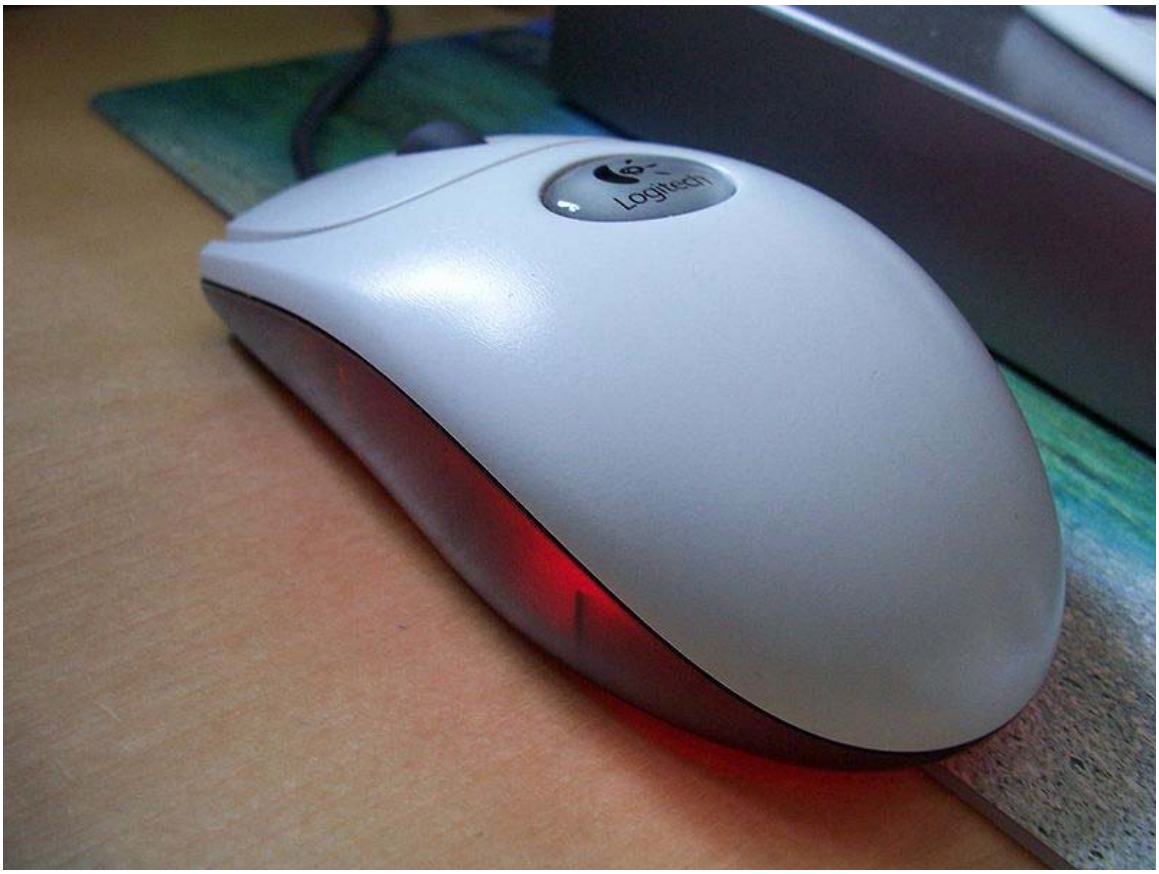
A 'keyboard' is a human interface device which is represented as a layout of buttons. Each button, or key, can be used to either input a linguistic character to a computer, or to call upon a particular function of the computer. Traditional keyboards use spring-based buttons, though newer variations employ virtual keys, or even projected keyboards.

Examples of types of keyboards include:

- Computer keyboard
- Keyer

- Chorded keyboard
- LPFK

Pointing devices



A computer mouse

A **pointing device** is any human interface device that allows a user to input spatial data to a computer. In the case of mice and touch screens, this is usually achieved by detecting movement across a physical surface. Analog devices, such as 3D mice, joysticks, or pointing sticks, function by reporting their angle of deflection. Movements of the pointing device are echoed on the screen by movements of the cursor, creating a simple, intuitive way to navigate a computer's GUI.

High-degree of freedom input devices

Some devices allow many continuous degrees of freedom as input. These can be used as pointing devices, but are generally used in ways that don't involve pointing to a location in space, such as the control of a camera angle while in 3D applications. These kinds of devices are typically used in CAVEs, where input that registers 6DOF is required.

Composite devices

Wii Remote with attached strap

Input devices, such as buttons and joysticks, can be combined on a single physical device that could be thought of as a composite device. Many gaming devices have controllers like this. Technically mice are composite devices, as they both track movement and provide buttons for clicking, but composite devices are generally considered to have more than two different forms of input.

- Game controller
- Gamepad (or joypad)
- Paddle (game controller)
- Wii Remote

Output device

An **output device** is any piece of computer hardware equipment used to communicate the results of data processing carried out by an information processing system (such as a computer) to the outside world.

In computing, input/output, or I/O, refers to the communication between an information processing system (such as a computer), and the outside world. Inputs are the signals or data sent to the system, and outputs are the signals or data sent by the system to the outside.

Examples of output devices:

- Speaker
- Headphones
- Screen (Monitor)
- Printer

Computer software

Computer software, or just **software**, is the collection of computer programs and related data that provide the instructions telling a computer what to do. We can also say software refers to one or more computer programs and data held in the storage of the computer for some purposes. Program software performs the function of the program it implements, either by directly providing instructions to the computer hardware or by serving as input to another piece of software. The term was coined to contrast to the old term hardware (meaning physical devices). In contrast to hardware, software is intangible, meaning it "cannot be touched". Software is also sometimes used in a more narrow sense, meaning application software only. Sometimes the term includes data that has not traditionally been associated with computers, such as film, tapes, and records.

Examples of computer software include:

- Application software includes end-user applications of computers such as word processors or video games, and ERP software for groups of users.
- Middleware controls and co-ordinates distributed systems.
- Programming languages define the syntax and semantics of computer programs. For example, many mature banking applications were written in the COBOL language, originally invented in 1959. Newer applications are often written in more modern programming languages.

- System software includes operating systems, which govern computing resources. Today large applications running on remote machines such as Websites are considered to be system software, because the end-user interface is generally through a graphical user interface, such as a web browser.
- Testware is software for testing hardware or a software package.
- Firmware is low-level software often stored on electrically programmable memory devices. Firmware is given its name because it is treated like hardware and run ("executed") by other software programs.
- Shrinkware is the older name given to consumer-purchased software, because it was often sold in retail stores in a shrink-wrapped box.
- Device drivers control parts of computers such as disk drives, printers, CD drives, or computer monitors.
- Programming tools help conduct computing tasks in any category listed above. For programmers, these could be tools for debugging or reverse engineering older legacy systems in order to check source code compatibility.

History

The first theory about software was proposed by Alan Turing in his 1935 essay *Computable numbers with an application to the Entscheidungsproblem (Decision problem)*. The term "software" was first used in print by John W. Tukey in 1958. Colloquially, the term is often used to mean application software. In computer science and software engineering, software is all information processed by computer system, programs and data. The academic fields studying software are computer science and software engineering.

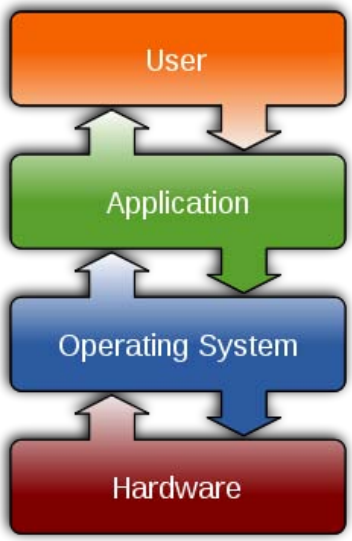
The history of computer software is most often traced back to the first software bug in 1946. As more and more programs enter the realm of firmware, and the hardware itself becomes smaller, cheaper and faster due to Moore's law, elements of computing first considered to be software, join the ranks of hardware. Most hardware companies today have more software programmers on the payroll than hardware designers, since software tools have automated many tasks of Printed circuit board engineers. Just like the Auto industry, the Software industry has grown from a few visionaries operating out of their garage with prototypes. Steve Jobs and Bill Gates were the Henry Ford and Louis Chevrolet of their times, who capitalized on ideas already commonly known before they started in the business. In the case of Software development, this moment is generally agreed to be the publication in the 1980s of the specifications for the IBM Personal Computer published by IBM employee Philip Don Estridge. Today his move would be seen as a type of crowd-sourcing.

Until that time, software was *bundled* with the hardware by Original equipment manufacturers (OEMs) such as Data General, Digital Equipment and IBM. When a customer bought a minicomputer, at that time the smallest computer on the market, the computer did not come with Pre-installed software, but needed to be installed by engineers employed by the OEM. Computer hardware companies not only bundled their software, they also placed demands on the location of the hardware in a refrigerated

space called a computer room. Most companies had their software on the books for 0 dollars, unable to claim it as an asset (this is similar to financing of popular music in those days). When Data General introduced the Data General Nova, a company called Digidyne wanted to use its RDOS operating system on its own hardware clone. Data General refused to license their software (which was hard to do, since it was on the books as a free asset), and claimed their "bundling rights". The Supreme Court set a precedent called Digidyne v. Data General in 1985. The Supreme Court let a 9th circuit decision stand, and Data General was eventually forced into licensing the Operating System software because it was ruled that restricting the license to only DG hardware was an illegal *tying arrangement*. Soon after, IBM 'published' its DOS source for free, and Microsoft was born. Unable to sustain the loss from lawyer's fees, Data General ended up being taken over by EMC Corporation. The Supreme Court decision made it possible to value software, and also purchase Software patents. The move by IBM was almost a protest at the time. Few in the industry believed that anyone would profit from it other than IBM (through free publicity). Microsoft and Apple were able to thus cash in on 'soft' products. It is hard to imagine today that people once felt that software was worthless without a machine. There are many successful companies today that sell only software products, though there are still many common software licensing problems due to the complexity of designs and poor documentation, leading to patent trolls.

With open software specifications and the possibility of software licensing, new opportunities arose for software tools that then became the de facto standard, such as DOS for operating systems, but also various proprietary word processing and spreadsheet programs. In a similar growth pattern, proprietary development methods became standard Software development methodology.

Overview



A layer structure showing where operating system is located on generally used software systems on desktops

Software includes all the various forms and roles that digitally stored *data* may have and play in a computer (or similar system), regardless of whether the data is used as *code* for a CPU, or other interpreter, or whether it represents other kinds of information. Software thus encompasses a wide array of products that may be developed using different techniques such as ordinary programming languages, scripting languages, microcode, or an FPGA configuration.

The types of software include web pages developed in languages and frameworks like HTML, PHP, Perl, JSP, ASP.NET, XML, and desktop applications like OpenOffice.org, Microsoft Word developed in languages like C, C++, Java, C#, or Smalltalk. Application software usually runs on an underlying software operating systems such as Linux or Microsoft Windows. Software (or firmware) is also used in video games and for the configurable parts of the logic systems of automobiles, televisions, and other consumer electronics.

Computer software is so called to distinguish it from computer hardware, which encompasses the physical interconnections and devices required to store and execute (or run) the software. At the lowest level, executable code consists of machine language instructions specific to an individual processor. A machine language consists of groups of binary values signifying processor instructions that change the state of the computer from its preceding state. Programs are an ordered sequence of instructions for changing the state of the computer in a particular sequence. It is usually written in high-level programming languages that are easier and more efficient for humans to use (closer to natural language) than machine language. High-level languages are compiled or interpreted into machine language object code. Software may also be written in an assembly language, essentially, a mnemonic representation of a machine language using a natural language alphabet. Assembly language must be assembled into object code via an assembler.

Types of software

Practical computer systems divide software systems into three major classes: system software, programming software and application software, although the distinction is arbitrary, and often blurred.

System software

System software provides the basic functions for computer usage and helps run the computer hardware and system. It includes a combination of the following:

- device drivers
- operating systems
- servers
- utilities
- window systems

System software is responsible for managing a variety of independent hardware components, so that they can work together harmoniously. Its purpose is to unburden the application software programmer from the often complex details of the particular computer being used, including such accessories as communications devices, printers, device readers, displays and keyboards, and also to partition the computer's resources such as memory and processor time in a safe and stable manner.

Programming software

Programming software usually provides tools to assist a programmer in writing computer programs, and software using different programming languages in a more convenient way. The tools include:

- compilers
- debuggers
- interpreters
- linkers
- text editors

An Integrated development environment (IDE) is a single application that attempts to manage all these functions.

Software topics

Architecture

Users often see things differently than programmers. People who use modern general purpose computers (as opposed to embedded systems, analog computers and supercomputers) usually see three layers of software performing a variety of tasks: platform, application, and user software.

- Platform software: Platform includes the firmware, device drivers, an operating system, and typically a graphical user interface which, in total, allow a user to interact with the computer and its peripherals (associated equipment). Platform software often comes bundled with the computer. On a PC you will usually have the ability to change the platform software.
- Application software: Application software or Applications are what most people think of when they think of software. Typical examples include office suites and video games. Application software is often purchased separately from computer hardware. Sometimes applications are bundled with the computer, but that does not change the fact that they run as independent applications. Applications are usually independent programs from the operating system, though they are often tailored for specific platforms. Most users think of compilers, databases, and other "system software" as applications.
- User-written software: End-user development tailors systems to meet users' specific needs. User software include spreadsheet templates and word processor

templates. Even email filters are a kind of user software. Users create this software themselves and often overlook how important it is. Depending on how competently the user-written software has been integrated into default application packages, many users may not be aware of the distinction between the original packages, and what has been added by co-workers.

Documentation

Most software has software documentation so that the end user can understand the program, what it does, and how to use it. Without clear documentation, software can be hard to use—especially if it is very specialized and relatively complex like Photoshop or AutoCAD.

Developer documentation may also exist, either with the code as comments and/or as separate files, detailing how the programs works and can be modified.

Library

An executable is almost always not sufficiently complete for direct execution. Software libraries include collections of functions and functionality that may be embedded in other applications. Operating systems include many standard Software libraries, and applications are often distributed with their own libraries.

Standard

Since software can be designed using many different programming languages and in many different operating systems and operating environments, software standard is needed so that different software can understand and exchange information between each other. For instance, an email sent from a Microsoft Outlook should be readable from Yahoo! Mail and vice versa.

Execution

Computer software has to be "loaded" into the computer's storage (such as the hard drive or memory). Once the software has loaded, the computer is able to *execute* the software. This involves passing instructions from the application software, through the system software, to the hardware which ultimately receives the instruction as machine code. Each instruction causes the computer to carry out an operation – moving data, carrying out a computation, or altering the control flow of instructions.

Data movement is typically from one place in memory to another. Sometimes it involves moving data between memory and registers which enable high-speed data access in the CPU. Moving data, especially large amounts of it, can be costly. So, this is sometimes avoided by using "pointers" to data instead. Computations include simple operations such as incrementing the value of a variable data element. More complex computations may involve many operations and data elements together.

Quality and reliability

Software quality is very important, especially for commercial and system software like Microsoft Office, Microsoft Windows and Linux. If software is faulty (buggy), it can delete a person's work, crash the computer and do other unexpected things. Faults and errors are called "bugs." Many bugs are discovered and eliminated (debugged) through software testing. However, software testing rarely – if ever – eliminates every bug; some programmers say that "every program has at least one more bug" (Lubarsky's Law). All major software companies, such as Microsoft, Novell and Sun Microsystems, have their own software testing departments with the specific goal of just testing. Software can be tested through unit testing, regression testing and other methods, which are done manually, or most commonly, automatically, since the amount of code to be tested can be quite large. For instance, NASA has extremely rigorous software testing procedures for many operating systems and communication functions. Many NASA based operations interact and identify each other through command programs called software. This enables many people who work at NASA to check and evaluate functional systems overall. Programs containing command software enable hardware engineering and system operations to function much easier together.

License

The software's license gives the user the right to use the software in the licensed environment. Some software comes with the license when purchased off the shelf, or an OEM license when bundled with hardware. Other software comes with a free software license, granting the recipient the rights to modify and redistribute the software. Software can also be in the form of freeware or shareware.

Patents

Software can be patented; however, software patents can be controversial in the software industry with many people holding different views about it. The controversy over software patents is that a specific algorithm or technique that the software has may not be duplicated by others and is considered an intellectual property and copyright infringement depending on the severity.

Design and implementation

Design and implementation of software varies depending on the complexity of the software. For instance, design and creation of Microsoft Word software will take much more time than designing and developing Microsoft Notepad because of the difference in functionalities in each one.

Software is usually designed and created (coded/written/programmed) in integrated development environments (IDE) like Eclipse, Emacs and Microsoft Visual Studio that can simplify the process and compile the program. As noted in different section, software is usually created on top of existing software and the application programming interface

(API) that the underlying software provides like GTK+, JavaBeans or Swing. Libraries (APIs) are categorized for different purposes. For instance, JavaBeans library is used for designing enterprise applications, Windows Forms library is used for designing graphical user interface (GUI) applications like Microsoft Word, and Windows Communication Foundation is used for designing web services. Underlying computer programming concepts like quicksort, hashtable, array, and binary tree can be useful to creating software. When a program is designed, it relies on the API. For instance, if a user is designing a Microsoft Windows desktop application, he/she might use the .NET Windows Forms library to design the desktop application and call its APIs like *Form1.Close()* and *Form1.Show()* to close or open the application and write the additional operations him/herself that it need to have. Without these APIs, the programmer needs to write these APIs him/herself. Companies like Sun Microsystems, Novell, and Microsoft provide their own APIs so that many applications are written using their software libraries that usually have numerous APIs in them.

Computer software has special economic characteristics that make its design, creation, and distribution different from most other economic goods. A person who creates software is called a programmer, software engineer, software developer, or code monkey, terms that all have a similar meaning.

Industry and organizations

A great variety of software companies and programmers in the world comprise a software industry. Software can be quite a profitable industry: Bill Gates, the founder of Microsoft was the richest person in the world in 2009 largely by selling the Microsoft Windows and Microsoft Office software products. The same goes for Larry Ellison, largely through his Oracle database software. Through time the software industry has become increasingly specialized.

Non-profit software organizations include the Free Software Foundation, GNU Project and Mozilla Foundation. Software standard organizations like the W3C, IETF develop software standards so that most software can interoperate through standards such as XML, HTML, HTTP or FTP.

Other well-known large software companies include Novell, SAP, Symantec, Adobe Systems, and Corel, while small companies often provide innovation.