Computer Architecture

MJ Rutter
mjr19@cam.ac.uk

Lent 2003

Bibliography

Computer Architecture, A Qualitative Approach, 3rd Ed., Hennessy, JL and Patterson, DA, pub. Morgan Kaufmann, £37.
Operating Systems, Internals & Design Principles, 3rd Ed., Stallings, W, pub. Prentice Hall, £30.

Both are thick (1000 pages and 800 pages respectively), detailed, and quite technical. Both are pleasantly up-to-date.

Typeset by FoilTEX

Contents

History

History: to 1970

1951	Ferranti Mk I: first commercial computer
	UNIVAC I: memory with parity
1953	EDSAC I 'heavily used' for science (Cambridge)
1954	Fortran I (IBM)
1955	Floating point in hardware (IBM 704)
1956	Hard disk drive prototype. 24" platters (IBM)
1961	Fortran IV
	Pipelined CPU (IBM 7030)
1962	Hard disk drive with flying heads (IBM)
1963	CTSS: Timesharing (multitasking) OS
	Virtual memory & paging (Ferranti Atlas)
1964	First BASIC
1967	ASCII (current version)
	GE635 / Multics: SMP (General Elect)
1968	Cache in commercial computer (IBM 360/85)
	Mouse demonstrated
	Reduce: computer algebra
1969	ARPAnet: wide area network
	Fully pipelined functional units (CDC 7600)
	Out of order execution (IBM 360/91)

History: the 1970s

1970 First DRAM chip. 1Kbit. (Intel)
First floppy disk. 8" (IBM)
1971 UNIX appears within AT&T
Pascal
First email
1972 Fortran 66 standard published
First vector computer (CDC)
First TLB (IBM 370)
ASC: computer with 'ECC' memory (TI)
1973 First 'Winchester' disk (IBM)
1974 First DRAM with one transistor per bit
1975 UNIX appears outside AT&T
Ethernet appears (Xerox)
1976 Apple I launched. $666.66
Cray I
Z80 CPU (used in Sinclair ZX series) (Zilog)
$5\frac{1}{4}$" floppy disk
1978 K&R C appears (AT&T)
TCP/IP
Intel 8086 processor
Laser printer (Xerox)
WordStar (early wordprocessor)
First VAX (11/780) and VMS (DEC)
1979 TEX

History: the 1980s

1980 Sinclair ZX80 £100 10^5 sold
Fortran 77 standard published

1981 Sinclair ZX81 £70 10^6 sold
$3\frac{1}{2}$" floppy disk (Sony)
IBM PC & MS DOS version 1 $3,285
SMTP (current email standard) proposed

1982 Sinclair ZX Spectrum £175 48KB colour
Acorn BBC model B £400 32KB colour
Commodore64 $600 10^7 sold
Cray X-MP (first multiprocessor Cray)
Motorola 68000 (commodity 32 bit CPU)

1983 Internet defined to be TCP/IP only
Apple IIe $1,400
IBM XT, $7,545
Caltech Cosmic Cube: 64 node 8086/7 MPP

1984 Apple Macintosh $2,500. 128KB, 9" B&W screen
Sinclair QL £400. 128KB
IBM AT, $6,150. 256KB
CD ROM

1985 LaTeX2.09
PostScript (Adobe)
Ethernet formally standardised
IEEE 748 formally standardised
Intel i386 (Intel's first 32 bit CPU)
X10R1 (forerunner of X11) (MIT)
C++

History: the RISCs

1986	MIPS R2000, RISC CPU (used by SGI and DEC)
	SCSI formally standardised
1987	Intel i860 (Intel's first RISC CPU)
	Acorn Archimedes (ARM RISC) £800
	SPARC I, RISC CPU (Sun)
	Macintosh II $4,000. FPU and colour.
	Multiflow Trace/200: VLIW
	X11R1 (MIT)
1989	ANSI C
1990	PostScript Level 2
	Power I: superscalar RISC (IBM)
	MS Windows 3.0
1991	World Wide Web / HTTP
	PVM
	Tera starts developing MTA processor
1992	PCI
	OpenGL
	OS/2 2.0 (32 bit a year before NT) (IBM)
	Alpha 21064: 64 bit superscalar RISC (DEC)

A Summary of History

The above timeline stops a decade before this talk will first be given. Computing is not a fast-moving subject, and little of consequence has happened in the past decade.

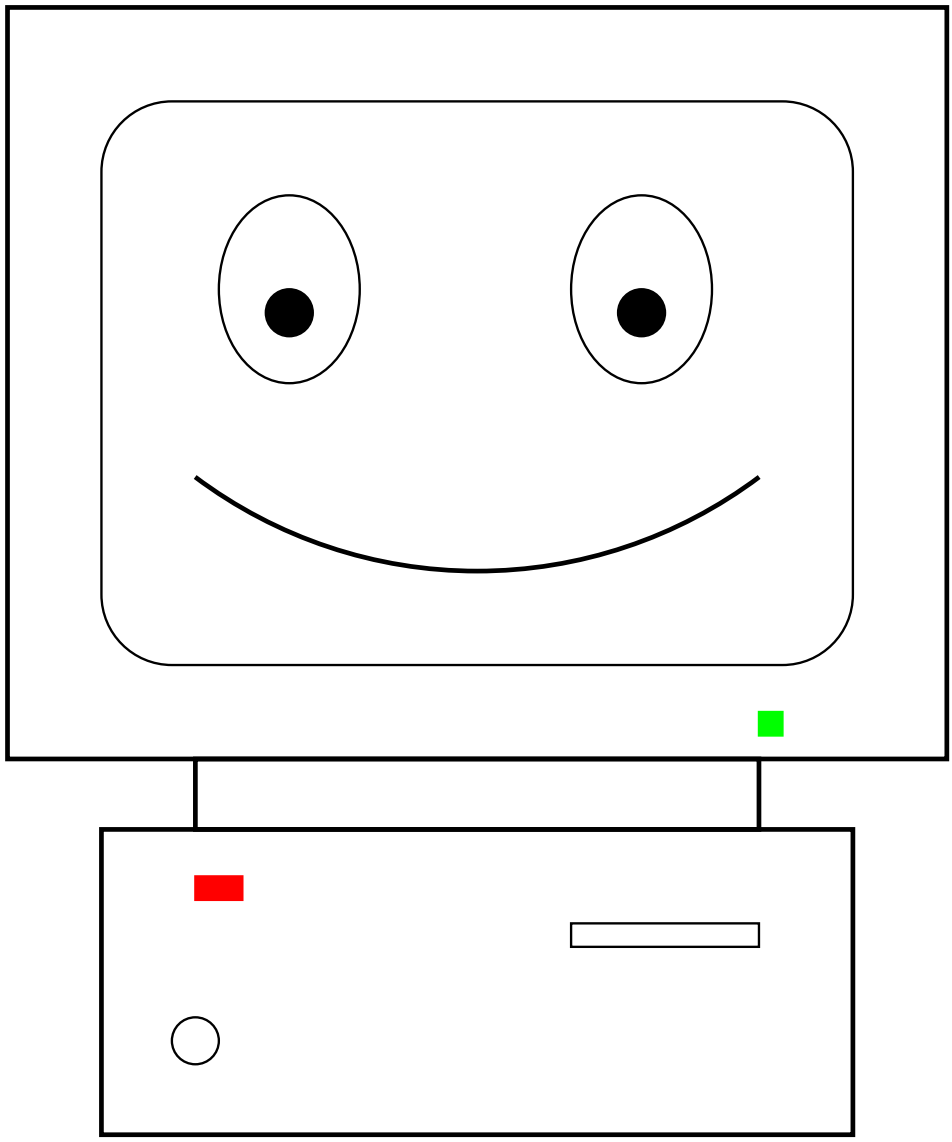
By 1970 the concepts of disk drives, floating point, memory paging, parity protection, multitasking, caches, pipelining and out of order execution have all appeared in commercial systems, and high-level languages and wide area networking have been developed.

The 1970s themselves add vector computers and error correcting memory, and implicit with the vector computers, RISC. The rest is just enhanced technology rather than new concepts.

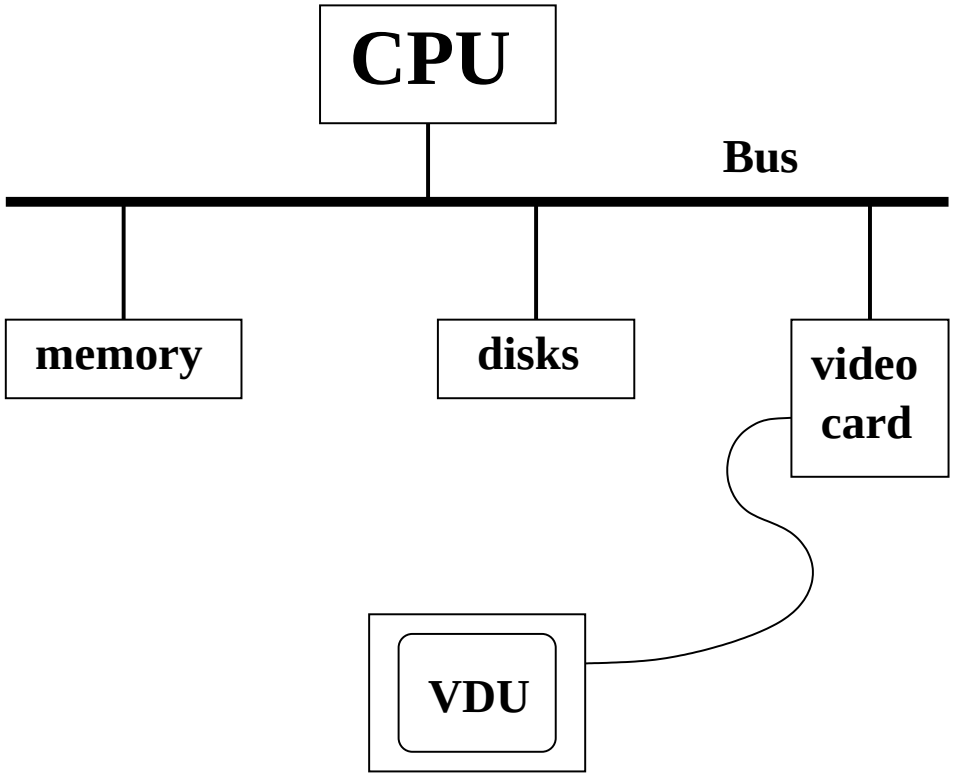
The 1980s see the first serious parallel computers, and much marketing in a home computer boom.

The CPU

The Computer



Inside the Computer



The Heart of the Computer

The CPU is the brains of the computer. Everything else is subordinate to this source of intellect.

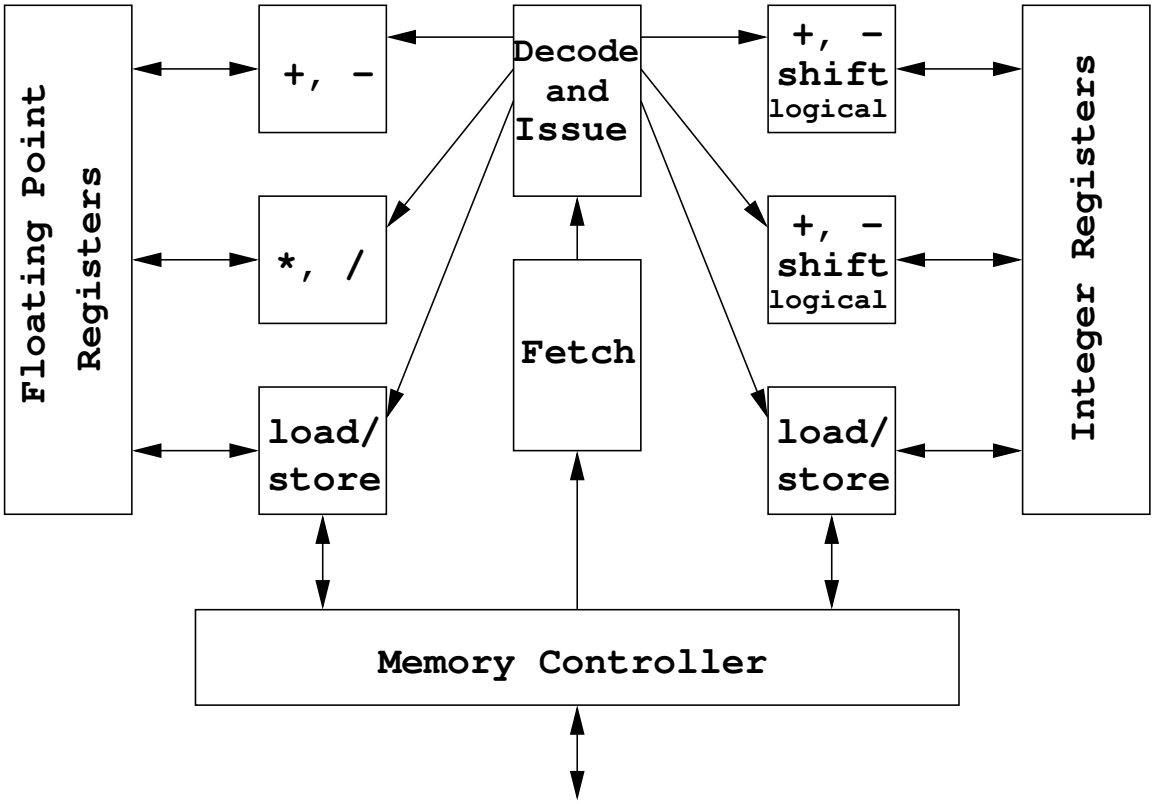
A typical modern CPU understands two main classes of data: integer and floating point. Within those classes it may understand some additional subclasses, such as different precisions.

It can perform basic arithmetic operations and comparisons, governed by a sequence of instructions, or $program$.

It can also perform comparisons, the result of which can change the $execution\ path$ through the program.

Its sole language is machine code, and each family of processors speaks a completely different variant of machine code.

Schematic of Typical RISC CPU



What the bits do

- Memory: not part of the CPU. Used to store both program and data.
- Instruction fetcher: fetches next machine code instruction from memory.
- Instruction decoder: decodes instruction, and sends relevant data on to. . .
- Functional unit: dedicated to performing single operation
- Registers: store the input and output of the functional units There are typically about 32 floating point registers, and 32 integer registers.

Partly for historical reasons, there is a separation between the integer and floating point parts of the CPU.

On some CPUs the separation is so strong that the only way of transferring data between the integer and floating point registers is via the memory. On some older CPUs (e.g. the Intel 386), the FPU (floating point unit) is optional and physically distinct.

Clock Watching

The best known part of a CPU is probably the $clock$. The clock is simply an external signal used for synchronisation. It is a square wave running at a particular frequency.

Clocks are used within the CPU to keep the various parts synchronised, and also on the data paths between different components external to the CPU. Such data paths are called $buses$, and are characterised by a $width$ (the number of wires (i.e. bits) in parallel) as well as a clock speed. External buses are usually narrower and slower than ones internal to the CPU.

Although sychronisation is important – every good orchestra needs a good conductor – it is a means not an end. A CPU may be designed to do a lot of work in one clock cycle, or very little, and comparing clock rates between different CPU designs is meaningless.

Typical instructions

Integer:

- arithmetic: $+$,$-$,$*$,$/$,negate
- logical: and, or, not, xor
- bitwise: shift, rotate
- comparison
- load / store (copy between register and memory)

Floating point:

- arithmetic: $+$,$-$,$*$,$/$,$\sqrt{}$,negate,modulus
- convert to/from integer
- comparison
- load / store (copy between register and memory)

Control:

- (conditional) branch (goto)

Most modern processors barely distinguish between integers used to represent numbers, and integers used to track memory addresses (i.e. pointers).

A typical instruction

`fadd f4,f5,f6`

add the contents of floating point registers 4 and 5, placing the result in register 6.

Execution sequence:

- fetch instruction from memory
- decode it
- collect required data (`f4` and `f5`) and sent to floating point addition unit
- wait for add to complete
- retrieve result and place in `f6`

Exact sequence varies from processor to processor.

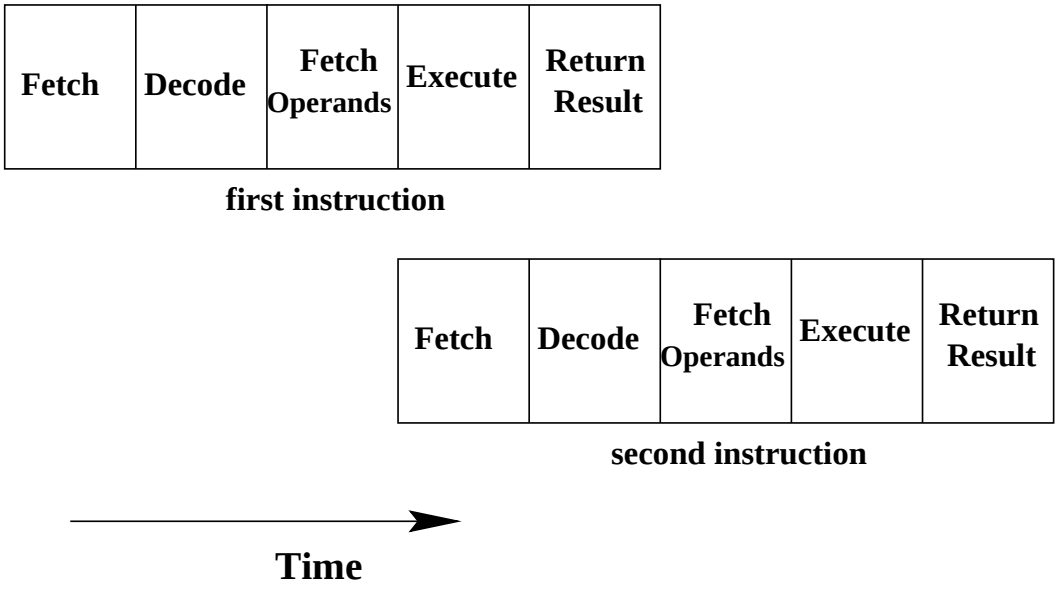
Always a *pipeline* of operations which must be performed sequentially.

The number of *stages* in the pipeline, or *pipeline depth*, can be between about 5 and 15 depending on the processor.

Making it go faster. . .

If each pipeline stage takes a single clock-cycle to complete, the previous scheme would suggest that it takes five clock cycles to execute a single instruction.

Clearly one can do better: in the absence of branch instructions, the next instruction can always be both fetched and decoded whilst the previous instruction is executing. This shortens our example to three clock cycles per instruction.



. . . and faster. . .

Further improvements are governed by *data dependency*. Consider:

```
fadd f4,f5,f6
fmul f6,f7,f4
```

(Add `f4` and `f5` placing the result in `f6`, then multiply `f6` and `f7` placing the result back in `f4`.)

Clearly the add must finish (`f6` must be calculated) before the multiply can start. There is a data dependency between the multiply and the add.

But consider

```
fadd f4,f5,f6
fmul f3,f7,f9
```

Now any degree of overlap between these two instructions is permissible: they could even execute simultaneously or in the reverse order and still give the same result.

. . . and faster

We have now reached one instruction per cycle, assuming data independency. The problem is now decoding the instructions.

Unlike written English, there is no mark which indicates the end of each machine-code word, sodecodingmustbedonesequentially.

The solution is to fix the 'word' length. If all instructions are four bytes long, then one knows trivially where the next, or next-but-ten instruction starts without decoding the intermediate ones.

Then multiple decoders can run in parallel, and multiple instructions can be issued per clock cycle. Such a processor is said to be $superscalar$, or $n\text{-}way\ superscalar$ if one wishes to specify how many instructions can be issued per cycle.

Keep it simple

With short, simple instructions, it is easy for the processor to schedule many overlapping instructions at once.

If a single instruction both read and wrote data to memory, and required the use of multiple functional units, such scheduling would be much harder.

This is part of the CISC vs RISC debate.

CISC (Complex Instruction Set Computer) relies on a single instruction doing a lot of work: maybe incrementing a pointer and loading data from memory and doing an arithmetic operation.

RISC (Reduced Instruction Set Computer) relies on the instructions being very simple – the above CISC example would certainly be three RISC instructions – and then letting the CPU overlap them as much as possible.

The VLIW EPIC

As an alternative to the CPU determining which instructions can be issued simultaneously in a dynamic fashion as the code executes, VLIW computers rely on the compiler to bundle the instructions appropriately, thus simplifying the logic on the CPU.

A single instruction word for a VLIW machine contains several independent instructions for different functional units with no data dependencies. The instruction decoder can simply take the bundle and issue them simultaneously without significant further thought.

However, if the compiler cannot fill all the slots in one word, the empty ones need to be filled with explicit 'nop' instructions. This can lead to large code sizes.

VLIW = Very Long Instruction Word
EPIC = Explicitly Parallel Instruction Code
nop = No OPeration

Within a functional unit

A functional unit may itself be pipelined. Considering again floating-point addition, even in base 10 there are three distinct stages to perform:

$9.67 * 10^5 + 4 * 10^4$

First the exponents are adjusted so that they are equal:

$9.67 * 10^5 + 0.4 * 10^5$

only then can the mantissas be added

$10.01 * 10^5$

then one may have to readjust the exponent

$1.001 * 10^6$

So floating point addition usually takes at least three clock cycles. But the adder may be able to start a new addition ever clock cycle, as these stages are distinct.

Such an adder would have a $latency$ of three clock cycles, but a $repeat$ or $issue\ rate$ of one clock cycle.

Breaking Dependencies

```
do i=1,n
  sum=sum+a(i)
enddo
```

This would appear to require three cycles per iteration, as the iteration `sum=sum+a(i+1)` cannot start until `sum=sum+a(i)` has completed. However, consider

```
do i=1,n,3
  s1=s1+a(i)
  s2=s2+a(i+1)
  s3=s3+a(i+2)
enddo
sum=s1+s2+s3
```

The three distinct partial sums have no interdependency, so one add can be issued every cycle.

Do not do this by hand. This is a job for an optimising compiler, as you need to know a lot about the particular processor you are using before you can tell how many paritial sums to use.

A Branch in the Pipe

So far we have assumed a linear sequence of instructions. What happens if there is a branch?

```
   double t=0.0; int i,n;
   for (i=0;i<n;i++) t=t+x[i];

# $17 contains n, # $16 contains x
  fclr $f0
  clr $1
  ble $17,L$5
L$6:
  ldt   $f1, ($16)
  addl  $1, 1, $1
  cmplt $1, $17, $3
  lda   $16, 8($16)
  addt  $f0, $f1, $f0
  bne   $3, L$6
L$5:
```

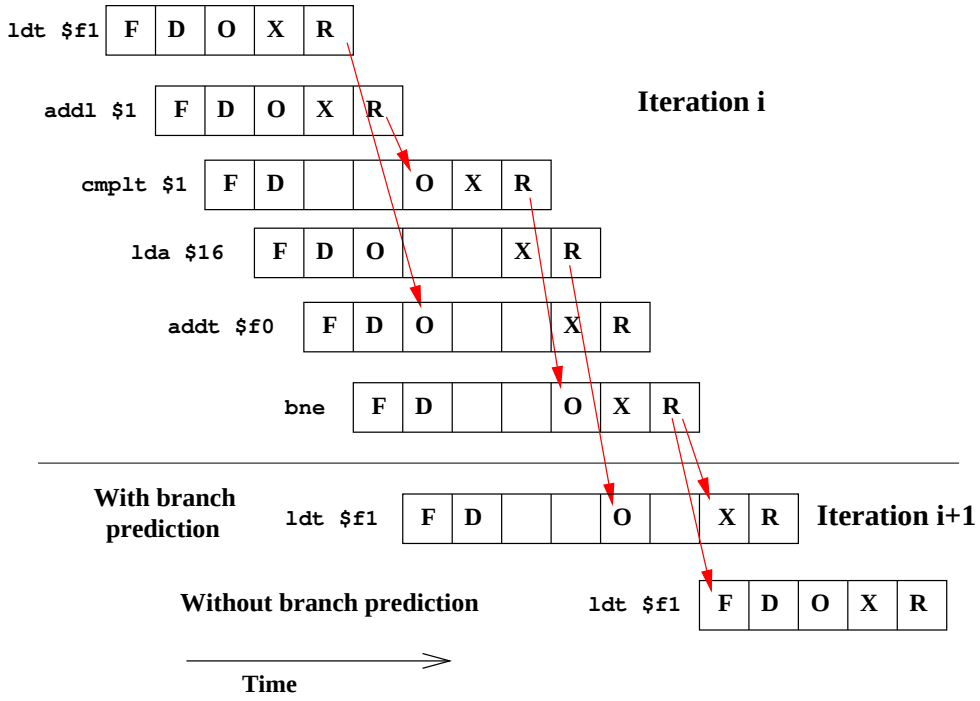
There will be a conditional jump or $branch$ at the end of the loop. If the processor simply fetches and decodes the instructions following the branch, then when the branch is taken, the pipeline is suddenly empty.

Assembler in More Detail

The above is Alpha assembler. The integer registers $1, $3, $16 and $17 are used, and the floating point registers $f0 and $f1. The instructions are of the form 'op a,b,c' meaning 'c=a op b'.

`fclr $f0`	Float CLeaR `$f0` – place zero in `$f0`
`clr $1`	CLeaR `$1`
`ble $17, L$5`	Branch if Less than or Equal on comparing $17 to (an implicit) zero and jump to `L$5` if less (i.e. skip loop)
L$6:	
`ldt $f1, ($16)`	LoaD `$f1` with value value from memory address $16
`addl $1, 1, $1`	$1=$1+1
`cmplt $1, $17, $3`	CoMPare $1 to $17 and place result in $3
`lda $16, 8($16)`	LoaD Address effectively $16=$16+8
`addt $f0, $f1, $f0`	`$f0=$f0+$f1`
`bne $3,L$6`	Branch Not Equal – if counter $\neq$n, do another iteration
L$5:	

Predictions



With the simplistic pipeline model of page 18, the loop will take 9 clock cycles per iteration if the CPU predicts the branch and fetches the next instruction appropriately. With no prediction, it will take 12 cycles.

A 'real' CPU has a pipeline *depth* much greater than the five slots shown here: usually ten to twenty. The penalty for a mispredicted branch is therefore large.

Note the *stalls* in the pipeline based on data dependencies (shown with red arrows) or to prevent the execution order changing. If the instruction fetch unit fetches one instruction per cycle, stalls will cause a build-up in the number of *in flight* instructions. Eventually the fetcher will pause to allow things to quieten down.

This is not the correct timing for any Alpha processor.

Speculation

In the above example, the CPU does not begin to execute the instruction after the branch until it knows whether the branch was taken: it merely fetches and decodes it, and collects its operands. A further level of sophistication allows the CPU to execute the next instruction(s), provided it is able to throw away all results and side-effects if the branch was mispredicted.

Such execution is called *speculative execution*. In the above example, it would enable the `ldt` to finish one cycle earlier, progressing to the point of writing to the register before the result of the branch were known.

More advanced forms of speculation would permit the write to the register to proceed, and would undo the write should the branch have been mispredicted.

Errors caused by speculated instructions must be carefully discarded. It is no use if
`if (x>0) x=sqrt(x)`
causes a crash when the square root is executed speculatively with `x=-1`, nor if
`if (i<1000) x=a(i)`
causes a crash when `i=2000` due to trying to access `a(2000)`.

Almost all current processors are capable of some degree of speculation.

Predication

Most CPUs have the branch instruction as their only conditional instruction, so that a code sequence such as:

```
  if (a<0) a=-a;
```

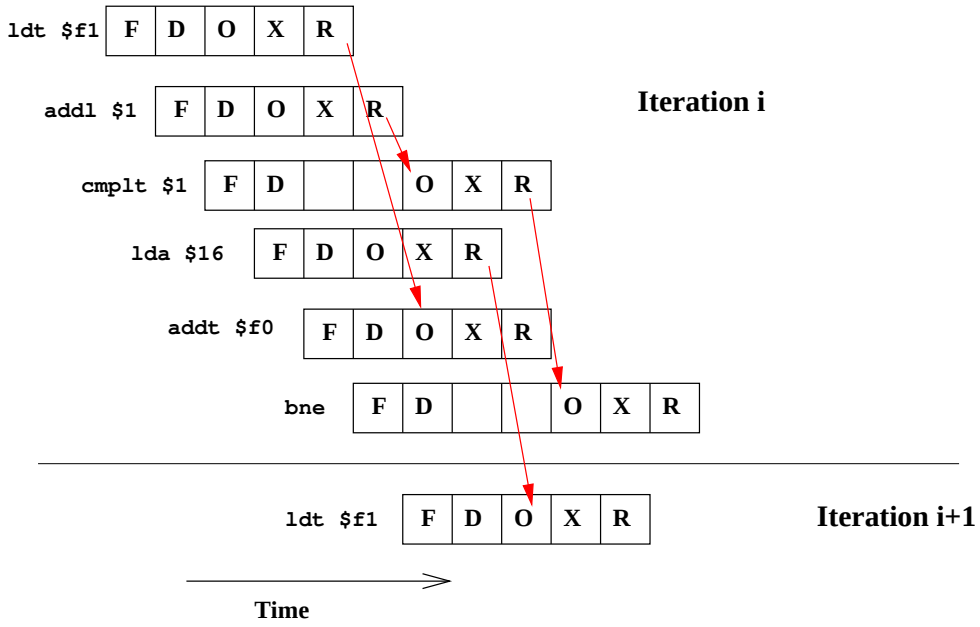
must be converted to

```
  if (a>=0) goto L1;
  a=-a;
L1:
```

This causes a large number of conditional branches with the problems mentioned above. With full predication, any instruction can be prefixed by a condition. This avoids interrupting the progress of the instruction fetching and decoding logic.

Many modern CPUs, including Alpha, can predicate on loads. Full predication is rare, although the ARM CPU achieved it in 1987.

OOO!



Previously the `cmplt` is delayed due to a dependency on the `addl` immediately preceeding it. However, the next instruction has no relevant dependencies. A processor capable of *out-of-order* execution could execute the `lda` before the `cmplt`.

The timing above assumes that the `ldt` of the next iteration can be executed speculatively and OOO before the branch. Different CPUs are capable of differing amounts of speculation and OOOE.

The EV6 Alpha does OOOE, the EV5 does not, nor does the UltraSPARC III. In this simple case, the compiler erred in not changing the order itself. However, the compiler was told not to optimise for this example.

Machine Code

Most RISC processors use a fixed instruction *word* of four bytes – the smallest convenient power of two.

An instruction may need to specify up to three registers, and, with 32 registers, 5 bits are needed to identify each, or 15 bits for three. The remaining 17 bits are plenty to specify the few dozen possible instructions.

Some instructions might need just two registers and an integer constant provided within the instruction itself. Many RISC processors allow for 16 bits of data to be stored in this manner, for a subset of their instructions.

Branch instructions need just a single register, and the destination is usually stored as an offset from the current position. This will always be a multiple of four, so the two lowest bits are not stored.

Unlike *byte*, which always means 8 bits, there is no precise definition of *word*. It usually means 4 bytes, the length of an instruction, except when talking about the 8086, when it means two bytes, or vector Crays, when it means eight bytes.

The IA32 instruction set, with its variable length, can place double precision floating point values as data within a single instruction, and must store all bits of its branches.

Not all possible bit sequences will be valid instructions. If the instruction decoder hits an invalid instruction, it objects. Under UNIX this results in the process receiving a SIGILL: SIGnal ILLegal instruction.

Meaningless Indicators of Performance

- MHz: the silliest: some CPUs take 4 clock cycles to perform one operation, others perform four operations in one clock cycle. Only any use when comparing otherwise identical CPUs.

- MIPS: Millions of Instructions Per Second. Theoretical peak speed of decode/issue logic.

- MTOPS: Millions of Theoretical Operations Per Second. Current favourite of the US Government.

- FLOPS: Floating Point Operations Per Second. Theoretical peak issue rate for floating point instructions. Loads and stores usually excluded. Ratio of $+$ to $*$ is usual fixed (often $1:1$).

- MFLOPS, GFLOPS, TFLOPS: 10^6, 10^9, 10^{12} FLOPS.

As we shall see later, most of these are not worth the paper they are written on.

Meaningful Indicators of Performance

The only really good performance indicator is how long a computer takes to run $your$ code. Thus my fastest computer is not necessarily your fastest computer.

Often one buys a computer before one writes the code it has been bought for, so other 'real-world' metrics are useful.

Unfortunately, there are not many good ones. Here is a critique of the main contenders.

The Guilty Candidates: Linpack

Linpack 100x100

Solve 100x100 set of double precision linear equations using fixed FORTRAN source. Pity it takes just 0.7 s at 1 MFLOPS and uses under 100KB of memory. Only relevant for pocket calculators.

Linpack 1000x1000 or nxn

Solve 1000x1000 (or nxn) set of double precision linear equations by any means. Usually coded using a blocking method, often in assembler. Is that relevant to your style of coding? Achieving less than 50% of a processor's theoretical peak performance is unusual.

Number of operations: O(n^3), memory usage O(n^2).
n chosen by manufacturer to maximise performance, which is reported in MFLOPS.

SPEC

SPEC is a non-profit benchmarking organisation. It has two CPU benchmarking suites, one concentrating on integer performance, and one on floating point. Each consists of around ten programs, and the mean performance is reported.

Unfortunately, the benchmark suites need constant revision to keep ahead of CPU developments. The first was released in 1989, the second in 1992, the third in 1995. None of these use more than 8MB of data, so fit in cache with many current computers. Hence a fourth suite was released in 2000.

It is not possible to compare results from one suite with those from another, and the source is not publically available.

Until 2000, the floating point suite was entirely Fortran.

Two scores are reported, 'base', which permits two optimisation flags to the compiler, and 'peak' which allows any number of compiler flags. Changing the code is not permitted.

SPEC: Standard Performance Evaluation Corporation (`www.spec.org`)

The glimmers of hope

Linpack, the return

Taking the 100x100 Linpack source and rewriting it to be 1000x1000 (or 2000x2000) does give a half-reasonable benchmark. Most computers achieve between 5 and 15% of their processor's peak performance on this code.

Streams

Streams (a public domain benchmark) does not really measure CPU performance, but rather memory performance. This is often rather more useful.

Various Results, SPEC

Processor	MHz	SpecInt	SpecFP
Alpha 21364	1150	877	1482
Itanium 2	1000	807	1431
Alpha 21264	1250	928	1365
Power4	1450	935	1295
Pentium4	3066	1107	1091
Athlon	2250	933	843
UltraSPARC III Cu	1050	610	827
Itanium	800	379	701
MIPS R14000	600	500	529
Power3-II	450	346	433
Pentium III	1000	442	335

For each CPU, the best result (i.e. fastest motherboard / compiler / clock speed) as of 1/2/03 is given.

Note that the Pentium4, Athlon and Pentium III are the only CPUs to have higher SpecInt scores than SpecFP.

Various Results, Streams and Linpack

Machine	Year	CPU/MHz	Streams	Linpack	dgesv
Pentium4	2002	P4/2400	1850	241	2750
Pentium4	2002	P4/1800	1140	140	1980
PentiumIII	1999	PIII/650	343	47	454
XP1000	1999	21264/500	980	146	683
PW500au	1998	21164/500	233	42	590
AS500/500	1996	21164/500	170	32	505

The 'Linpack' column is for the 2000x2000 Fortran version, whereas the `dgesv` column is the same problem using the vendor's supplied maths library.

The faster P4 uses RAMBUS memory, the slower SDRAM. Similarly the two 21164 machines have different memory subsystems, but identical processors.

Representing Integers

Computers store bits, each of which can represent either a 0 or 1.

For historical reasons bits are processed in groups of eight, called $bytes$. One byte is sufficient to store one English character.

Most CPUs can handle integers of different sizes, typically some of 1, 2, 4 and 8 bytes long.

For the purposes of example, we shall consider a half-byte integer (i.e. four bits).

Being Positive

This is tediously simple:

Bits	number
0000	0
0001	1
0010	2
0011	3
0100	4
. . .	. . .
1111	15

This is the obvious, and universal, representation for positive integers: binary.

One more obvious point: 4 bits implies 2^4 combinations. Whatever we do, we can represent only 16 different numbers with 4 bits.

Being Negative

There are many ways of being negative.

Sign-magnitude

Use first bit to represent sign, remaining bits to represent magnitude.

Offset

Add a constant (e.g. 8) to everything.

One's complement

Reverse all the bits to represent negative numbers.

Two's complement

Reverse all the bits then add one to represent negative numbers.

Chaos

Bits	s-m	off	1's	2's
0000	0	-8	0	0
0001	1	-7	1	1
0010	2	-6	2	2
0111	7	-1	7	7
1000	-0	0	-7	-8
1001	-1	1	-6	-7
1110	-6	6	-1	-2
1111	-7	7	-0	-1

Of these possibilities, two's complement is almost universally used.

Having only one representation for zero is usually an advantage, and having zero being the bit pattern '$000\ldots$' is also a good thing.

Adding up

Again, trivial:

$$0101 + 1001 = 1110$$

Otherwise known as

$$5 + 9 = 14$$

But note how this would read using the various mappings for negative numbers:

- sign-mag: $5 + (-1) = -6$
- offset: $(-3) + 1 = 6$
- 1's: $5 + (-6) = -1$
- 2's: $5 + (-7) = -2$

Clearly not all mappings are equal.

Overflow

$$5 + 12 = 1$$

maybe not, but

$$0101 + 1100 = 0001$$

as there is nowhere to store the first bit of the correct answer of 10001. *Integer arithmetic simply wraps around on overflow.*

Interpreting this with 1's and 2's complement gives:

- 1's: $5 + (-3) = 1$
- 2's: $5 + (-4) = 1$

This is why two's complement is almost universal. An adder which correctly adds unsigned integers will correctly add two's complement integers. A single instruction can add bit sequences without needing to know whether they are unsigned or two's complement.

Ranges

bits	unsigned	2's comp.
8	0 to 255	-128 to 127
16	0 to 65535	-32768 to 32767
32	0 to 4294967295	-2147483648 to 2147483647
64	0 to 1.8×10^{19}	-9×10^{18} to 9×10^{18}

Uses:

- 8 bits: Latin character set
- 16 bits: Graphics co-ordinates
- 32 bits: General purpose
- 64 bits: General purpose

Note that if 32 bit integers are used to address bytes in memory, then 4GB is the largest amount of memory that can possibly be addressed.

Similarly 16 bits and 64KB, for those who remember the BBC 'B', Sinclair Spectrum, Commodore64 and similar.

Text

Worth a mention, as we have seen so much of it. . .

American English is the only language in the world, and it uses under 90 characters. Readily represented using 7 bits, various extensions to 8 bits add odd European accented characters, and £.

Most common mapping is ASCII.

0000000-0011111	control codes
0100000	space
0110000-0111001	0 to 9
1000001-1011010	A to Z
1100001-1111010	a to z

Punctuation fills in the gaps.

The contiguous blocks are pleasing, as is the single bit distinguishing upper case from lower case.

Note that only seven bits, not eight, are used. The 'control codes' are special codes for new line, carriage return, tab, backspace, new page, etc.

ASCII = American Standard Code for Information Interchange
Other main mapping is EBDIC, used (mainly) by old IBM mainframes.

Multiplying and Dividing

Multiplication is harder and (typically) slower than addition. It also causes numbers to increase rapidly in magnitude.

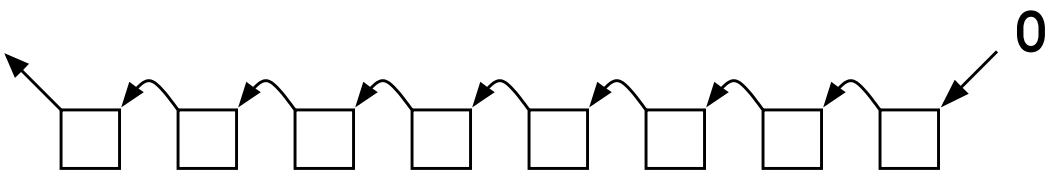
Some processors have expanding multiply instructions, e.g. for multiplying two 16 bit numbers, and keeping all 32 bits of the result. Other instructions simply truncate the result to the size of the operands.

Some processors have no integer multiply instruction. Examples include the old and simple, such as the Z80, and the relatively modern RISC range PA-RISC (HP).

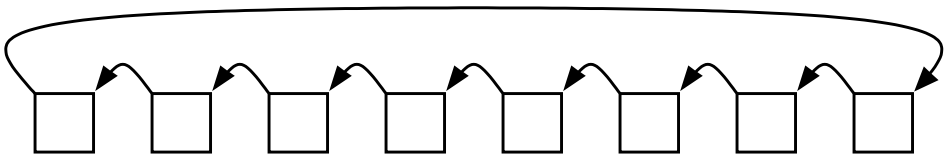
Division is yet more complex, and much rarer in code. Hence even fewer processors support it. Alpha does not.

Shifting and Rotating

All processors have both shift and rotate instructions. Shift moves the bits along, filling in with zeros, whereas rotate fills in with the bits shifted out of the other end of the register. Illustrated for an 8 bit register.



shift left



rotate left

A left shift by n positions corresponds to $\times 2^n$.

A right shift by n positions corresponds to dividing by 2^n with the remainder lost.

One can simulate multiplication from a combination of shifts, adds and comparisons.

Logical operations

The other class of operations that all processors can do is bitwise logical operations. The common operations are provided:

	and	or	xor
0 0	0	0	0
0 1	0	1	1
1 0	0	1	1
1 1	1	1	0

These operations crop up surprisingly frequently. For instance:

x and $7 = 0$ implies x is divisible by eight.

$(x + 7)$ and -8 is smallest number $\geq x$ divisible by 8

(any letter) or $32 =$ corresponding lower case letter (ASCII)

(any letter) and $95 =$ corresponding upper case letter (ASCII)

xor is used for trivial encryption and for graphics cursors, because $(a \text{ xor } b) \text{ xor } b \equiv a$.

Typical functional unit speeds

Instruction	Latency	Issue rate
iadd/isub	1	1
and, or, etc.	1	1
shift, rotate	1	1
load/store	1-2	1
imul	3-15	3-15
fadd	3	1
fmul	2-3	1
fdiv	15-25	15-25
fsqrt	15-25	15-25

In general, most things 1 or 2 clock cycles, except integer $\times$, and floating point $\div$ and $\sqrt{}$.

'Typical' for processors such as DEC Alpha, MIPS R10000 and similar RISC processors. Very recent processors tend to have longer fp latencies: 4 for `fadd` and `fmul` for the UltraSPARC III, 5 and 7 respectively for the Pentium 4.

Those slow integer multiplies are more common that it would seem at first. Consider:

```
double precision x(1000),y(500,500)
```

The address of `x(i)` is the address of `x(1)` plus $8 * (i - 1)$. That multiplication is just a shift. However, `y(i,j)` is `y(1,1)` plus $8 * ((i - 1) + (j - 1) * 500)$. A lurking multiply!

C does things very differently, but not necessarily better.

Floating Point

Floating Point Numbers

Perhaps the most important area of scientific computing, but probably not the most well understood.

Let us begin by revising our knowledge of base-10 'scientific notation' and assuming that every number we write shall be written as a four digit signed $mantissa$ and a two digit signed exponent.

$$\pm 0.XXXX * 10^{\pm XX}$$

e.g.

$$0.1000 * 10^{1}$$

or

$$0.6626 * 10^{-33}$$

(As is conventional, the mantissa, M, is restricted to $0.1 \leq M < 1$.)

Representable numbers

Using this notation, we can represent at most four million distinct numbers. Having merely six digits which can range from 0 to 9, and two signs, there are only 4,000,000 possible combinations.

The largest number we can represent is $0.9999 * 10^{99}$, the smallest $0.1000 * 10^{-99}$, or $0.0001 * 10^{-99}$ if we do not mind having fewer than four digits of precision in the mantissa.

Algebra

$$\boldsymbol{a + b = a \not\Rightarrow b = 0}$$

$$0.1000 * 10^1 + 0.4000 * 10^{-3} = 0.1000 * 10^1$$

$$\boldsymbol{(a + b) + c \neq a + (b + c)}$$

$$\begin{aligned}
&(0.1000 * 10^1 + 0.4000 * 10^{-3}) + 0.4000 * 10^{-3} \\
&\quad = \quad 0.1000 * 10^1 + 0.4000 * 10^{-3} \\
&\quad = \quad 0.1000 * 10^1 \\
&0.1000 * 10^1 + (0.4000 * 10^{-3} + 0.4000 * 10^{-3}) \\
&\quad = \quad 0.1000 * 10^1 + 0.8000 * 10^{-3} \\
&\quad = \quad 0.1001 * 10^1
\end{aligned}$$

Algebra (2)

$$\boldsymbol{\sqrt{a^2} \neq |a|}$$

$$\sqrt{(0.1000 * 10^{-60})^2} = \sqrt{0.0000 * 10^{-99}} = 0$$

$$\boldsymbol{a/b \neq a \times 1/b}$$

$$\begin{aligned} 0.6000 * 10^1 / 0.7000 * 10^1 &= 0.8571 \\ 0.6000 * 10^1 \times (1/0.7 * 10^1) &= 0.6000 * 10^1 \times 0.1429 \\ &= 0.8574 \end{aligned}$$

Zeros, Underflows, and Denormals

Whereas $1-1=0$ is indisputable, other zeros are more doubtful.

$0.1*10^{-99}/0.1*10^{10}=0$ is an example of $underflow$. The real answer is non-zero, but smaller than the smallest representable number, so it is/must be expressed as zero. A sign can be retained: $-0.1*10^{-99}/0.1*10^{10}=-0$.

$0.1*10^{-99}/2$ is more awkward. Should it be kept as $0.05*10^{-99}$, which breaks the rule that the mantissa must be greater than 0.1, and risks nonsense as precision is slowly lost, or should it be set to zero? The former is called gradual underflow, and results in $denormalised$ numbers. Flushing denormalised numbers to zero is the other option.

Many processors cannot compute directly with denormalised numbers. As soon as the FPU encounters one, it signals an error and a special software routine needs to tidy up the mess. This can make computing with denormals hundreds of times slower than computing with normalised numbers. Given that their use also implies a precision loss, flushing to zero is often best.

Binary fractions

Follow trivially from decimal fractions:

$$0.625 = 2^{-1} + 2^{-3} = 0.101_2$$

but note that some finite decimal fractions are not finite binary fractions

$$0.2_{10} = 0.0011001100110011\ldots_2$$

(although any finite binary fraction is a finite decimal fraction of the same number of digits)

n_m is a common way of expressing 'interpret n as a number in base m.' Of course, m itself is always base-10.

Computers and IEEE

IEEE 754 defines a way both of representing and manipulating floating point numbers on a computer. Its use is almost universal.

In a similar fashion to the above decimal format, it defines a sign bit, a mantissa of fixed length, and an exponent. Naturally everything is in base 2, and the exponent is not signed, but rather it has a constant offset added to it: 127 in the case of single precision.

IEEE requires that the simple arithemetic operators return the nearest representable number to the true result, and and consequently that $a + b = b + a$ and $ab = ba$.

Note that the IEEE mantissa is an example of sign-magnitude storage, and exponent offset. Two's compliment is not universal.

IEEE Example

As an example of a single precision number:

$$5\frac{3}{4} = 101.11_2 = 0.10111_2 * 2^3$$

This is stored as a sign (0 for $+$), an 8 bit exponent biased by 127, so 10000010 here, and then a 23 bit mantissa. Because the first digit of a normalised mantissa is always 1, that digit is not stored. This leaves the sequence

$$01000001001110000000000000000000$$

So this bit sequence represents $5\frac{3}{4}$ when interpreted as a single precision IEEE floating point number, or 1094189056 when interpreted as a 32 bit integer.

The above is perfectly valid, but very different, when interpreted as a `real` or an `integer`. Nothing tags a value to make it clear that it is integer or floating point: a programmer must keep track of what was stored where!

Nasty numbers

A few other 'special' numbers exit, for dealing with overflows, underflows, $\sqrt{-1}$ and other problems. For this reason, two of the possible exponent bit-sequences (all ones and all zeros) are reserved.

For an overflow, the resulting 'infinity' is represented by setting the sign bit appropriately, the mantissa equal to zero, and the exponent equal to all ones.

Something which is 'Not a (real) Number', such as $0/0$ or $\sqrt{-1}$, is represented similarly but the mantissa is set non-zero. This is normally reported to the user as 'NaN'.

Zero is represented by setting all bits to zero. However the sign bit may still be one, so $+0$ and -0 exist. For denormalised numbers the exponent is zero, and all bits of the mantissa are stored, for one no longer has a leading one.

In comparisons, $+0$ and -0 compare as equal.

When reading rubbish bit sequences as doubles, one expects merely one in 2000 to appear as a `NaN`.

Signals

Sometimes it is useful for a program to abort with an error as soon as it suffers an overflow, or generates a NaN. Less often it is useful for underflows to stop a program.

By convention Fortran tends to stop on overflows and NaNs, whereas C does not and expects the programmer to cope.

If the code does stop, it will do so as a result of receiving a signal from the floating point unit, and it will complain `SIGFPE`: SIGnal Floating Point Exception.

Also by convention, integer overflow wraps round silently. The conversion of a real to an integer when the real is larger than the largest possible integer might do almost anything. In Java it will simply return the largest possible integer.

Ranges

	Precision	
	Single	Double
Bytes	4	8
Bits, total	32	64
Bits, exponent	8	11
Bits, mantissa	23	52
Largest value	$1.7 * 10^{38}$	$9 * 10^{307}$
Smallest non-zero	$6 * 10^{-39}$	$1 * 10^{-308}$
Decimal digits of precision	c.7	c.15

Other representations result in different ranges. For instant, IBM 370 style encoding has a range of around 10^{75} for both single and double precision.

IEEE is less precise about extended double precision formats. Intel uses an 80 bit format with a 16 bit exponent, whereas many other vendors use a 128 bit format.

Rounding

Rounding errors have a habit of building up. With IEEE's round-to-nearest, a simplistic estimation of the relative error is

$$2^{-25} * \sqrt{n}$$

after n single-precision operations. Other forms of arithmetic can be much worse. IBM 370 style rounds by truncation (i.e. towards zero). This is simple to implement, but leads to relative errors building as

$$2^{-25} * n$$

in its single-precision form. This can be disasterous.

IEEE also provides for rounding by truncation, but this is rarely used!

Backwards and Forwards

$$\sum_{n=1}^{N} \frac{1}{n}$$

Consider summing this series forwards $(1..N)$ and backwards $(N..1)$ using single precision arithmetic.

N	forwards	backwards	exact
100	5.187378	5.187377	5.187378
1000	7.485478	7.485472	7.485471
10000	9.787613	9.787604	9.787606
100000	12.09085	12.09015	12.09015
1000000	14.35736	14.39265	14.39273
10000000	15.40368	16.68603	16.69531
100000000	15.40368	18.80792	18.99790

The smallest number such that $15 + x \neq x$ is about $5 * 10^{-7}$. Therefore, counting forwards, the total stops growing after around two million terms.

This is better summed by doing a few hundred terms explicitly, then using a result such as

$$\sum_{n=a}^{b} \frac{1}{n} \approx \log\left(\frac{b+0.5}{a-0.5}\right) + \frac{1}{24}\left((b+0.5)^{-2} - (a-0.5)^{-2}\right) + O(a^{-4})$$

The Quadratic Formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$30x^2 + 60.01x + 30.01 = 0$$

Roots are -1 and $-1\frac{1}{3000}$.

Single precision arithmetic and the above formula give no roots!

number	nearest representable single prec. no.
30	30.0000000000
30.01	30.0100002289. . .
60.01	60.0099983215. . .

Even with no further rounding errors, whereas $4 * 30 * 30.1 = 3601.2$ and $60.01^2 = 3601.2001$, $60.0099983215\ldots^2 = 3601.199899\ldots$.

The following gives no roots when compiled with a K&R C compiler, and repeated roots with ANSI C

```
void main(){
   float a=30,b=60.01,c=30.01,d;
   d=b*b-4*a*c;
   printf("%18.15f\n",(double)d);
}
```

The Logistic Map

$$x_{n+1} = 4x_n(1 - x_n)$$

n	single	double	correct
0	0.5200000	0.5200000	0.0520000
1	0.9984000	0.9984000	0.9984000
2	0.0063896	0.0063898	0.0063898
3	0.0253952	0.0253957	0.0253957
4	0.0990019	0.0990031	0.0990031
5	0.3567998	0.3568060	0.3568060
10	0.9957932	0.9957663	0.9957663
15	0.7649255	0.7592756	0.7592756
20	0.2214707	0.4172717	0.4172717
30	0.6300818	0.0775065	0.0775067
40	0.1077115	0.0162020	0.0161219
50	0.0002839	0.9009089	0.9999786
51	0.0011354	0.3570883	0.0000854

With just three operations per cycle, this series has even double precision producing rubbish after just 150 elementary operations.

Making Life Complex

Processors deal with real numbers only. Many scientific problems are based on complex numbers. This leads to major problems.

Addition is simple

$$(a + ib) + (c + id) = (a + c) + (b + d)i$$

and subtraction is similar.

Multiplication is slightly tricky:

$$(a + ib) * (c + id) = (ac - bd) + (bc + ad)i$$

What happens when $ac - bd$ is less than the maximum number we can represent, but ac is not?

What precision problems do we have if ac is approximately equal to bd?

The Really Complex Problem

$$(a + ib)/(c + id) = \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2}$$

This definition is almost useless!

If N is the largest number we can represent, then the above formula will produce zero when dividing by any number x with $|x| > \sqrt{N}$.

Similarly, if N is the smallest representable number, it produces infinity when dividing by any x with $|x| < \sqrt{N}$.

This is not how languages like Fortran, which support complex arithmetic, do division: they use a more complicated algorithm which we shall quietly ignore.

Hard or Soft?

The simple operations, such as $+$, $-$ and $*$ are performed by dedicated pipelined pieces of hardware which typically produce one result each clock cycle, and take around four clock cycles to produce a given result.

Slightly more complicated operations, such as $/$ and $\sqrt{}$ may be done with $microcode$. Microcode is a tiny program on the CPU itself which is executed when a particular instruction, e.g. $/$, is received, and which may use the other hardware units on the CPU multiple times.

Yet more difficult operations, such as trig. functions or logs, are usually done entirely with software in a library. The library uses a collection of power series or rational approximations to the function, and the CPU needs evaluate only the basic arithmetic operations.

The IA32 range is unusual in having microcoded instructions for trig. functions and logs. Even on the PentiumIII and Pentium4, a single such instruction can take over 200 clock cycles to execute. RISC CPUs tend to avoid microcode.

Soft denormals

```
x=1d-20
y=1d-28
n=1e8

do i=1,n
  x=x+y
enddo
```

This yields x=2E-20 in under half a second on a 466MHz EV6. If x and y are scaled by dividing by 2^{955} before the loop, and multiplied by the same factor afterwards, the loop takes 470s.

The EV6 hardware cannot handle denormals, so software emulation was used for each addition. Ditto most other RISC CPUs.

With the default compiler flags, Alphas flush denormals to zero, and thus get an answer of x=1E-20 in under a quarter of a second after scaling. Full IEEE compliance costs a factor of two anyway, and over a factor of 2000 with denormals present.

Dividing slowly

The simplest way for a CPU designer to impliment division is to use the long division process familiar to all(?) school children. This requires only shifts, subtractions and comparisons, but is horribly slow. Each iteration of the algorithm yields one more bit of the quotient, and there are 52 to find. . .

The Intel 486 used a variation of this technique, and like all IA32 Intel processors it worked to 64 bits of precision internally. It is not too surprising it took 73 clock cycles to do a division!

The Pentium used a more sophisticated variation, effectively doing the calculations in base 4 so that two bits of the quotient were produced at every step. This reduced the time to a 'mere' 38 clock cycles, or twice as slow as any serious CPU. The Pentium II is no better.

Conquering division

If one has a fast hardware multiplier, there is a better way to do division: one can use a Newton-Raphson like iterative algorithm.

$$a/b = a * 1/b$$

$$x_{n+1} = 2 * x_n - b * {x_n}^2$$

and, for reasonable starting guesses, this series will converge to $1/b$. As an example with $b = 6$.

n	x_n
0	0.2
1	0.16
2	0.1664
3	0.16666624
4	0.166666666655744

Division in more detail

Returning to our base-10 example, let us consider the initial guess for the reciprocal.

$$0.XXXX * 10^{XX}$$

the important parts are the exponent and the first digit of the mantissa:

$$0.A * 10^{X}$$

and the reciprocal is (approximately)

$$0.B * 10^{1-X}$$

where B is chosen from a table such as

A	B
1	7
2	4
3	3
4,5,6	2
7,8,9	1

e.g. given $0.6 * 10^{1}$, we guess $0.2 * 10^{0}$.

Converging

Applying the above iterative formula four times to any guess is sufficient to converge to the four significant figures we are using. As the convergence is exponential, converging to 10 or 20 significant figures would not be hard.

For simplicity, we can choose to iterate for a fixed number of steps, rather than checking the convergence ever.

Most processors use a similar lookup table and convergent series combination. The bigger the table, the fewer iterations are required. Division typically takes around 20 clock cycles, enough time for about 4 iterations through the a loop like the above.

Square roots

A similar approach to that for division is taken for square roots, and often reciprocal square roots too. The latter operation is common in graphics (normalising vectors), so a special instruction is often supplied which is about twice as fast as doing the square root and reciprocal separately. Again it takes around 20 clock cycles to extract a square root.

And again it is possible to do it without multiplications, as shown by Intel whose Pentium takes typically 70 cycles and 486 over 80 for a square root.

Memory

- Memory technologies
- Parity and ECC
- Going faster: wide bursts
- Going faster: caches

Memory Technologies

ROM Read Only Memory: contents set at fabrication and unchangeable.

PROM Programable ROM: contents written once electronically.

EPROM Erasable PROM: contents may be erased using UV light, then written once.

EEPROM Electronically EPROM: contents may be erased electronically a few hundred times.

RAM Random Access Memory: contents may be read and changed with 'equal' ease.

DRAM Dynamic RAM: the cheap and common flavour. Contents lost if power lost.

SRAM Static RAM: contents may be retained with a few μW.

An EPROM with no UV window is equivalent to a PROM. A PROM once written is equivalent to a ROM. SRAM with a battery is equivalent to EEPROM.

Most 'ROMs' are some form of EEPROM so they can have their contents upgraded without physical replacement: also called $Flash\ RAM$, as the writing of an EEPROM is sometimes called *flashing*.

RAM

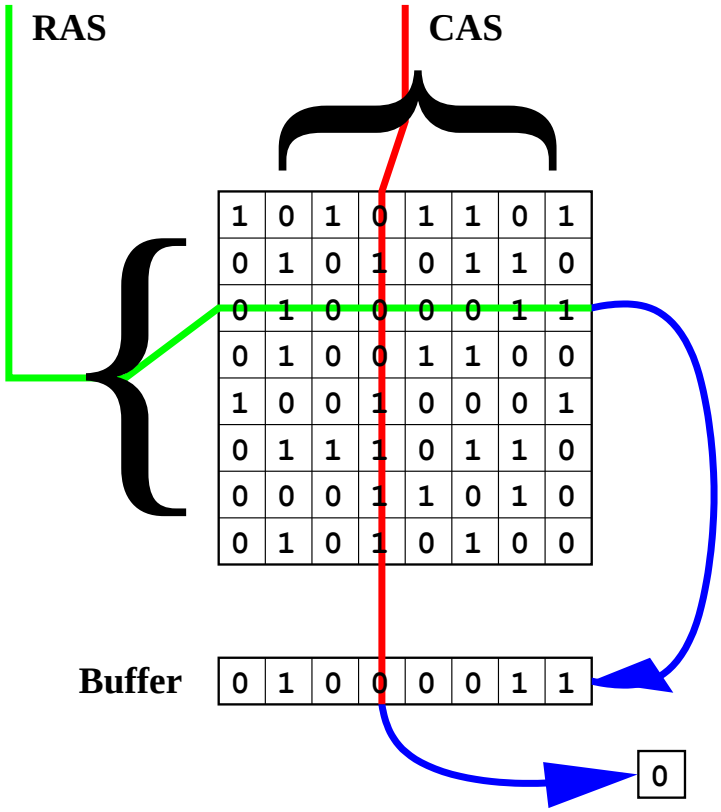
A typical DRAM cell consists of a single capacitor and field effect transistor. It stores a single bit, and has barely changed since 1974.

The slow leak of charge onto or off the capacitor is accounted for by refreshing the memory periodically (thousands of times a second). The bits are simply read, then written back. This refreshing is usually done by circuitary on the motherboard, rather than the memory module or the CPU.

Conversely SRAM has four or six transistors per bit, and needs no refreshing.

SRAM comes in two flavours: that optimised for low-power data retention (pocket diaries), and that optimised for speed (cache). DRAM's requirement for refreshing leads to a very much higher power consumption when idle than SRAM.

DRAM in Detail



DRAM cells are arranged in (near-)square arrays. To read, first a row is selected and copied to a buffer, from which a column is selected, and the resulting single bit becomes the output. This example is a 64 bit DRAM.

This chip would need 3 *address lines* (i.e. pins) allowing 3 bits of address data to be presented at once, and a single `data line`. Also two pins for power, two for CAS and RAS, and one to indicate whether a read or a write is required.

Of course a 'real' DRAM chip would contain several tens of million bits.

DRAM Read Timings

To read a single bit from a DRAM chip, the following sequence takes place:

- Row placed on address lines, and Row Access Strobe pin signalled.
- After a suitable delay, column placed on address lines, and Column Access Strobe pin signalled.
- After another delay the one bit is ready for collection.
- The DRAM chip will automatically write the row back again, and will not accept a new row address until it has done so.

The same address lines are used for both the row and column access. This halves the number of addess lines needed, and adds the RAS and CAS pins.

Reading a DRAM cell causes a significant drain in the charge on its capacitor, so it needs to be refreshed before being read again.

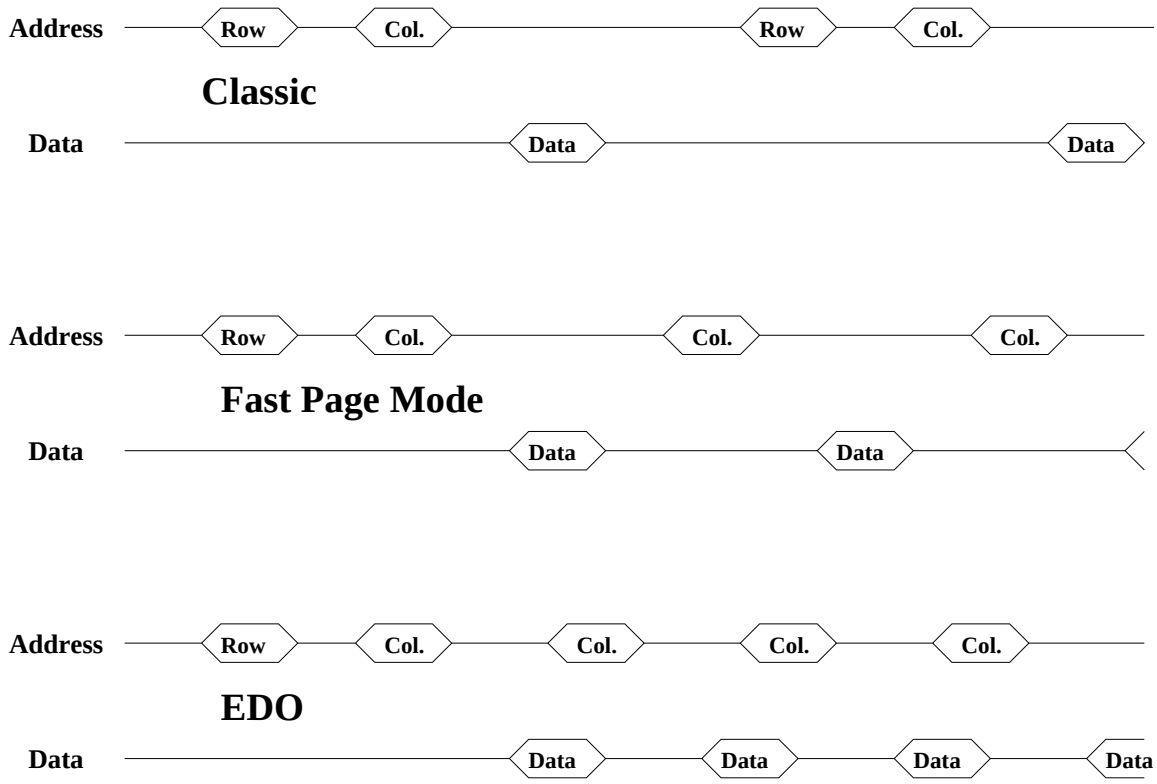
More Speed!

The above procedure is tediously slow. However, for reading consecutive addresses, one important improvement can be made.

Having copied a whole row into the buffer (which is usually SRAM), if another bit from the same row is required, simply changing the column address whilst signalling the CAS pin is sufficient. There is no need to wait for the chip to write the row back, and then to rerequest the same row. Thus Fast Page Mode (FPM) DRAM.

Extended Data Out (EDO) is similar, but allows the column address for the next read to be given whilst the data from the previous read are being read.

DRAM Timings compared



The first two lines show 'classic' DRAM, with both row and column addresses being sent before each data transfer. The next two lines show FPM, with the row addresses being omitted after the first. Finally EDO overlaps the column addresses with the data.

Time increases left to right in the above diagrams.

Measuring time

The three graphs show memory of identical $latencies$, that is, time from the start of a new read to the data being available.

They also have identical $cycle\ times$, that is, the time from the start of one read, to the start of another unrelated read.

They have very different $bandwidths$, that is, how rapidly data streams out when requested sequentially. A group of sequential reads is often called a $burst$.

Classic, FPM and EDO memory is sold on latency (50-80ns typically), although the time between data items in a burst is less than half the latency for EDO.

SDRAM

SDRAM is a major current memory technology. It is advertised as being very fast: introduced at 66MHz (15ns), soon moving to 100MHz (10ns) then 133MHz (7.5ns). Much faster than that 60ns EDO stuff current when SDRAM was first introduced?

Not really. 'Headline' SDRAM times are those between data items in a burst, and EDO was already down to 25ns for this. SDRAM latencies are typically four to five clock cycles, so 100MHz SDRAM has a similar latency to 50ns EDO. SDRAM is optimised for bursts: during a burst the CPU does not have to keep changing the column address, the data are sent automatically.

As SDRAM is usually sold in modules which output 64 bits at once, PC100 SDRAM (100MHz) has a peak bandwidth of 800MB/s.

The 'S' in SDRAM stands for synchronous: the bus has a clock and all signals on it are synchronised to the clock pulses.

DDR-SDRAM is similar, except that data are transfered twice each clock-cycle, thus doubling the bandwidth, and not touching the latency. It is rated on bandwidth, not clock speed, so 133MHz DDR-SDRAM calls itself 266MHz (sic) or PC2100.

RAMBUS

The RDRAM vs (DDR-)SDRAM debate is too complex to be presented here, except to say that RAMBUS provides yet another way of getting at a DRAM core which is, in essence, very similar to that on a SDRAM or EDO RAM chip.

RDRAM uses a very fast (400MHz or 533MHz) bus, over which data is transfered twice each clock cycle. The bad news is that the data bus is only 16 bits wide, so PC800 RDRAM, which runs at 400MHz, has a peak data rate of 1600MB/s. Intel's Pentium4 chipsets take RDRAM modules in pairs, to give a theoretical 3.2GB/s with PC800 parts.

RDRAM is not magic: its latency is still around 40ns.

The double data rate 533MHz bus RDRAM uses is fairly close to magic. The issues of path length and termination become quite exciting, and it uses *clock forwarding* (different clock pulses for data travelling in different directions), because Einstein has difficulty synchronising things any other way. (The speed of light is just one foot per nanosecond in vacuum.)

Packaging it up

DRAM is usefully sold in small, plug-in modules:

30pin SIMMs: obsolete, bus width 8 or 9 bits, capacity up to 16MB.

72pin SIMMs: obsolete, bus width 32 or 36 bits, capacity up to 64MB, plain, FPM or EDO.

168pin DIMMs: bus width 64 or 72 bits, SDRAM.

184pin RIMMs: bus width 16 or 18 bits, RDRAM.

SIMM: Single In-line Memory Module (the contacts on each side of the module are equivalent)
DIMM: Dual In-line Memory Module (84 distinct contacts on each side for 168pin)
RIMM: Rambus In-line Memory Module

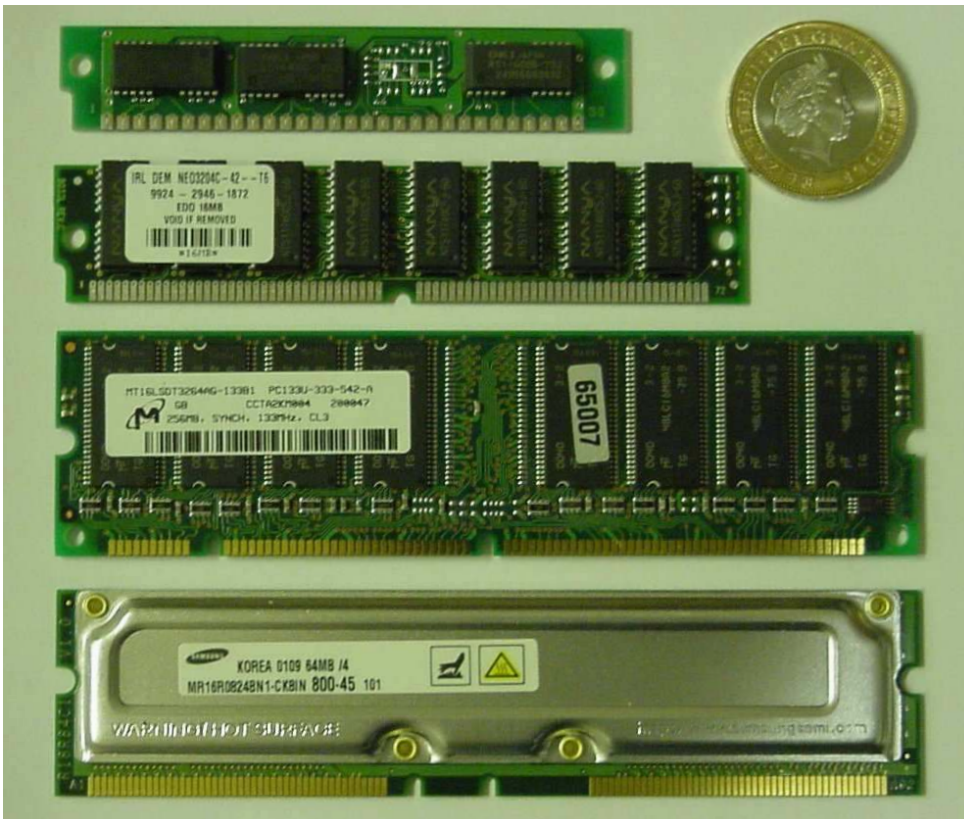
The SIMMs have just 12 address lines, limiting capacity to 2^{24} words.

The individual chips on a SIMM used to supply one bit each, so there would be eight on an 8-bit SIMM. Later chips supplied several (e.g. four) bits each, but internally still had one bit per array of cells. More recent DRAM chips have four or eight bits per cell.

Modules must be installed in groups sufficient to fill the width of the bus being used: four 30pin SIMMs for a 32 bit bus, two 72pin SIMMs for a 64 bit bus, etc.

DIMMs and RIMMs are capable of identifying themselves to the motherboard using a protocol called Serial Presence Detect. This reveals capacity, speed and type, and allows the motherboard to configure itself appropriately.

Modules in Pictures



30 pin SIMM 1MB, 70ns, parity. Two of the chips are identical supply four bits each, the third supplies one bit to make up the nine.

72 pin SIMM 16MB, 60ns, EDO, 32 bit, four bits from each of eight chips.

168 pin DIMM 256MB, 133MHz, 64 bit. Eight more chips on the other side of the module.

184 pin RIMM 64MB, 16 bit, PC800. Note the metal heat-spreader to assist cooling.

Notches prevent misinsertion. The two-pound coin is 28mm in diameter.

Other forms of memory include 144 pin SO-DIMMs (Small Outline DIMM) used in laptops.

Speed Required

A typical CPU runs at, say, 750 MHz. It can perform a double precision floating-point addition and multiplication each clock-cycle, which is four pieces of data in, and two out, so potentially six memory references.

So a latency of about 0.2 ns, maybe 2 ns if latencies can be overlapped, and a bandwidth of 36 GB/s, are needed.

The fastest SDRAM chips currently available, 133MHz DDR-SDRAM, have a latency of 40 ns, and a peak bandwidth of 4GB/s if configured in a 128 bit bus. Thus the bandwidth is disappointing, and the latency dreadful.

And it gets worse: other necessary control chips between the CPU and the RAM chips increase the latency (by over a factor of two) and reduce the bandwidth.

The lowest latency machine in TCM (2003) has a latency to main memory of 100ns.

More Conservative

Maybe somewhat less performance is actually needed. Consider a dot product (a common operation). At each step, two elements are loaded, a multiplication and an addition done, and nothing stored, for the total will be kept in a register.

In other words, two memory references, not six, so 'only' 12 GB/s of memory bandwidth needed.

However, those 667 MHz EV6 based Alphas in TCM have an achievable memory bandwidth of about 1GB/s, so a dot product will achieve a mere 125 MFLOPS, or 10% of the CPU's theoretical MFLOPS rating.

Vector computers are somewhat different. The Hitachi S3600 which used to be in Cambridge had a peak performance of 2 GFLOPS, but a memory bandwidth of over 12 GB/s.

Wider Still and Wider

One obvious bandwidth enhancer is to increase the width of the memory bus. PCs have certainly done this: 8086 and 286 used 16 bit memory buses, 386 and 486 used 32 bits and the Pentium and above 64 bits. 'Real' workstations tend to use 128 or even 256 bit buses.

There is an obvious problem: a 256 bit bus does require 256 tracks leading to 256 different pins on the CPU, and going beyond this gets really messy. Commodity memory modules provide just 64 bits each, so they must be added in fours for a 256 bit bus.

It also fails to address the latency issue.

Bus widths of things currently in TCM:
PentiumIII: 64 bit
Alpha (XP900): 128 bit
Alpha (XP1000): 256 bit

Caches: the Theory

The theory of caching is very simple. Put a small amount of fast, expensive memory in a computer, and arrange automatically for that memory to store the data which are accessed frequently.

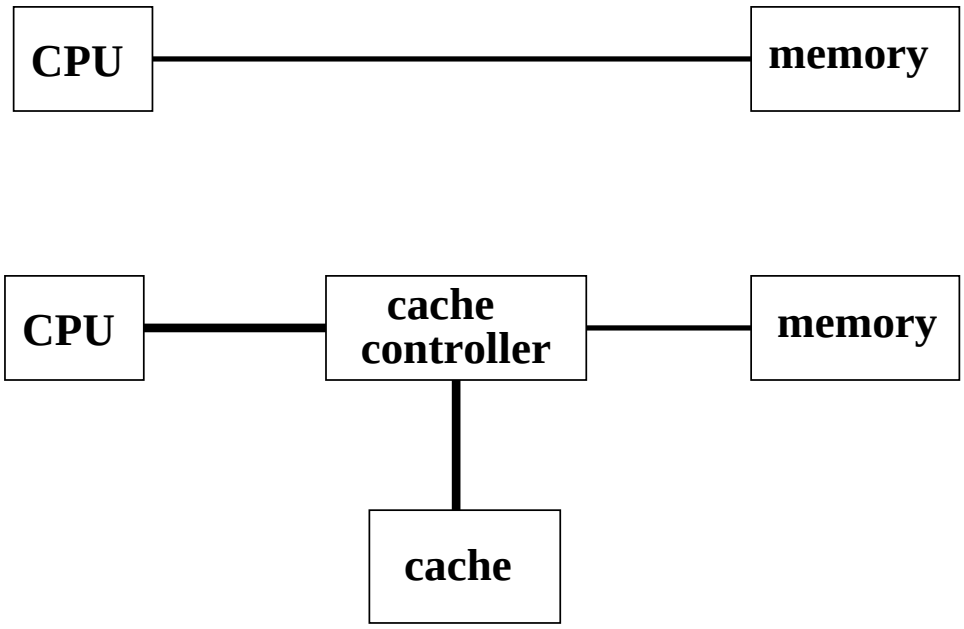
One can then define a cache $hit\ rate$, that is, the number of memory accesses which go to the cache divided by the total number of memory accesses. This is usually expressed as a percentage.

One of the reasons for the expense of the fast SRAM used in caches is that it requires around six transistors per bit, not one.

The first paper to describe caches was published in 1965 by Maurice Wilkes (Cambridge). The first commercial computer to use a cache was the IBM 360/85 in 1968.

Caches: the Problem

Main memory may be slow compared to a CPU, but it is not that slow. Any cache control logic must be very fast (and therefore very simple) or there will be no improvement in speed.



A badly designed cache controller can be worse than no cache at all.

The Anomalies

For a (particularly horrible) program, the following run-times are observed:

Processor	Speed (MHz)	Time (s)
Pentium	100	33.0
Pentium	90	22.4
Pentium	120	21.7 and 14.1
Pentium	133	12.7
21064 (EV4)	266	6.0
486DX2	66	3.5
Pentium MMX	166	1.1
21264 (EV5)	500	0.75
PentiumII	350	0.25

The Cache Controller

Conceptually this has a simple task:

- Intercept every memory request
- Determine whether cache holds requested data
- If so, read data from cache
- If not, read data from memory *and* place a copy in the cache as it goes past.

However, the second bullet point must be done $very$ fast, and this leads to the compromises.

An aside: Hex

A quick lesson in hex (base-16 arithmetic) is due at this point. Computers use base-2, but humans tend not to like reading long base-2 numbers.

Humans also object to converting between base-2 and base-10.

However, getting humans to work in base-16 and convert between base-2 and base-16 is easier.

Hex uses the letters A to F to represent the 'digits' 10 to 15. As $2^4 = 16$ conversion to and from binary is done trivially using groups of four digits.

Converting to/from Hex

```
0101 1101 0010 1010 1111 0001 1100 0011

 5    C    2    A    F    1    B    3
```

So

$$01011101001010101111000111000011_2$$

$$= 5\mathrm{C2AF1B3}_{16} = 1546318259$$

As one hex digit is equivalent to four binary digits, two hex digits are exactly sufficient for one byte.

Hex numbers are often prefixed with '`0x`' to distinguish them from base ten.

When forced to work in binary, it is usual to group the digits in fours as above, for easy conversion into hex or bytes.

Our Computer

For the purposes of considering caches, let us consider a computer with a 1MB address space and a 64KB cache.

An address is therefore 20 bits long, or 5 hex digits.

Suppose we try to cache individual bytes. Each entry in the cache must store not only the data, but also the address in main memory it was taken from, called the tag. That way, the cache controller can look through all the tags and determine whether a particular byte is in the cache or not.

So we have 65536 single byte entries, each with a $2\frac{1}{2}$ byte tag.

tag	**data**

A Disaster

This is bad on two counts.

A waste of space

We have 64KB of cache storing useful data, and 160KB storing tags.

A waste of time

We need to scan 65536 tags before we know whether something is in the cache or not. This will take far too long.

Lines

The solution to the space problem is not to track bytes, but *lines*. Consider a cache which deals in units of 16 bytes.

```
64KB = 65536 * 1 byte
     =  4096 * 16 bytes
```

We now need just 4096 tags.

Furthermore, each tag can be shorter. Consider a random address:

`0x23D17`

This can be read as byte `7` of line `23D1`. The cache will either have all of line `23D1` and be able to return byte number 7, or it will have none of it.

Getting better. . .

A waste of space?

We now have 64KB storing useful data, and 8KB storing tags. Considerably better.

A waste of time

Scanning 4096 tags may be a 16-fold improvement, but is still a disaster.

Causing trouble

Because the cache can store only full lines, if the processor requests a single byte which the cache does not hold, the cache then requests the full line from the memory so that it can keep a copy of the line. Thus the memory might have to supply $16\times$ as much data as before!

A Further Compromise

We have 4096 tags, potentially addressable as tag 0 to tag 0xFFF.

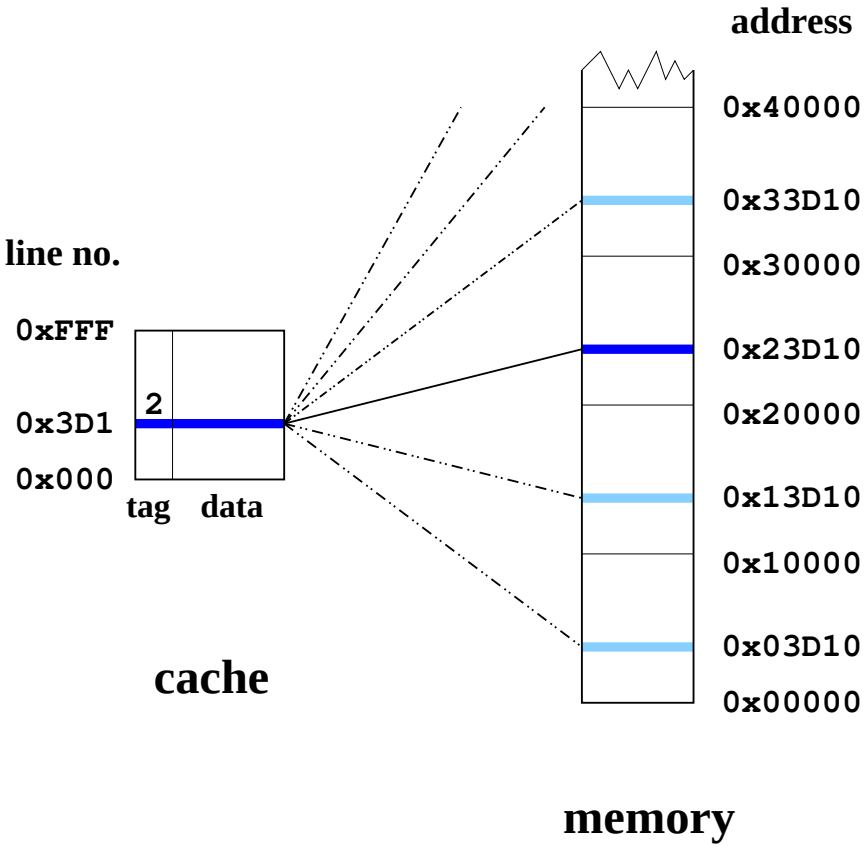
On seeing an address, e.g. `0x23D17`, we discard the last 4 bits, and scan all 4096 tags for the number `0x23D1`.

Why not always use line number `0x3D1` within the cache for storing this bit of memory? The advantage is clear: we need only look at one tag, and see if it holds the line we want `0x23D1`, or one of the other 15 it could hold: `0x03D1`, `0x13D1`, etc.

Indeed, the new-style tag need only hold that first hex digit, we know the other digits! This reduces the amount of tag memory to 2KB.

Direct Mapped Caches

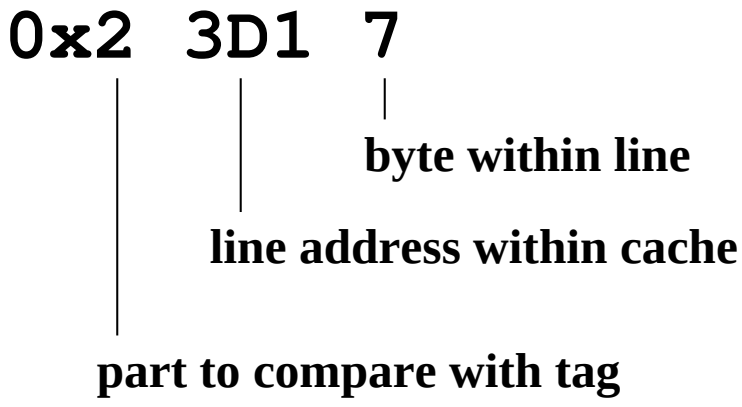
We have just developed a *direct mapped* cache. Each address in memory maps directly to a single location in cache, and each location in cache maps to multiple (here 16) locations in memory.



Success?

- The overhead for storing tags is 3%. Quite acceptable, and much better than 250%!
- Each 'hit' requires a tag to be looked up, a comparison to be made, and then the data to be fetched. Oh dear. This $tag\ RAM$ had better be *very* fast.
- Each miss requires a tag to be looked up, a comparison to fail, and then a whole line to be fetched from main memory.
- The 'decoding' of an address into its various parts is instantaneous.

The zero-effort address decoding is an important feature of all cache schemes.



The Consequences of Compromise

At first glance we have done quite well. Any contiguous 64KB region of memory can be held in cache. (As long as it starts on a cache line boundary)

E.g. The 64KB region from `0x23840` to `0x3383F` would be held in cache lines `0x384` to `0xFFF` then `0x000` to `0x383`

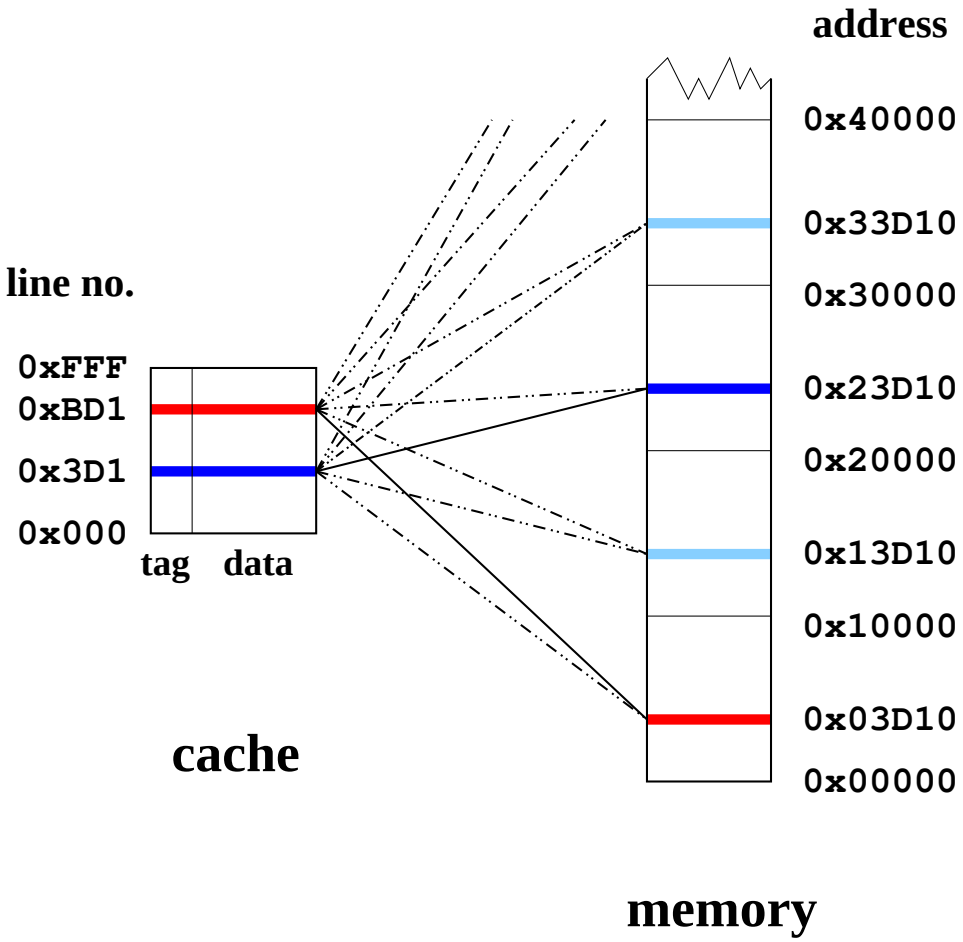
Even better, widely separated pieces of memory can be in cache simultaneously. E.g. `0x15674` in line `0x567` and `0xC4288` in line `0x428`.

However, consider trying to cache the two bytes `0x03D11` and `0x23D19`. This cannot be done: both map to line `0x3D1` within the cache, but one requires the memory area from `0x03D10` to be held there, the other the area from `0x23D10`.

Repeated accesses to these two bytes would cause cache *thrashing*, as the cache repeatedly caches then throws out the same two pieces of data.

Associativity

Rather than each line in memory being storable in just one location in cache, why not make it two?



Now we have a *2 way (set) associative* cache.

An n-way associative cache has n possible places for storing each location in memory, needs to read n tags to check whether something is in the cache, and needs $\log_2 n$ extra tag bits to keep track of things.

Victim Caches

Victim Caches, or Anti Thrashing Entries, are a cheap way of increasing the effective associativity of a cache. One extra cache line, complete with tag, is stored, and it contains the last line expelled from the cache proper.

This line is checked for a 'hit' in parallel with the rest of the cache, and if a hit occurs, it is moved back into the main cache, and the line it replaces is moved into the ATE.

Some caches have several ATEs, rather than just one.

```
double precision a(2048,2),x
do i=1,2048
  x=x+a(i,1)*a(i,2)
enddo
```

Assume a 16K direct mapped cache with 32 byte lines. `a(1,1)` comes into cache, pulling `a(2-4,1)` with it. Then `a(1,2)` displaces all these, at it must be stored in the same line, as its address modulo 16K is the same. So a(2,1) is not found in cache when it is referenced. With a single ATE, the cache hit rate jumps from 0% to 75%, the same that a 2-way set associative cache would have for this algorithm.

Policies: Write Back vs Write Through

Should data written by the CPU modify merely the cache if those data are currently held in cache, or modify the memory too? The former, $write\ back$, can be faster, but the latter, $write\ through$, is simpler.

With a write through cache, the definitive copy of data is in the main memory. If something other than the CPU (e.g. a disk controller or a second CPU) writes directly to memory, the cache controller must $snoop$ this traffic, and, if it also has those data in its cache, update (or invalidate) the cache line too.

Write back caches add two problems. Firstly, anything else reading directly from main memory must have its read intercepted if the cached data for that address differ from the data in main memory.

Secondly, on ejecting an old line from the cache to make room for a new one, if the old line has been modified it must first be written back to memory.

Each cache line therefore has an extra bit in its tag, which records whether the line is modified, or $dirty$.

Policies: Allocate on Write

If a cache is write-back, and a write occurs which is a cache miss, should the cache line be filled? For the corresponding read event, the answer would always be 'yes', otherwise the cache would never be used!

If the data just written are read again soon afterwards, filling is beneficial, as it is if a write to the same line is about to occur. However, caches which allocate on writes perform badly on randomly scattered writes.

Each write of one word is converted into $reading$ the cache line from memory, modifying the word written in cache and marking the whole line dirty. When the line needs discarding, the whole line will be written to memory. Thus writing one word has be turned into two lines worth of memory traffic.

The PentiumPro allocates on writes, and the Pentium did not. Certain codes therefore ran slower on the otherwise-faster PentiumPro.

A partial solution to this problem is to break a line into equal sub-blocks, each with its own dirty bit. If only one sub-block has been modified, just that sub-block is written back to memory when the line is discarded. This is useful even for caches which do not allocate on writes.

Policies: LRU vs Random replacement

With an n-way associative cache, when caching a new line, in which of the n possible locations in the cache should it be placed? A randomly-chosen line can be ejected, or the Least Recently Used.

LRU is easy for a 2-way associative cache: it is simply not the most recently used, and a single bit shared by the two ways can keep track of which was most recently accessed.

For higher associativities, a pseudo-LRU algorithm is often used: random, but excluding most recently used line. This reduces book-keeping.

As should now be clear, not all caches are equal!

Write Buffers

It is usual for a CPU to provide a small number of write buffers. These store data due to be written to the main memory, but by performing the writes by preference when the memory system is otherwise idle, they can permit reads to continue without stalling the CPU until the data are written.

Naturally all reads are checked against writes pending in these buffers for consistency, otherwise a write followed by an immediate read from the same address would get the old data from memory, not the new data which was overtaken whilst it was in the write buffer.

These buffers also $collapse$ writes. Two 32-bit writes to consecutive locations will be issued as a single transaction on a 64-bit bus, if possible, etc.

A decent write buffer can produce a significant fraction of the gain of a write-back cache. However, a write-back cache also needs a write buffer, so that writes caused by ejecting modified cache lines can be delayed if necessary to reduce conflicts with reads.

Not All Data are Equal

If the cache controller is closely associated with the CPU, it can distinguish memory requests from the instruction fetcher from those from the load/store units. Thus instructions and data can be cached separately.

This almost universal $Harvard\ Architecture$ prevents poor data access patterns leaving both data and program uncached.

The term 'Harvard architecture' comes from an early American computer which used physically separate areas of main memory for storing data and instructions. No modern computer does this.

A Hierarchy

The speed gap between main memory and the CPU core is so great that there are usually multiple levels of cache.

The first level, or *primary cache*, is small (typically 16KB to 128KB), physically attached to the CPU, and runs as fast as possible.

The next level, or *secondary cache*, is larger (typically 256KB to 8MB), and usually placed separately on the motherboard. In some systems it is completely independent of the CPU.

Typical times in clock-cycles to serve a memory request would be:

primary cache	1-3
secondary cache	5-25
main memory	30-300

Cf. functional unit speeds on page 50.

Intel tends to make small, fast caches, compared to RISC workstations which tend to have larger, slower caches. Some machines have *tertiary caches* too.

Cache size

The logic core of a CPU is typically between 1 million and 30 million transitors. A cache which runs as fast as the CPU core, must be approximately the same size, for the core will run as fast as possible, and bigger things tend to be slower due to thermal problems, electrical capacitance, and the finite speed of light.

As one bit in SRAM requires six transistors, and extra bits for tag and parity will be needed, fifty transistors per byte of cache is probably an underestimate. So a 256KB cache requires over 12 million transistors, and a 1MB cache would be bigger than any CPU core.

Thus most CPUs have around 128KB of primary cache, and also the tags for the secondary cache, on the same die as the core. Much more, and it becomes mostly cache.

The bulk of the secondary cache is normally physically separate, and running at a half or a third of the speed of the CPU to reduce power consumption.

One million transistors for an i486 core (1989). Around thirty million for an IBM Power4 (2002).

Line Size

Whenever the CPU wants a single byte which is not in the cache, the cache controller will fetch a whole line.

This is good if the CPU then requests the following item from memory: it is probably now in cache.

This is bad if the CPU is jumping randomly around: the cache will cause unnecessary memory traffic.

As current DRAM is so much faster at consecutive accesses than random accesses, filling a cache line which is four or even eight times the width of the data bus takes only about twice as long as filling one the same size as the data bus.

For a fixed total size of cache, doubling the line size halves the number of tags required, and reduces the tag length by one bit too. The UltraSPARC III Cu procesor has 16,384 tags for its secondary cache, and a line size of 64, 256 or 512 bytes depending whether the cache is 1MB, 4MB or 8MB in size. The longer lines are broken into sub-blocks of 64 bytes with independent 'dirty' and 'valid' bits.

Explicit Prefetching

One spin-off from caching is the possibility of *prefetching*.

Many processors have an instruction which requests that data be moved from main memory to primary cache when it is next convenient.

If such an instruction is issued ahead of some data being required by the CPU core, then the data may have been moved to the primary cache by the time the CPU core actually want them. If so, much faster access results. If not, it doesn't matter.

If the latency to main memory is 100 clock cycles, the prefetch instruction ideally needs issuing 100 cycles in advance, and many tens of prefetches might be busily fetching simultaneously. Most current processors can handle a couple of simultaneous prefetches. . .

Implicit Prefetching

Some memory controllers are capable of spotting certain access patterns as a program runs, and prefetching data automatically. Such prefetching is often called *streaming*.

The degree to which patterns can be spotted varies. Unit stride is easy, as is unit stride backwards. Spotting different simultaneous streams is also essential, as a simple dot product:

```
do i=1,n
  d=d+a(i)*b(i)
enddo
```

leads to alternate unit-stride accesses for `a` and `b`.

IBM's Power3 processor, and Intel's Pentium 4 both spot simple patterns in this way. Unlike software prefetching, no support from the compiler is required, and no instructions exist to make the code larger and occupy the instuction decoder. However, streaming is less flexible.

That Code

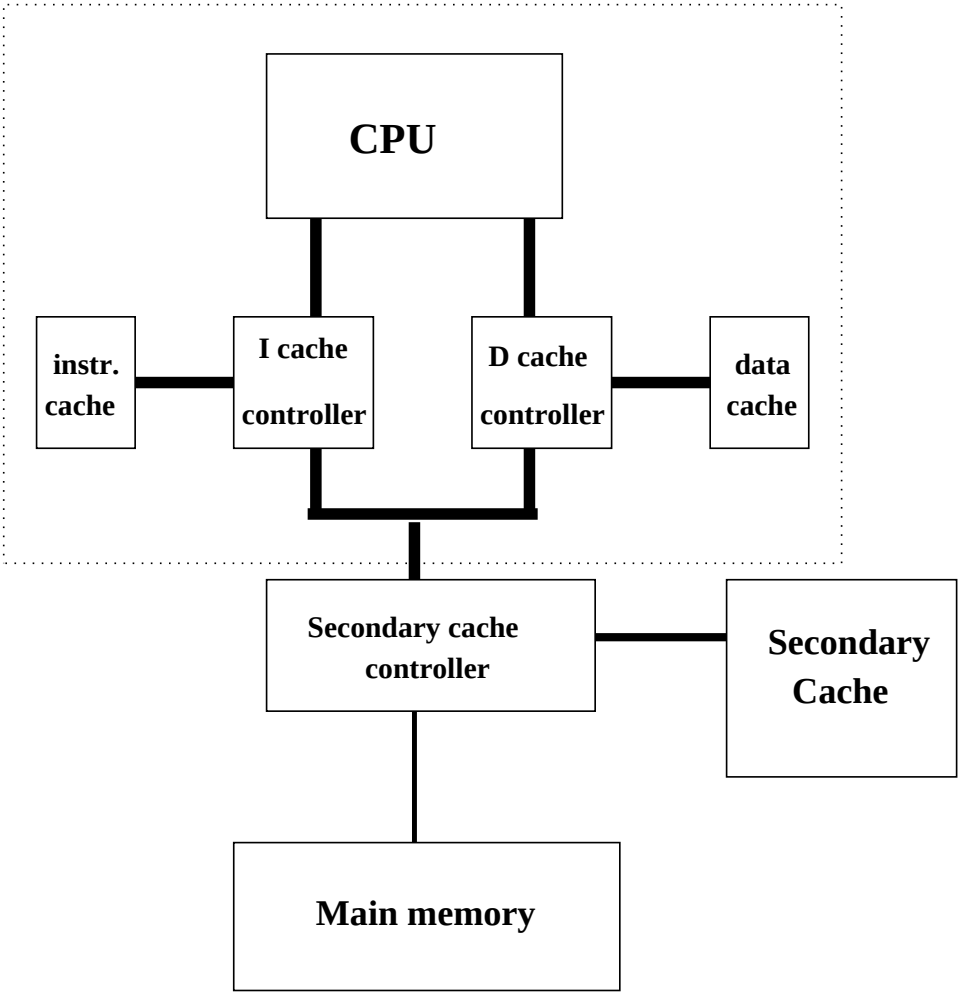
```
unsigned char* a;
for(j=0;j<10001;j++)
  for(i=0;i<2048;i++)
    a[i]+=a[i+8192]+a[i+16384]+a[i+24576];
```

Processor	Primary Data Cache		Time
	Size	Assoc	(s)
Pentium	8KB	2	12.7 to 33.0
21064A (EV4)	16KB	1	6.0
486DX2	8KB	4	3.5
Pentium MMX	16KB	4	1.1
21164 (EV5)	8KB+96KB	1+3	0.75
Pentium II	16KB	4	0.25

The EV5 has a fast 96KB secondary cache in the CPU as well as the 8KB cache.

The above code can be cached in a 32KB direct mapped cache, 16KB 2 way associative, or 8KB 4 way associative.

Big and Free


CPU
instr. cache
I cache controller
D cache controller
data cache
Secondary cache controller
Secondary Cache
Main memory

Clock multiplying

If the CPU has to fetch all its data and instructions from memory external to it, there is no point in the CPU core running faster than that bus.

If the CPU includes a cache, then, for some code, no external memory references will be needed. The CPU is no longer limited by the speed of the external bus.

So one runs the CPU faster than the external bus.

CPU	bus	core	ratio
486DX4	33	100	3
Pentium	66	200	3
Pentium III	133	1000	$7\frac{1}{2}$
Pentium4	533	3066	$5\frac{3}{4}$
21064 (EV4)	39	275	7
21164 (EV5)	83	500	6
21264 (EV6)	333	500	$1\frac{1}{2}$
21264C (EV6.8)	500	1250	$2\frac{1}{2}$

Limits to Clock Multiplying: Thermal

The heat produced by a CPU is proportional to its clock speed. Cooling is a major limiting factor.

Once the CPU gets too hot, thermally excited carriers begin to swamp the intrinsic carriers introduced by the n and p doping. With the low band-gap of silicon, the maximum junction temperature is around $90°$C, or just $50°$C above the air temperature which most computers can allegedly survive.

Current techniques allow around 100W to be dissipated from a chip with forced air cooling.

As far as power consumption is concerned, a chip looks like a capacitor which is charged, then discharged, through a resistor at a frequency f. Each data line or storage cell acts as a tiny capacitor, whose charge state changes every time its logic state changes. The power consumption is thus fCV^2.

'Shrinking' a processor, by using more advanced fabrication techniques with smaller features, reduces C and permits V to be reduced. Hence the 5V used for processor cores in the late 1980s has dropped to about 2V today (2002), and the feature size on the die has dropped from 1μm to under 0.2μm over the same period.

Limits to Clock Multiplying: Cache Misses

Unless the primary cache hit rate is well over 90%, it is pointless considering multipliers of more than three.

If the hit rate is large, then the CPU core is effectively decoupled from the rest of the motherboard. If low, it will spend much time waiting for data to be provided to it. Large, highly associative, write-back caches help to reduce this problem

One way around this problem is to speed up the secondary cache. One could put its controller and maybe its tag RAM on the CPU module, and use a separate bus to the external cache memory. That bus might be wider or faster than the main memory bus.

The EV6 Alpha, UltraSPARC III, Pentium II and Power 3 all have their secondary cache controllers on die, and a separate faster bus (i.e. with a lower multiplier) for cache traffic.

Which is faster, a 133MHz core with the external cache and other external features running at 66MHz (or 33MHz), or a 150MHz core with the external cache and other external features running at 60MHz (or 30MHz)? The 12% increase in core speed can be completely offset by the 9% decrease in the speed of everything else. See the 133MHz and 150MHz Pentiums for more details.

The Relevance of Theory

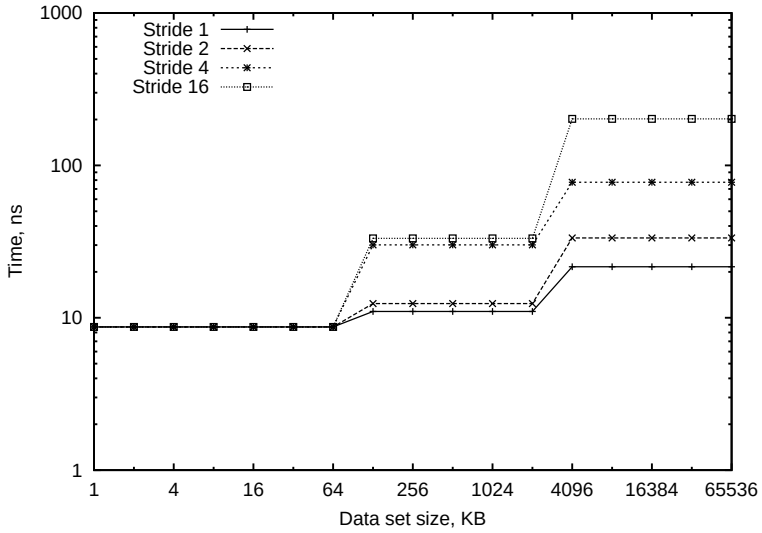
```
integer a(*),i,j

j=1
do i=1,n
  j=a(j)
enddo
```

This code is mad. Every iteration depends on the previous one, and significant optimisation is impossible.

However, the memory access pattern can be changed dramatically by changing the contents of `a`. Setting `a(i)=i+1` and `a(k)=1` will give consecutive accesses repeating over the first `k` elements, whereas `a(i)=i+2`, `a(k-1)=2` and `a(k)=1` will access alternate elements, etc.

Classic caches

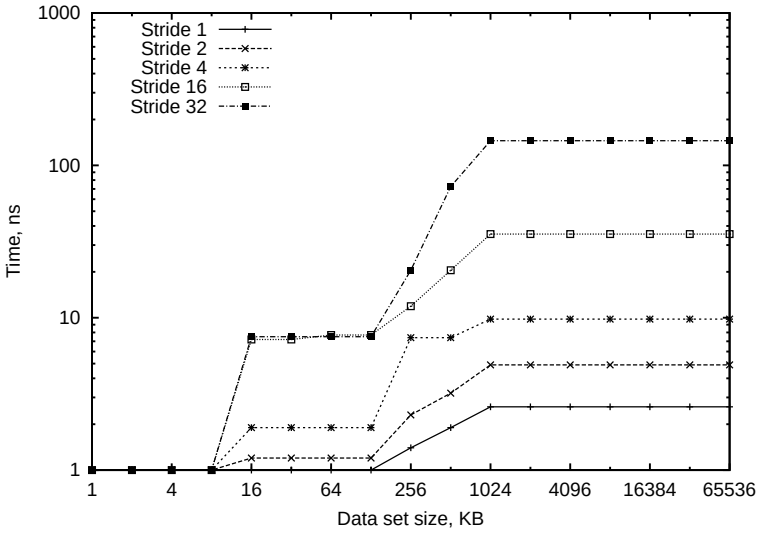


With a 16 element (64 bytes) stride, we see access times of 8.7ns for primary cache, 33ns for secondary, and 202ns for main memory. The cache sizes are clearly 64KB and 2MB.

With a 1 element (4 bytes) stride, the secondary cache and main memory appear to be faster. This is because once a cache line has been fetched from memory, the next 15 accesses will be primary cache hits on the next elements of that line. The average should be $(15 * 8.7 + 202)/16 = 20.7$ns, and 21.6ns is observed.

The computer used for this was a 463MHz XP900. It has 64 byte cache lines.

Performance Enhancement



This is a 2.4GHz Pentium4. A very fast 8KB primary cache is clearly seen, and a 512KB secondary less clearly so. The surprise is the speed of the main memory, and the factor of four difference between its latency at a 64 byte and 128 byte stride.

The explaination is automatic hardware prefetching into the secondary cache. For strides of up to 64 bytes inclusive, the hardware notices the memory access pattern, even though it is hidden at the software level, and starts fetching data in advance automatically.

The actual main memory latency is disappointing: a 2.4GHz core and 400MHz RDRAM has yielded 145ns, compared to 202ns with a 463MHz core and 77MHz SDRAM on the XP900. The slowest Alpha currently in TCM (175MHz EV4) manages 292ns, the fastest computer (for this) 100ns (a 933MHz Pentium III), the slowest 850ns (a 195MHz R10K).

But is it correct?

All forms of DRAM need refreshing as the charge leaks of their capacitors. More sudden leaks, such as those caused by ionisation events (cosmic rays or natural radioactive decay), or insulation becoming slightly marginal with age, must also be dealt with.

Increased miniturisation decreases the charge difference between a stored ‘1’ and ‘0’, so modern chips are intrinsically more susceptible than older chips.

If a bit in memory ‘flips’, the consequences for a program could be disastrous. The program could simply crash, as a jump instruction has its destination changed to something random.

Worse, it could continue, but give a completely wrong answer, as important data could have been changed. The change could be a sign, an order of magnitude, or merely a fraction of a percent: it could be wrong, but not obviously wrong.

Parity: going for a crash

The simplest scheme for rescuing this situation is to use a $parity$ bit.

This is an extra bit of memory which stores a one if an odd number of those bits which it is checking is set to one, or zero otherwise. If a single bit flips spontaneously, this parity bit will then disagree with the parity of the stored value, and the error is detected.

The problem is what should one do next? The usual answer is to cause the computer to halt immediately: stopping is safer than continuing to calculate using a program or data known to be corrupt.

A slightly more sophisticated response is to terminate the process which caused the read which produced the error. One cannot let the process continue feeding it duff data, so killing it is the best option.

Most parity memory uses one parity bit for every 8 bits of real data: a 12.5% overhead.

ECC to the rescue

A better scheme is to use an $Error\ Correcting\ Code$. The standard scheme can correct for any single bit error, and detect any two bit error and some three bit errors. This contrasts with simple parity, which can correct nothing, detects all single bit errors, and no two bit errors. This standard level of ECC is sometimes known as SECDED: Single Error Corrected, Double Error Detected.

An ECC scheme is more expensive in terms of bits. Whereas parity requires a single bit to protect an n bit word, the usual ECC scheme requires $2 + \log_2 n$. For an 8 byte word, this overhead is again 12.5%.

The Hamming Code

Consider storing check bits in bit positions 2^n, $n \geq 0$, and data in the other bit positions. Thus to store eight data bits, one has the following arrangement:

```
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12

C1 C2 D1 C3 D2 D3 D4 C4 D5 D6  D7  D8
```

Then let `C1` record the parity of those bit positions, excluding itself, with a 1 at the end of the binary representation of their position. In other words

```
C1 = B3 + B5  + B7  + B9  + B11
```

Similarly for `C2` those bit positions with a 1 second from the right of their binary representation.

```
C2 = B3 + B6  + B7  + B10 + B11

C3 = B5 + B6  + B7  + B12
C4 = B9 + B10 + B11 + B12
```

The Hamming Code (2)

Alternatively, consider this backwards:

B6 will be checked by C2 and C3 because $6 = 110_2$ with ones in the second and third places.

If, when we read a word from memory, and recalculate the check bits, we discover that, say, C4 and C1 disagree with our calculation, we know that bit 1001_2 is in error.

Because a bit can exist in just two states, once we know it is wrong, we can just invert it, and it must then be correct.

Qui custodiet ipsos custodes?

The above analysis works for any single bit error. If a check bit itself gets corrupted, it does not cause a data bit to be 'corrected' erroneously, for the binary representation of the bit position of the error will contain a single one, and all those positions contain check bits, not data. No-one needs to guard the guards.

However, to ensure that all possible two bit errors are detected, as well as all one bit errors corrected, a single extra bit is added, which is simply records the parity of all of the other bits.

Otherwise, most 2 bit errors would be 'corrected' by converting them to 3 bit errors!

Wall to Wall ECC

Bit errors are most likely to occur in DRAM: there is a lot of it, and the charge stored is tiny. They are less likely to occur in SRAM (cache RAM) or registers, however the consequences of errors occurring there would be more devastating.

Almost all processors perform some form of error checking and/or correcting on the contents of their caches and registers.

Cheap and nasty systems designed for wordprocessing and games tend not to bother for their main memory, because of the price disadvantage: extra circuitry and extra memory required to store the check bits.

Those computers with memory buses wider than 64 bits have a proportionately lower overhead for ECC. An 8 bit bus would need 5 correction bits, a 256 bit bus just 10.

The good, bad and ugly

Who does what?

Machine	Error code
Intel 386, 486	8 bit parity
Pentium	usually none
PentiumII	none or ECC
Pentium4, SDRAM	usually none
Pentium4, RDRAM	none or ECC
Mac, 68k	none
Mac, PPC	none?
DEC Alpha	ECC
Most workstations	ECC

where ECC means one bit correction, two bit detection.

Considering that parity has been used since the 1950s, and SECDED since the 1970s, this is poor.

Does it all matter?

TCM has 33 DEC Alphas which log single bit errors. In October 1999, the following error rates were seen:

No errors	29 machines
One error	2 machines
Two errors	1 machine
42 errors	1 machine

Without ECC tcm29 would be noticeably unusable and would have a memory chip replaced. Tcm28, with one error every three weeks over October and November, would have a memory chip replaced too.

In the absence of any error detection, tcm29 would be unusable, and it would be unclear what needed replacing. Tcm28 would not stand out, but might give occasional erroneous results, and waste much time as a result. This is the position of all of TCM's non-RDRAM PCs. . .

I have heard it said that a Canadian court ruled that a company was negligent in doing payroll calculations on a machine with no parity checking, and required it to do all such calculations twice.

Memory Management

Memory Management

An operating system ought to have a mechanism for assigning memory to different processes on request.

It ought also have a mechanism for enforcing its decisions: that is, for preventing processes using memory which they have not been allocated.

We shall start by considering a simple and bad memory management strategy: that used by MS DOS. It was vaguely appropriate for the sort of personal computers in use in the 1980's, but has many deficiencies.

Memory, the DOS way

DOS's use of memory typically looks as follows:

DOS provides functions for requesting a new block of memory, or freeing or resizing an existing one.

In the above picture `command.com` cannot grow its memory block: it is firmly surrounded.

Fragmentation

Suppose `Program 1` loads another program, and exits itself. This will leave a memory map looking as follows:

Now the largest single free block of memory is much smaller than the total free memory.

The 640K limit is really a 1088K limit, but as the video memory for graphics modes always starts at 640K, the largest free block is always less than 640K. Sometimes some of the memory between 640K and 1088K can be usefully reclaimed.

Anarchy

Under DOS, what happens if a program tries to access memory marked as ‘free’, or owned by the system, without attempting to reserve it for itself?

Nothing special: the access happens just as if the memory had been correctly reserved.

Any program can overwrite any other program or the operating system.

Intentionally or accidentally.

MacOS had a memory manager with just these properties too.

What went wrong?

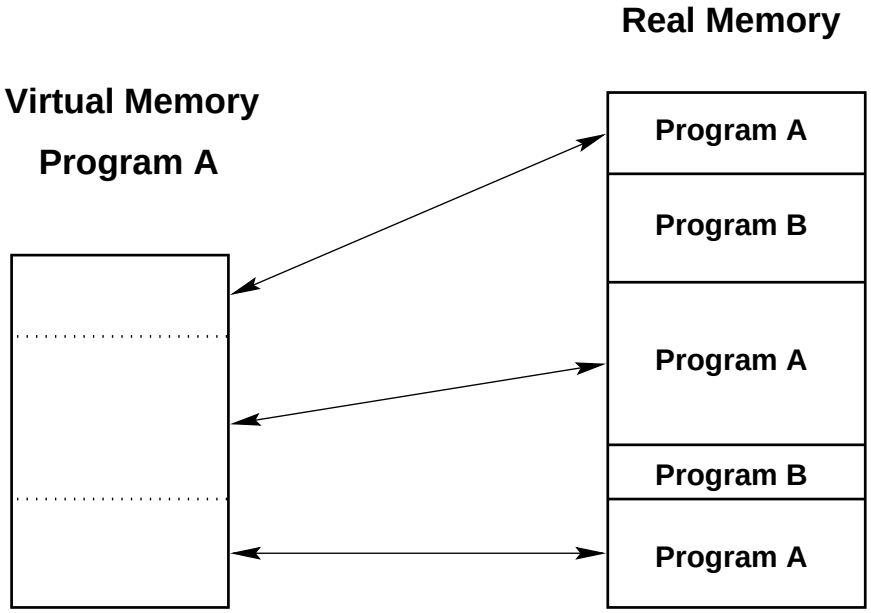
Actually, very little. The memory management of DOS or early versions of MacOS was about as good as one could achieve on the processors then available (8086 and 68000 respectively).

To improve on the above, a little help from the processor is required.

Clearly a program wishes to see a contiguous area of memory: if a programmer requests a 1MB array, he gets upset if the OS says "No, but you can have a 384K array, two 256K arrays, and one 128K array instead."

Virtual addressing

All memory references made by a program are intercepted and have their addresses *translated* by the CPU before they reach the real physical memory.



Fragmentation occurs in the real, physical memory, and not in the virtual address space seen by the program.

The same virtual address in two different processes can refer to two different physical addresses. The converse is rarer but also possible.

When OS/2, Windows9x or Linux runs two DOS applications simultaneously, each DOS application sees an address range from 0K to 640K inhabited by just itself and a copy of DOS. The corresponding physical addresses will be very different.

Address translation in more detail

Various shortcuts occur to make the above scheme reasonable.

Firstly, a complete table of the mapping from every virtual byte to every real byte would be rather big. For a 32 bit machine, one would need four bytes to store the real address corresponding to every virtual byte. . .

So the mapping is done on the granularity of $pages$ not bytes. A page is typically about 4KB and is the smallest unit of memory the operating system can allocate to a process.

The OS keeps a $page\ table$ telling it for every virtual page given to every process, where the real page resides. Entries say things like 'the page from 260K to 264K in process 5's address space is to be found from 2040K to 2044K in physical memory, reading and writing permitted.'

Not quite there

For a 32 bit machine with 4KB pages, the bottom 12 bits of an address are simply an offset into a page, whilst the top 20 bits give the page number. These 20 bits are used as an index into the page table, which will return the physical address of the page.

Each page table entry needs 20 bits for the physical address, and maybe a few spare bits to mark such things as invalid pages, and whether the page can be read, written, or have code executed from it. So say 32 bits, or four bytes.

So for every 32 bit process one needs a 4MB page table, so that every virtual address can be looked up therein. Not quite: we need to do a little better, and *much* better for 64 bit machines.

A Two Tier System

Each process has one page table directory, and at least one page table. Each is one page long, and contains 1024 entries.

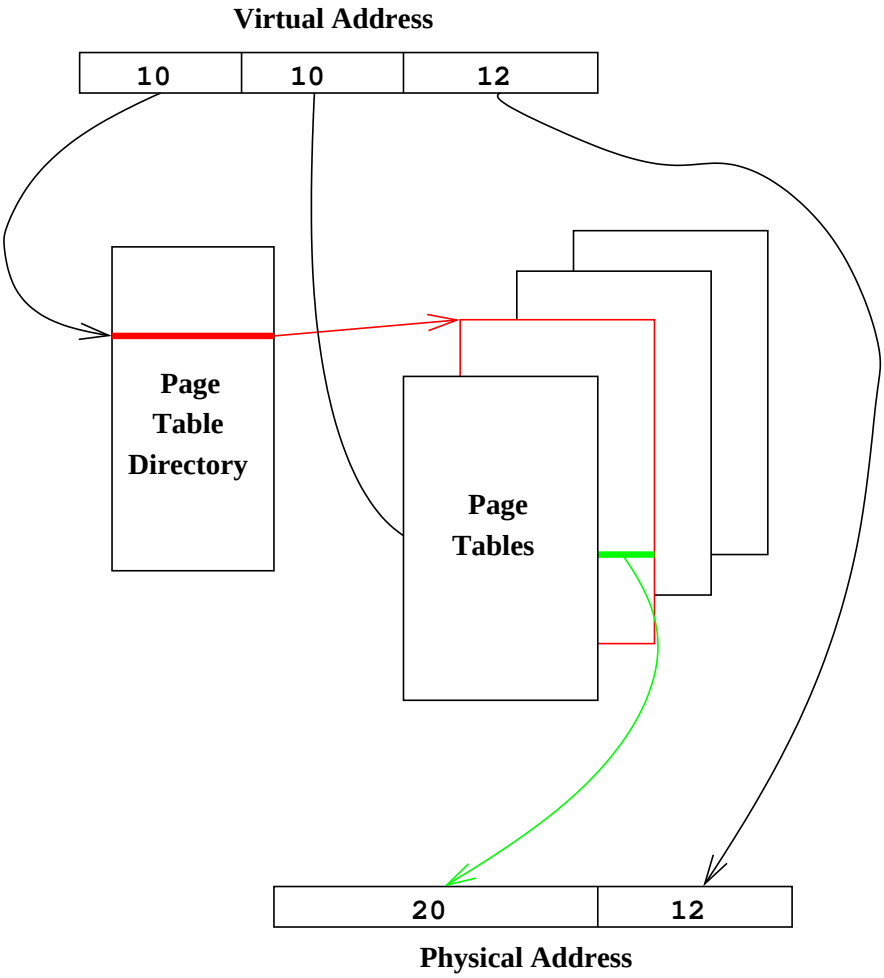
The first ten bits of an address are used as an index into the page table directory. If no virtual address with that starting sequence is valid, the directory will indicate this, and a *page fault* will occur. Otherwise, twenty bits indicating the position in physical memory of the page table for the 1024 pages in that address range will be found.

The next ten bits index this second page table. Again, either an invalid address will be indicated, or twenty bits corresponding to the physical address of the page containing the virtual address being translated.

The final twelve bits are an index into that page.

UNIX tends to announce page faults with SIGSEGV, and terminates the process. This will also happen on attempts to modify read-only pages. SEGV refers to SEGment Violation, as historically pages have been grouped into segments. Windows has called them 'Unrecoverable Application Errors' and a 'General Protection Faults.'

A Two-Level Page Table



A 32 bit virtual address with the top ten bits indexing a single page table directory, and thus giving the address of a page containing the page table entries relevant for the next ten bits of the virtual address. This then contains a twenty bit page number to give a 32 bit physical address when combined with the final twelve bits of the virtual address. The page table will also contain protection information.

Each process has its own virtual address space, and hence its own page tables, but half a dozen pages of page table is sufficient for most small programs.

Beyond 32 Bits

This system does not scale well. For a 64 bit virtual address, not only do the page table entries need around 64 bits, not 32, but one would need 2^{26} entries in each of the directory and page tables. Thus with a minimum of two tables of 2^{26} 8 byte entries, each process would have 1GB of page table.

One solution to this is that used by the Alpha processor when running Tru64 UNIX. The page size is 8KB, so can contain 1024 64 bit entries. The bottom 13 bits are now the index with the page, and there are three levels of page table, not two, each indexed by 10 bits from the virtual address. This accounts for 43 bits of virtual address, and that is all that there are. An Alpha running Tru64 UNIX does not provide a 64 bit virtual address space, but 43 bits (8TB) is enough for most people.

IBM's AIX uses an *inverted page table*, which is a completely different solution to this problem.

Efficiency

This is still quite a disaster. Every memory reference now requires two or three additional accesses to perform the virtual to physical address translation.

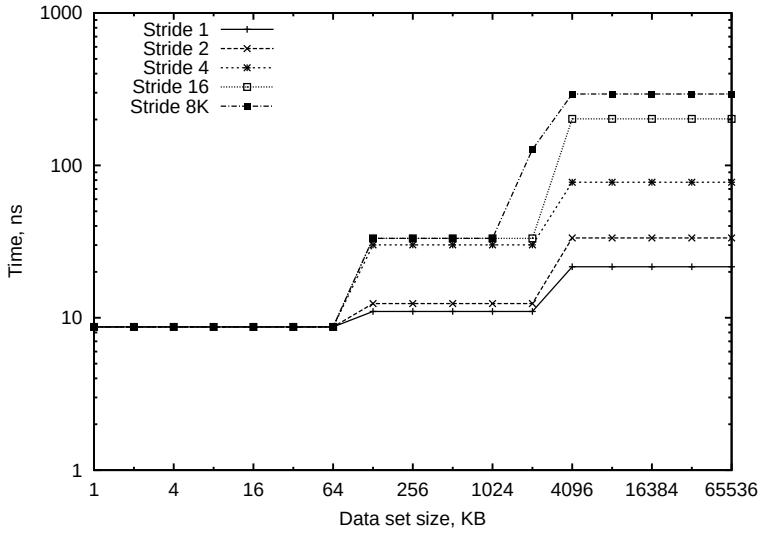
Fortunately, the CPU understands pages sufficiently well that it remembers where to find frequently-referenced pages using a special cache called a TLB. This means that it does not have to keep asking the operating system where a page has been placed.

Just like any other cache, TLBs vary in size and associativity, and separate instruction and data TLBs may be used. A TLB rarely contains more than 1024 entries, often far fewer.

Even when a TLB miss occurs, it is rarely necessary to fetch a page table from main memory, as the relevant tables are usually still in secondary cache, left there by a previous miss.

TLB = translation lookaside buffer
ITLB = instruction TLB, DTLB = data TLB if these are separate

TLBs at work



This is a repeat of the graph on page 123, but with an 8KB stride added. The XP900 uses 8KB pages, and has a 128 entry DTLB. Once the data set is over 1MB, the TLB is too small to hold its pages, and, with an 8KB stride, a TLB miss occurs on every access, taking 92ns in this case.

Given that three levels of page table must be accessed, it is clear that the relevant parts of the page table were in cache: indeed, at least one part must have been in primary cache. A mere 92ns is a best case TLB miss recovery time, and still represents 43 clock cycles, or 172 instruction issue opportunities.

More paging

Having suffering one level of translation from virtual to physical addresses, it is conceptually easy to extend the scheme slightly further. Suppose that the OS, when asked to find a page, can go away, read it in from disk to physical memory, and then tell the CPU where it has put it. This is what all modern OSes do (UNIX, OS/2, Win9x / NT, MacOS), and it merely involves putting a little extra information in the page table entry for that page.

If a piece of real memory has not been accessed recently, and memory is in demand, that piece will be paged out to disk, and reclaimed automatically (if slowly) if it is needed again. Such a reclaiming is also called a page fault, although in this case it is not fatal to the program.

Rescuing a page from disk will take around 20ms, compared with under 200ns for hitting main memory. If just one in 10^5 memory accesses involve a page-in, the code will run at half speed, and the disk will be audibly 'thrashing'.

The ps command reports not only how much virtual address space a program is using, but how many of those pages are resident in physical memory.

The union of physical memory and the page area on disk is called *virtual memory*. Virtual addressing is a prerequisite for virtual memory, but the terms are not identical.

Less paging

Certain pages should not be paged to disk. The page tables themselves are an obvious example, as is much of the kernel and parts of the disk cache.

Most OSes (including UNIX) have a concept of a *locked*, that is, unpageable, page. Clearly all the locked pages must fit into physical memory, so they are considered to be a scarce resource. On UNIX only the kernel or a process running with root privilege can cause its pages to be locked.

Much I/O requires locked pages too. If a network card or disk drive wishes to write some data into memory, it is too dumb to care about virtual addressing, and will write straight to a physical address. With locked pages such pages are easily reserved.

Certain 'real time' programs which do not want the long delays associated with recovering pages from disk request that their pages are locked. Examples include CD writing software.

Swapping

The terms ‘swapping’ and ‘paging’ are often used interchangably. More correctly paging refers to discarding individual pages to the swap device, whereas swapping refers to removing all the pages associated with a single process to the swap device in one operation. Once swapped in this fashion, a process must necessarily be suspended.

Swapping is the older and simpler mechanism, and works well on, a PC running several interactive applications. Clearly just one application can interact with the PC’s one user at once, so wholly removing the other processes from memory is fine. It may take several seconds to restart a swapped-out process.

Paging permits a single process to use more memory than physically present. Swapping does not.

Whether paging or swapping, the area of disk used is called the $swap\ space$.

The total amount of memory usable might be the size of the swap space, the size of physical memory plus swap space, or greater than this, depending on the OS. The last case is ‘impossible’: the OS claims to have more memory available than it does, overcommitting swap space, and will behave badly if all programs try to use all the memory they have been allocated.

Page sizes

A page is the smallest unit of memory allocation from OS to process, and the smallest unit which can be paged to disk. Large page sizes result in wasted memory from allocations being rounded up, longer disk page in and out times, and a coarser granularity on which unused areas of memory can be detected and paged out to disk.

Small page sizes result in more TLB misses, as the area of virtual address space 'covered' by the TLB is simply the number of TLB entries multiplied by the page size.

Large-scale scientific codes which allocate hundreds of MB of memory benefit from much larger page sizes than a mere 8KB. However, a typical UNIX system has several dozen small processes running on it which would not benefit from a page size of a few MB.

Mix and match

Many modern CPUs support multiple page sizes, such as the Pentium which supports 4KB or 4MB, or the UltraSPARC III which supports 8K, 64K, 512K and 4MB. The EV6 Alpha allows a single TLB entry to refer to one, eight, 64 or 512 consecutive pages, thus effectively increasing the page size.

However, this reintroduces fragmentation problems. If a Pentium wants to allocate a 4MB page, or an Alpha wants to allocate 512 consecutive 8KB pages, there must be 4MB of contiguous free physical memory. This problem cannot occur if all pages are the same size.

Thus operating system support for large pages is rarer than hardware support. Solaris 9 (introduced 2002) is one of the few OSes which supports different page sizes for different processes.

Cambridge HPCF people will be familiar with these concepts. The S3600 vector Hitachi used 1MB pages for programs using the vector unit, and 4KB pages for others. The SR2201 parallel machine had three special TLB entries for user code, called Block TLBs, each of which covered a 32MB page. The SR2201 did not page to disk, but could get its physical memory too badly fragmented to use its BTLB. The S3600 paged from its main SRAM memory to a 'disk' of DRAM.

Alignment

To keep circuitary simple, a 64 bit bus cannot transfer an arbitary 8 bytes, but eight bytes to/from an address which is a multiple of eight. Similarly a 64 byte cache line will start at an address which is a multiple of 64, and a 4KB page will start on a 4KB boundary.

If data in memory are also $naturally$ aligned, then a single load/store will involve no more than one cache line per cache, one TLB entry, and will not require multiple bus transfers in the same direction. It will be faster than a misaligned load/store.

Some processors permit the use of misaligned data, at a performance cost. Others do not have hardware support for misalignment, and will either be rescued by software (at an enormous speed penalty), or will stop the process with SIGBUS.

The IA32 range permits all alignments. The Alpha range does not, requiring 4 byte objects to be aligned on 4 byte boundaries, 8 byte objects on 8 byte boundaries, etc.

‘Free’ memory

Memory is not free, indeed, in most computers it costs more than the CPU or disk drives. . .

Memory which is idle is therefore a waste, and most OSes use idle memory to increase the size of their disk cache: just as a small amount of fast SRAM acts as a cache for slower DRAM, so a small amount of DRAM can act as a cache for a yet slower disk drive.

Most variants of UNIX (inc. Linux) and Win9x do this.

A small amount of memory (c. 100 pages) is typically kept genuinely free for emergencies, whereas other unused memory is available to the disk cache.

The UNIX command ‘`vmstat`’ shows how many pages are completely unused, as well as information on paging activity.

The Digital UNIX Way

The scheme is used by Digital UNIX 4.0 to 5.1A is:

- at least 128 pages are 'always' kept free. Paging will not occur whilst more memory than this is free.
- swapping will not occur whilst there are more than 74 pages free.
- if the disk cache is bigger than 20% of total memory, it will be shrunk rather than paging a process.
- if the disk cache is using between 10% and 20% of memory, it will fight processes on equal terms.
- if the disk cache is under 10%, it has priority over other processes for memory.

A reasonable estimate of 'free' memory is thus those pages actually unused, plus the amount by which the disk cache is above 20% of total memory.

The 10% and 20% 'watermarks' are configurable. They have been changed to 5% and 10% on TCM's larger-memory Alphas. DEC offers no explicit guidance on reasonable values.

Digital UNIX becomes very unhappy if the disk cache is forced below the lower watermark.

The command '`free`' (Linux, Digital UNIX (TCM only)) shows the current disk cache size.

Segments

A program uses memory for many different things. For instance:

- The code itself
- Shared libraries
- Statically allocated uninitialised data
- Statically allocated initialised data
- Dynamically allocated data
- Temporary storage of arguments to function calls and of local variables

These areas have different requirements.

What We Want

Code
Read only, executable, fixed size

Shared libraries
Read only, shared, executable, fixed size

Static data
Read-write, non-executable, fixed size

Dynamic data
Read-write, non-executable, variable size

Temporary data
Read-write, non-executable, frequently varying size

Stacks of Text?

These regions are given distinct address ranges and are called *segments*. Each segment is managed differently to give it the required properties. The common UNIX names for the segments are:

Code	text
Initialised static data	data
Uninitialised static data	bss
Dynamic data	heap
Temporary data	stack
Shared libraries	shared text

An executable file must contain the first two, plus a record of the size of the third. These three sizes are reported by the '`size`' command.

Often read-only data (constants) are placed in the text section, for this section will be read-only.

What Went Where?

Determining which of the above data segments a piece of data has been assigned to can be difficult. One would strongly expect C's `malloc` and F90's `allocate` to reserve space on the heap. Likewise small local variables tend to end up on the stack.

Large local variables really ought not go on the stack: it is optimised for the low-overhead allocation and deletion needed for dealing with lots of small things, but performs badly when a large object lands on it. However compilers sometimes get it wrong.

UNIX limits the size of the stack segment and the heap, which it 'helpfully' calls 'data' at this point. See the '`limit`' command (`csh`) or '`ulimit`' (`sh`).

Because `limit` and `ulimit` are internal shell commands, they are documented in the shell man pages (`tcsh` and `bash` in TCM), and do not have their own man pages.

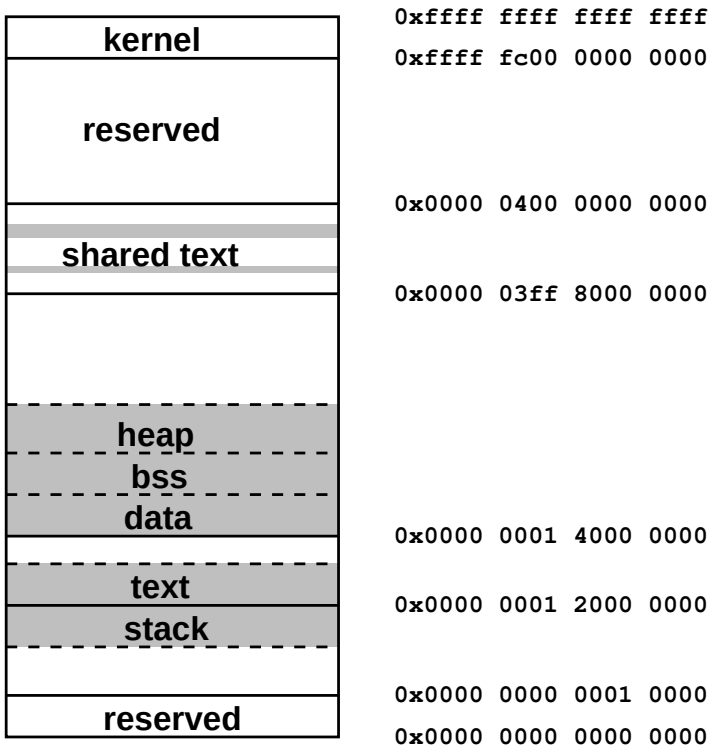
Sharing

If multiple copies of the same program or library are required in memory, it would be wasteful to store multiple identical copies of their unmodifiable read-only pages. Hence many OSes, including UNIX, keep just one copy in memory, and have many virtual addresses refering to the same physical address. A count is kept, to avoid freeing the physical memory until no process is using it any more!

UNIX does this for shared libraries and for executables. Thus the memory required to run three copies of Netscape is less than three times the memory required to run one, even if the three are being run by different users.

Two programs are considered identical by UNIX if they are on the same device and have the same inode. See a the section on filesystems for a definition of an inode.

A UNIX Memory Map

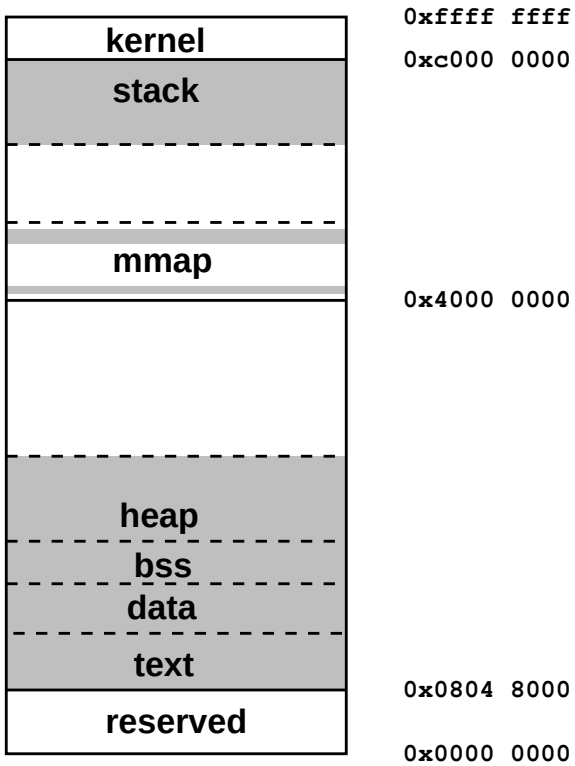


N.B. This view is per process, not for the whole machine.

This particular bizarre layout is based on that used by Digital UNIX 4.0. Note that this layout imposes artificial limits, such as approx 4GB for the stack, and 512MB for the text segment. Such limits tend to be much more severe when one is squeezing into a 32 bit address space, rather than the 64 bit (43 bit usable) space here.

Shared data and `mmap` region omitted for simplicity.

Another UNIX Memory Map



This is roughly the layout used by Linux 2.4 on 32 bit machines. Note the shorter addresses than for Digital UNIX.

The `mmap` region deals with shared libraries and large objects allocated via `malloc`, whereas smaller `malloc`ed objects are placed on the heap in the usual fashion. Note too that if one uses `mmap` or shared libraries at all, the largest contiguous region is under 2GB.

Note in both cases the area around zero is reserved. This is so that null pointer dereferencing will fail: ask a C programmer why this is important.

Memory Maps in Action

Under Linux, one simply needs to examine `/proc/[pid]/maps` using `less` to see a snapshot of the memory map for any process one owns. It also clearly lists shared libraries in use, and some of the open files.

Under Solaris one must use a program called `pmap` in order to interpret the data in `/proc`.

With Digital UNIX less information is available, and it is harder to extract. In TCM a utility called `pmap` exists which will display some information in a similar fashion to the Solaris program.

Files in `/proc` are not real files, in that they are not physically present on any disk drive. Rather attempts to read from these 'files' are interpretted by the OS as requests for information about processes or other aspects of the system.

CPU Families

CPU Families

CPUs tend to come in *families*, each successive member of a family being *binary compatible* with all preceding members: that is, being able to run any machine code written for preceding members.

This means that later members of a family must have the same registers and instructions as earlier members, or, if not the same, a superset of them.

Each family will have its own assembly language, and potentially different numbers of registers and even types of instructions.

Whereas high-level languages, such as C and FORTRAN, are *portable* between different CPU families, the low-level assembler or machine code certainly is not.

Common Families

8086 Intel 8086, 80286, and all IA32

IA32 Intel 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Cyrix 586 and M1, AMD K6 and Athlon.

Motorola 68K 68000, 68020, 68030, 68040, 68060

Alpha 21064 (EV4), 21164 (EV5), 21264 (EV6)

Power Power, Power2, Power3, Power4

PowerPC 601, 603, 604, 620, 7400

MIPS64 R8000, R10000, R12000, R14000

SPARC SPARC I, SPARC II, SuperSPARC, HyperSPARC, UltraSPARC, UltraSPARC II & III

All except the non-IA32 8086 line, the M68K line and the Alphas are still being developed, with future members planned.

Some minor members have been omitted.

Inter-family differences: IA32 vs Alpha

	IA32	Alpha
Integer Regs	8 x 32 bit	32 x 64 bit
F.P. Regs	8 x 80 bit	32 x 64 bit
Memory operands	Yes	No
Has trig functs	Yes	No
Instruction length	1 - c.14 bytes	4 bytes

There are many other differences, such as the Alpha having integer and FP instructions of the form 'a op b $\longrightarrow$ c' where a, b and c must be registers. IA32 uses the form 'a op b $\longrightarrow$ a' for integer operations, and a or b may be references to data stored in memory.

The Alpha uses the naming convention $0 to $31 for its integer registers, and $f0 to $f31 for its floating point registers. The majority of these registers are equivalent. IA32 calls its integer registers %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp and %esp for odd historical reasons. For IA32 there are many tedious restrictions concerning which instructions may act on which registers.

Both have an additional register holding to the address of the current instruction: the *instruction pointer*. A branch instruction simply modifies this special register. Both also reserve one register for pointing to the end of the program's stack: $30 for Alpha, and %esp for IA32. Alpha has a register, $31, whose value is fixed at zero. IA32 does not.

More IA32 vs Alpha

```
   double t=0.0; int i,n;
   for (i=0;i<n;i++) t=t+x[i];
```

```
; %edx contains n
; %ecx contains x
  fldz
  xorl %eax,%eax
  cmpl %edx,%eax
  jge .L3

.L5:
  faddl (%ecx,%eax,8)
  incl %eax
  cmpl %edx,%eax
  jl .L5
.L3:
```

```
# $17 contains n
# $16 contains x
  fclr $f0
  clr $1

  ble $17,L$5

L$6:
  ldt   $f1, ($16)
  addl  $1, 1, $1
  cmplt $1, $17, $3
  lda   $16, 8($16)
  addt  $f0, $f1, $f0
  bne   $3, L$6
L$5:
```

Both sides slightly abbreviated, but many differences are clear. Different mnemonics are used (Float LoaD Zero vs Float CLeaR), and certainly different binary representations. IA32 has a special instruction to increment (add one to) a register, Alpha does not. IA32 can move data from memory directly to the FP adder without passing through a register, Alpha cannot. Etc.

The IA32 Family in Detail: the Ancestors

8086

Introduced 1978. 16 bit registers and data bus, 20 bit address bus (1MB), four clock cycles to do simplest instruction, separate FPU (8087). Integer registers each usable as a pair of 8 bit registers (e.g. `ax` is `ah` and `al`). Clock speeds 4 to 10MHz. 29,000 transitors.

80286

Introduced 1982. Address bus increased to 24 bits (16MB), simple operations complete in three cycles, separate FPU (80287). Clock speeds 8 to 12MHz. 134,000 transistors.

These things are antique, and firmly CISC. They have many, many bizarre instructions, not least for dealing with binary coded decimal (what? never mind. . .) However, code written for one of these will still run on a Pentium4 designed over two decades later.

Although the 8086 is the first ancestor as far as binary compatibility is concerned, that is, the ability to run machine code written for one process on another, it is not a completely 'clean' design, and enjoys a degree of compatibility with the 8080, which is 4 years older.

IA32: the Start

80386

Introduced 1985. Registers extended to 32 bits, e.g. the `ax` register is now the bottom half of the `eax` register, and two new registers added. Data and address bus extended to 32 bits (4GB). Virtual memory (paging etc.) with 32 entry 4-way associative TLB, multitasking, device protection. Simple operations complete in two cycles. Separate FPU (80387). Clock speeds 16 to 33MHz. Different modes of operation in order to keep full 8086 compatibility. This major increase of functionality has almost everything a modern CPU has, and just 275,000 transistors.

i486

Introduced 1989. Almost no changes in functionality, but core redesigned. Cache controller with an 8KB 4-way associative write-through cache, two write buffers, and the FPU, placed on the main CPU. Pipelined integer core does basic operations in a single cycle. Bus can transfer a cache line (16 bytes) in five cycles, compared to two cycles per 4 bytes for the 80386. Clock speeds 20 to 50MHz. The **i486DX2** and **i486DX4** versions have a 2:1 or 3:1 core:bus frequency ratio. Clock speeds 50MHz to 100MHz.

The Pentium: the last CISC

Pentium

Introduced 1993. Very few changes in functionality, but many in implimentation. Again a redesigned core, now superscalar for integer operations, with dual issue possible in some circumstances. The FPU is pipelined for the first time, and is made much faster. The cache is split as a 2-way associative 8KB instruction cache, and similar write-back data cache. Branch prediction and 4MB pages introduced. The data bus width is increased to 64 bits, and runs at 60 or 66MHz. The core runs at an integer or half-integer multiple of this, from 60MHz to 200MHz.

PentiumMMX

Adds Intel's MultiMedia eXtensions, instructions which pack eight 8-bit, four 16-bit, two 32-bit or one 64-bit integer into a floating-point register, and do simulateous adds / shifts / etc. on them. Useful for some graphics work. Also has 'add with saturate', e.g. $200 + 180 = 255$ for an 8-bit example. Branch prediction improved, and caches made 4-way associative again and increased to 16KB.

PentiumPro / Pentium II / Pentium III

Introduced 1995. Yet another redesigned core (often called P6). The core is a superscalar RISC CPU, and the CISC Intel instructions are converted to 'micro-ops' and passed on to the core, with one CISC instruction potentially producing several μ-ops. The core can overlap and reorder the simple μ-ops with greater ease than it could the CISC IA32 instructions, and yet old IA32 code runs happily.

The PPro does not support MMX, the PII (1997) does, and the PIII (1999) also supports SSE. SSE adds support for four single precision floats in one of eight new SSE registers. Very important for certain 3D effects, but not for science, as is single precision.

The PPro has the secondary cache on the CPU and running at the same speed (150MHz to 200MHz). The PII moves it off die, but keeps it on the physical CPU module, and runs it at half CPU speed. It also introduces 100MHz bus speeds for the faster PIIs. Similarly faster PIIIs use 133MHz buses, and up to 1.2GHz cores.

Pentium4

Introduced 2001. Another redesign, again with a RISC core and a conversion step. This time the instruction cache is moved after the conversion logic, so the μ-ops get cached. SSE2 introduced, which permits two doubles in an SSE register. Useful at last!

Primary data cache only 8KB write through, combined secondary cache 256KB initially, later 512KB. DTLB 64 entries. Automatic hardware prefetching. Xeon models support multiprocessor designs.

Clock speeds are very high: 1.4GHz to 3GHz. However, many latencies are longer than the PIII, and the pipeline is longer so mispredicted branches are more costly. Faster than a PIII, but not by as much as the increase in clock speed would suggest, unlike every previous family member which would outperform its immediate predecessor even when run at the same clock speed.

Data bus, still 64 bit, run at 400MHz initially, 533MHz later, giving much higher potential than the PIII.

Marketing gimmicks

8088 an 8086 slowed by an 8 bit data bus.

386SX an 80386 slowed by a 16 bit data bus and with a 24 bit address bus.

486SX an i486 slowed by the removal of the FPU.

Celeron a PII, PIII or P4 slowed by reducing the amount of secondary cache (or removing it altogether), and reducing the bus speed. Hence all Celerons support MMX, but only some SSE or SSE2.

The above are slightly cheaper to make than the corresponding 'real' CPUs, and are sold at significantly lower prices.

MMX MultiMedia eXtensions
SSE Streaming SIMD Extensions

IA64

Intel's latest range of processors is called the IA64 range (Itanium and Itanium2), and is not fully compatible with IA32 code. It differs from IA32 in many ways:

- Integer registers are 64 bit
- There are 128 fp registers
- It uses VLIW (3 instructions per word)
- It has predicate bits

It is not really that new: almost everything it does had already been done a decade ago.

The Itaniums have much larger secondary caches than the IA32 range: 1.5MB for the Itanium, 3MB for the Itanium2.

The Alpha Family

21064 (EV4)

The first Alpha, the 21064 or EV4, was introduced in 1992. It was a fresh, new design with thirty-two 64 bit integer registers, thirty-two 64 bit floating-point registers (supporting IEEE and VAX formats), and a superscalar pipelined RISC architecture. Supported paging and protection. Separate intruction and data caches, each 8KB direct mapped write through. Twelve entry ITLB, and 32 entry DTLB, both fully associative. Address bus 34 bit. Can issue two instructions per clock cycle, at most one floating point. CPU speeds 100MHz to 266MHz. Data bus 128 bit, 33 to 40MHz.

21164 (EV5)

1995. Added a small, 96KB 3-way associative write-back secondary cache, ITLB increased to 48 entry, DTLB to 64 entry. Two integer pipelines, one FP add pipeline, one FP multiply pipeline. Four way superscalar core, 40 bit address bus. The 21164A (EV5.6) added Alpha's form of 'MMX' for dealing more efficiently with small integer data. Core speeds 266MHz to 600MHz.

21264 (EV6)

1999. Primary caches now 64KB 2-way associative, write back for data. Both TLBs 128 entry. Out-of-order execution supproted by the core. Secondary cache controller on the CPU core, supporting 2MB to 16MB external direct mapped write-back caches. Software prefetch added. Core speeds 500MHz to 1.25GHz. Data bus 64 bit 333MHz (500MHz later).

Although the theoretical peak speed of the EV5 and EV6 is the same, the many enhancements on the EV6 – bigger, better caches, OOOE, much higher bandwidth to memory – ensure that the EV6 easily outperforms the EV5.

Unfortunately the Alpha range got cancelled by Compaq for dubious political reasons. The 21364 (EV7) will probably arrive in 2003, at least two years late. It has the same core as the EV6, but a faster secondary cache, a redesigned memory interface, and much of the logic needed to create multiprocessor machines incorporated. The 21464 (EV8) has been cancelled. It would have had a redesigned core supporting 4-way SMT.

How Many Bits?

	Z80	8086	Pentium	Alpha
Integer registers	8/16	8/16	8/16/32	64
FP registers	-	(80)	80	64
Address registers	16	16	16/32	43
Data Bus	8	16	64	64
Address Bus	16	20	36	41
Conventional answer	8	16	32	64

The question is ill-defined.

The answer used to be the largest integer operand size for simple operations.

Today the answer is usually the size of the address space.

A 16-bit computer can address 64KB, a 32-bit computer 4GB, and a truely 64-bit computer 16EB.

The 'Alpha' above is the EV6.

Families and Optimisation

Although code compiled for the 80386 will run on the Pentium4, and code compiled for an EV4 Alpha will run on an EV6, this is not ideal. Not only will all the additions to the instruction set be ignored by such code, the scheduling optimisations may be inappropriate.

Whereas the EV5 will issue an FP $\times$ and $+$ simultaneously if there are no data dependencies, the EV4 cannot do so, so for it there is no point in trying to pair $\times$ and $+$. Similarly, a 80386 or i486 will fail to predict the branch at the end of a loop, and benefit from 'hiding' the resulting pipeline drain by the latency of a floating-point instruction, such tricks make no sense for later models, which will predict the branch.

Of course, if one does use extensions to the instruction set, backwards compatibility suffers. Code using Intel's MMX instructions will not run on an 80386, i486, plain Pentium or a PentiumPro.

Code using the full EV6 instruction set will run on an EV4 under Digital UNIX: the missing instructions are emulated in software provided by the OS. This happens extremely slowly.

Video Hardware

Video

A short section, but something ought to be said about the main means of communication from computer to human.

A computer monitor is a dumb cathode ray tube. The signals it receives from the computer control fairly directly the scanning of the three electron beams (red, green and blue) across the screen.

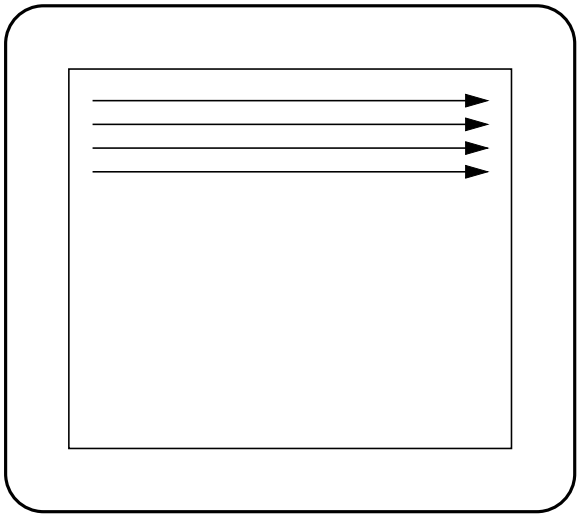
The coloured phosphors that the electrons strike glow for a few tens of milliseconds after they have been energised. After that, the picture disappears.

The intensity of each gun is controlled in an analogue fashion, so that a monitor can display an infinite range of colours and brightnesses. The resolution of the monitor is limited by the fineness of the grid of red, green and blue phosphors. Monochrome monitors need no such grid.

The video signal

The electron beams scan lines from left to right, going from the top to the bottom of the screen. This results in three characteristic frequencies.

- Vertical refresh: the rate at which the whole picture is repeated.
- Horizontal scan rate: the rate at which individual lines are produced.
- Dot rate: the rate at which pixels are passed.



Humans can perceive flicker if the vertical refresh rate is below about 65 Hz. Below around 72Hz humans do not notice flicker, but are still adversely affected by it.

pixel: PIcture ELement – the dots which make up an image.

Video hardware

The graphics card contains an area of memory which stores the current screen image, and some circuitry to scan through that memory producing the video signal, via a digital to analogue converter which changes the binary numbers stored in the memory to the analogue levels required by the monitor.

The amount of memory required is governed by the total number of pixels and the colour depth. Monochrome displays require one bit per pixel, full colour displays 24 bits per pixel to give 256 independent levels for the red, green and blue guns.

Older displays use a palette: a look-up table with typically 16 or 256 entries and which returns a 24 bit colour value. Thus for each pixel one need store only the four or eight bit palette index, not 24 bits.

The worst display considered vaguely acceptable these days, 1024x768 with 256 colours, requires 768KB of memory. A trivial amount now, but an enormous amount a few years ago. 1280x1024 in 24 bit requires 4MB, or slightly more if the pixels need aligning on 4-byte boundaries.

Bandwidths are also high, as the video circuitary needs to read the whole display once for every scan, so about 75 times per second.

TFT LCDs

A thin film transistor liquid crystal display is not at all like a CRT. It works at a fixed resolution, with each dot being addressable in a manner similar to a DRAM chip. However, unlike a DRAM chip, each pixel stores an analogue value to give different intensities.

The liquid crystal layer merely absorbs light, so a white backlight shines through this layer, which has three cells, one red, one green and one blue, for each pixel. The response time of the liquid crystal is somewhat slower than that of the CRT, and the decay time in the absence of refresh is so long that the refresh flicker is not usually apparent.

Given that the image is stored in RAM on the video card, converting this to an analogue signal suitable for a CRT, and then reconverting this to a signal suitable for a RAM-like device, is silly. It also works poorly: try displaying alternate black and white pixels (e.g. an `xterm` scrollbar) on such a display – they will 'crawl' spontaneously. So one should use a different form of signal between a graphics card and an LCD monitor, and this is what DVI is (and the old SVGA (or 15 pin D-sub) connector isn't).

Acceleration

Graphics adapters were 'dumb', the computer's CPU explicitly writing every pixel into their memory.

Then processing power appeared on the adapter itself able to do simple operations, such as filling, copying & scrolling rectangular regions. This $2D\ acceleration$ made a large difference to graphics performance.

Without acceleration, to scroll a window the CPU would have to copy the contents to a slightly different location in the video memory, with each pixel moving twice across the CPU's bus.

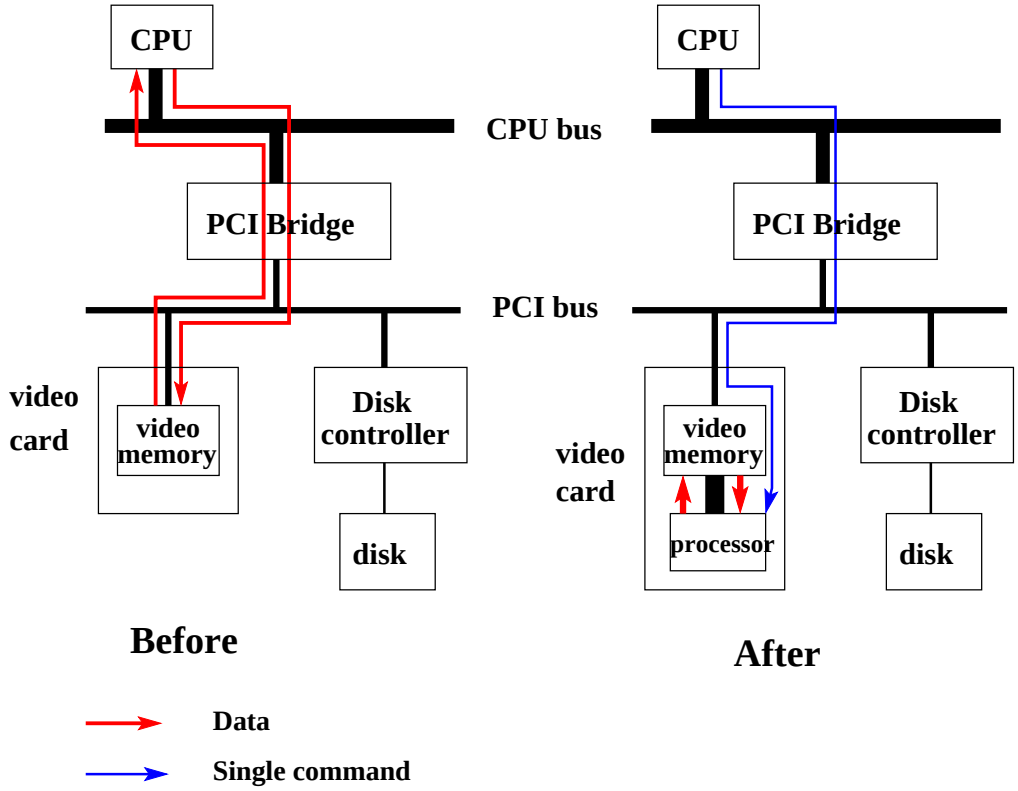
With acceleration, the CPU sends a short instruction to the graphics card, and then lets the graphics card get on with the problem. The card will use its own internal bus, which will be many times faster than the host computer's bus.

Thus the computer's CPU and bus are freed immediately, and the graphics operation completes several times faster.

On PCs, the original VGA and SVGA standards were unaccelerated. Chips such as the S3 and ATI's Mach32 were early, and incompatible, examples of adding acceleration to the core SVGA functionality.

Acceleration in pictures

Herewith a diagram of the data flow for an unaccelerated and accelerated move or scroll operation.



When PC graphics cards call themselves '64 bit' or '128 bit' they are usually referring to the width of the internal bus on the card. When they call themselves 8, 16 or 24 bit they are referring to the colour depth . . .

If the memory on the video card is more than needed for storing the image, the extra can be used to store frequently-needed objects which can then be quickly copied to the part which is being displayed.

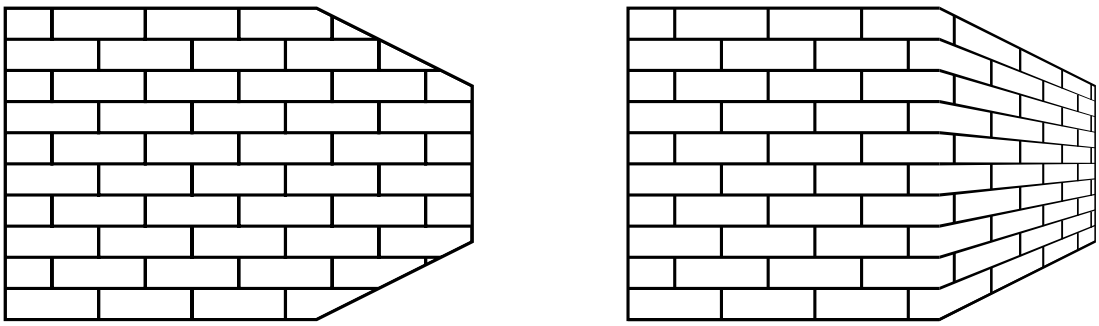
OpenGL

OpenGL is a development (1992) of an older, propriatory, SGI graphics library. It provides for all the primitive operations one might wish for in 3D graphics: partial transparency, texture mapping, colour interpolation, lighting effects, object lists with 3D co-ordinates, rotations, etc. Its basic unit of operation is the triangle. (Any polygon can constructed from triangles, and triangles can (approximately) tile curved surfaces.)

Although OpenGL could be converted to standard bitmap by software, this was slow. SGI had video cards which did most of the operations themselves in hardware – 3D acceleration – but these were very expensive (thousands of pounds).

However, towards the end of the 1990s OpenGL suddenly took off as it did precisely what games programmers needed, and the hardware had become affordable.

Texture Mapping



Two identical polygons with a simple brick texture mapped onto them. The left has a simple 2D texture mapping, the right a simple 3D mapping.

OpenGL would cope with lighting effects as well as doing the simple geometric transform required above.

A video card with hardware GL support has a processor optimised for the sort of single-precision floating-point operations involved in 3D graphics. It will be highly RISC, highly pipelined, and probably superscalar, and may well have better performance than the computer's main CPU. The processor on the graphics card is sometimes called the 'GPU' (Graphics PU), and needs memory for storing textures and triangle co-ordinates as well as the image itself.

OpenGL has been adopted by Windows and MacOS, as well as the UNIX world.

A final speed enhancement on modern computers is the use of a dedicated bus for the video card. This AGP (Advanced Graphics Port) bus is simpler than PCI (it can only cope with a single device), faster than PCI, and is not shared with disk drive controllers etc.

Parallel Computers

Not Parallel: Multitasking

A single CPU can run only one program at once. Multitasking is an illusion for the confusion of gullible humans.

The processor runs one program for a $timeslice$, typically 1 to 100ms, then switches to another. The shorter the timeslice, the less humans will notice.

When the CPU performs a $process\ switch$, it must save to memory all its registers and reload the set relevant to the new process. This will take hundreds of clock cycles. The restarted process will also find the caches mostly, or entirely, storing data relevant to the previous process.

The more registers a CPU has, the more expensive a process switch is, although the flushing of caches, TLBs and branch prediction history is a significant hidden cost too. The longer the timeslice, the less time is wasted switching.

Inequality

If the operating system knows a process is waiting for input (disk, network, human), it will not give that process any timeslices until input is ready for it. Such a process will be marked as *waiting* rather than *running*. The arrival of input might cause an immediate process switch to be triggered, with the timeslice of whatever process was running being interrupted. Thus fast response to I/O events is achieved.

The part of the operating system responsible for assigning priorities to processes is called the *scheduler*. The priorities need not be equal.

The UNIX `ps` command shows processes waiting for input in a state of 'wait' or 'sleep'. Only those in a state of 'run' are actively competing for CPU cycles.

The *load* or *load average* is UNIX's term for the number of proceses is the 'run' state averaged over a short period. The `uptime` command reports three averages, over 1, 5 and 15 minutes on most UNIXes, and 5s, 30s, and 1 minute on Tru64.

Under UNIX the `nice` and `renice` commands can be used to decrease the scheduling priority of any process you own. The priority cannot be increased again, unless one is root. (If you use `tcsh` or `csh` as your shell, `nice` is a shell built-in and is documented in the shell man page. Otherwise, it is `/usr/bin/nice` and documented by `man nice` in the usual way.)

Co-operate or be Pre-empted

Early forms of MacOS and Windows used *co-operative* multitasking. Each process was responsible for giving back control to the scheduler, and would retain the CPU until that point. Naughty or buggy programs could thus prevent multitasking.

With *pre-emptive* multitasking, the process need know nothing of multitasking, for it will be automatically and unavoidably suspended at the end of its allotted time. Thus UNIX, Win9x, WinNT, and most modern OSes.

Pre-emptive multitasking needs support from the CPU. The 80386 was the first Intel processor to support this, although all the 68000 range have been capable.

Privilege

Modern CPUs associate a privilege level with each piece of code, and support at least two such levels. The lower is forced to use virtual addressing, cannot access any hardware directly (video, disk, ethernet card, PCI bus, etc.), and cannot change scheduling priorities. The higher can use physical addressing, access all hardware, and do anything.

UNIX runs just the kernel at the higher level, with all processes running at the lower. Whenever a process accesses disk, video or network, or allocates memory, it must send the request via the kernel. The kernel then applies appropriate restrictions, restricting root slightly less than other users.

The interface between the two privilege levels is carefully designed to prevent a normal process being able to run its own code with full privilege.

Early, cheap CPUs designed for single-user computers, e.g. the 8086 and Z80, did not support this concept at all.

In any OS, the kernel should be as small as possible, for bugs in the kernel have the greatest potential for mischieve.

Parallel Computers: the Concepts

Modern supercomputers are generally $parallel\ computers$. That is, they have more than one CPU.

Usually 50-500 CPUs. The CPUs themselves are usually standard workstation processors, hence 'cheap.'

Some tasks are clearly suited to being done by a 'farm' of 'workers' working simultaneously, whilst others are not. As two examples:

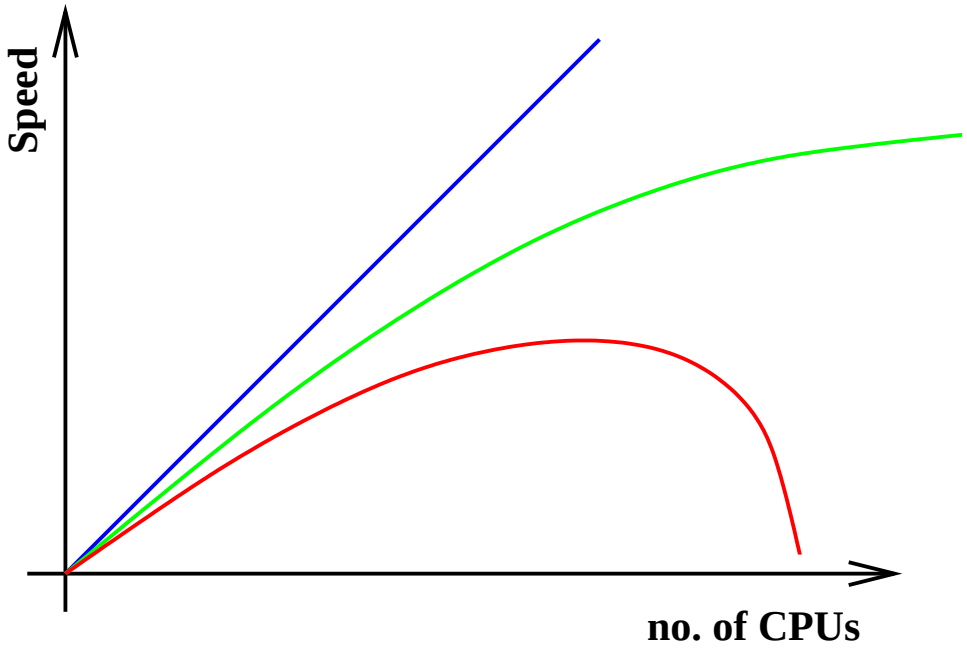
Integration of differential equation over very many timesteps. Clearly one cannot start the 5,000th timestep until the 4,999th has been finished. The process is fundamentally serial.

Factorisation of a large number. The independent trial factors from 2 to $\sqrt{n}$ are readily distributed amongst multiple processors.

A simple example of parallelisation has alrady been seen in the various 'multimedia' instructions. This is known as SIMD parallelism: Single Instruction Multiple Data. The parallism discussed in this section is MIMD (Multiple. . .).

Scaling

How much faster does a code run when spread over more CPUs?



From top to bottom:
Linear scaling (rare!)
Amdahl's Law (see below)
The Real World

Notice that the speed is not monotonic in the number of CPUs

Amdahl's Law

Amdahl was a pioneer of supercomputing and an employee of IBM.

This law assume that a program splits neatly into an unparallelisable part, and a completely parallelisable part. It claims:

$$t_n = t_s + t_p/n$$

The total run time on n processors is the time for the serial part of the code, plus the time the parallel part would take on a single processor divided by the number of processors.

Consider $t_s = 0.2$ and $t_p = 0.8$. Then $t_1 = 1.0$, $t_{32} = 0.225$ and $t_\infty = 0.2$.

On 32 processors the speedup is $4.5\times$ and the efficiency is just 14%.

Bigger is better

Suppose t_s and t_p scale differently with problem size.

Assume t_s scales as N and t_p as N^3 and consider a problem $4\times$ as large as before. Now

$t_s = 0.8$ and $t_p = 51.2$ giving $t_1 = 52$ and $t_{32} = 2.4$.

Now the speedup on 32 processors is $21\times$, and the efficiency is now over 67%.

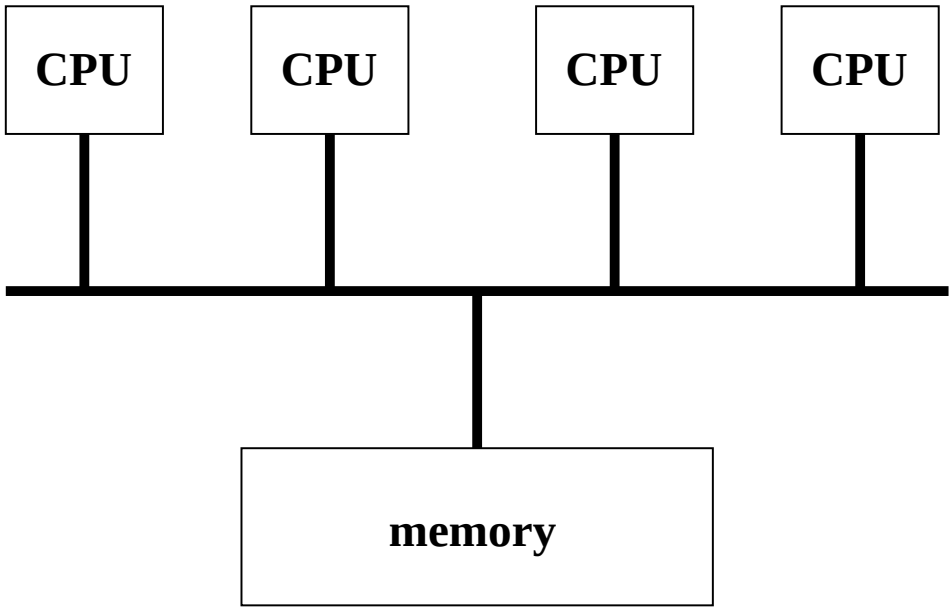
Supercomputers like *big* problems.

Conversely, workstations hate big problems, as their various caches become less effective and their overall efficiency falls.

SMP: Bused Based

SMP (Symmetric Multi Processor, Shared Memory Processor) describes a particular class of multi-CPU computer.

The original, bus-based, SMP computer simply has multiple CPUs attached to a single system bus.



The architecture is *symmetric* (all CPUs are equivalent), and the memory is *shared* between them.

Two Heads are Better than One?

As in a conventional, single-CPU computer, the single processor typically spends between 75 and 95% of its time waiting for memory, trying to 'feed' two or more CPUs from one memory bank is clearly crazy. The memory was, and is, the bottleneck. The CPU was not.

However the design is cheap, simple, still common, and therefore worth exploring further.

SGI's PowerChallenge, DEC's TurboLaser and many dual processor machines (including Intel's) have this architecture. DEC's DS20 and DS25, and Sun's SunBlade 2000, do not.

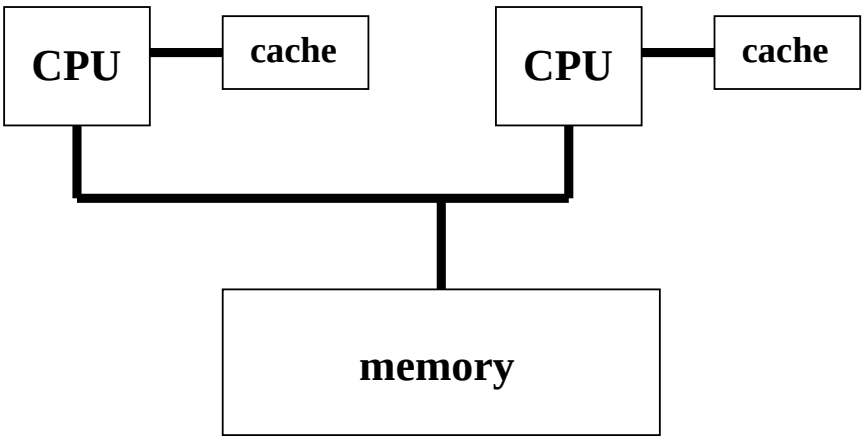
Shared memory

As all processors access the same main memory, it is easy for different parts of a program executing on different processors to exchange data. One CPU can write an array into memory, possibly from disk, possibly as the result of a calculation, then all other CPUs can read it with no further effort.

Programming is thus simple: all the data are in one place, and there is merely the little matter of dividing up the millions of instructions to be executed in a long loop between the multiple eager processors – a job so simple that the compiler can do it automatically.

Except it is not quite that simple.

Cache coherency



Processor A reads a variable from memory. Later, it reads the same variable, which it can now get directly from its cache, without troubling the system bus.

Only it can't. For what if processor B has modified that variable, and processor A needs the new value?

If processor B has a write back cache, the new value may not even have reached the main memory, with the current value being held in processor B's cache only.

Snoopy caches

The trivial solution is to abandon all caches.

An easy solution is to ban write-back caches, and to ensure that each cache '$snoops$' the traffic on the system bus, and, if it sees a write to a line it is currently caching, it must either update itself automatically, or mark its copy as being invalid.

These solutions severely compromise one's cache architecture, and often lead to a SMP machine generating more traffic to the main memory than a uniprocessor machine would running the same code. Thus a SMP machine can fail to reach the performance of a single-processor workstation based on the same CPU.

With either of these solutions, the definitive data are always those in the main memory.

Even single CPU workstations have a lesser version of this problem, as it is common for the CPU and the disk controller to be able to read and write directly to the main memory. However, with just two combatants, the problem is fairly easily resolved.

MESI solutions

A typical SMP has extra bits associated with each cache line, which mark it as being on one of four states:

- Modified (i.e. dirty)
- Exclusive (in no other cache)
- Shared (possibly in other caches too)
- Invalid

Modified implies exclusive, and a line must be exclusive before it can be modified.

A line fill for a read starts by ensuring that no other cache has the line modified, then loading the line marked as 'shared.' A fill for a write must ensure than any other cache with that line shared marks it invalid. In either case any cache with it 'modified' (there can be only one) writes it back to memory.

Thus a line can be:
In no caches
In one cache and marked as modified
In one or more caches and modified in none

Directory Entries vs Broadcasting

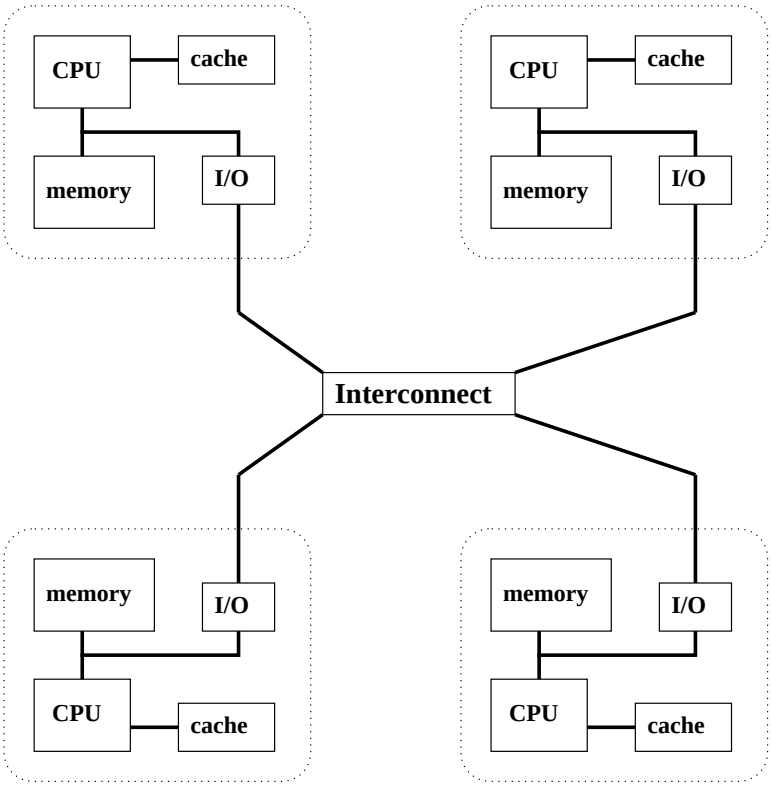
Two techniques are used for communicating with the other cache controllers. One is simply that details of all line fills are broadcast to all cache controllers, and the fill does not progress until the other controllers have had an opportunity to reveal that they held the line.

This is a simple technique, but the broadcast *coherency traffic* scales as the square of the number of caches, so it does not perform well beyond about eight CPUs.

Alternatively, each line in main memory can have a *directory entry* associated with it, which records which caches have copies of the line. Then a fill need simply check the directory, contact only those caches listed (probably none), and proceed, updating the directory as it does so.

MPP: Breaking the Memory Bottleneck

Rather than multiple processors sharing the same, 'global', memory area, each processor could have its own private memory, and no global memory. Adding processors adds more pools of private memory with their separate buses, and the total memory bandwidth increases in step with the number of processors. Such a computer is called a *distributed memory computer* or *massively parallel processor*



Breaking the Code

This arrangement is so far removed from the traditional model of a computer, that traditional code does not run on it. The programmer must be prepared to think in terms of multiple processors working on his program at once, each with its own private memory, and any interprocessor communication being explicitly requested.

Fortunately this is not nearly as hard as it might sound, and there are standard programming models to assist. Thus one can write code for a Cray T3E, using C or FORTRAN with MPI, and be confident that it will run, unmodified, on an IBM SP, a Beowulf cluster, or on a machine not yet developed. One merely has to follow the relevant standards and not be lured down the road of vendor-specific extensions. . .

MPI (1994) and PVM (1991, now obsolete) standardised the programming model for MPPs. Before PVM, each vendor had its own way of doing things.

Topologies

There are many different ways of connecting nodes together, as ever governed by cost and practicality.

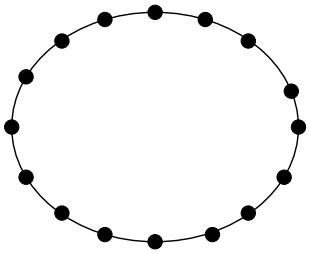
Two useful ways of characterising a network are the 'diameter', the maximum number of hops from one node to another, and the bisectional bandwidth, the bandwidth between two halves of the machine.

	Bandwidth	Diameter
Ring	2	$N/2$
2D Grid	$\sqrt{N}$	$2\sqrt{N}$
2D Torus	$2\sqrt{N}$	$\sqrt{N}$
Hypercube	$N/2$	$\log_2 N$
Tree	2	$2\log_2 N$
Fat tree	$N/2$	$2\log_2 N$
X-bar	$N/2$	1
3D X-bar	$N/2$	3

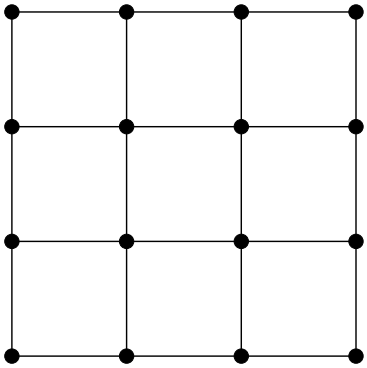
The Cray T3D is a 2D torus, the IBM SP2 a fat tree, the SGI Origin2000 a form of hypercube, and the Hitachi SR2201 a 3D X-bar.

Ideally the network topology should not be apparent to the user.

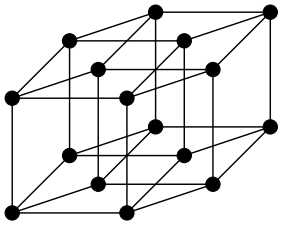
16 Nodes. . .



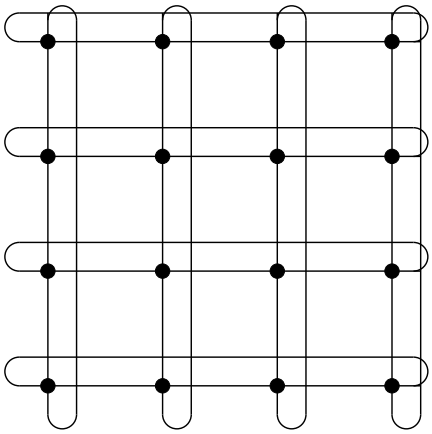
Ring (1D torus)



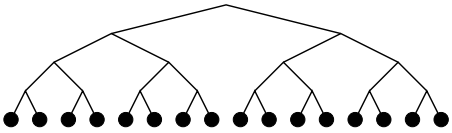
2D mesh



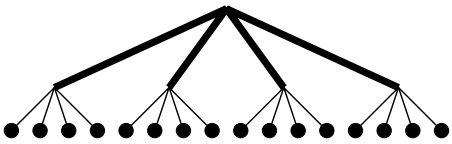
Hypercube



2D torus



Tree (log 2)



Fat Tree (log 4)

Performance

Another important characteristic of the interconnect is its raw performance, both bandwidth and latency. These are most usefully measured using a standard interface such as MPI, and not using the hardware directly.

Ideally the time to transmit a packet is simply

latency + size / bandwidth

If size $<$ latency $\times$ bandwidth, then the latency will dominate.

Also ideally communication between a pair of nodes is unaffected by any other communications happening simultaneously between other nodes. Such a network is called $non\text{-}blocking$.

Typical figures are 200 to 350 MB/s bandwidth and 10 to 30 μs latency. Clusters using 100MBit/s ethernet typically run at around 10 MB/s and 120 μs.

Parallelisation Overheads

Amdahl's law assume that there are no overheads associated with parallelisation. This is certainly a gross approximation.

Consider the case where each node must exchange data with every other node at some point in the program: some sort of rearranging of an array spread over all the nodes. E.g. an FFT

Each node must send $n-1$ messages of size a/n where a is the size of the distributed array. Even assuming that the nodes can do this simultaneously, the time taken will be

$$(n-1) \times \left(\lambda + \frac{a}{n\sigma}\right) \approx n\lambda + \frac{a}{\sigma}$$

where λ is the latency and σ the bandwidth.

Amdahl revisited

A better form of Amdahl's law might be

$$t_n = t'_s + t_p/n + c\lambda n$$

where $t'_s > t_s$.

Now t_n is no longer a monotonically decreasing function, and its minimum value is governed by λ.

This form stresses that the quality of the interconnect can be more important than the quality of the processors.

Hence 'cheap' PC clusters work well up to about 16 nodes, and then their high latency compared to 'real' MPPs starts to be significant.

Programming Example

Consider doing an enormous dot product between two arrays previously set up. The SMP code might look as follows:

```
! Let's hope the compiler optimises
! this loop properly

t=0.0
do i=1,100000000
  t=t+a(i)*b(i)
enddo
```

Easy to write, but little control over whether it is effective!

To be fair, HPF (High Performance Fortran) and OpenMP (a set of directives to Fortran and C) permit the programmer to tell an SMP compiler which sections of code to parallelise, and how to break up arrays and loops. One day I might meet someone using such a language for real research.

Programming, MPP

```
! Arrays already explicitly distributed
! Do the dot product for our bit

t_local=0.0
do i=1,nmax  ! nmax approx 100000000/ncpus
  t_local=t_local+a(i)*b(i)
enddo

! Condense results

call MPI_AllReduce(t_local,t,1,   &
   MPI_DOUBLE_PRECISION, MPI_SUM, &
   MPI_COMM_WORLD)
```

(Those MPI calls are not half as bad as they look once one is used to them!)

All the variables are local to each node, and only the MPI call causes one (`t`) to contain the sum of all the `t_local`'s and to be set to the same value on all nodes. The programmer must think in terms of multiple copies of the code running, one per node.

The Programming Differences

With MPP programming, the programmer explicitly distributes the data across the nodes and divides up the processing amongst the nodes. The programmer can readily access the total number of CPUs and adjust the distribution appropriately.

Data are moved between nodes by explicitly calling a library such as MPI.

With SMP, the compiler tries to guess how best to split the task up amongst its CPUs. It must do this without a knowledge of the physical problem being modeled. It cannot know which loops are long, and which short.

Artificial intelligence vs human intelligence usually produces a clear victory for the latter!

SMP: The Return

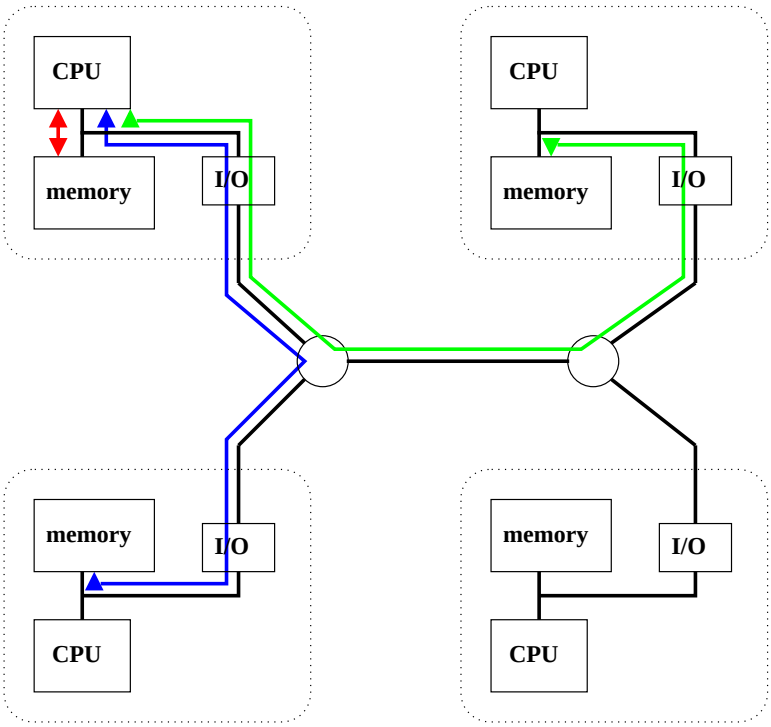
Most modern SMP machines are not bus based. Internally they are configured like MPPs, with the memory physically distributed amongst the processors. Much magic makes this distributed memory appear to be global.

This (partially) addresses the poor memory bandwidth of the bus based SMP machines.

However, there are problems. . .

And magic costs money, and, in this case tends to degrade performance over an MPP, providing instead increased flexibility.

NUMA / cc-NUMA



Four nodes in a tree configuration giving three different memory access times: on node, nearest neighbour and next-nearest neighbour.

If caches are to be added, the lack of a single common bus to snoop requires that a broadcast or directory coherency protocol be used.

NUMA = Non Uniform Memory Access
cc-NUMA = Cache Coherent NUMA

The Consequences of NUMA

If a processor is mangling an array, it now matters crucially that that array is stored in the memory on that processor's node, and not on memory the other side of the machine. Getting this wrong can drop the performance by a factor of three or more instantly.

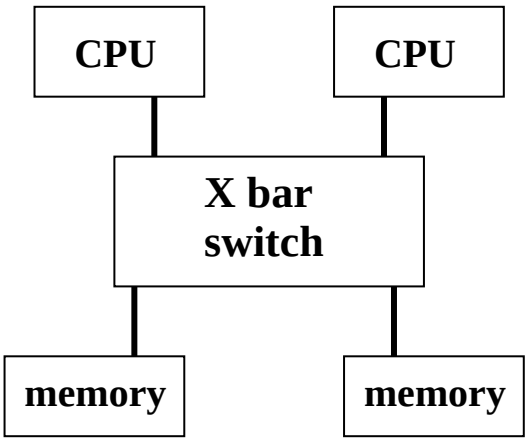
Whereas with MPP all memory accesses are guaranteed to be local, as one cannot access remote memory except by explicit requests at the program level, with SMP the compiler has many ways of getting things wrong.

```
for(i=0;i<10000000;i++)
   t+=x[i]*y[i];
```

Consider this on a two node NUMA machine. If the code is split so that node A stores the first 5000000 elements of each array, and does the first half of the loop, then optimal performance is obtained. If node A stores the whole of `x` and node B the whole of `y`, then much reduced performance will result.

Modern, small SMPs

A decent, modern, small SMP machine, such as Compaq's DS20 or ES40, IBM's Sphinx, SGI's Octane or Sun's SunBlade2000 uses a rather different architecture.

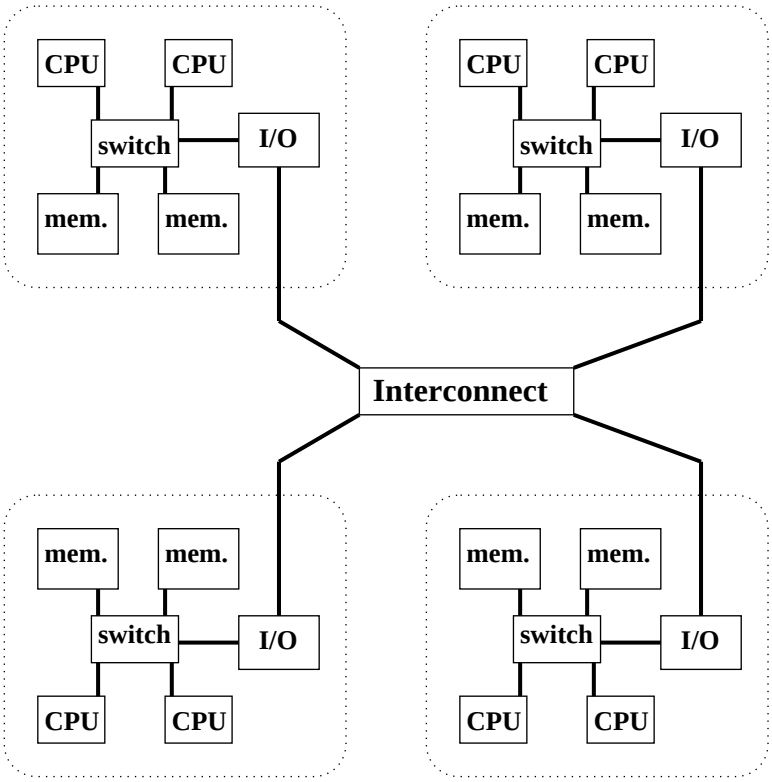


In this two CPU example, there are two distinct memory 'banks' and two CPUs joined by a switch which will let either CPU talk to either memory bank whilst the other pair also talk simultaneously. This fails when both CPUs wish to access data in the same memory bank.

This sort of design works for up to 4 or maybe 8 CPUs. After that point the crossbar switch becomes very expensive, and the chance that the CPUs are not fighting for the same memory bank rather low.

Modern, large MPPs

The latest MPP designs (Hitachi SR8000, IBM SP3, Compaq SC45) join SMP nodes like the above.



Such a machine is awkward to program, as one has both internode and intranode parallelism to address.

Modern, large SMPs are just the same. The Origin2000 has two CPUs per node, the Origin3000, Compaq GS320 and SunFire15K have four.

Multithreading

Whether in a uni- or multi-processor computer, the CPU is often used very inefficiently, with most of its functional units idle waiting for memory to respond or data dependencies to be resolved. It is rare for a four-way superscalar CPU to be able to issue four instructions simultaneously.

Conventional multitasking is not the answer. This software-driven process-switching takes thousands of clock cycles, so is useful for latencies caused by disk drives, networks and humans.

However, there are rarely data dependencies between processes, so in some sense multitasking is the answer.

A multithreading processor gains multiple banks of registers, one per 'thread' (process) which will be run simultaneously. These processes share access to the functional units, caches, instruction decoding logic, etc.

SMT

There are different ways of achieving multithreading. Some change thread every clock-cycle, whereas the more advanced Simultaneous MultiThreading architecture allows instructions from different threads to be issued in the same clock-cycle.

The extra logic on the CPU need to keep track of a modest number of threads is very small, increasing the CPU size by less than 10%. The gain is zero if the computer is only ever running a single thread, but the throughput can increase by over half when two threads are run.

Two MultiThreading Architectures (MTAs) currently exist. One, developed by Tera (now Cray), supports 128 threads per processor (prototype delivered 1998). The other, Intel's 'Pentium4 with Hyperthreading' (2002), supports two threads per processor. The now-cancelled EV8 Alpha was to support four-way SMT, but other MTAs are in development.

Permanent Storage

Disk Drives

Are remarkably boring, but worthy of mention because people do rely on them rather a lot. . .

Remarkably standard, with just two interfaces dominating the market for hard disks and CD ROMs: SCSI at the expensive end, EIDE (aka UDMA) at the cheap end.

SCSI: Small Computer Systems Interface, a general-purpose interface which can support scanners and tape-drives, and, depending on the flavour of SCSI, several metres of external cable. Each SCSI interface (or *channel*) can support seven devices.
EIDE: Enhanced Integrated Drive Electronics. Designed for internal disk drives only, with short cable lengths and just two devices per channel.

Physical considerations

A single hard disk contains a spindle with multiple $platters$. Each platter has two magnetic surfaces, and at least one head 'flying' over each surface. The heads do fly, using aerodynamic effects in a dust-free atmosphere to maintain a very low altitude. Head crashes (head touching surface) are catastrophic. There is a special 'landing zone' at the edge of the disk where the heads must settle when the disk stops spinning.

The size of a drive is such that it fits into a standard $3\frac{1}{2}$" drive bay, which is just 10cm wide and 1" tall for the whole assembly.

Spin speeds were 3,600 rpm in the mid 1980s, and now 7,200 to 15,000 rpm. Capacity has grown over the same period from typically 20MB to typically 60GB.

Drive bays are 1" tall, or $1\frac{3}{4}$" tall (half height), or $3\frac{1}{2}$" tall (full height). Their width is 10cm (called '$3\frac{1}{2}$ inch') or 15cm ('$5\frac{1}{4}$ inch'), though the imperial width measurements refer to the size of floppy disk taken by a drive which fits in given width. Laptops use yet smaller drives.

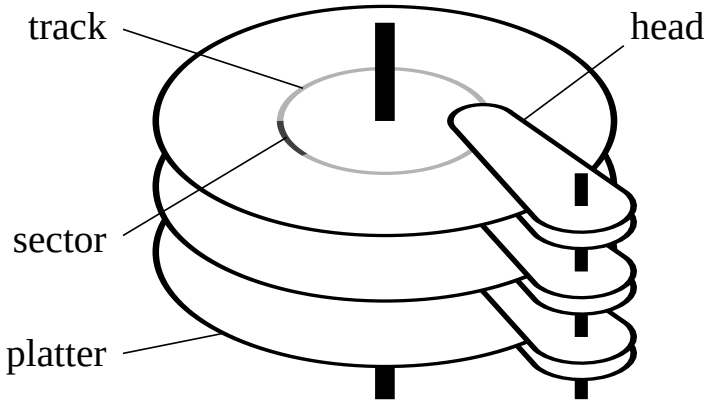
Although the heads move only radially, the air is dragged into tangential motion by the spinning platters, and in this air stream the heads fly.

Data Storage and Access Times

Data are written in concentric *tracks* on each platter. Each track is subdivided into *sectors*. An individual sector typically records just 512 bytes.

For data to be read, the disk heads have to move into position, and then wait for the correct piece of disk to rotate past. The head *seek time* is typically around 7 ms, and the rotational latency is 3 ms at 10,000 rpm.

In other words, the bandwidth is about 20 times lower than main memory, but the latency is over 30,000 times higher.



This disk has three platters and six heads. In reality the heads are much smaller than shown above.

A modern (IBM) 36GB disk has 5 glass platters with a total of 10 heads. It records at 13,500 tracks per inch, and 260,000 bits per inch along the track. The raw error rate is about 1 in 10^{12} bits, reducing to 1 in 10^{14} after automatic correction. The sustained data transfer rate from the physical disk is 15 to 30MB/s.

Floppy drives

The original floppy drives (8 inch and $5\frac{3}{4}$ inch) were genuinely floppy. The current $3\frac{1}{2}$ inch variety are rigid, and have the following specification:

Two sides, with one head per side
Eighty tracks per side, 135 tracks per inch
18 sectors per track
512 bytes per sector

Total unformatted capacity: $2\times80\times18\times512 = 1440$K.

The disk is spun at 360 rpm, and the heads are in contact with the disk surface.

This specification has been static since the late 1980s, as has the bizarre, pre-EIDE interface that most floppy drives use.

CD drives

There are many obvious differences between a CD drive and a hard disk drive. A CD is physically 12cm in diameter, and single-sided. The drive therefore fits into the older 15cm wide bays.

The single head is optical, and is physically larger than the tiny magnetic sensors used for hard drives. Thus seek times are around ten times higher at 80ms.

The transfer rate for an audio CD player is 150KB/s, and data drives are expressed as multiples of this, so a 40× drive is 6MB/s.

The data are written onto a single spiral track, starting at the centre. The capacity is around 650MB.

Wot no files?

Disk drives do no more than store blocks of data. The blocks are typically 512 bytes, and the commands between the computer and disk drive look like:
Give me block number 43578
Write these 512 bytes to block 1473

A disk drive has no concept of a 'file'.

Different operating systems conjure files out of disk drives in different ways. We shall consider a couple in detail.

File systems: the requirements

First let us consider what a file system needs.

- a concept of a 'file' as an ordered set of disk blocks.
- a way of refering to a file by a textual name.
- a way of keeping track of free space on the disk.
- a concept of subdirectories.

There are other things which would be useful too, as shall be discussed.

The data which describes the files is called 'metadata', as opposed to the plain data which the files contain.

An example: FAT16

As a first example, we shall consider FAT16, the filesystem used by DOS.

The name means 'File Allocation Table, 16 bit,' and a prominent feature is the FAT.

The disk is divided into fewer than 2^{16} clusters, each of which is then identified by a 16 bit number.

The FAT is a table with one 16 bit entry per cluster. If the entry is 0, the cluster is unused, if 65535, the cluster is the last in a file, and otherwise the FAT entry contains the cluster number of the next cluster in the file.

The limit of just under 65536 clusters per disk can make clusters quite large leading to poor use of space. On a 1GB partition, the cluster size would be 16K, leading to an average of 8K wasted per file.

On partitions of under 32MB, the cluster size is 512 bytes, or one block, the smallest possible size.

Chains

FAT entry number	value
0	1
1	2
2	65535
3	0
4	65535
5	8
6	65535
7	0
8	6

Here we see two free clusters (3 and 7) and three files occupying clusters 0, 1 and 2, cluster 4, and clusters 5, 8 and 6. Such sequences of clusters in the FAT are called ‘chains’.

So the FAT has already given us the concept of a file, but not of a filename.

The metadata in the FAT are so important that DOS stores the FAT twice at the beginning of a disk.

A directory

Immediately following the two copies of the FAT is the root directory. Like every other directory, it contains a 32 byte entry per file, with the following information:

File name (8 bytes)
File extension (3 bytes)
File attributes (1 byte)
Last modified time (4 bytes)
Starting FAT entry (2 bytes)
File size (4 bytes)
Reserved (10 bytes)

The bits in the attribute byte indicate things such as whether the entry is a file or a subdirectory, whether it is read-only, whether it should be hidden from directory listings, etc.

The root directory is of fixed length. No other directory is.

Every subdirectory contains at least two entries. One, called '..', which describes its parent directory, and one, called '.', which describes itself.

Simple operations

File Deletion
The directory entry has the first byte zeroed, and the corresponding FAT entries are marked free.

File Creation
An unused directory entry is found and used, and a FAT chain of at least one block created.

File Renaming
Only the directory entry needs changing.

Appending to a file
The file length in the directory needs modifying, and possibly a new cluster allocating and the FAT changing, as well as writing the data.

etc.

Consistency

There are many ways in which a DOS filesystem can become inconsistent. A consistent one has the following properties:

- The two copies of the FAT are identical
- The FAT contains chains, but no loops.
- Every chain has precisely one directory entry pointing at it.
- Every directory entry points to the beginning of a chain.
- The filesizes in the directory entries are consistent with the corresponding chain lengths.

The programs `chkdsk` and `scandisk` check these consistency issues.

`chkdsk` = CHecKDiSK

The 1.4MB DOS Floppy

Having just 2880 sectors of 512 bytes, 16 bits per FAT entry is excessive, so a FAT12 format is used. The sectors are used as follows:

Sector no.	Use
0	Boot sector and format description
1-9	FAT, 1st copy
10-18	FAT, 2nd copy
19-32	root directory
33-2879	data (1423.5K)

Hence the largest file one can store on a 1440K DOS-formatted floppy is 1423.5K, and the root directory can hold 224 entries.

If a 16 bit FAT had been used, then 2880 entries would require 5760 bytes or 12 sectors, rather than just 9. An extra 3K would be lost.

Sector 0 contains information such as the size of the FAT (12 or 16 bit), the size of the root directory, the cluster size, etc.

Other FATs

FAT32 was recently introduced, and makes the obvious extension to the size of the FAT. Thus smaller cluster sizes can be used on large disks.

VFAT is a FAT-like filesystem which supports long, mixed case filenames. It does this by using several of FAT's directory entries for each file, keeping a FAT-like one holding a 'short' file name, and marking the additional ones as hidden files taking zero space so that one rarely sees them listed. The resulting disk is fully usable by a system which supports FAT but not VFAT. VFAT does have the air of a nasty hack, rather than a well-thought-out solution.

VFAT uses Unicode not ASCII to store filenames. This permits all sorts of exciting foreign characters, at the expense of using two bytes per letter, not one.

The UNIX file system

Every UNIX vendor has one (or more) file systems of his own. However, the traditional UNIX file system (UFS) has the following features.

Unlike DOS, which splits its metadata between the FAT and the directory entry, UNIX has three locations: the block bitmap, the index node (inode) and the directory entry.

The block bitmap simply contains one bit for each cluster (block) on the disk, and marks whether the cluster is free. One can have up to 2^{24} or 2^{32} clusters typically.

The directory entry is also simple: a variable-length field containing the name, and a field giving an index into the inode table.

The original UNIX filesystem was even simpler, with fixed-length 16 byte directory entries containing a 14 character name and a two byte i-node number.

Again every subdirectory contains explicit entries for '.' and '..' giving its own and its parent's inode number.

The inode table

The inode table follows the block bitmap at the beginning of the disk. It is of fixed size, containing a fixed number of fixed-length records (typically 128 bytes each), each describing one file. Each record contains:

File length
File ownership (user and group)
File 'creation', modification and last access times
File access permissions
The number of directory entries pointing at this file
A list of the first ten clusters occupied by the file
Three pointers to clusters containing details of further clusters used

Again, the block bitmap, inode table and directory entries must all be consistent.

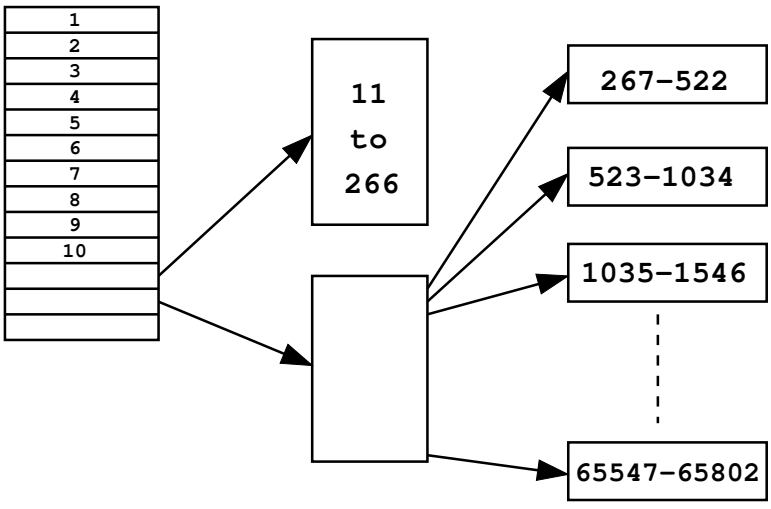
The file and group ownership records the numeric user id (typically 32 bits), not the eight character textual user name.

The program `fsck` checks for consistency. `fsck` = File System CHeck.

Large files

Files smaller than 10 blocks have the complete list of blocks used in their inode. Longer files use an entry which points to a disk block filled with a list of the next blocks used. If a block number is 4 bytes long, and a block is 1K, this gives another 256 blocks.

For larger files, the inode has another entry pointing to a block filled with entries pointing to blocks containing the rest of the list! This adds another 65000 or so blocks.



In this example, one would need another level of indirection to support files larger than 64MB. In practice, the block size is probably 4K, and this scheme will therefore work up to 4GB.

File types

So far we have seen two file types: ordinary files and directories. UNIX also has two forms of link.

The first, the hard link, is not a new file type at all. One merely has two directory entries pointing at the same inode. As the inode stores the information about file length and access times, there are no consistency problems as there would be with FAT.

One can construct two directory entries pointing a the same chain within FAT, but if the file is modified using one name, the file length stored under the other directory entry will not be changed, and a mess results.

The link count in the inode keeps track of how many directory entries point to that inode, and only when deletion reduces the count to zero are the inode and data blocks actually freed.

All directory entries pointing at the same inode are equivalent, and must reside on the same filesystem.

Hard links to directories are not permitted.

Symbolic links

A symbolic, or soft, link is a new file type. The file simply contains an indirection, saying 'don't look at me, look over there instead.'

```
tcm30:/usr/sbin> ls -l /usr/sbin/sendmail
lrwxrwxrwx  1 root  system  24 Sep 12 1998
      sendmail -> /usr/local/exim/bin/exim
```

Any references to `/usr/sbin/sendmail` will be redirected to `/usr/local/exim/bin/exim`. The `l` at the beginning of the permissions bits indicates that this is a symbolic link. The rest of the permissions are ignored (the file is *not* world-writable!).

The file length, 24 bytes, is the number of characters in the filename linked to. This name is stored as the file 'data'.

The link and the file linked to are quite distinguishable, and need not be on the same filesystem.

UNIX will check for circular paths in symlinks.

More speed

Having to move the disk heads to the beginning of the disk to fiddle with the FAT or inode table, then out to the centre to deal with the real data, is clearly inefficient.

All file systems except FAT deal with this in practice by dividing the disk into zones, and placing the part of the block bitmap etc. dealing with each zone within that zone. Therefore the metadata is (probably) physically close to the real data it describes.

Also, to avoid scanning large amounts of the inode table when looking for a free entry, a bitmap showing which entries in the inode table are free is often maintained.

A zone could be called a group.

The ext2 floppy

Ext2 is a variant of the UNIX filesystem. When used on a 1.44MB floppy, it arranges things in roughly the following way, using blocks of 1KB.

Cluster no.	Use
0	Unused
1	Superblock
2	Group descriptor
3	Free block bitmap
4	Free inode bitmap
5-49	Inode table (360 entries)
50	Root directory
51-1439	data (1389K)

The disk will be filled by a 1382K file, for seven blocks are needed for storing the block list for such a file.

Ext2 is much more configurable than FAT16. One could reduce the total number of inodes to 32, giving a maximum file size of 1423K. The block size can also be changed, but 1K is the minimum.

Fragmentation

For optimal speed, a file should be stored in a single set of contiguous blocks. However, once files start being deleted, the free space on a disk becomes fragmented, and files subsequently written are in danger of being fragmented too. The situation tends to get worse as the disk gets fuller.

DOS's allocation strategy is very poor: it always allocates the first available free cluster whenever a file needs to grow. It takes no account of which clusters the file is currently occupying.

UFS has two weapons to control this. Firstly a more intelligent block allocation algorithm which tries to avoid excessive fragmentation. Secondly, it always keeps 5% of the blocks free, which also tends to reduce fragmentation.

These 'reserved blocks' can be used by root. It gives the OS the chance to clear up, or at least die gracefully, if a user fills up an important disk, for the OS can still find free blocks if it wants them.

FAT vs HFS vs Ext2

(HFS is the Apple filesystem used from the Mac+ to MacOS 8.2)

	FAT16	HFS	ext2
Max filename length	8+3	31	255
Directory separator	\	:	/
Max no of clusters	2^{16}	2^{16}	2^{32}
Last modify time	Yes	Yes	Yes
Creation time	No	Yes	Yes
Last access time	No	No	Yes
Creator (application)	No	Yes	No
Owner (user)	No	No	Yes
Max file size	2GB	2GB	2GB
Max disk size	2GB	2GB	16TB(?)

HFS has one odd feature: all files consist of two 'forks'. The data fork is simply a stream of bytes, like a normal file on any other system. The resource fork is a set of records rather like a simple database. It often holds fonts, menu structures, preferences, icons and similar things. File transfer between MacOS and the rest of the world is considerably complicated by this unusual feature.

Ext2 does support filesizes of up to 2^{64} bytes, but not all implimentations manage this. Linux 2.4 does.

Partitioning

One might wish to put several filesystems on the same physical disk. Perhaps FAT16 and ext2 for a Windows / Linux dual boot computer, or perhaps two FAT16 filesystems because one has a 4GB disk.

This is done by breaking the disk into *partitions*. The disk starts with a *partition table*, which describes the number of partitions on the disk, and where they each start and finish. The OS uses a partition as if it were a complete independent disk, and thus the term 'logical disk' is sometimes used.

Partitions cannot be resized or moved without destroying the filesystem they contain unless much magic is applied.

If one has a 1GB disk and wishes to run Windows95, one can choose a single partition with a 16K cluster size, or, maybe, two 512MB partitions each with an 8K cluster size and with the advantage that the FAT for the second half of the disk is (probably) then stored in the middle of the disk, physically closer to the data it describes. A poor man's zoning is thus achieved.

The downside occurs when each partition has 20MB free and you wish to write a 30MB file. . .

The partition table usually exists, even if it shows just one partition using the whole disk.

Still slow

Disk drives are mechanical devices, and thus are always going to be much slower than solid-state devices such as memory. Vaguely realistic performance data for current (2000) machines would be:

	Disk	Memory
Latency	5-15 ms	100-200 ns
Bandwidth	10-30 MB/s	300-1000 MB/s

The latency for disk drives, caused by the time taken to move the heads to the correct part of the disk, is particularly poor.

Even if the disk heads do not need moving, just waiting for the disk to rotate to the correct position takes a while. The typical half revolution at 10,000 rpm is 3 ms.

Caching

Using memory to store frequently-accessed files in a cache makes a dramatic improvement to perceived disk performance. Even caching just things like the block bitmap, FAT and frequently-used directories can make a marked difference to performance costing relatively little memory.Such metadata are usually preferentially cached.

For more speed, *write behind* caching occurs. When a program writes to a file, the data are written to the memory cache only, and the program can continue immediately. Later, the data are written out to the disk at its slow speed whilst the program continues to run.

Just like a write back memory cache, this form of write back disk cache needs dirty bits to record which entry need writing back to the disk.

Writing to disk does not over stress a modern CPU: if it has nothing else to do it will be idle waiting for the disk to process the data for much of the time.

Write collapsing

Consider deleting fifty files from a FAT-based directory. This results in the following operations:

```
do i=1,50
  move heads to directory
  read directory
  write out directory with zero in 1st byte
    of filename of file deleted
  move heads to 1st FAT
  read it
  write out with relevant chain marked free
  heads are now at second copy of FAT
  fix that too
done
```

With two long head seeks per file, or one hundred in total, this will take about a second.

Writes collapsed

Consider the alternative with a write-behind cache:

```
do i=1,50
  read directory from cache
  write out directory to cache with a zero
   in 1st byte of filename of file deleted
  read FAT from cache
  write it back with relevant chain freed
  ditto second FAT
done

move heads to directory
write modified directory from cache to disk
move heads to FAT
write modified FATs to disk
```

With just two head movements, this will be done in well under a tenth of a second.

Intelligent caches will also reorder writes to minimise head movements.

Inconsistencies

A file system will be consistent before and after a file is deleted, but not during the deletion: the directory might be changed but the FATs not.

And clearly with a write-behind cache, the data on the disk need not be the same the data in the cache.

Hence it is important to tell a computer to finish all disk operations and to send all modified data from its cache to the disk before turning it off. This is called *flushing* the cache, or *syncing* the disks.

('Syncing' abbreviates 'synchronising', so is similarly pronounced.)

Any filesystem which records last access times (such as UFS) will be frequently modifying data on disk.

UNIX systems, and some versions of Windows, will detect if they have been turned off without being shutdown properly, and check their disks for consistency when they are next turned on. If they have been shutdown correctly, they don't bother.

Though `fsck` and `scandisk` can often autorepair a filesystem to a consistent state, it is worth pointing out that consistency and correctness are different: formatting a disk also reduces its filesystem to a consistent state, but in a slightly unhelpful manner.

Journalling filesystems

Because checking filesystem consistency is painful on large fileservers – it can often take over an hour – various filesystems which never need a full consistency check have been developed.

They all work by keeping a log, or journal, of operations which they are about to do. Deleting a UNIX file might be broken down as:

```
write to journal 'I am about to remove this
  directory entry, free this inode, and mark
  these clusters as free.'
do the above
remove the journal entry
```

After a crash, the journal is scanned and those entries which have not been completed are finished.

A journalling filesystem must flush the journal from cache to disk before attempting the updates described by the journal.

Digital UNIX has AdvFS as a journalled filesystem, Irix has xfs, AIX has jfs, Linux has ext3, and WinNT has NTFS.

Journal problems

Journalling produces a significant performance penalty, as every write is turned into two: one to the journal, and one to the real file. For this reason most journalled filesystems only journal metadata.

Journalling metadata can ensure that the filesystem remains consistent, and guards against the type of errors which can cause whole directories to vanish. The contents of files can still be corrupted by crashes.

Journalling data as well as metadata is a serious performance penalty, and requires a much bigger area for the journal. Many journalling filesystems do not support data journalling at all.

The final problem with journalling is that hardware errors or bugs in the OS can still cause a journalled filesystem to become inconsistent. Because the recovery tools for journalled filesystems are used less frequently, they tend to be less tested and less effective.

Linux' ext3 and Solaris' UFS support journalling and still use the same layout as the older, non-journalled, filesystem they are based on. Hence the old recovery tools are valid.

Remote files

It is often convenient to use files physically located on a remote computer as though they were stored locally. This UNIX, MacOS and Windows can all do, and you do every time you use TCM's computers, for your UNIX home directory is physically on `tcms`, and your NT home directory `tcmp`. Neither do you usually touch directly.

That UNIX, MacOS and Windows use three completely incompatible protocols for this will be no surprise.

In all cases there is a speed and reliability penalty to pay compared to local disk access, but the increase in convenience can be great.

Disk drives not only typically have a higher bandwidth than networks, but also a lower latency, especially once the overheads of going through the networking protocols is considered.

On the other hand, it makes it possible to use a machine with no internal disk drive.

Remote trouble

File sharing can cause various forms of trouble. Multiple computers might try to change the same file at once. The network might die at an inopportune moment. And then there is security.

NFS, the remote filesystem that UNIX uses, leaves the task of imposing access restrictions on files to the client computers. That is, TCM's server exports everyone's home directory to all TCM's machines. The individual machines are responsible for keeping other users from editing my files. This requires the client to run a secure OS, and rules out NFS exporting to DOS, Win98 or MacOS.

Under Windows, the server imposes the access controls having determined which user is logged on to the client machine.

Multiple filesystems

DOS, Windows and MacOS present each filesystem to the user as a separate 'disk drive.' With DOS, they are called friendly things like `C:`, `D:` and `E:`, whereas MacOS pops up icons with configurable textual names.

UNIX does things rather differently. It presents a single directory tree with a single root directory. Different filesystems are then grafted on to that tree. On a typical TCM Alpha, there are three filesystems resident on local disks: `/`, `/usr` and `/temp`. There are also several remote filesystems including `/u/tcms` (where the home directories reside), `/var/spool/mail` (where email is delivered), and `/usr/local/shared` (where many applications are to be found).

The joins between these filesystems are almost invisible to the user, and programs like '`mv`' automatically switch between doing a rename if moving within a filesystem, to a copy then delete if moving between filesystems.

'`df -k .`' will tell you where you really are.

Mounting filesystems

The process of 'grafting on' a filesystem under UNIX, or mounting it, is always done explicitly (unlike DOS which finds all local filesystems itself). If a filesystem is mounted as being modifiable, it is immediately marked as being 'dirty.'

Unmounting, which will happen on shutdown or when requested, causes all cached data referring to that filesystem to be written out, and then the dirty bit reset. A crash leaves the dirty bit set, and prompts `fsck` to run.

With most UNIXes only root can mount or unmount. TCM's linux machines allow users to mount `/cdrom` (where present) and `/floppy`. Remember to unmount things before trying to eject them.

With CDs, being read-only, it hardly matters, but the eject button will not work until you do. With floppies, being read-write, it does matter, and the eject button will work even if you don't.

Multiple programs

What happens when two programs try to manipulate the same file? Chaos, often.

As an example, consider a password file, and suppose two users change their entries 'simultaneously.' As the entries need not be the same size as before, the following might happen:

User A reads in password file, changes his entry in his copy in memory, deletes the old file, and starts writing out the new file.

Before A has finished, user B reads in the password file, changes his entry in memory, deletes the old, and writes out the new.

It is quite possible that A was part way through writing out the file when B started reading it in, and that B hit the end of file marker before A had finished writing out the complete file. Hence B read a truncated version of the file, changed his entry, and wrote out that truncated version.

Locking

The above scenario is rather too probable. It is unlikely that one can write out more than a few 10s of KB before there is a strong chance that your process will lose its scheduling slot to some other process.

UNIX tacked on the concept of file locking to its filing systems. A 'lock' is a note to the kernel (nothing is recorded on disk) to say that a process requests exclusive access to a file. It will not be granted if another process has already locked that file.

Because locking got tacked on later, it is a little unreliable, with two different interfaces (`flock` and `fcntl`), and a very poor reputation when applied to remote filesystems over NFS.

As the lock is recorded in the kernel, should a process holding a lock die, the lock is reliably cleared. This does not happen as reliably over NFS, because the lock is recorded in the kernel of the server, not in that of the machine the process is running on.

Microsoft, trying to be positive, refers to 'file sharing' not 'file locking.'

Dot locking

Another exciting form of locking is called ‘dot locking’. In this scheme, if an application wishes to lock a file called ‘`foo`’, it merely creates a file called ‘`foo.lock`’.

This has the advantage of requiring no support from the kernel.

It has the disadvantages of requiring other applications to understand the meaning of these `.lock` files, and of lock files being left around should the application crash or otherwise die unexpectedly.

Quotas

Another feature which got 'tacked on' later, but one which might interest some of you. . .

Quota information is recorded on a per-disk basis, usually in a file in the top directory of that disk. Every time a file changes size, the quota information for the relevant user is changed too. If the quota information becomes out-of-step with the real disk usage, there is nothing to correct it.

Except that on boot Digital UNIX machines tend to recalculate all their quota information based on how much data is really on the disks. This can cause people's disk usage as seen by the quota system to jump suddenly when the fileserver is rebooted. . .

Quotas over NFS do not work very well: a write when over quota returns an error and sets `errno` to 69. Few applications report the cause back the the user in a friendly fashion.

Mirrors

Another way of increasing reliability is for the OS to maintain identical data on two separate disks. The combination is treated as a single $virtual$ disk, with any attempt to write to a block modifying the relevant block on both disks. If one physical disk fails, there is no data loss.

The filing system accesses only the virtual disk, the mirroring occuring one level lower than the filing system. The filing system thus needs no modification.

Drawbacks include costing twice as much, being slightly slower for writing, and, whereas shutting the machine down properly will mark the mirrors as being synchronised, not doing so will potentially leave the mirrors different. This then needs to be checked and corrected by reading every block from both disks: much slower than a file system consistency check.

RAID

RAID introduces more ways of building virtual disks out of physical disks. Three levels are commonly used.

Level 0 is simple concatenation: take n 72GB disks, and treat as a single $n \times 72$GB disk.

Level 1 is mirroring.

Level 5, which requires at least three physical disks, is a mixture of mirroring and concatenation, where the capacity for n disks is $(n-1)\times$ that of one, and a single disk failure produces no data loss.

RAID: Redundant Array of Inexpensive/Independent Disks.

Level 0 is very sensitive to failure: one disk fails, and all the data are lost. Level 5, which uses parity blocks, can be quite slow for writing, as parity blocks will need updating, possibly requiring additional reads. Rebuilding a level 5 RAID set after a power cut is also very slow.

CDs

A CD can contain any filing system, just as a normal disk can. However, most CDs use the ISO9660 system. This was developed as a 'lowest common denominator', so filenames were limitted to MS DOS's '8.3' scheme, and most exotic features were absent. VMS-style version numbers were included though.

Extensions were quickly produced, and the 'Rock Ridge' extensions are almost universal. They permit long mixed case file names, symbolic links and UNIX-style file ownership and permissions. The latter are almost useless, for the numeric user ID is (necessarily) used. I (mjr19) am 2719 on CUS, 264 in Physics, 10084 on the HPCF. . .

Microsoft just had to produce its own extensions, called Joliet, incompatible with Rock Ridge but adding little extra.

CD-Rs

Write-once CDs are awkward. As we now understand writing a file to a disk involves the rewriting of a directory sector and of the FAT or inode table. Thus write-once media are useless.

There are three solutions:

- Prepare an image on a rewritable disk, then, when complete, copy to the CD-R.
- As above, but do it all in memory.
- Use a completely different filesystem.

I regard the first solution as being the sanest: it is how CDs in TCM are burnt.

Rewritable CDs (CD-RW) do not avoid this problem, for a CD-RW is really a CD-R with an extra option of 'erase whole disk'.

Tapes

No discussion of filing systems would be complete without a word about tape drives.

A tape drive is not a disk drive.

That should be obvious: a disk drive might have a head seek time of 8 ms, a tape drive is likely to have one of over 30 s. It is simply not reasonable to treat a tape drive as though it were a disk drive.

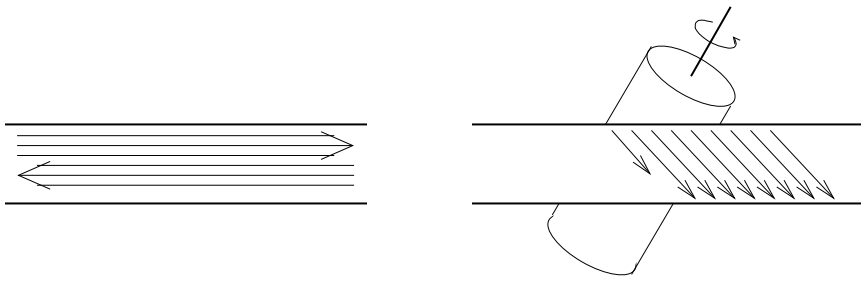
Tapes ideally store a single file each. Just data are stored, with no metadata (name, length, owner etc). The only metadata that a tape drive really understands are the 'end of file' mark and 'end of tape' mark. Thus it is possible to put several files on one tape, and then index the result by hand with a pen.

There are schemes for using the first sector of a tape to store a brief index, but unfortunately these schemes appear to be far from completely universal.

Tape Technologies

There are two competing technologies used in tapes. Linear and sepentine recording use tracks parallel to the length of the tape, often laid down in multiple passes. An example is DLT 8000, where the head records four tracks at once across part of the width of a $\frac{1}{2}$" tape, and then moves down slightly and reverses direction for the next pass, finally building up 208 tracks.

The other method is helical scan, used by DAT tapes and VHS video recorders. The tracks are oblique to the length of the tape, and created by a spinning cylindrical head. The requirement to wind the tape partially around the head stretches the tape slightly, and reduces reliability. Problems also arise if the angle of the head changes, either over time or between drives.



Serpentine Helical

Current tapes

Currently (2003) all tape drives offer automatic data compression as they record. They then 'cheat', by quoting capacities and transfer rates assuming a 2:1 compression ratio. As data are often uncompressible, the 'raw' uncompressed sizes are given here.

DAT: 4mm tape, helical scan. DDS4 gives 20GB per tape and 3MB/s.

DLT: $\frac{1}{2}$" tape, serpentine. DLT 8000 is 40GB per tape and 6MB/s.

LTO / S-DLT: Two competing $\frac{1}{2}$" serpentine standards giving around 100GB per tape and 15MB/s.

AIT: 8mm helical scan, 100GB per tape 12MB/s.

DAT: Digital Audio Tape (DDS: Digital Data Storage)
DLT: Digital Linear Tape
LTO: Linear Tape Open, consortium of IBM, HP and Seagate.
S-DLT: Super DLT. Quantum.
AIT: Advanced Intelligent Tape. Sony.

Note it takes over 2 hours to read any of the above tapes in full.

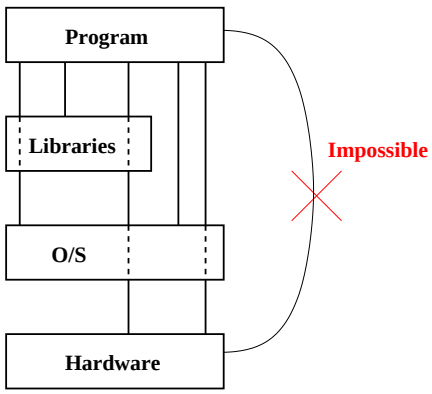
Practical Programming

Programs, Libraries and OSes

The operating system has full control of all aspects of the hardware. Anything which requires action from the hardware – reading a file, writing to the screen, allocating memory – must be handled by the OS.

The OS is both fair and friendly. It prevents other people reading your files, and other processes writing over your memory. It will also create the concept of a file from the blocks on a disk, and a network connection from the packets arriving at its network card.

Libraries are only friendly. They consist of simple, unprivileged code which one can use in an identical fashion to a subroutine of one's own creation. They exist to save reinventing the wheel too often.



Libraries (1)

Whereas programming languages provide standard maths functions, CPUs do not. Very few CPUs can deal with any transcendental functions, yet most languages have some. A library provides for the shortfall, different libraries for C, F77 and F90.

Similarly programming languages provide standard ways of doing input and output, e.g. `printf` in C and `write` in Fortran. The OS does not provide precisely these functions, but a library exists to convert them into whatever operation(s) are necessary to provide that functionality from the OS. Indeed, most programming languages provide no mechanism for calling the OS directly.

Thus the same piece of Fortran or C can be compiled and then run on different operating systems and CPU with libraries providing the translation between the features of the CPU and OS, and the features required by the programming language.

Libraries (2)

The other common use for libraries is to solve the more difficult maths problems: numerical integration, matrix manipulation, FFTs, etc. Various collections of routines exist: BLAS, LAPACK, NAG, etc. Using one of these is usually simpler, quicker, and more reliable than trying to code a similar algorithm oneself.

BLAS just deals with elementary vector and matrix operations, with a matrix-matrix multiply being about the most complicated. LAPACK contains algorithms for eigen problems, and uses BLAS to do the fundamental operations required. NAG includes much more: PDEs, integration, pseudorandom numbers, FFTs, minimisation, root finding. It also uses BLAS for the fundamental operations.

Most vendors offer versions of BLAS and LAPACK, and maybe FFTs, optimised for their own hardware. Alphas have `cxml`, IBMs have `essl`, Intel has `mkl` etc. A well-optimised BLAS library helps LAPACK and NAG run faster.

BLAS: Basic Linear Algebra System
LAPACK: Linear Algebra PACKage
NAG: Numerical Algorithms Group (commercial, available for most platforms)
`cxml`: Compaq eXtended Maths Library (originally `dxml` (Digital))
`essl`: Engineering and Scientific Subroutine Library
`mkl`: Maths Kernel Library

Compiling and Linking

Compiling is the process of converting a high-level language, such as C or Fortran, to the machine code relevant for a particular processor. The output is an *object file*, traditionally with the suffix `.o`. The translation should be completely accurate, and give the most efficient possible program. Most compilers achieve the former, very few the latter.

The object file contains multiple sections: machine code, initialised data, a list of functions called but not present, and a list of functions provided.

Linking is the process of adding in the relevant library routines to produce an *executable* program. A statically-linked program is entirely self-contained, and, when run, will call the OS directly as necessary.

Being dynamic

Dynamic linking is the alternative to static linking. In this case most of the linking is done at run-time, not compile time. It has several advantages.

The executables are smaller: they no longer need contain copies of bits of libraries.
Libraries tuned for the processor the code is being run on, rather than the processor the code was compiled on, can easily be used.
Only one copy of any library needs to be in memory at once, no matter how many programs are using it.
Updates to the libraries take immediate effect, without the need for recompilation.

There are disadvantages too:

Equivalent libraries to those found by the compiler at compile time need to be found at run time.
There is (usually) a (tiny) overhead every time a dynamically linked function is called.

Windows 3.0 and later do dynamic linking, the libraries having the suffix `.dll` (Dynamic Link Library). Most Unixes support dynamic linking, and use the suffix `.so` (Shared Object).

Calling Conventions

When calling any function the arguments must be made available to the function, the CPU must branch to the start of the function's code, and, at the end, the function must return its result(s), and execution continue at the next instruction in the calling code.

The stack is the area of memory usually used for this. One of the CPU's registers, the stack pointer, always points to the top of the stack, and on this stack are placed, in order, the arguments to the subroutine, followed by the address to return to when finished, and then a branch to the routine occurs. The routine reads its arguments from the stack, places its results on the stack, reads the return address and jumps back, having adjusted the stack pointer approriately. The routine will also use the stack for storing any small local variables it may wish to use.

There are obvious optimisations to this scheme: if only one or two arguments are expected, why not leave them in registers? Similarly for the return address.

Vagueness

The previous slide is deliberately vague. There is no one way of transferring data to and from subroutines. However, the caller and the callee must agree on what to do!

UNIX is mostly written in C, and every UNIX comes with a C library and has an associated compiler (not always free though!). This defines the calling convention for C for that flavour of UNIX.

It does not define it for C++ or Fortran, which need calling features which C does not have. If the vendor supplies C++ and Fortran compilers and libraries, others will usually follow those conventions. If not, chaos.

Hence Linux, which has many Fortran compilers which cannot use each other's libraries as the various compiler writers have done things differently.

Name mangling

```
double modulus(double x){return(fabs(x));}

double modulus(double *x, int n){
   int i;
   double m;
   for(i=0,m=0;i<n;i++) m+=x[i]*x[i];
   return(sqrt(m));
}
```

Two functions with the same name, distinguishable by argument type and number. Legal in C++, but the compiler must generate unique names for these functions so that the linker sees them as distinct. No standard exists for this *name mangling*.

F90 achieves this function *overloading* in a subtly different fashion which avoids this issue.

Even plain F77 must do some name mangling: the UNIX linker is case-sensitive, and F77 is not, so all names must be converted to a consistent case. They usually gain underscores too, to avoid unexpected name clashes with functions in the system libraries.

Optimisation

Optimisation is the process of producing a machine code representation of a program which will run as fast as possible. It is a job shared by the compiler and programmer.

The compiler uses the sort of highly artificial intelligence that programs have. This involves following simple rules without getting bored halfway through.

The human will be bored before he starts to program, and will never have followed a rule in his life. However, it is he who has the Creative Spirit.

This section discussed some of the techniques and terminology used.

Loops

Loops are the only things worth optimising. A code sequence which is executed just once will not take as long to run as it took to write. A loop, which may be executed many, many millions of times, is rather different.

```
do i=1,n
  x(i)=2*pi*i/k1
  y(i)=2*pi*i/k2
enddo
```

Is the simple example we will consider first, and Fortran will be used to demonstrate the sort of transforms the compiler will make during the translation to machine code.

Simple and automatic

CSE

```
do i=1,n
  t1=2*pi*i
  x(i)=t1/k1
  y(i)=t1/k2
enddo
```

Common Subexpression Elimination. Rely on the compiler to do this.

Invariant removal

```
t2=2*pi
do i=1,n
  t1=t2*i
  x(i)=t1/k1
  y(i)=t1/k2
enddo
```

Rely on the compiler to do this.

Division to multiplication

```
t2=2*pi
t3=1/k1
t4=1/k2
do i=1,n
  t1=t2*i
  x(i)=t1*t3
  y(i)=t1*t4
enddo
```

after which

```
t1=2*pi/k1
t2=2*pi/k2
do i=1,n
  x(i)=i*t1
  y(i)=i*t2
enddo
```

The compiler won't do this by default, as it breaks the IEEE standard subtly. However, there will be a compiler flag to make this happen: find it and use it!

Conversion of `x**2` to `x*x` will be automatic.

Remember multiplication is many times faster than division, and many many times faster than logs and exponentiation.

Another example

```
  y=0
  do i=1,n
    y=y+x(i)*x(i)
  enddo
```

As machine code has no real concept of a loop, this will need converting to a form such as

```
  y=0
  i=1
1   y=y+x(i)*x(i)
    i=i+1
    if (i<n) goto 1
```

At first glance the loop had one fp add, one fp multiply, and one fp load. It also had one integer add, one integer comparison and one conditional branch. Unless the processor supports speculative loads, the loading of `x(i+1)` cannot start until the comparison completes.

Unrolling

```
  y=0
  do i=1,n-mod(n,2),2
    y=y+x(i)*x(i)+x(i+1)*x(i+1)
  enddo
  if (mod(n,2)==1) y=y+x(n)*x(n)
```

This now looks like

```
  y=0
  i=1
  n2=n-mod(n,2)
1   y=y+x(i)*x(i)+x(i+1)*x(i+1)
    i=i+2
    if (i<n2) goto 1
  if (mod(n,2)==1) y=y+x(n)*x(n)
```

The same 'loop overhead' of integer control instructions now deals with two iterations, and a small $coda$ has been added to deal with odd loop counts.

Rely on the compiler to do this.

The compiler will happily unroll to greater $depths$ (2 here, often 4 or 8 in practice), and may be able to predict the optimum depth better than a human, because it is processor-specific.

Reduction

This dot-product loop has a nasty data dependency on `y`: no add may start until the preceeding add has completed. However, this can be improved:

```
t1=0 ; t2=0
do i=1,n-mod(n,2),2
  t1=t1+x(i)*x(i)
  t2=t2+x(i+1)*x(i+1)
enddo
y=t1+t2
if (mod(n,2)==1) y=y+x(n)*x(n)
```

There are no data dependencies between `t1` and `t2`. Again, rely on the compiler to do this.

This class of operations are called reduction operations for a 1-D object (a vector) is reduced to a scalar. The same sort of transform works for the sum or product of the elements, and finding the maximum or minimum element.

Prefetching

```
y=0
do i=1,n
  prefetch_to_cache x(i+8)
  y=y+x(i)*x(i)
enddo
```

As neither C nor Fortran has a prefetch instruction in its standard, and not all CPUs support prefetching, one must rely on the compiler for this.

This works better after unrolling too, as only one prefetch per cache line is required. Determining how far ahead one should prefetch is awkward and processor-dependent.

It is possible to add directives to one's code to assist a particular compiler to get prefetching right: something for the desperate only.

Loop Elimination

```
do i=1,3
  a(i)=0
endo
```

will be transformed to

```
a(1)=0
a(2)=0
a(3)=0
```

Note this can only happen if the iteration count is small *and* known at compile time. Replacing '3' by '`n`' will cause the compiler to unroll the loop about 8 times, and will produce dire performance if `n` is always 3.

Loop Fusion

```
do i=1,n
  x(i)=i
enddo
do i=1,n
  y(i)=i
enddo
```

transforms trivially to

```
do i=1,n
  x(i)=i
  y(i)=i
enddo
```

eliminating loop overheads, and increasing scope for CSE. Good compilers can cope with this, a few cannot.

Assuming `x` and `y` are real, the implicit conversion of `i` from integer to real is a common operation which can be eliminated.

Strength reduction

```
double a(2000,2000)

do j=1,n
  do i=1,n
    a(i,j)=x(i)*y(j)
  enddo
enddo
```

The problem here is finding where the element `a(i,j)` is in memory. The answer is $8(i-1)+16000(j-1)$ bytes beyond the first element of `a`: a hideously complicated expression.

Just adding eight to a pointer every time `i` increments in the inner loop is much faster, and called strength reduction. Rely on the compiler again.

Inlining

```
function norm(x)
double precision norm,x(3)

norm=x(1)**2+x(2)**2+x(3)**2
end function
...
a=norm(b)
```

transforms to

```
a=b(1)**2+b(2)**2+b(3)**2
```

eliminating the overhead of the function call.

Often only possible if the function and caller are compiled simultaneously.

Instruction scheduling and loop pipelining

A compiler ought to move instructions around, taking care not to change the resulting effect, in order to make best use of the CPU. It needs to ensure that latencies are 'hidden' by moving instructions with data dependencies on each other apart, and that as many instructions as possible can be done at once. This analysis is most simply applied to a single pass through a piece of code, and is called *code scheduling*.

With a loop, it is unnecessary to produce a set of instructions which do not do any processing of iteration `n+1` until all instructions relating to iteration `n` have finished. It may be better to start iteration `n+1` before iteration `n` has fully completed. Such an optimisation is called *loop pipelining* for obvious reasons..

Sun calls 'loop pipelining' 'modulo scheduling'.

Consider a piece of code containing three integer adds and three fp adds, all independent. Offered in that order to a CPU capable of one integer and one fp instruction per cycle, this would probably take five cycles to issue. If reordered as $3\times$(integer add, fp add), it would take just three cycles.

Debugging

The above optimisations should really never be done manually. A decade ago it might have been necessary. Now it has no beneficial effect, and makes code longer, less readable, and harder for the compiler to optimise!

However, one should be aware of the above optimisations, for they help to explain why line-numbers and variables reported by debuggers may not correspond closely to the original code. Compiling with all optimisation off is occassionally useful when debugging so that the above transformations do not occur.

Loop interchange

The conversion of

```
do i=1,n
  do j=1,n
    a(i,j)=0
  enddo
enddo
```

to

```
do j=1,n
  do i=1,n
    a(i,j)=0
  enddo
enddo
```

is one loop transformation most compilers do get right. There is still no excuse for writing the first version though.

Matrix Multiplication

$$c_{ij} = a_{ik} b_{kj}$$

```
do i=1,n
  do j=1,n
    t=0.
    do k=1,n
      t=t+a(i,k)*b(k,j)
    enddo
    c(i,j)=t
  enddo
enddo
```

The number of FP operations is clearly $2n^3$.

Some timings, for a 463MHz (926MFLOPS peak) XP900:

n=2032	933s	18MFLOPS
n=2048	1348s	13MFLOPS

The problem

The inner loop contains one fp add, one fp multiply, one fp load with unit stride (`b`), and one fp load with stride `n` (`a`). The arrays are around 32MB each.

The 2MB secondary cache on the XP900 is direct mapped, with 32,768 lines of 64 bytes. Thus the lowest 8 bits of an address are an offset within a line, and the next 15 bits are a tag index. The DTLB has 128 entries each covering an 8K page.

For `n`=2032, every load for `a` is a cache and TLB miss for `i=j=1`. For `j`=2, every load for `a` is a cache hit and a TLB miss: over 2000 TLB entries would be needed to cover the first column just read. A cache hit because 2032 cache lines are sufficient, and the cache has 32,768 lines.

For `n`=2048, the same analysis applies for the TLB. For the cache, because the stride is 2^{14} bytes, the bottom 14 bits of the address, and hence the bottom 6 of the tag index, are the same for all `k`. Thus only 512 different cache lines are being used, and one pass of the loop would need 2048 if all are to remain in cache, so all are cache misses.

Blocking

```
do i=1,n,2
  do j=1,n
    t1=0.
    t2=0.
    do k=1,n
      t1=t1+a(i,k)*b(k,j)
      t2=t2+a(i+1,k)*b(k,j)
    enddo
    c(i,j)=t1
    c(i+1,j)=t2
  enddo
enddo
```

Now two elements of a are used every time a cache line of a is fetched. The number of cache misses is halved, and the speed doubles. The obvious extension to use eight elements (all of the 64 byte cache line) achieves 73MFLOPS for n=2048 and 98MFLOPS for n=2032.

Note that t1 to t8 will be stored in registers, not memory.

Loop transformations

The compiler used claims to be able to do some of the above automatically. Specifying `-O5` achieves this (`-fast` is insufficient), and manages 164MFLOPS on the original code.

However, specifying `-O5` on the code after blocking by hand by a factor of eight produces something which runs about three times slower than not using `-O5`.

So with current compilers automatic loop transformations are slightly dangerous: sometimes they make code much faster, sometimes much slower. They work best on very simple structures, but even then they can make debugging awkward.

Laziness

```
call dgemm('n','n',n,n,n,1d0,a,n,b,n,0d0,c,n)
```

The `dgemm` routine is part of the BLAS library and can evaluate

$$c_{ij} = \alpha a_{ik} b_{kj} + \beta c_{ij}$$

Although this is much more general than we require, it achieves 800MFLOPS using the same operation count as before.

The library may have special cases for $\alpha = 1$ and $\beta = 0$. Even if not, there are only n^2 of these operations.

Compaq's own cxml library gave 800MFLOPS. NAG's BLAS gave just 120MFLOPS.

`c=matmul(a,b)` is tempting, and achieves just 13MFLOPS (Compaq Fortran V5.5-1877), and used 32MB of stack, so one can guess how that is implimented. With `-O5` too it achieves 385MFLOPS, so the optimisation flags affect intrinsics. Compaq's compiler is quite bad in this regard.

What was wrong with our 100MFLOPS code? The TLB miss on every cache line load of `a` prevents any form of prefetching working for this array.

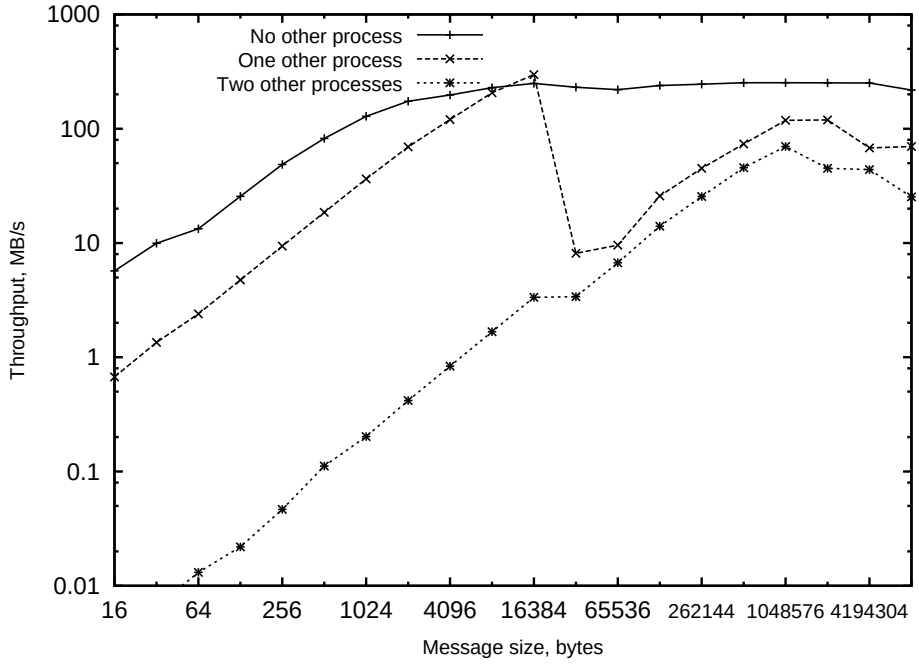
Practical Parallelism

Any parallel algorithm will involve passing messages from one process to another. If both processes are executing simultaneously on separate processors, this can be very rapid. A process waiting for a message should *spin wait*: constantly checking to see if the message has arrived, and not yield its scheduling slot, for the expected latency for a message is a few μs, whereas a scheduling slot will be a few thousand μs.

If two processes are run on a single CPU, a process waiting for a message should immediately yield its scheduling slot, so that the process sending the message gets some CPU time and can send it.

In either case, large messages will have to be broken into smaller fragments as they are sent, the processes effectively sharing a buffer, the first filling it, then waiting until it has been emptied before it is able to refill it.

The slowdown



Transfer rate for various sized packets using two MPI processes on a dual processor machine.

With no other processes, the latency is about 3μs and the bandwidth 250MB/s. With one other process, the latency is 24μs and the bandwidth 120MB/s. The point at which multiple packets are needed for a single transfer (32KB) is clearly seen. With two other processes, the latency is 5000μs and the bandwidth 40MB/s. The details depend greatly on how the scheduler distributes processes amongst processors.

No-one who cares about latencies runs MPI with more than one process per processor!

Note that when running four serial processes on a dual processor machine, each will run twice as slowly as they would if just two had been run. With parallel code, the slowdown could be a factor of one thousand.

The Compilers

```
f90 -fast -o myprog myprog.f90 func.o -lnag
```

That is options, source file for main program, other source files, other objects, libraries. Order does matter (to different extents with different compilers), and should not be done randomly.

Yet worse, random options whose function one cannot explain and which were dropped from the compiler's documentation two major releases ago should not occur at all!

The compile line is read from left to right. Trying

```
f90 -o myprog myprog.f90 func.o -lnag -fast
```

may well apply optimisation to nothing (i.e. the source files following `-fast`). Similarly

```
f90 -o myprog myprog.f90 func.o -lnag -lcxml
```

will probably use routines from NAG rather than cxml if both contain the same routine. However,

```
f90 -o myprog -lcxml myprog.f90 func.o -lnag
```

may also favour NAG over cxml with some compilers.

Common compiler options

`-lfoo` and `-L`

`-lfoo` will look first for a shared library called `libfoo.so`, then a static library called `libfoo.a`, using a particular search path. One can add to the search path (`-L${HOME}/lib` or `-L.`) or specify a library explicitly like an object file, e.g. `/temp/libfoo.a`.

`-O`, `-O`**n** and `-fast`

Specify optimisation level, `-O0` being no optimisation. What happens at each level is compiler-dependent, and which level is achieved by not specifying `-O` at all, or just `-O` with no explicit level, is also compiler dependent. `-fast` requests fairly aggressive optimisation, including some unsafe but probably safe options, and probably tunes for specific processor used for the compile.

`-c` and `-S`

Compile to object file (`-c`) or assembler listing (`-S`): do not link.

`-g`

Include information about line numbers and variable names in `.o` file. Allows a debugger to be more friendly, and may turn off optimisation.

More compiler options

`-C`

Attempt to check array bounds on every array reference. Makes code much slower, but can catch some bugs. Fortran only.

`-r8`

The `-r8` option is entertaining: it promotes all single precision variables, constants and functions to double precision. Its use is unnecessary: code should not contain single precision arithmetic unless it was written for a certain Cray compiler which has been dead for years. So your code should give identical results whether compiled with this flag or not.

Does it? If not, you have a lurking reference to single precision arithmetic.

The rest

Options will exist for tuning for specific processors, warning about unused variables, reducing (slightly) the accuracy of maths to increase speed, aligning variables, etc. There is no standard for these.

IBM's equivalent of `-r8` is `-qautodbl=dbl4`.

Fortran 90

Fortran 90 is the langauge for numerical computation. However, it is not perfect. In the next few slides are described some of its many imperfections.

Lest those using C, C++ and Mathematica feel they can laugh at this point, nearly everything that follows applies equally to C++ and Mathematica. The only (almost completely) safe language is F77, but that has other problems.

Most of F90's problems stem from its friendly high-level way of handling arrays and similar objects.

Slow arrays

```
a=b+c
```

Humans do not give such a simple statement a second glace, quite forgetting that depending what those variables are, that could be an element-wise addition of arrays of several million elements. If so

```
do i=1,n
  a(i)=b(i)+c(i)
enddo
```

would confuse humans less, even though the first form is neater. Will both be treated equally by the compiler? They should be, but many early F90 compilers produce faster code for the second form.

Big surprises

```
a=b+c+d
```

really ought to be treated equivalently to

```
do i=1,n
  a(i)=b(i)+c(i)+d(i)
enddo
```

if all are vectors. Many early compilers would instead treat this as

```
temp_allocate(t(n))
do i=1,n
  t(i)=b(i)+c(i)
enddo
do i=1,n
  a(i)=t(i)+d(i)
enddo
```

This uses much more memory than the F77 form, and is much slower.

Sure surprises

```
a=matmul(b,matmul(c,d))
```

will be treated as

```
temp_allocate(t(n,n))
t=matmul(c,d)
a=matmul(b,t)
```

which uses more memory than one may first expect. And is the `matmul` the compiler uses as good as the `matmul` in the BLAS library? Not if it is Compaq's compiler.

I don't think Compaq is alone in being guilty of this stupidity. See IBM's `-qessl=yes` option. . .

Note that even `a=matmul(a,b)` needs a temporary array. The special case which does not is `a=matmul(b,c)`.

More sure surprises

```
allocate(a(n,n))
...
call wibble(a(1:m,1:m))
```

must be translated to

```
temp_allocate(t(m,m))
do i=1,m
  do j=1,m
    t(j,i)=a(j,i)
  enddo
enddo
call wibble(t)
do i=1,m
  do j=1,m
    a(j,i)=t(j,i)
  enddo
enddo
```

Array slicing and reshaping may be automatic, but it takes a lot of time and memory.

The temporary array is unnecessary if `m=n`, or if the call is `a(:,1:m)`, but early compilers will use it anyway, being the simple approach which always works.

Type trouble

```
type electron
   integer :: spin
   real (kind(1d0)), dimension(3) :: x
end type electron

type(electron), allocatable :: e(:)
allocate (e(10000))
```

Good if one always wants the spin and position of the electron together. However, counting the net spin of this array

```
s=0
do i=1,n
  s=s+e(i)%spin
enddo
```

is now slow, as an electron will contain 4 bytes of spin, 4 bytes of padding, and three 8 byte doubles, so using a separate spin array so that memory access was unit stride again could be eight times faster.

What is `temp_allocate`?

Ideally, an `allocate` and `deallocate` if the object is 'large', and placed on the stack otherwise, as stack allocation is faster, but stacks are small and never shrink. Ideally reused as well.

```
a=matmul(a,b)
c=matmul(c,d)
```

should look like

```
temp_allocate(t(n,n))
t=matmul(a,b)
a=t
temp_deallocate(t)
temp_allocate(t(m,m))
t=matmul(c,d)
c=t
temp_deallocate(t)
```

with further optimisation if `m=n`. Some early F90 compilers would allocate all temporaries at the beginning of a subroutine, use each once only, and deallocate them at the end.

Precision

```
complex (kind(1d0)) :: c
real (kind(1d0)) :: a,b,pi
...
pi=3.1415926536
c=cmplx(a,b)
```

This should read

```
pi=3.1415926536d0
c=cmplx(a,b,kind(1d0))
```

for both a constant and the `cmplx` function default to single precision.

Some compilers automatically correct the above errors.

Note also that π expressed to full double precision is not the above value: either use

```
real (kind(1d0)) :: pi
pi=4*atan(1d0)
```

or

```
real (kind(1d0)), parameter :: pi=3.14159265358979 3d0
```

(The latter has the advantage that one cannot accidently change the value of π in the program, the former that it is less likely to be mistyped.)

`c=(0.2d0,0.4d0)` is sensible, as `(,)` produces a complex constant of the same precision as the real constants in the brackets.

Precision again

```
real*8 x
real(8) :: y
```

The first is a ubiquitous F77 extension. The second is a foolish misunderstanding: some compilers may use a `kind` value of 8 to represent an 8 byte double precision number, but nothing in the standard says they should use eight rather than three (as a few do), or anything else.

```
double precision x
real (kind(1d0)) :: y
```

is the correct F77 and F90 respectively.

```
integer, parameter :: dp=kind(1d0)
real (dp) :: y
```

is a common (and correct) F90 construction.

Other languages

So that I am not accused of bias,

`http://www.tcm.phy.cam.ac.uk/~mjr/C/`

discusses why C is even worse. . .