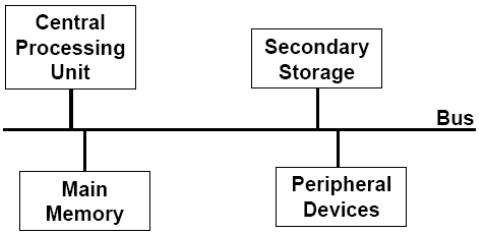
An Overview of Computer Hardware

Any study of operating systems requires a basic understanding of the components of a computer system.

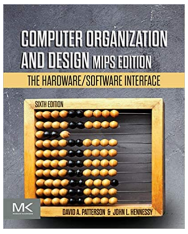
Although the variety of computer system configurations is forever changing, as (new) component types employ different standards for their interconnection, it is still feasible to discuss a simple computer model, and to discuss components' roles in operating systems.



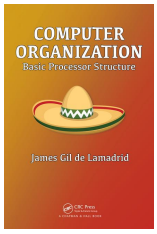
Traditionally, we consider four main structural components:

- The **Central Processing Unit, or CPU**, undertakes arithmetic and logical computation, and directs most input and output services from memory and peripherals. There may be multiple processors in a system, each executing the (same, single) operating system or user/application programs.
- **Main Memory, or RAM** (Random Access Memory) is used to store both instructions and data. Processors read and write items of memory both at the direction of programs (for data), and as an artifact of running programs (for instructions).
- **Secondary Storage** and **Peripheral Devices**, (or input/output modules) and their I/O controllers, move data to and from the other components usually to provide longer-term, persistent storage of data (disks, tapes),
- A **communications bus**, or system bus, connects the processor(s), main memory, and I/O devices together, providing a "highway" on which data may travel between the components. Typically only one component may control the bus at once, and *bus arbitration* decides which that will be.

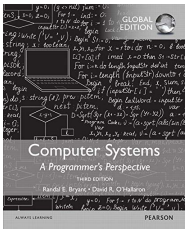
Excellent, albeit expensive, computer organisation texts



Computer Organization and Design (MIPS Edition): The Hardware/Software Interface, by David A. Patterson and John L. Hennessy. Morgan Kaufmann Publishers, 6th edition, 2021.



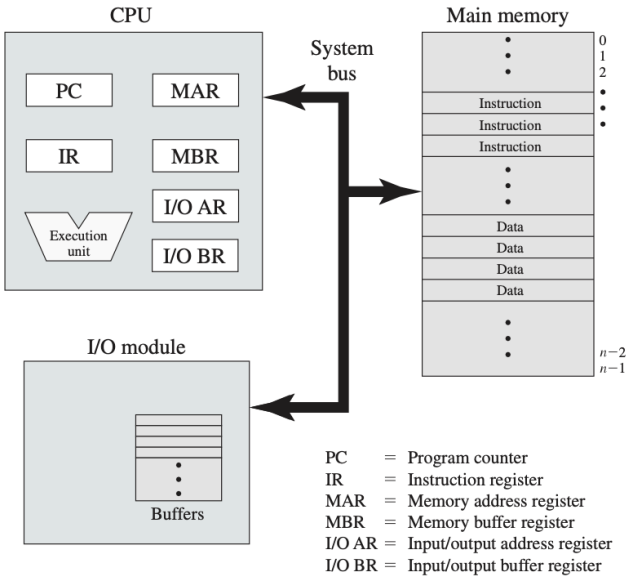
Computer Organization: Basic Processor Structure, James Gil de Lamadrid, Chapman and Hall/CRC, Published February 23, 2018, 372pp, ISBN 9781498799515.



Computer Systems: A Programmer's Perspective, Randal Bryant and David O'Hallaron, October 2015, 1120pp, ISBN 1292101768.

Basic Computer Components

Many OS textbooks (often in their 1st or 2nd chapters) outline a traditional computer model, in which the CPU, main memory, and I/O devices are all interconnected by a single system bus (figures are taken from Stallings' website).



Instruction and data fetching

- The CPU *fetches* a copy of the contents of uniquely-addressed memory locations, by identifying the required location in its MAR (Memory Address Register).
- Depending on *why* the CPU requested the memory's value, it executes the contents as an *instruction*, or operates on the contents as *data*.
- Similarly, the CPU locates data from, or for, distinct locations in the I/O devices using its I/O-AR (Address Register).

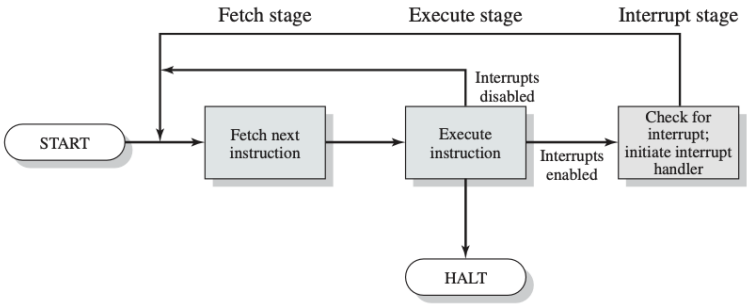
Role of operating systems

The role of the OS in managing the flow of data to and from its CPU and I/O devices, made very challenging by the wide variety of devices.

The OS attempts to attain *maximum throughput* of its computation and data transfer.

Processor scheduling attempts to keep the (expensive) processor busy at all times, by *interleaving* computation and communication.

While *waiting* for a slow device to complete its I/O transfer, the CPU may be able to undertake other activities, such as performing some computation or managing faster I/O.



Processor Registers

As well as special logic to perform arithmetic and logic functions, the processor houses a small number of very fast memory locations, termed *processor registers*.

- Data in registers can be read and written very rapidly (with a typical access time of 0.5-3ns). If the required data is available in registers, rather than main memory, program execution may proceed 10-500X faster.
- Different types of processors have varying number of registers, For example, some processors have very few (3-16), some have many (32-100s).
- The number of general-purpose CPU registers, and the *width* of each register (measured in bits, e.g. 64-bit registers), contribute to the power and speed of a CPU.
- Processors place constraints on how some registers are used. Some processors expect certain types of data to reside in specific registers. For example, some registers may be expected to hold integer operands for integer arithmetic instructions, whereas some registers may be reserved for holding floating-point data.

The Role of Processor Registers

All data to be processed by the CPU must first be copied into registers - the CPU cannot, for example, add together two integers residing in RAM.

Data must first be copied into registers; the operation (e.g. addition) is then performed on the registers and the result left in a register, and that result (possibly) copied back to RAM.

Registers are also often used to hold a *memory address*, and the register's contents used to indicate which item from RAM to fetch.

The transfer of data to and from registers is completely transparent to users (even to programmers).

Generally, we only employ *assembly language programs* to manipulate registers directly. In compiled high-level languages, such as C, the compiler translates high-level operations into low-level operations that access registers.

Register types

Registers are generally of two types:

User-accessible registers -
are accessible to programs, and may usually be read from and written to under program control. Programs written in an assembly language, either by a human programmer or generated by a compiler for a high-level language, are able to read and write these registers using specific instructions understood by the processor which usually accept the names of the registers as operands.

The user-accessible registers are further of two types:

- **Data registers** hold values before the execution of certain instructions, and hold the results after executing certain instructions.
- **Address registers** hold the addresses (not contents) of memory locations used in the execution of a program, e.g.
  - *the memory address register (MAR)* holds the address of memory to be read or written;
  - *the memory buffer register (MBR)* holds the memory's data just read, or just about to be written;
  - *index registers* hold an integer offset from which of memory references are made; and
  - *a stack pointer (SP)* holds the address of a dedicated portion of memory holding temporary data and other memory addresses.

Control and status registers -
hold data that the processor itself maintains in order to execute programs, e.g. the *instruction register*(IR) holds the current instruction being executed, and the *program counter* (PC) holds the memory address of the*next* instruction to be executed.

Special registers reflect the *status of the processor.* The *processor status word* (PSW) reflects whether or not the processor may be interrupted by I/O devices and whether privileged instructions may be executed, and it uses *condition bits* to reflect the status of recently executed operations.

In order evaluate results, and to determine if branching should occur, the PSW may record -

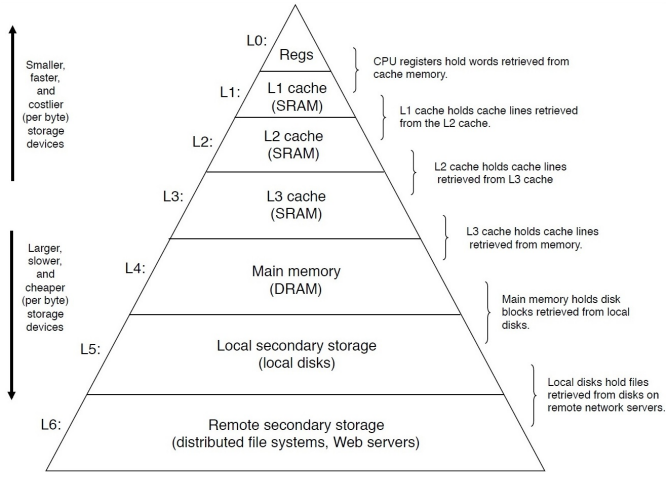
- whether an arithmetic operation overflowed,
- whether an arithmetic operation performed a carry,
- whether a division by zero was attempted,
- whether the last comparison instruction succeeded or failed.

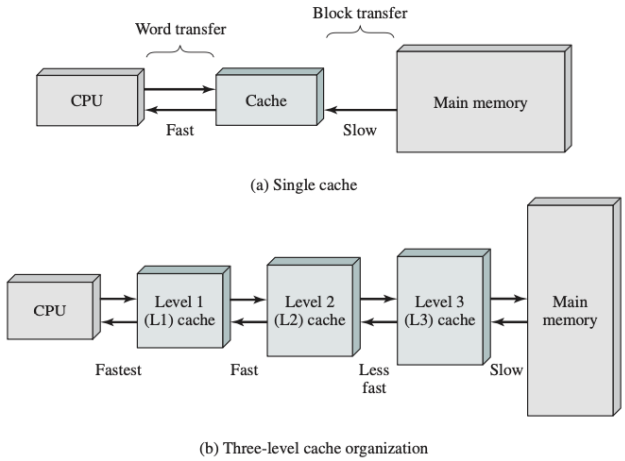
The Memory Hierarchy

The role of memory is to hold instructions and data until they are requested by the processor (or, some devices). While it is easy to make a case for as much memory as possible, having too much can be wasteful (financially) if it is not all required.

We also expect memory to be able to provide the necessary data, as quickly as possible, when called upon. Unfortunately, there is a traditional trade-off between cost, capacity, and access time:

- the faster the access time, the greater the cost per bit,
- the greater the capacity, the smaller the cost per bit and, the greater the capacity, the slower the access time.

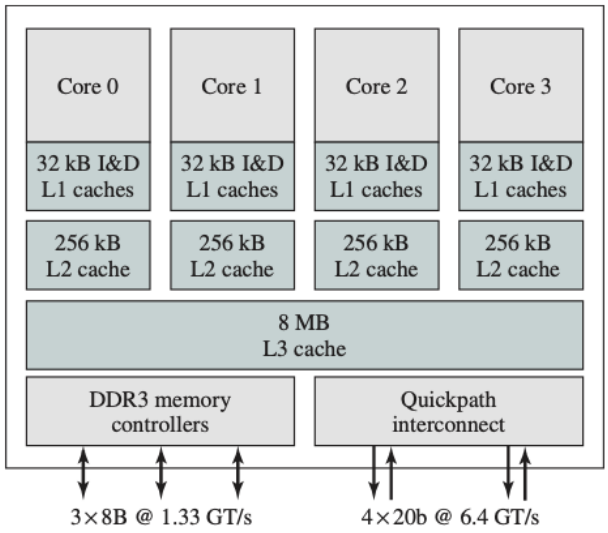




The Memory Hierarchy, *continued*

The solution taken is not to rely on a single, consistent form of memory, but instead to have a *memory hierarchy*, constrained by requirements and cost.

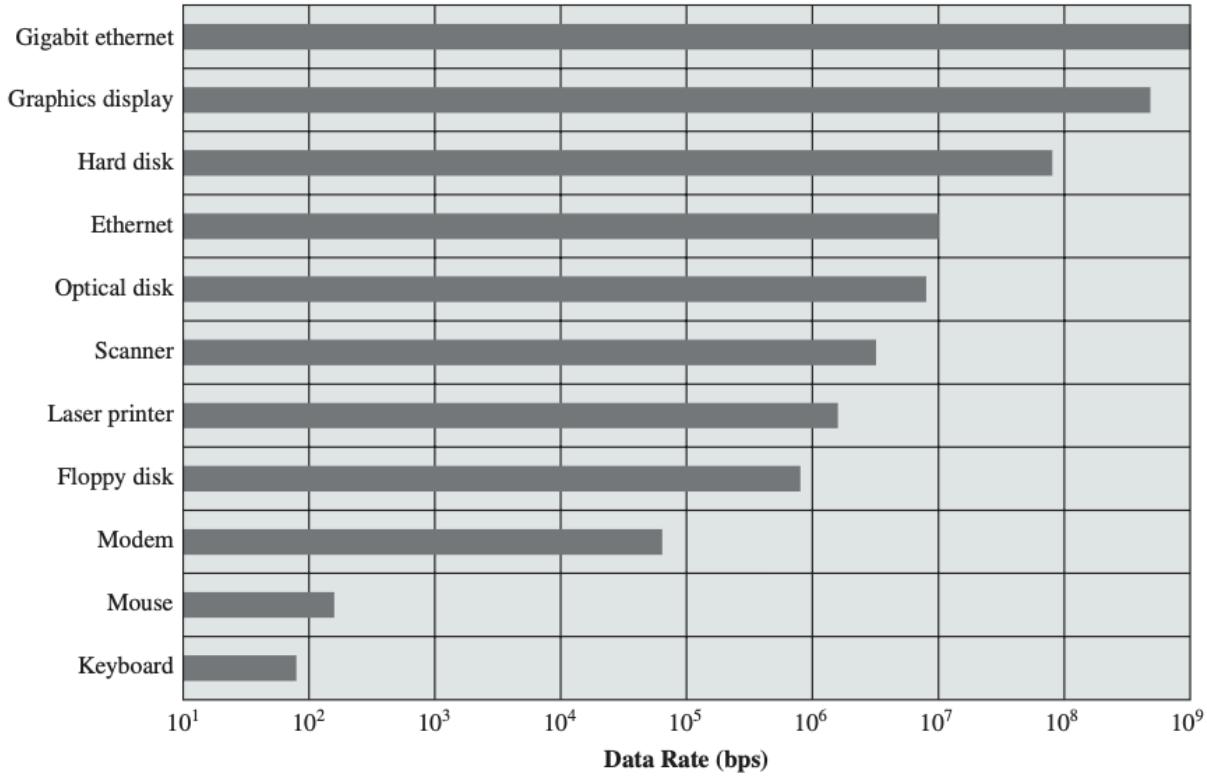
Memory	Access-time	Capacity	Technology	Managed by
Registers	0.5-3ns	1-4KB	custom CMOS	compiler
Level-1 cache (on-chip)	0.4-4ns	8KB-256KB	SRAM	hardware
Level-2 cache (on-chip)	4-8ns	256KB-8MB	SRAM	hardware
Level-3 cache	6-16ns	4MB-64MB	SRAM	hardware
Main memory (RAM)	10-60ns	64MB-128GB	DRAM	operating system
hard disk	3M-10M ns	128MB-24,000GB	magnetic	operating system
solid-state disk (SSD)	0.5M-1M ns	16GB-18,000GB	DRAM/SRAM	operating system



For example, a contemporary laptop or home computer system may include:

- a modest amount of cache memory (1MB) to deliver data as quickly as possible to the processor,
- a larger main memory (8GB) to store entire programs and less-frequently required data, and
- long term, persistent storage in the form of a hard disk (1TB), or SSD (256GB).

The Range of I/O Device Data Rates



See also Wikipedia's List of interface bit rates.

Units of data: bits, bytes, and words

- The basic building block is the **bit** (binary digit), which can contain a single piece of binary data (true/false, zero/one, high/low, etc.).

  Although processors provide instructions to set and compare single bits, it is rarely the most efficient method of manipulating data.

- Bits are organised into larger groupings to store values encoded in binary bits. The most basic grouping is the **byte**: the smallest normally *addressable* quantum of main memory (which can be different from the minimum amount of memory fetched at one time).

  In modern computers, a byte is almost always an 8-bit byte, but history has seen computers with 7-, 8-, 9-, 12-, and 16-bit bytes.

- A **word** is the default data size for a processor. The word size is chosen by the processor's designer and reflects some basic hardware issues (such as the width of internal or external buses).

  The most common word sizes are 32 and 64 bits; historically words have ranged from 16 to 60 bits.

- It is very common to speak of a processor's **wordsize**, such as a 32-bit or 64-bit processor.
  However, different sources will confuse whether this means the size of a single addressable memory location, or the default unit of integer arithmetic.

  Some processors require that data be *aligned*, that is, 2-byte quantities must start on byte addresses that are multiples of two; 4-byte quantities must start on byte addresses that are multiples of four; etc.

  Some processors allow data to be *unaligned*, but this can result in slower execution as the processor may have to align the data itself.

On the interpretation of data

We have seen that computer systems store their data as bits, and group bits together as bytes and words.

However, it is important to realise that the processor can interpret a sequence of bits only in context: on its own, a sequence of bits means nothing.

A single 32-bit pattern could refer to:

- 4 ASCII characters,
- a 32-bit integer,
- 2 x 16-bit integers,
- 1 floating point value,
- the address of a memory location, or
- an instruction to be executed.

No *meaning* is stored along with each bit pattern: it is up to the processor to apply some *context* to the sequence to ascribe it some meaning.

For example, a sequence of integers may form a sequence of valid processor instructions that could be meaningfully executed; a sequence of processor instructions can always be interpreted as a vector of, say, integers and can thus be added together.

Critical errors occur when a bit sequence is interpreted in the wrong context. If a processor attempts to execute a meaningless sequence of instructions, a *processor fault* will generally result: Linux announces this as a "bus error". Similar faults occur when instructions expect data on aligned data boundaries, but are presented with unaligned addresses.

On the interpretation of data, *continued*

As an example of how bytes may be interpreted in different ways, consider the first few hundred bytes of the disk file */bin/ls*. We know this to be a program, and we expect the operating system to interpret its contents to be a program, and request the processor to execute its contents (a mixture of instructions and data).

However, another program could read the bytes from */bin/ls* and interpret them in other ways, e.g. as 32-bit integers:

```
prompt> od -i /bin/ls
0000000  1179403647       65793           0           0
0000020      196610           1   134518416          52
0000040       66628           0     2097204     2621447
0000060     1638426           6          52   134512692
0000100   134512692         224         224           5
0000120           4           3         276   134512916
0000140   134512916          19          19           4
0000160           1           1           0   134512640
.......
```

or as octal (8-bit) bytes:

```
prompt> od -b /bin/ls
0000000 177 105 114 106 001 001 001 000 000 000 000 000 000 000 000 000
0000020 002 000 003 000 001 000 000 000 220 226 004 010 064 000 000 000
0000040 104 004 001 000 000 000 000 000 064 000 040 000 007 000 050 000
0000060 032 000 031 000 006 000 000 000 064 000 000 000 064 200 004 010
0000100 064 200 004 010 340 000 000 000 340 000 000 000 005 000 000 000
0000120 004 000 000 000 003 000 000 000 024 001 000 000 024 201 004 010
0000140 024 201 004 010 023 000 000 000 023 000 000 000 004 000 000 000
0000160 001 000 000 000 001 000 000 000 000 000 000 000 000 200 004 010
.......
```

or as ASCII characters:

```
prompt> od -c /bin/ls
0000000 177   E   L   F 001 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020 002  \0 003  \0 001  \0  \0  \0 220 226 004  \b   4  \0  \0  \0
0000040   D 004 001  \0  \0  \0  \0  \0   4  \0      \0  \a  \0   (  \0
0000060 032  \0 031  \0 006  \0  \0  \0   4  \0  \0  \0   4 200 004  \b
0000100   4 200 004  \b 340  \0  \0  \0 340  \0  \0  \0 005  \0  \0  \0
0000120 004  \0  \0  \0 003  \0  \0  \0 024 001  \0  \0 024 201 004  \b
0000140 024 201 004  \b 023  \0  \0  \0 023  \0  \0  \0 004  \0  \0  \0
0000160 001  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0  \0 200 004  \b
.......
```

And each interpretation could be correct, depending on context.