## Några ord om hur datorer fungerar

A very simple model of a computer:


The CPU contains the ALU, arithmetic and logic unit and the control unit. The ALU performs operations such as +, -, *, / of integers and Boolean operations.

The control unit is responsible for fetching, decoding and executing instructions.

The memory stores programs and data.

I/O-devices are disks, keyboards etc.

The CPU contains several registers, such as:

- PC, program counter, contains the address of the next instruction to be executed
- IR, instruction register, the executing instruction
- address registers
- data registers

Filsystemet (filerna) ligger på (hård)-diskar.



Minnesbussen består normalt av en adressbuss och databuss. Databussen är kanske 64 "kablar" i bredd (man skickar alltså bitar parallellt och inte seriellt) och adressbussen 32.

Allt arbete i datorn sker i takt. Det finns flera klockor som håller takten, och olika delar kan gå olika snabbt. Minnesbussen brukar t.ex. gå i lägre takt än CPUn.

Om det står 2 GHz Pentium 4 så är CPUn en Pentium 4 som tillverkas av Intel och processorklockan tickar 2 miljarder tick per sekund. En klockcykel är tiden mellan två tick. Man säger att klockfrekvensen är 2 GHz. Bussen mot minnet kanske håller minst 266 MHz. Låt nu en klockcykel svara mot en sekund

| 1 clockcykel (2 GHz) | en sekund |
|---|---|
| minnet | $\mathcal{O}$(10 s) |
| disk | två månader |
| tangentbord 10 nedslag/s | 1.5 år |

---

Moderna datorer har RISC CPUer:

RISC - Reduced Instruction Set Computer
(ersatte CISC - Complex Instruction Set Computer)

1980, David Patterson, Berkeley, RISC-I, RISC-II
1981, John Hennessy, Stanford, MIPS
$\approx$ 1986, commercial processors

A processor whose design is based on the rapid execution of a sequence of simple instructions rather than on the provision of a large variety of complex instructions.



## Operativsystemet (the operating system) OS

Operativsystemet är den uppsättning program som bland annat ansvarar för följande:

- filsystemet (`cp`, `mv`, `rm`)
- processhantering (program som kör i datorn)
- resursfördelning (om t.ex. flera användare på samma dator)
- minnes-tilldelning, minnes-skydd
- kontakten med perifera enheter (diskar, skrivare etc.)
- kommunikation med andra datorer och processer
- kommunikation med användaren

Några vanliga operativsystem:

Unix, olika varianter (linux, irix, aix, solaris, hpux, ...)
MacOS (Macintosh Operating System)
MS-DOS (Microsoft Disk Operating System), PC
Windows XP, 2000, NT, 95, 98, PC

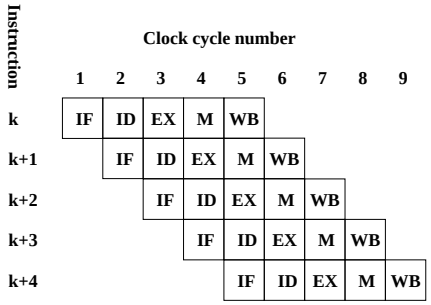
Fönstersystemet, X11 Window System, kompilatorer etc.
intar en mellanställning och räknas inte nödvändigtvis till OS.

Under unix så är kommandon, som `ls`, kompilerade C-program (eller kommandofiler). När man ger kommandot `ls` läser shellet, `tcsh`, detta och ser till att exekvera (via systemanrop) filen `/binl/ls` som är det kompilerade `ls.c`-programmet (svarar mot `a.out`).



## Pipelining - performing a task in several steps

Analogi: löpande band (en instruktion utförs i flera steg, stages)




CPUerna kan dessutom arbeta på flera instruktioner parallellt, man säger att de är superskalära.

Om vi inriktar oss på flyttalsberäkningar så brukar FPUn (Floating Point Unit) i CPUn kunna utföra en addition/subtraktion och en multiplikation parallellt.

Dessa operationer är dessutom pipelinade (dålig svenska), så om man kan hålla FPUn sysselsatt så kan man få en summa och en produkt per klockcykel. Observera dock att själva additionen (multiplikationen) tar mer än en klockcykel. Division är normalt inte pipelinad och brukar ta runt 20 klockcykler.

Det finns FPUer som kan göra två + och två * färdiga per klockcykel. Det finns dessutom (bla vissa Intel-CPUer) som har en vektor-dator som kan arbeta parallellt på korta vektorer (2-4 element) SSE (Streaming SIMD Extensions); fins olika versioner.

flop = floating point operation.
flops = plural of flop or flop / second.

Top floating point speed =
number of flops / clock cycle × clock frequency

Why do we often only get a small percentage of these speeds?
Is it possible to reach the top speed (and how)?
Example on a 167 MHz Sun; top speed 334 Mflops:

```
Instr.   fl. p. registers

fmuld    %f4,%f2,%f6              faddd    %f4,%f2,%f6
faddd    %f8,%f10,%f12            faddd    %f8,%f10,%f12
fmuld    %f26,%f28,%f30           faddd    %f4,%f2,%f6
faddd    %f14,%f16,%f18           faddd    %f8,%f10,%f12
fmuld    %f4,%f2,%f6              faddd    %f4,%f2,%f6
faddd    %f8,%f10,%f12            faddd    %f8,%f10,%f12
                          ...

331.6 Mflops                      166.1 Mflops
```

---

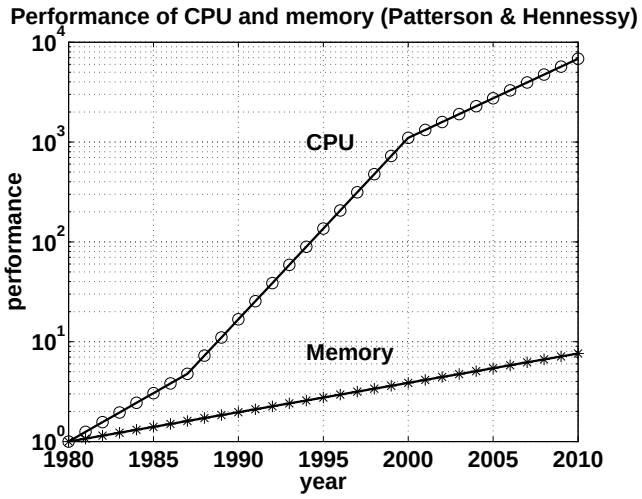
```
fdivd    %f4,%f2,%f6              fdivd    %f4,%f2,%f6
faddd    %f8,%f10,%f12            fdivd    %f8,%f10,%f12
fdivd    %f4,%f2,%f6              fdivd    %f4,%f2,%f6
faddd    %f8,%f10,%f12            fdivd    %f8,%f10,%f12
                          ...
15.1 Mflops                       7.5 Mflops
                                22 clockcykler för en fdivd
```

So, the answer is sometimes

- provided we have a suitable instruction mix and that
- we do not access memory too often



Memory is the problem - caches

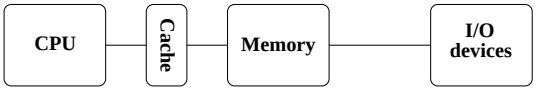



CPU: increase 1.25 improvement/year until 1986,
and a 1.52 improvement/year until 2004 and a 1.20
improvement/year thereafter.
DRAM (dynamic random access memory) 1.07 improvement/year.



DRAM slow and cheap.

Use SRAM (static random access memory) fast & expensive for cache.




SRAM will retain a value as long as power is supplied, unlike DRAM which must be regularly refreshed. It is however, still volatile, i.e. it will lose its contents when the power is switched off, in contrast to ROM.

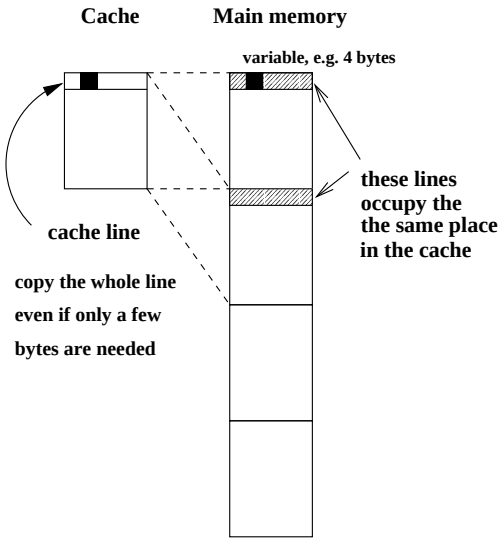
SRAM is usually faster than DRAM but since each bit requires several transistors (about six) you can get less bits of SRAM in the same area.

Each bit in a DRAM is stored using a capacitor and a transistor. Due to leakage the capacitor discharges gradually and the memory cell loses the information. Therefore, to preserve the information, the memory has to be refreshed periodically.

SRAM usually costs more per bit than DRAM and so is used for the most speed-critical parts of a computer (e.g. cache memory) or other circuit.



Direct mapped cache



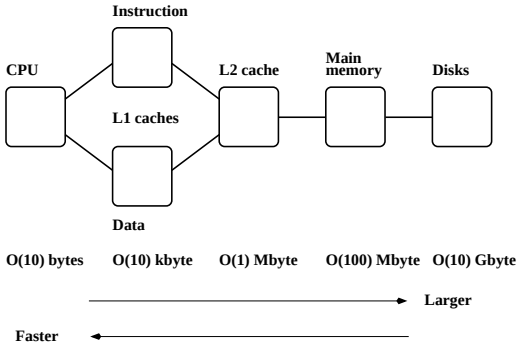

The cache lines are a few 10 to hundreds of bytes long.
Detta varierar mellan olika datorsystem.

It is a good idea to fetch cache lines, instead of individual variables, provided we access data sequentially in memory.

Bilden visar den enklaste typen av cache-organisation.
Moderna datorer har mer komplicerade varianter.

Programmet ligger också i minnet så vi vill ha en cache även för instruktioner.

## L1 and L2 caches




Dessa värden uppdateras. Minnena blir större och snabbare.

Några (få) datorer har även en L3-nivå.

Memory hierarchy

På följande sidor kommer några enkla "benchmarks".



## Another example - Horner's method

```
subroutine horner(px, x, coeff, n)
  integer           j, n
  double precision  px(n), x(n), coeff(0:4)
  double precision  xj

  do j = 1, n
    xj    = x(j)
    px(j) = coeff(0) + xj * (coeff(1) + xj * &
                             (coeff(2) + xj * &
                             (coeff(3) + xj * coeff(4))))
  end do

end
```

Take `n = 1000` and call the routine $10^6$ times.

Here are the results on a 900 MHz Sun and a
900 MHz Intel/HP Itanium 2 och 2.4GHz Opteron.

I compiled using full optimisation, `cc -fast`.
Sun's `cc` is faster than `gcc`. I assumed aliasing.
Executed using `java -server`, faster than `java -client`.
Jag använder SSE2 (vektorisering) på Opteron.

| System | Fortran | C/C++ | Java | `-client` |
|---|---|---|---|---|
| Sun | 4.7 | 37.0 | 38.2 | 54 |
| Itanium | 2.3 | 8.8 | 46.7 | - |
| Opteron | 2.8 | 3.1 | - | - |

Don't forget to optimise: f90 with no optimisation takes 82 s.



## Mitt benchmark i komplex form.

Fortran har `complex` som standard, C++ har det i standardbiblioteket. När det gäller Java så hämtade jag en klass från nätet. AL står för Javas `ArrayList` och `V` för `Vector` (trådsäker). Dessa kräver att `double` bakas in i en `Double`.

Allt är kört på 900 MHz Sun Blade. Tiderna är dels i sekunder och dels normaliserade så att Fortran tar tiden 1.

| Datatyp | Fortran | C++ | Java | AL | V |
|---|---|---|---|---|---|
| Double (norm) | 1 | 7.9 | 8.1 | 28 | 214 |
| Double (sek) | 4.7 | 37.0 | 38.2 | 132 | 1006 |
| Double complex (norm) | 1 | 3.2 | 33.8 | 46.5 | 84.8 |
| Double complex (sek) | 18 | 65.2 | 609.0 | 837 | 1526 |

OO behöver inte vara så långsamt. Notera `C++`. B. Stroustrup, The Design and Evoloution of C++, Addison-Wesley, 1994.

Om man gör en modul i Fortran 90 och skapar en egen komplex typ, `my_complex`, och ser till att koden inlinas i proceduren så går det lika snabbt som med Fortrans egen `complex`.

Fortran 90 klarar överlagring så jag fick ändra deklarationer och bildandet av konstanter, `x(j) = my_complex(1.0, 2.0)` t.ex.

Så här ser loopen ut i Java som inte stödjer överlagring:

```
for (int j = 0; j < n; j++) {
  xj    = x[j];  // Detta lönar sig!
  px[j] = coeff[0].add(xj.mul(coeff[1].add(xj.mul(
          coeff[2].add(xj.mul(coeff[3].add(xj.mul(
          coeff[4])))))))));
}
```



## Ett exempel på diverse egenheter

Exemplet har kompilerats på en 1.6 GHz Pentium 4, 512 Mbyte, Red Hat, Linux med Intels egen kompilator, ifc.

```
program main
  integer, parameter :: n = 2000
  double precision   :: A(n, n)

  do k = 1, 10
    call init(A, n)
  end do

  print*, A(1, 1)
end

subroutine init(A, n)
  double precision :: A(n, n)

  do j = 1, n
    do k = 1, n
      A(j, k) = 123.456
    end do
  end do

end
```

```
% ifc main.f90 init.f90
% time a.out
   123.456000000000
5.730u 0.140s 0:05.89 99.6%
```

Vi ber nu kompilatorn optimera koden:

```
% ifc -O3 -tpp7 -xW main.f90 init.f90
% time a.out
0.700u 0.100s 0:00.80 100.0%
```

Detta enkla program kan vi dock optimera själva:

```
    do k = 1, n
  do j = 1, n
      A(j, k) = 0.0
    end do
  end do

% ifc main.f90 init.f90   Ingen optimering
% time a.out
0.710u 0.100s 0:00.81 100.0%
```

Initiera en gång och byt de resterande nio anropen mot:

```
  do k = 1, n
    do j = 1, n
      A(j, k) = 324425.7d0 + 2.425d0 * A(j, k)
    end do
  end do

% time a.out
0.710u 0.100s 0:00.80 101.2%
```

Så i detta fall kostar inte räkneoperationerna något extra.

Detta är inte ovanligt. Det är minnesaccesserna som tar tid.



## Blocking and large strides

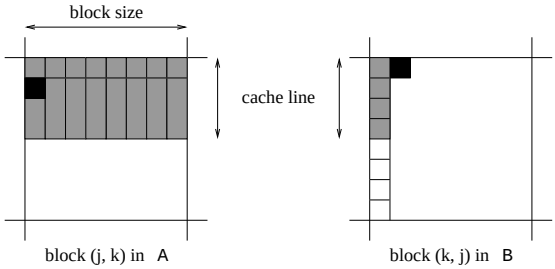
Sometimes loop interchange is of no use.

```
s = 0.0
do row = 1, n
  do col = 1, n
    s = s + A(row, col) * B(col, row)
  end do
end do
```

Blocking is good for data re-use, and when we have large strides.

Partition `A` and `B` in square sub-matrices each having the same order, the block size.

Treat pairs of blocks, one in `A` and one in `B` such that we can use the data which has been fetched to the L1 data cache.
Looking at two blocks:



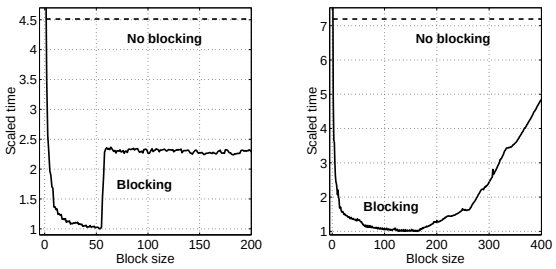

The block size must not be too large. Must be able to hold all the grey elements in `A` in cache (until they have been used).



This code works even if `n` is not divisible by the block size).

```
  ! first_row = the first row in a block etc.

  do first_row = 1, n, block_size
    last_row = min(first_row + block_size - 1, n)
    do first_col = 1, n, block_size
      last_col = min(first_col + block_size - 1, n)
      do row = first_row, last_row      ! sum one block
        do col = first_col, last_col
          s = s + A(row, col) * B(col, row)
        end do
      end do
    end do
  end do
```




$n = 2000$.

Note the speedups (4.5 and 7.2).



## BLAS

(the Basic Linear Algebra Subprograms).

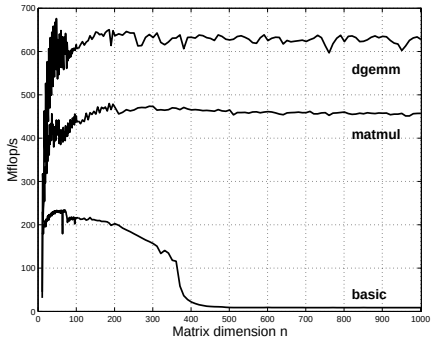
BLAS1: `y := a*x + y` one would use `daxpy`

BLAS2: `dgemv` can compute `y := a*A*x + b*y`

BLAS3: `dgemm` forms `C := a*A*B + b*C`

`daxpy`: $\mathcal{O}(n)$ data, $\mathcal{O}(n)$ operations
`dgemv`: $\mathcal{O}(n^2)$ data, $\mathcal{O}(n^2)$ operations
`dgemm`: $\mathcal{O}(n^2)$ data, $\mathcal{O}(n^3)$ operations, data **re-use**

Matrix multiplication: "row times column", slow.
Blocking is necessary.




375 MHz machine, start two FMAs per clock cycle, top speed is 750 million FMAs per second.

Lapack. Tuned libraries.