Notes on Computer Systems: A Programmer's Perspective by Bryant & O'Hallaron

February 22, 2019 — March 25, 2019

Contents

0.1 A Tour of Computer Systems

- Traces the lifetime of a `hello.cpp` program from creation, runtime, output, through to termination.

```
#include <stdio.h>

int main()
{
    printf(``hello, world\n'');
}
```

0.1.1 Information is Bits + Context

- ASCII – most common representation for text characters. Each character is represented by a byte-sized integer
- ``All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits.'' (pg. 3)

0.1.2 Programs are Translated by Other Programs into Different Forms

- Individual programs (such as in C) are translated by *compilers* into machine–language instructions, stored as binary files (also called an *executable*).
- Four different phases of this process (using a C program as an example):
  - *Preprocessing*: Modifies the original code according to directives begining with the # character (such as #include statements) and outputs a `.i` file.
  - *Compilation phase*: Translates the preprocessed code into an *assembly–language* program (a `.s` file). Important since this is a common output language for all compilers for all high–level languages (e.g. C and Fortran compilers output programs in the same assembly–language)
  - *Assembly*: The assembly–language program is translated into *machine–language* instructions, called a *relocatable program* (`.o` file)
  - *Linking*: Links the program to external libraries (such as the C standard library) to output an *executable object file*.

0.1.3 It Pays to Understand How Compilation Systems Work

- Gives several reasons why it's important to understand how compilers work:
  - **Optimizing program performance**. Optimization of code relies on understanding how a compiler turns certain statements into machine–language (i.e. `switch` statements vs `if--else` statements)
  - **Understanding link–time errors**. What to certain link errors mean, how can we resolve them?
  - **Avoiding security holes**. Buffer overflow is one of the main source of vulnerabilities, so understanding data types and structures is important

0.1.4 Processors Read and Interpret Instructions Stored in Memory

- At this point in our example, our program is compiled into an executable.
- How do we run it? What happens to it when we do?
- The shell is a base way humans interact with a computer. Run programs by typing `./$PROGRAM`, where `$PROGRAM` is the name of the executable.

Hardware Organization of a System

- Many different, interacting parts go into a modern computer system (Figure 0.1.4 on the following page).
- This subsection goes through each of the subsystems in turn.

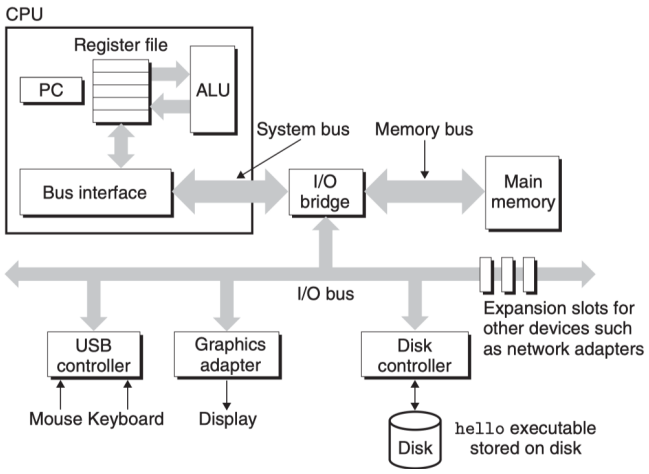

Figure 1: Organization of a modern computer system (pg. 8).

Buses

- Each fixed–size chunk of bytes carried by a bus is called a *word*, with a size of 4 bytes in 32 bit machines and 8 bytes in 64 bit machines (most modern machines are 64 bit)

I/O Devices

- Connect the system to the external world (e.g. keyboard, mouse, etc.)
- Each I/O device is connected to the I/O bus using some type of controller, which are chips on the device or motherboard.

Main Memory

- Volatile (i.e. temporary) memory (Random Access Memory (RAM)) which holds programs and the data they manipulate.
- Linear arrays of bytes, each with their own index (address), starting at index 0.
- The size of the data items stored in memory depend on their declared type (int, float, etc.) in whatever program it is that the memory belongs to.

Processor

- The Central Processing Unit (CPU) executes instructions stored in main memory.
- Contains a word–sized storage device (also called a register) called a Program Counter (PC), which points at an address in main memory containing some machine–language instructions.
- Reads the instruction from memory, interprets the bits, executes the instruction, then updates the PC to point to the next instruction (which may or may not be contiguous in memory)
- Only performs a few types of operations, revolving around the main memory, the register file, and the arithmetic/logic unit (ALU):
  - Load: Copy a byte or word from main memory into a register.
  - Store: Copy a byte or word from a register into main memory.
  - Operate: Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words, store the result in a register.
  - Jump: Extract a word from the instruction itself and copy it to the PC

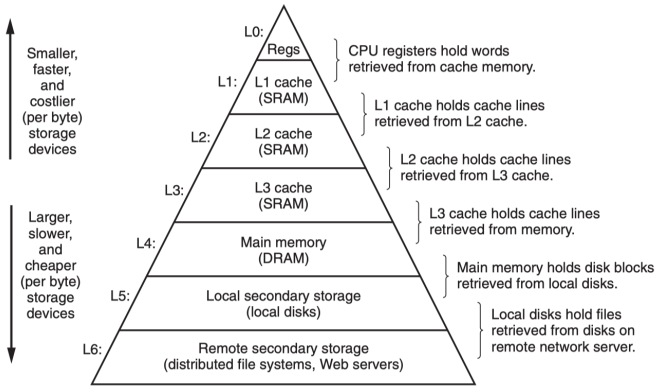

Figure 2: Example of a memory hierarchy.

Running the `hello` Program

- While typing `./hello` in the shell, each character is read into the register and stored in memory.
- The shell then loads the executable into memory from the disk, without passing through the processor by going through the memory bus.
- Once loaded into memory, the processor begins executing the machine–language instructions within the compiled code (some good figures on pages 11 and 12 showing each step of this process mapped out on copies of Figure 0.1.4 on the previous page).

Caches Matter

- IMPORTANT: ‘‘[...] a system spends a lot of its time moving information from one place to another.’’ (pg. 12)
- This information movement overhead slows down parts of the program which depend on it.
- Larger storage devices are slower than smaller storage devices.
- Register files store only a few hundred bytes of data while main memory holds billions, but a processor can read data from a register file almost 100 times faster than from main memory (called the *processor–memory gap*).
- To deal with this, there is a hierarchy of caches (see Figure 0.1.4 for more) which serve as temporary staging areas for information.
- Information can be staged in this cache hierarchy so the processor can more quickly access data.

Storage Devices Form a Hierarchy

- Storage devices of all type in a computer form a *memory hierarchy* (Figure 0.1.4).
- As you move up the hierarchy, memory becomes faster to access at a higher monetary cost.
- The storage at one level acts as a cache for the storage one rung below it.

0.1.5 The Operating System Manages the Hardware

- Note that when outlining running the program (Section 0.1.4) the program itself never accessed the keyboard, display, disk, or main memory.
- This is instead done by the computer’s operating system (OS).
- Operating systems have two main purposes:
  - Protect the hardware from runaway programs.

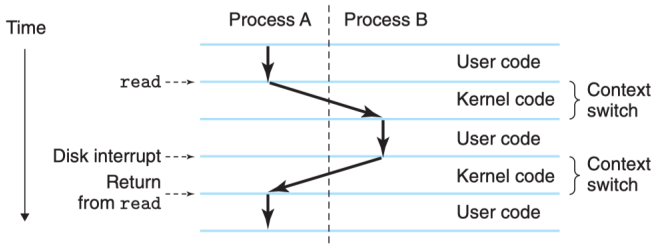

Figure 3: Minimal example of context switching between two concurrent processes.

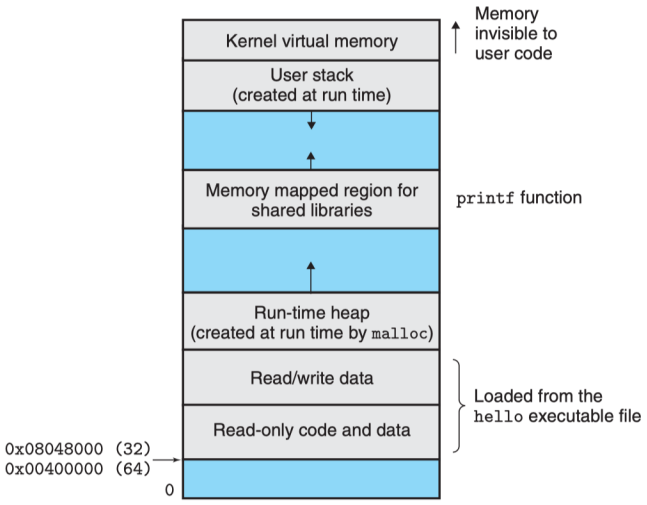

Figure 4: Virtual address space hierarchy.

  - Provide higher–level, simple, and uniform mechanisms for interacting with low–level hardware.

- An operating system has three fundamental abstractions: processes, virtual memory, and files.

Processes

- A process is the operating system's abstraction for a running program.
- A single CPU can run multiple processes concurrently by switching between them (called context switching). The CPU keeps track of all the information about a running processes, such as the register file, its data in main memory, etc., which is called a *context*. The CPU can then switch between processes by storing one context, switching, and restoring the context (Figure 0.1.5).
- A processes can contain multiple execution units called *threads*, each running in the context of its umbrella processes and sharing the same code and data.
- IMPORTANT: It is much easier to share data between multiple threads than it is between multiple processes (i.e. it is much easier to parallelize code in a multithreaded manner than in a multiprocessor manner).

Virtual Memory

- Abstraction that gives the illusion that each process has exclusive access to memory (see virtual memory)
- Each process has the same view of memory, called the *virtual address space*
- Hierarchical: topmost for code and data common to all processes, lower region for code and data specific to user created processes (Figure 0.1.5)

- Well–defined spaces of memory space (* denotes important concepts, which I've heard of before):
  - **Program code and data**: Code begins at same address for all processes, followed by data that correspond to global C variables, initialized directly from the executable.
  - **Heap***: Initialized at run–time. Expands and contracts as a result of memory allocation calls in whatever program it is that's running (e.g. C/Cpp's `malloc`).
  - **Shared libraries**: Holds the code and data for shared libraries (e.g. C/Cpp's STL)
  - **Stack***: Area where compiler implements function calls of a program. Expands with each function call and contracts when returned from the function.
  - **Kernel virtual memory**: The part of the operating system which permanently resides in memory.

Files

- ''A *file* is a sequence of bytes, nothing more and nothing less.'' (pg. 19)

0.1.6 Concurrency and Parallelism

- Parallelism refers to using concurrency to make code faster (i.e. running multiple, simultaneous activities/processes)
- Three levels of abstraction:

Thread–Level Concurrency

- The concept of using threads to run multiple control flows executing within a single processes.
- Multi–core processor: Processors with multiple cores per chip.
  - These cores have their own L1 and L2 cache, but share higher level caches and interfaces to main memory.
- Multi–threaded processor: Processors which allow multiple flows of control per core.
  - Each core has multiple copies of some CPU hardware, such as registers and PCs, while sharing interfaces with other parts like ALUs.
  - Multithreaded processors decide which thread to execute on a cycle–per–cycle basis.
- Improves system performance in two ways:
  - Reduces the need for single cores to simulate concurrency by context switching.
  - Can run single applications faster, iff **the program is written such that it can utilize multiple threads effectively.**

Instruction–Level Parallelism

- Refers to the ability of a processor to execute multiple instructions at once.
- Each individual instruction can take longer, but a processor can execute 100s of instructions simultaneously
- Pipelining: when actions required to execute instructions are partitioned into discrete steps
- Most modern processors are *superscalar*, meaning they have execution rates faster than one instruction per cycle

Single–Instruction, Multiple–Data (SMID) Parallelism

- Allows single instructions to trigger multiple operations run in parallel.
- An example of this is the ability of modern processors to add four pairs of single–precision floating–point numbers at once

Part I

Program Structure and Execution

Chapter 1

Representing and Manipulating Information

- Computer process information as binary signals called bits
- Individual bits are meaningless, sets of bits have meaning.
- Three important representations of numbers:
  - **Unsigned**: Based on traditional binary notation, represent numbers greater than or equal to 0.
  - **Two's–complement**: Represent signed real numbers.
  - **Floating–point**: Base–two representation of scientific notation, for real numbers.
- Limited numbers of bits are used to represent numbers. Going over this can cause an overflow
- **IMPORTANT**: Floating–point arithmetic is not associative, as precision is not infinite.
  - Example: (3.14 + 1E20) – 1E20 will most likely evaluate to 0.0 on most machines, whereas 3.14 + (1E20 – 1E20) usually evaluates to 3.14.
  - Integers encode a narrow range of numbers precisely, floating–point numbers encode a wide range approximately.

1.1 Information Storage

- Information is stored in eight bit blocks called a byte. A program views memory as a large array of bytes, called its virtual memory
- Every byte is identified by a unique number called an address, the set of which compose virtual address space.
- All allocation and management of memory at the program level is done in virtual address space.
  - Languages associate type information (e.g. `int`) with each byte of memory it uses, but the machine–level program the languages generate have no type information themselves.

1.1.1 Hexadecimal Notation

- A value of a byte ranges from 00000000_2 to 11111111_2 in binary notation and from 0_{10} to 255_{10} in decimal integer notation.
- Writing numbers in binary notation is tedious, use base–16 numbers, called hexadecimal
  - A value of a single byte ranges from 00_{16} to FF_{16}
  - A hex number is denoted by a starting string, either `0x` or `0X`
- A lot of stuff about interconverting between them, but I won't replicate it here (pp. 34-37)

C/C++type	32–bit	64–bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char*`	4	8
`float`	4	4
`double`	8	8

Table 1.1: Type sizes in bytes for various C/C++types

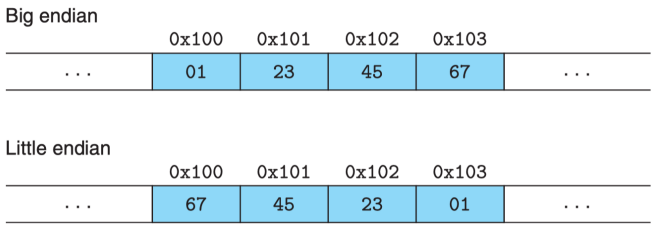

Figure 1.1: Different conventions for bit ordering.

1.1.2 Words

- Computers have different word sizes, which indicate the nominal size of integer and pointer data.
- Puts a maximum limit on virtual address space: a machine with a word size of w has virtual addresses on the interval $[0, 2^w - 1]$, giving a program access to 2^w bytes
  - This is why 32–bit machines are capped at 4 GB of RAM ($2^{32} \simeq 4.3 \times 10^9$ bytes)

1.1.3 Data Sizes

- Machines have instructions for manipulating blocks of memory on 1, 2, 4, and 8 byte intervals.
- Language types have different byte sizes (see table 1.1.3 for C/C++type sizes)

1.1.4 Addressing and Byte Ordering

- Two conventions must be established to make things consistent:
  - What the address of an object will be
  - How bytes are ordered in memory
- A multi–byte object is stored as a contiguous sequence of bytes
  - The object's address is the smallest address of the bytes it spans
- Assume we have a w–bit object with the bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit and x_0 is the least.
  - If w is a byte (multiple of 8), the bits can be grouped as bytes, with $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$ being the most significant byte and $[x_7, x_6, \ldots, x_{w-0}]$ being the least
  - Two conventions fr ordering bytes (Figure 1.1.4):
    - **Little endian**: The least significant byte is listed first
    - **Big endian**: The most significant byte is listed first
  - Most modern processors are *bi-endian*
  - Bit ordering is usually unimportant, but there are several cases where it is critical to know which one is being used:

* Binary data transfer between two machines over a network
* When looking at machine–level code (such as decompiled C/C++code)
* When circumventing a language's typing, e.g. using `cast` in C/C++

1.1.5 Representing Strings

- In C, a string is represented as an array of characters terminated by a null character, `0`
- Text is more platform independent than binary data

Representing Code

- Different machine types have different and incompatible instruction codings
  - For the function below, Table 1.1.5 show the different machine–level encodings for some architectures

```
int sum(int x, int y)
{
   return x + y;
}
```

Linux 32	`55 89 e5 8b 45 0c 03 45 08 c9 c3`
Linux 64	`55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3`
Sun	`81 c3 e0 08 90 02 00 09`
Windows	`55 89 e5 8b 45 0c 03 45 08 5d c3`

Table 1.2: Different machines have different character encodings.

- **IMPORTANT**: The computer has no information about the original program it's running, everything is just bits to it.

Introduction to Boolean Algebra

- Most of this is just review for me, so I won't discuss a lot of it

Operation	Math Symbol	Meaning
~	$\neg$	not
&	$\wedge$	and
|	$\vee$	or
^	$\oplus$	exclusive–or (xor)

Table 1.3: Boolean symbols and their meaning

- The boolean operations can also operate on *bit vectors*, strings of binary numbers of a fixed length w
  - Example: let a and b be the bit vectors $[a_{w-1}, a_{w-2}, \ldots, a_0]$ and $[b_{w-1}, b_{w-2}, \ldots, b_0]$
  - $a\&b$ is a bit vector of length w where the ith element is $a_i \& b_i$
- Goes on to describe some bitwise and logical operations, which I will not replicate.

1.2 Integer Representations

- Describes two different ways bits can be used to encode integers, unsigned and signed.

C/C++ Data Type	Minimum	Maximum
`char`	-127	127
`unsigned char`	0	255
`short int`	-32 767	32 767
`unsigned short int`	0	65 535
`int`	-32 767	32 767
`unsigned int`	0	65 636
`long int`	-2 147 483 647	2 147 483 647
`unsigned long int`	0	4 294 967 295
`long long int`	-9 223 372 036 854 775 807	9 223 372 036 854 775 807
`unsigned long long int`	0	18 446 744 073 709 551 615

Table 1.4: Guaranteed ranges of C/C++ integral types.

1.2.1 Integral Data Types

- *Integral data types*: types which represent finite ranges of integers. Table 1.2.1 shows the guaranteed range of integral data types (guaranteed on both 32– and 64–bit systems)
  - The guaranteed ranges differ from the ''typical ranges'' (shown on pg. 57), in that several data types have different maximums (notably `unsigned long int`), and the symmetric range of the negative and positive extrema

Unsigned Encodings

- Assume we have an integer data type of w bits and a bit vector, $\vec{x}$, $[x_{w-1}, x_{w-1}, \ldots, x_0]$
  - If we treat $\vec{x}$ as a number written in binary, this is the unsigned encoding interpretation of $\vec{x}$, expressed as

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

    where $B2U_w$ means ''binary to unsigned'' of length w and ''$\doteq$'' means ''defined as''.
  - This can be thought of as a mapping of sets of ones and zeros to nonnegative integers.
    - For example,

$$B2U_4([0001]) = (0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 0 + 0 + 0 + 1 = 1$$
$$B2U_4([0101]) = (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 0 + 4 + 0 + 1 = 5$$

- The unsigned binary representation is important in that every number in its interval, $[0, 2^{w-1}]$, has a unique encoding as a w–bit value.
  - This means the function $B2U_w$ is a *bijection*; every bit vector of length w is associated with a unique value

Two's–Complement Encodings

- Two's–complement encoding is the common computer representation of signed numbers
  - Defined by interpreting the most significant bit of the word to have a negative weight,

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

  - This negative weight bit is called the *sign bit*. The encoded number is negative when the sign bit is set to 1, and positive when it is set to 0.
  - Below show two example maps

$$B2U_4([0101]) = (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 0 + 4 + 0 + 1 = 5$$
$$B2U_4([1011]) = (-1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = -8 + 0 + 2 + 1 = -5$$

    * Note that the mapping for 5 and −5 are **not** symmetric.

- What is the range of values this method can encode?
  - The smallest representable number is the bit vector $[10\cdots0]$, with an integer value of $T_{\min,w} \doteq -2^{w-1}$, and the largest representable number is the bit vector $[01\cdots1]$, with an integer value of $T_{\max,w} \doteq \sum_{i=0}^{w-2} 2^1 = 2^{w-1} - 1$
  - With a 4–bit vector, $T_{\min,w} = -8$ and $T_{\max,w} = 7$
  - $\therefore B2T_w : \{0,1\}^w \to \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$
  - As with the unsigned encoding, $B2T_w$ is a bijection (each bit vector has a unique encoding)

Conversions Between Signed and Unsigned Encodings

- **IMPORTANT**: C/C++handles casting on the bit–level: casting between signed and unsigned encodings keeps the bit values identical but changes how they are interpreted.
  - Suppose we have the bit vector $[1011]$.
    * In two's–complement, the first bit, the sign bit, is interpreted as −8, whereas in an unsigned encoding it is interpreted as +8. So the negative values in two's–complement are increased by $2^4 = 16$ in an unsigned encoding (−5 becomes +11).
- When mapping a signed number to its unsigned counterpart, negative numbers are converted to large positive numbers and positive numbers remain unchanged. When going from unsigned to signed, large numbers ($> 2^{w-1}$, the maximum value for signed numbers) go to negative values and numbers $< 2^{w-1}$ remain unchanged (Figure **??** on page ??).

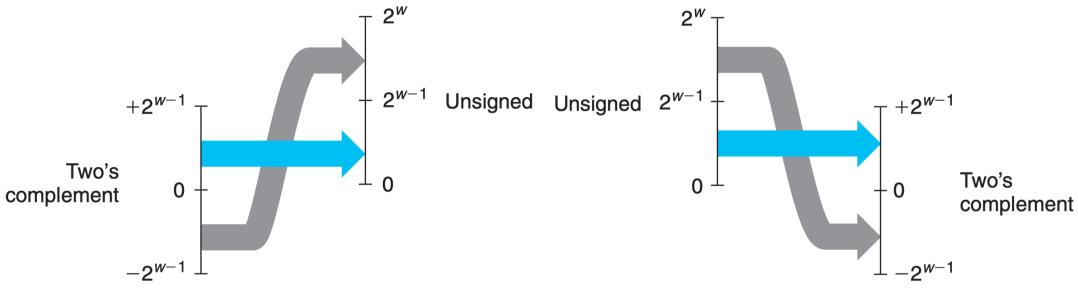

Figure 1.2: Interconversions between signed and unsigned numbers.

Glossary

ALU

Arithmetic/Logic Unit. Computes new data and address values.

assembly–language

A common language to which compilers translate code.

bus

Electrical conduits that carry bytes of information between other components in a computer.

cache

Temporary staging area for data which a processor might need in the near future. Between the register and main memory. Often in a hierarchy, from L1, which can hold 10000+ bytes and be read almost as fast as a register, to L2, which can hold 100k—1m bytes of memory but is slower to access than the L1 cache (though still faster than main memory). Newer machines can also have an L3 cache.

compiler

Program which translates ASCII code into a machine–language executable (e.g. g++).

concurrency

General concept of a system with multiple, simultaneous activities.

context switching

The way in which a CPU switches between multiple processes running concurrently.

CPU

Central Processing Unit. The ‘‘brain’’ of the computer, which executes instructions stored in main memory.

DRAM

Dynamic Random Access Memory. Refers to the memory type of main memory (i.e. sticks of RAM)

heap

Area of the user's virtual address space which expands and contracts with each call to memory allocation and freeing functions in the program that's running (e.g. C/C++'s `malloc` and `free`)

machine–language

Binary language in which executables are written (as translated from ASCII by compilers).

multi–core processor

A processor which has multiple CPUs, called cores, on a single chip. Most modern processors are multi–core.

multi–threaded processor

A processor which can run multiple threads per core. Most Intel Core i7 and all AMD Ryzen processors are multi–threaded.

OS

Operating System. Software interposed between the computer's hardware and the programs which run on it. Examples are Windows and macOS.

parallelism

The concept of using concurrency (multiple, simultaneous activities) to make a system or program run faster.

PC

Program Counter. Also called a *register*. Word–sized storage device pointing to an address in memory containing machine–language instructions.

pipelining

Process in which a processor partitions a instruction into discrete steps which it can handle in stages.

process

An operating system's abstraction for a running program. Multiple processes can be run concurrently.

RAM

Random Access Memory. Volatile memory which stores currently running programs and the data they manipulate.

register file

Small storage device consisting of word–sized registers.

SMID

Single–Instruction, Multiple–Data. Concurrency abstraction where modern processors can have instructions perform multiple operations in parallel.

SRAM

Static Random Access Memory. Refers to the memory type of CPU caches (L1 and L2).

stack

Area of the user's virtual address space which expands and contracts with each function call and return during the execution of a program.

virtual memory

An abstraction that provides each process with the illusion that it has exclusive access to main memory.

word

Fixed–sized chunks of bytes (4 bytes in 32 bit systems, 8 bytes in 64 bit systems) carried along buses to different components in a computer.