Computer Hardware

CPU

The CPU is an integrated circuit (IC) consisting of millions of transistors, and microscopic wires to connect them. To create a CPU, the IC design is etched onto a wafer of silicon called a die. Error-free dies are cut and mounted in a package, with the die's pads connected to the package pins, allowing direct connection with other components within the PC.

Multi core processors are ICs with multiple independent CPUs manufactured onto them, with shared main memory.

Moore's law -> Number of transistors in an IC doubles every 18-24 months.

CPU microarchitecture components:

- Datapath - performs data processing operations and includes the ALU and registers.
- Control - Tells the datapath, memory and I/O devices what to do - acts as a bridge between the datapath and memory.
- Cache - small, fast, expensive on-chip memory used to store memory items that need to be accessed regularly

Memory

Memory is a set of locations into which data can be stored. Each location has a unique address and stores 1 byte of data. Words can be formed by dealing with multiple bytes at a time - a word is the amount of data the processor can handle at once. Different computer architectures have different word sizes.

Memory types —>

The cost and performance of memory is generally proportional to its physical distance from the CPU.

- The **hard disk** usually consists of rapidly rotating disks with magnetic heads (although solid state drives, which have no moving parts, are becoming more common). Hard disks are used for secondary storage and tend to be cheap but relatively slow.
- **Random Access Memory** (**RAM**) takes the form of an integrated circuit and comes in two basic types.
  - **Dynamic RAM** (**DRAM**) is where a bit of data is stored using a transistor/capacitor combination. It is relatively cheap although it needs to be refreshed (because capacitors tend to 'leak') and is slow.
  - **Static RAM** (**SRAM**) is where a bit of data is stored by a **flip-flop**, which incorporates 4-6 transistors. It is stable and fast although takes up more space.
- **Caches** are expensive memory that are used to store rapidly accessed items. They are generally organised hierarchically into levels. A **level 1 cache** usually consists of static RAM and is built into the CPU. A **level 2 cache** again usually consists of static RAM and might be either built into the CPU or be at a short physical distance away.

Buses

A bus is a communication device that can be thought of as a series of parallel wires through which data can travel. There are 3 main buses within a processor architecture: the address bus, the data bus and the control bus:

- The **address bus** holds addresses of locations in main memory. The width of the address bus determines the size of addressable memory. For example, an address bus that is 32 bits wide can carry at most 2^{32} different addresses. So, as each memory address holds 1 byte then addressable memory has size 4 Gbytes.

- The **data bus** carries the contents of memory locations. The width of the data bus determines the word-size of the computer. Note that although each memory location holds 1 byte of data, a computer with a word-size of 32 bits, for example, can be considered to hold 32 bits of data in each location, for the contents of some 'location' are considered to consist of the contents of 4 contiguous bytes of memory.

- The **control bus** is used to transfer information between the CPU and various other devices within the processor. Typical information includes signals saying whether data is being read or written and detailing information transmitted from input/output devices.

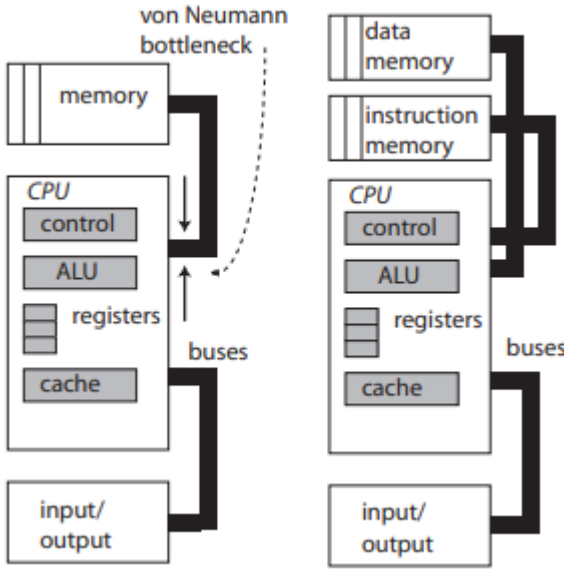
↑ not that sort of bus

Von Neumann Architecture

Von Neumann architecture is a fundamental computer architecture, the idea behind it being that the stored programs and the data are both held in the same memory, allowing for self modifying programs. This can however lead to a bottleneck - data and instructions have to be fetched in sequential order so the CPU has to spend time idle while waiting for the data to be fetched in order for the instruction to be executed. This bottleneck resulted in the introduction of cache.

Harvard Architecture

Harvard architecture uses separate memories for data and instructions with dedicated buses, allowing for data and instructions to be fetched simultaneously, eliminating the von Neumann bottleneck. Modern computer architectures use a modified form of Harvard architecture, where the instructions and data are kept in separate memories with dedicated buses, but the processor allows instructions to be treated as data and vice versa to allow for self modifying programs.

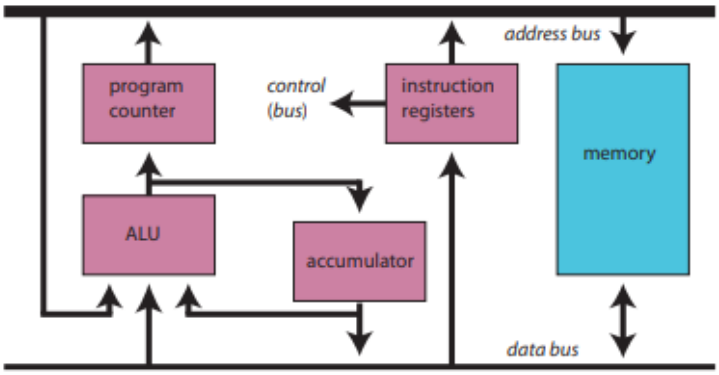


Von Neumann vs Harvard architecture

Fetch-etc. Cycle

Fetch-Decode-Fetch-Execute-Writeback Cycle

Every time we get taught about this cycle they keep adding bits to it…. Started off as the fetch-execute cycle, then decode got added at some point and now its got so much going on that if I put the full name as the title of this page it wouldn't even come close to fitting.



Instruction fetch

The first phase of the cycle involves sending the address of the instruction to memory, and the instruction at that address being returned to the CPU.

The CPU puts the value stored within the program counter (which is the address of the next instruction to be executed) onto the address bus. This value is read by memory and the contents of that memory location (i.e. the instruction) are put onto the data bus and sent to the instruction registers in the CPU.

Instruction Decode

The instruction word stored in the instruction registers is decoded by the internal logic to provide control signals to the ALU to tell it what to do to execute this instruction. The value currently stored in the PC is pushed onto the address bus, where the ALU can read it and increment this value by the word size and write it back to the PC, so the PC now contains the next instruction address.

Operand Fetch

The instruction registers provide the address of the data (operand) to be processed to the address bus. The data is supplied to the data bus and sent to the CPU, ready for processing by the ALU.

Execute

Processing is performed by the ALU on the operand according to the instruction. If there is a result (i.e. the result of an addition operation) then this is put into the accumulator. If the instruction is a branch instruction, then the PC would have been updated to the address of the next instruction being branched to.

Write-back

The result from the accumulator is written back to memory via the data bus if necessary.

Instruction Set

Instruction Set Architecture (ISA)

The ISA (which is in no way comparable to a swan) provides an interface between hardware and software and includes everything programmers need in order to directly control the processor. The primary component of the ISA is its assembly language, which consists of low level instructions that directly translate to machine code. The ISA also includes —>

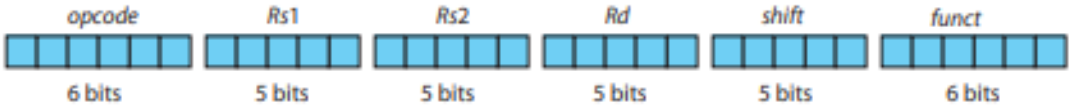
- the organisation and structure of programmable storage (namely the registers and main memory that is useable by the programmer);
- the instruction sets and formats (namely the processor instructions available to the user in order to access and manipulate data residing in memory);
- the methods for addressing and accessing data items within memory, e.g., absolute ('go to this location'), indirect ('go to the location stored in this location'), using offsets ('go to this location plus this offset number'); and
- the process of execution, e.g., the fetch-decode-fetch-execute cycle.

MIPS

MIPS is an example of an ISA. In MIPS, there are 2^{32} memory locations and the word length is 4 bytes (meaning address bus and control bus are both 32 bits wide). There are also 32 bit registers, including $zero which always stores 0 and some general purpose registers $s1, …, $s7.

All instructions are 32 bits wide, and there are 3 types of instructions: R-Type, I-Type and J-Type.

R-Type (**R**egister instructions) have the following format in machine code:

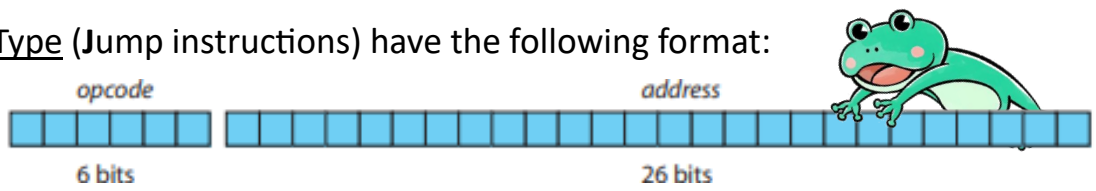


Where Rs1, Rs2 are source registers, Rd is the destination register, shift is the shift amount used by shift instructions (otherwise 0) and funct is a supplement for the opcode.

I-Type (**I**mmediate instructions) have the following format:

Where Rs is the source register, Rd is the destination register and address is a memory address or a constant (immediate), depending on the instruction.

J-Type (**J**ump instructions) have the following format:



Where address is a memory address.

(Note - J-Type instructions generally do not come with a frog included)

Operating Systems

What on 🌍 is an Operating System?

An operating system provides the interface between applications (software) and hardware. It manages the machine's resources (memory, processor cycles, I/O etc) so they are used efficiently and safely.

The operating system is needed as there are generally more processes running at a time than there are processors in the computer.

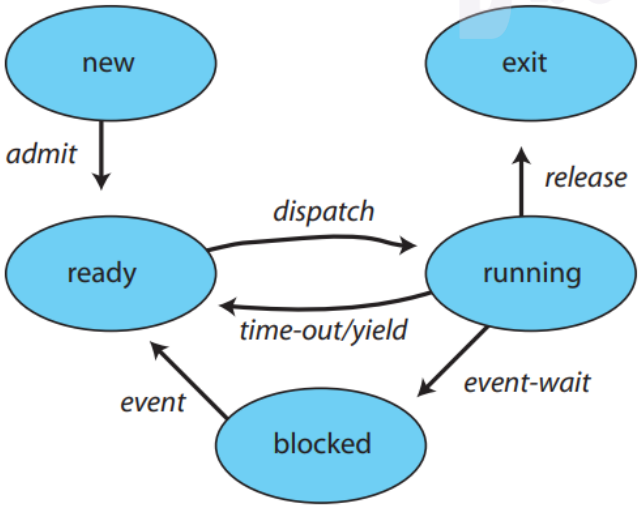
The OS provides an abstraction of the hardware for use by software so that all access to hardware is through the OS. The OS accesses the hardware through ISA instructions and system calls (used to organise programs' access to memory, and to ensure data security, which prevents programs from accessing memory locations allocated to other programs)

Functions of an OS

- The operating system **virtualizes** a machine. Hardware has low-level physical resources with complicated, idiosyncratic interfaces. An operating system provides abstractions that present clean interfaces so as to make the computer easier to use. For example, whilst the hard disk is a mechanical beast consisting of sectors and tracks, the operating system presents it as a well-organised file store. Also, virtualization means that the operating system can make a physical machine behave like an idealised virtual machine so as to aid portability. For example, Unix runs on many very different computer systems and programs can be ported across systems with very little effort.
- The operating system starts and stops programs. For example, it ensures that when a program is stopped, the memory allocated to this program is freed up for use by other programs and the processor is allocated to some other task (which is presumably in some queue waiting for processor access).
- The operating system manages memory. To see what we mean by this, consider compiling some C program. The compiled program will require memory so that it might run. However, it may be that the memory required is already being used by some currently executing program. The operating system will ensure that the compiled C program can still run by using memory that the compiled program *thinks* is the memory requested whereas in reality the memory actually *used* is different.
- The operating system handles input/output. Whilst programs are running, users are typing on keyboards, outputs are being sent to screens and so on. The operating system ensures that all of these activities are coordinated. For example, an operating system handles **interrupts** which might be caused, for example, by pressing some keyboard key and result in the execution of some program being paused or aborted.
- The operating system maintains and provides access to the file system. It ensures that programs and users only have access to their own files and not those of others, and that users can access their files in a well-structured way.
- The operating system deals with networking. For example, it continuously monitors the network for incoming traffic and sends data packets as output when requested by some program.
- The operating system maintains security through firewalls, and tries to ensure that executing programs don't have access to parts of the system from which they can cause malicious behaviour.
- The operating system facilitates error handling and recovery when programs don't do what they are supposed to do. It tries to provide some limitation as to the potentially disastrous effects of badly written or maliciously written programs.

Processes

Process Life Cycle



- **new**: the process is being created;
- **ready**: the process is not executing on the CPU but is ready to execute;
- **running**: the process is executing on the CPU;
- **blocked**: the process is waiting for an event (and so not executable); and
- **exit**: the process has finished.

There are the following state transitions, named according to the action:

- **admit**: the control overheads as regards a process have been set up and the process is moved to the run queue;
- **dispatch**: the scheduler allocates the CPU to an executable process (the **scheduler** is the component of the operating system that determines which processes should be run and when);
- **timeout/yield**: the executing process is forced to/volunteers to release access to the CPU;
- **event-wait**: a process is waiting for an input/output event, for example, and gives up access to the CPU;
- **event**: an event occurs and wakes up a process; and
- **release**: a process terminates and releases access to the CPU (and other resources).

Process Control Block (PCB)

The PCB for a process contains:

- the process's unique ID;
- the current state of the process (new, exit, ready, etc.);
- CPU scheduling information such as the process priority, whether there are events pending and so on;
- the program counter, detailing the next instruction to be executed (this is only available for processes that are not running, having been paused for some reason);
- the current values of other CPU registers (again, this is only available for processes that are not running, having been paused for some reason);
- memory management information, such as the (current) location of the address space;
- scheduling and accounting information, such as when the process was last run and the amount of CPU time that has elapsed so far; and
- a reference to the next and previous PCB in the list of PCBs.

When the OS switches between processes A and B (context switching), the state of the CPU registers etc are saved into PCB A, and the state from PCB B is restored so process B can begin executing from where it left off last time.

Mutual Exclusion

A process consists of one or more threads (sequences of instructions executed sequentially) and an address space of memory locations that the process (i.e. any of the threads) can read from or write to. When there are multiple threads, issues can arise when two threads have access to the same shared memory location at the same time. Segments of code that can be executed by multiple concurrent threads which could lead to conflicting changes to shared variables are called critical sections, and mutual exclusion is used to prevent two threads from executing code within the critical section at a time to prevent conflicts such as —>

c has value 0
thread 1 reads the value of counter *c*, namely 0
thread 1 increments its value, to 1
thread 2 reads the value of counter *c*, namely 0
thread 1 writes its value back to *c*
so now c has value 1
thread 2 increments its value, to 1
thread 2 writes its value back to *c*
so now c has value 1