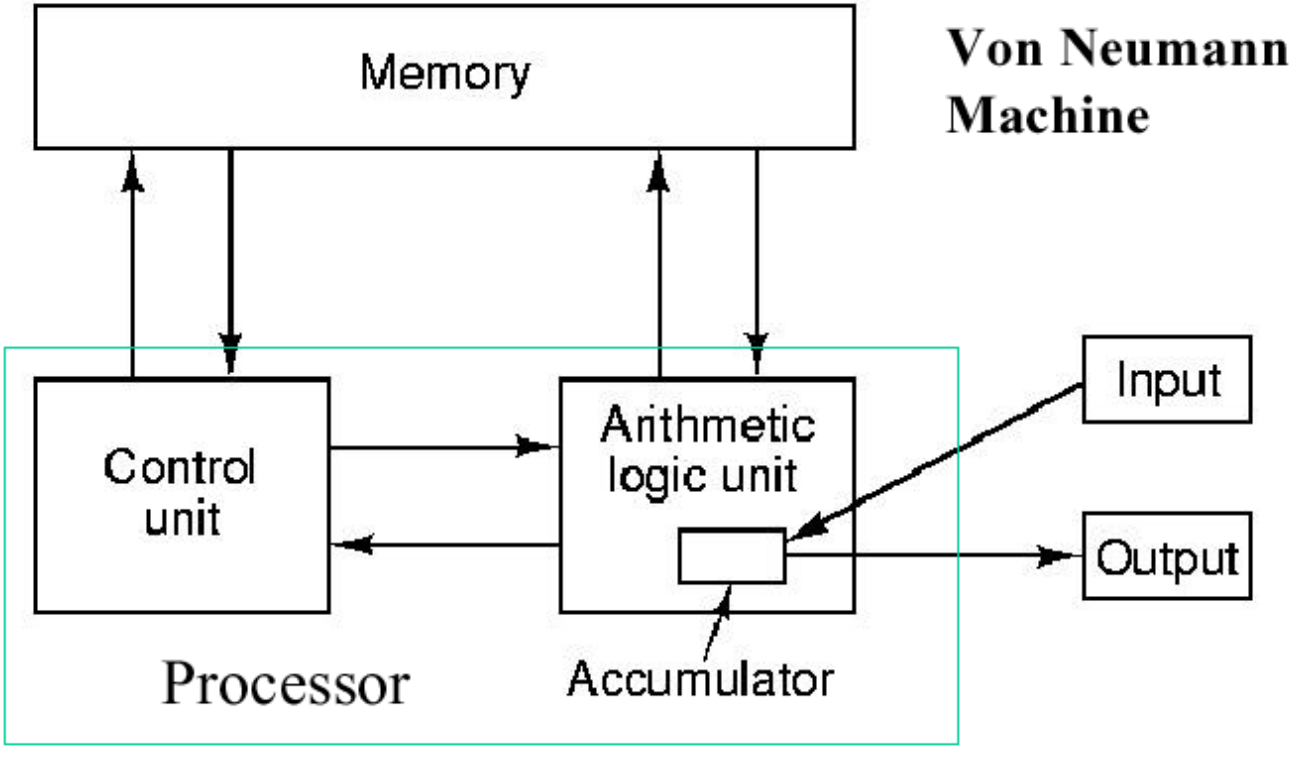
Computer Architecture

The basic composition of current computers looks like this (von Neumann machine):



Source: M.S.Schmalz

This is a simplified picture

Cache

Cache is (auxilliary) memory that can be faster accessed than the principal memory.

Typically, a programm will access in sequence either *adjacent* memory locations or even the *same* memory locations again.

On the other hand, different storage media have different capacities but also different access times:

One can classify memory (or data sources) by capacity but also by access speed:

- Capacity: How much data can be stored (without replacing media).
- Transfer Rate: How much data can be transferred per time unit.
- Latency: How long does it take for data to arrive initially.

  Example: Train carriage filled with hard disks.

In Processor: It is expensive to have external parts run at the processor frequency. Typically the access rate here is substantially lower. (Factor 10-20: 2.4GHz Processor/133MHz Bus)

It therefore makes sense to have (comparatively) low capacity memory which can be accessed fast.

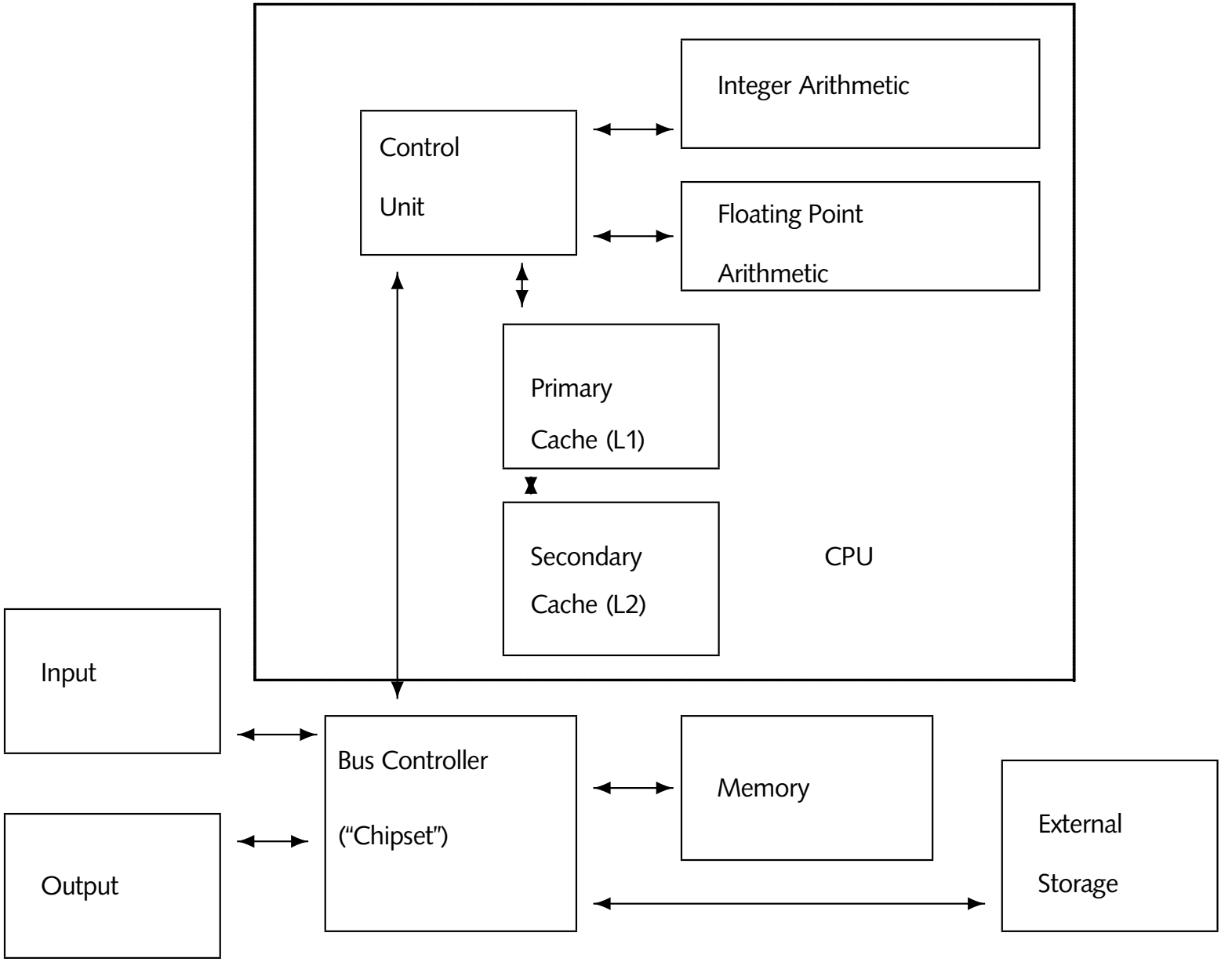
Typically memory access go “through the cache”:

- If the data is in the cache (hit) use it.
- If the data is not in the cache (miss) fill the cache (or some part of it) with the “block” of data that contains the demanded data.
- Advantage: Transfer of large amount of data occurs once in chunk: Wait only once for latency.
- Different strategies on how to transfer data back in the main memory.

The same idea applies to mass storage (hard disks) where the crucial problem is latency: Often the disk has a cache that will hold the contents of a whole track, even if only one sector is read. Here cache access time only has to compare with the access speed to the disk.

Modern processors also often keep an instruction cache which holds in memory the next few operations. In some cases it is thus possible to execute independent subsequent commands in parallel.

Together, we get the following structure for a modern computer:



Cache Misses

Since cache access is faster (probably by a magnitude), programs performance can suffer if memory is accessed in a way that causes frequent cache misses.

A typical example is matrix access in the wrong storage order.

On the other hand reloading a cache (at cache miss) costs time and therefore it is sometimes advantageous not to load too much cache.

As a compromise, modern processors have two caches, a smaller "immediate" cache and a larger (level 2) cache.

Both caches are usually on the processor microchip, sometimes also an external cache exists.

Cache size can be a substantial difference between “expensive” and “cheap” machines:

Name	Processor	Speed	L1 Cache	L2 Cache
`stokes`	Sparc	150Mhz(?)	16kB	4MB
`schur`	P III	1GHz	16kB	256kB
`wielandt`	P4 XEON	2.4GHz	8kB	512kB

The P4 has a smaller L1 cache than its predecessors. On the other hand it uses faster RAM (RDRAM or DDRAM).

Example: ASUS P4PE Motherboard

- CPU

  Socket 478 for Intel Pentium 4/ Celeron up to 3GHz+

  On-die 512KB/256KB L2 Cache with full speed

  Intel Hyper-Threading Technology ready
- Chipset: Intel 82845PE MCH, Intel 82801DA ICH4
- Front Side Bus: 533/400 MHz
- Memory

  3 x 184-pin DIMM Sockets support max. 2GB PC2700/PC2100 (FSB533) or PC2100/PC1600 (FSB400) non-ECC DDR SDRAM memory
- Expansion Slots: 1 x AGP 4X (1.5V only), 6 x PCI
- IDE: 2 x UltraDMA 100 / 66 / 33

Cache Issues

If we access a memory area, the *stride* is the distance of sucessively accessed memory locations.

Stride 1: Successive locations.

Array column (in C): Stride n

If we access data with a stride larger than the cache size we typically encounter a cache miss every time.

Try to avoid this by better programming.

Small benchmarks give unrealistic performance.

If stride is not coprime to size of cache lines, there can be problems.

Hardware issue: memory cycle time — interleaving.

Performance

If we want to determine the performance of a particular program (or to compare different computers) we are interested in actual runtimes, not just in asymptotic results.

"Runtime" might mean or include:

- "Wall clock time".
- Processor time excluding time for starting the program &c.
- I/O time – time to read input and produce the output
- User interaction time – how long do *I* have to work, but I'm willing to keep the computer running while doing other work.

Under `Unix` we can use the `time` command to measure the runtime of a process:

```
/usr/bin/time myprogram myprogramparameters
  [bla bla bla]
real        10.1
user         5.2
sys          1.2
```

Note: The (t)c-shell has a `time` comand build in. If you use this shell and only type `time` you get this command. Its output differs slightly.

In-program timing

The `Unix time` function measures the whole programn run time. This includes loading the program and initialization phases.

In cases of short runtimes, this can give a strong bias or even distort the results beyond recognition.

Remedies:

- Run the main calculation (within a loop) several times inside the program and divide the runtime by the number of iterations.
- Get the system time inside the program and subtract timings to get the runtime for parts of the program. Caveat: System time is returned in a structure, not just a number.

Benchmarks

Benchmarks are standardized (sets of) programs that can be run on various architectures to compare their performance in certain areas. Typically benchmarks fall in on of the following categories:

- Toy benchmarks ("Landmark-Number") – Loops with some commands
- Real Programs (or calculation routines) on simulated or standardized data.
- Synthetic benchmarks: Collection of various tasks such as arithmetic, memory access, register access, I/O.
- User benchmark: How long does it take to perform a set task.
- If a benchmark consist of several tests, its results often are averaged (or weighted averaged) to obtain a single "performance number".

- Benchmarks often are not only processor but processor/compiler dependent. When translation to another language even coding fluency might influence.

Performance measures that are frequently used are:

MIPS Million instructions per second.
Depends on how powerful instructions are.

MFLOPS Floating point operations.
Drawback: Measures only float arithmetic, no difference of relative cost of addition or division.

Whetstone A synthetic benchmark with loops and arithmetic (both integer and float). Originally (1976) an ALGOL program. (Named after town in Leicestershire.)

Dhrystone Synthetic benchmark (1984 Ada, now mostly in C) program intended to be representative for system (integer)

programming. Based on published statistics on use of programming language features.

Due to small size only cache memory is tested. Modern compiler often can optimize much away, in particular in version 1.

(However, quoted MIPS numbers are often based on version 1).

TPC Benchmarks for databases: Access/Modify/Create records.

Sysmark/BapCo Synthetic Windows user/Performance benchmark.

Linpack Linear algebra, based on predecessor of LAPack. Mainly measures matrix/vector multiplication.

SPEC

"Standard Performance Evaluation Corporation" — `http://www.spec.org`. Produces sets of benchmark suites.

The aim is to have "standardized" benchmarks that permit to compare processor performance over a wide variety of tasks, languages and operatinmg systems.

Test results are submitted by system maufacturers and are accessible on the Web.

For example: `SPEC CPU`: Computer intensive arithmetic, versions for integers and for floating points.

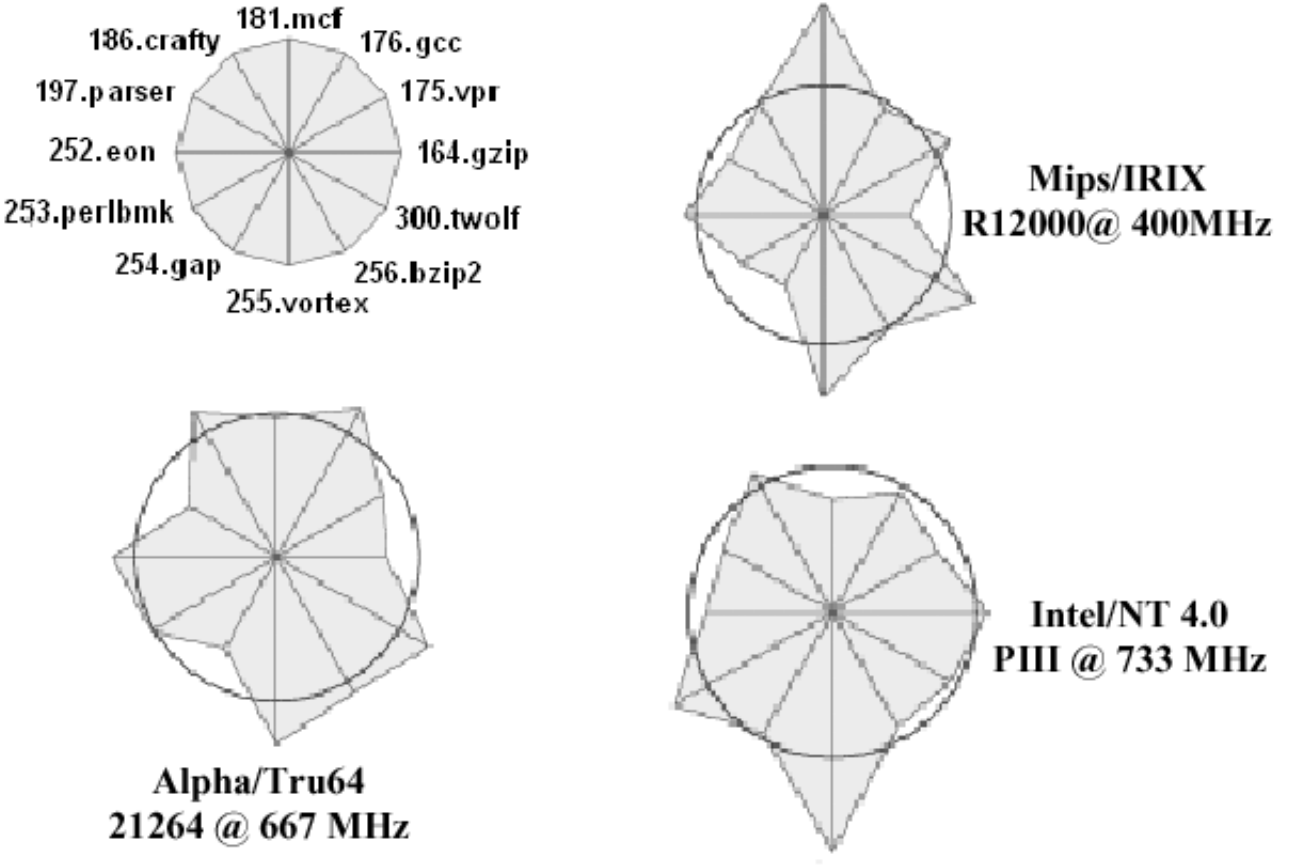
Tests are regularly updated to keep abreast of developments.

The programs in the SPEC CPU 2000 suite comprise:

164.gzip	(C, int)	Compression
175.vpr	(C, int)	FPGA circuit placement and routing
176.gcc	(C, int)	C programming language compiler
181.mcf	(C, int)	Combinatorial optimization
186.crafty	(C, int)	Chess game
197.parser	(C, int)	Word/language processing
252.eon	(C++, int)	Computer visualization
253.perlbmk	(C, int)	PERL programming language
254.gap	(C, int)	Group theory, interpreter
255.vortex	(C, int)	Object-oriented database
256.bzip2	(C, int)	Another compression program
300.twolf	(C, int)	Place and route simulator

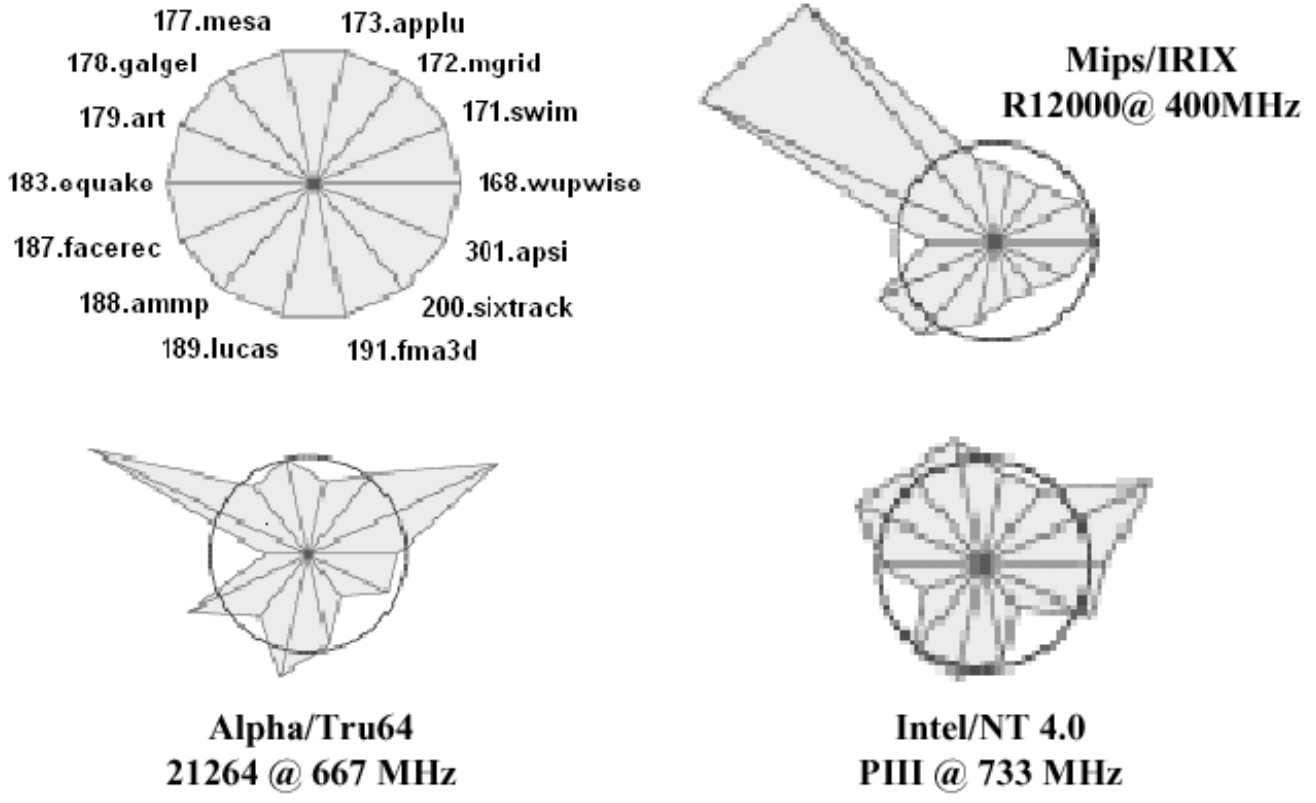
168.wupwise	(F77, float)	Physics/quantum chromodynamics
171.swim	(F77, float)	Shallow water circulation modelling
172.mgrid	(F77, float)	Multigrid solver: 3D potential field
173.applu	(F77, float)	Parabolic/elliptical PDEs
177.mesa	(C, float)	3D graphics library
178.galgel	(F90, float)	Computational fluid dynamics
179.art	(C, float)	Image recognition/neural networks
183.equake	(C, float)	Seismic wave propagation simulation
187.facerec	(F90, float)	Image processing: Face recognition
188.ammp	(C, float)	Computational chemistry
189.lucas	(F90, float)	Number theory/Primality testing
191.mfa3d	(F90, float)	Finite-element crash simulation
200.sixtrack	(F77, float)	High energy nuclear physics accelerator de
301.apsi	(F77, float)	Meteorology: Pollutant distribution

Performance of different tests might differ substantially on different processors (longer lines indicate better performance):



Source: E.Mafla

For float the difference is even more extreme:



Source: E.Mafla