An Introduction to Computational Physics

MJ Rutter
mjr19@cam.ac.uk

Michaelmas 2004

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. TCM has copies of both.

 ©2004 MJ Rutter

Disclaimers

The information herein is believed to be accurate, but it might not be.

Very many trademarks are used, they are all the property of their respective owners.

Standard rant: the PostScript file which created the whole of this booklet will fit onto a standard 1.44MB floppy disk.

Contents

Introduction **4**

The CPU **13**

instructions 19

performance measures 30

integers 36

floating point 50

algorithms 68

Pipelines **76**

Memory **90**

technologies 91

caches 111

Common CPU Families **133**

Permanent Storage **152**

Operating Systems **170**

multitasking 183

memory management 186

Parallel Computers **214**

Programming **249**

libraries 250

applied programming 256

optimisation 294

the pitfalls of F90 319

languages 328

Index **339**

Introduction

The Problem

Love 'em or loath 'em, computers are here to stay. Even 'pencil and paper' theorists use them for writing articles, finding other people's articles, secret sessions with Mathematica (behind closed doors), and maybe, sometimes, numerical work.

Others are more openly reliant on the things.

In order to get the best out of what can be a very useful tool, it is helpful to understand a little about how they work. If you don't, your competitors still will, and they will be able to do a better job of pushing the boundaries of what is possible as a result.

Floreat Domus

All good things come from Cambridge, and computing is no exception. Babbage (Trinity, Lucasian Professor of Mathematics) is generally regarded as the father of modern computers for the 'Difference Engine' he invented in 1821, a version of which was constructed in 1854.

The move to electronic computers came in around 1944 with Colossus built at Bletchley Park by a team containing many Cambridge men (Turing, Newman, Tutte and others).

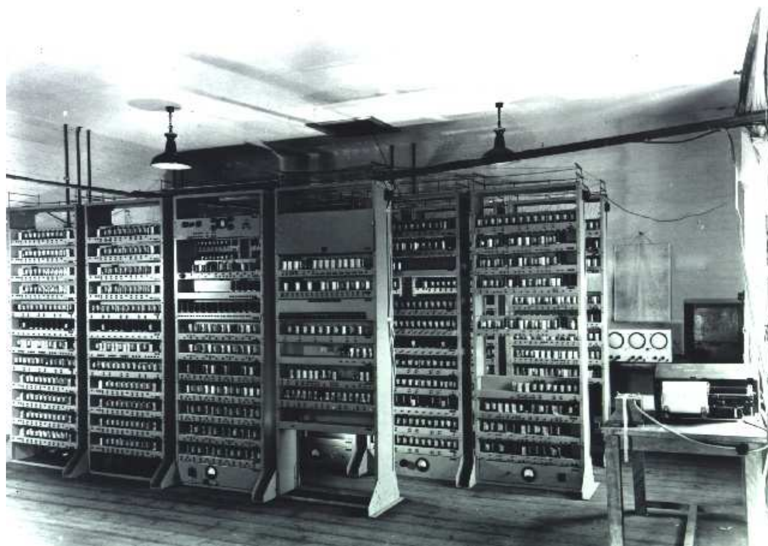
The Manchester Small Scale Experimental Machine (1948) was the first genuinely programmable, fully-electronic digital computer. A year later Cambridge followed with EDSAC.

Other claimants include ENIAC (US, 1945), an electronic and nearly-programmable computer, Zuse Z3 (Germany, 1944), an electromechanical programmable computer

The Mathematical Laboratory

EDSAC ran its first program on 6th May 1949. It was run and built by the forerunner of the University's Computing Service, the Mathematical Laboratory.

In 1950 it started doing real scientific work. Early fields of study included X-ray crystallography and radio astronomy. In 1958 EDSAC was replaced by EDSAC-2, another Cambridge-built valve-based machine, which lasted until 1965. EDSAC-2's successor, Titan, was programmable in Fortran from 1966.



EDSAC-1

World 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 TLB (IBM 370)
	ASC: computer with 'ECC' memory (TI)
1974	First DRAM with one transistor per bit
1975	UNIX appears outside AT&T
	Ethernet appears (Xerox)
1976	Apple I launched. $666.66
	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)
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
	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	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
	Tera starts developing MTA processor
1992	PCI
	OpenGL
	JPEG
	OS/2 2.0 (32 bit a year before NT) (IBM)
	Alpha 21064: 64 bit superscalar RISC (DEC)
1993	PDF version 1.0 (Adobe)
	MS Windows NT 3.1 (the first version. . .)
1994	LaTeX2e
	MPI

A Summary of History

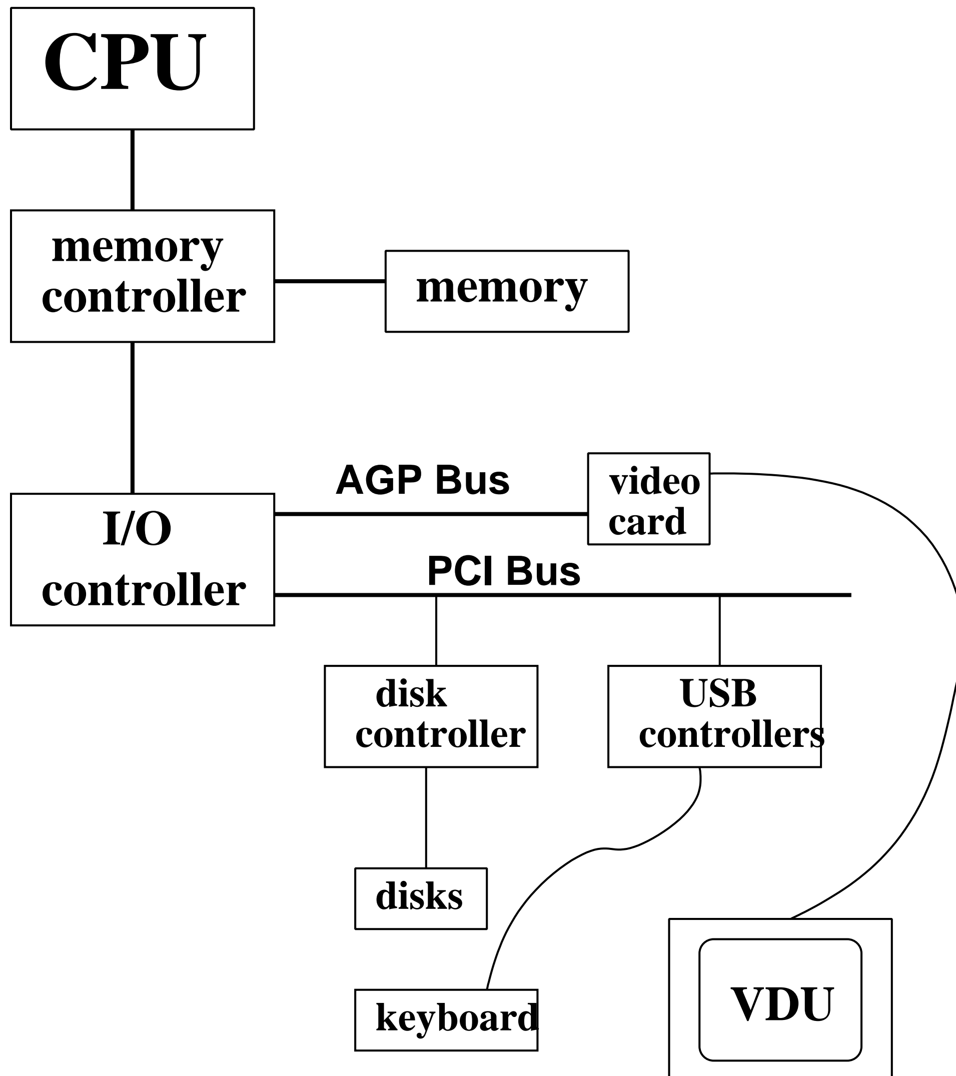
The above timeline stops a decade before this talk will first be given. Computing is not a fast-moving subject, and little of consequence has happened in the past decade.

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

The CPU

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.

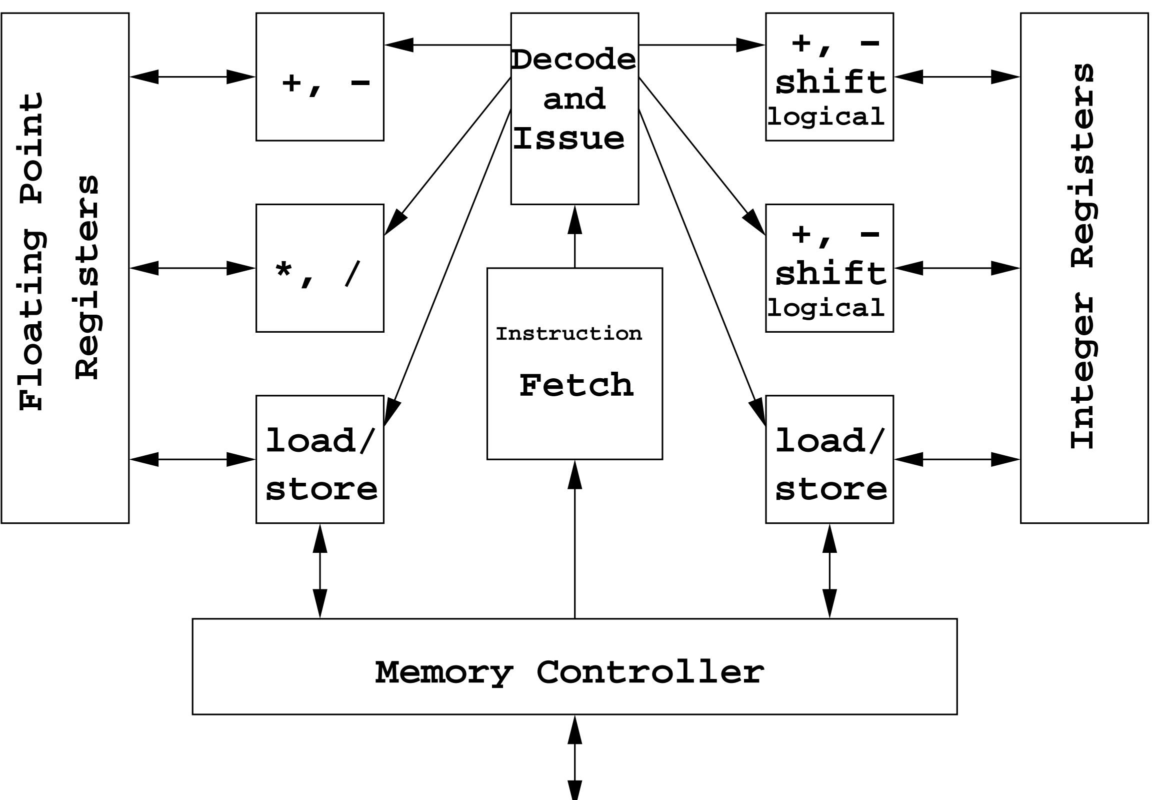
A typical modern CPU understands two main classes of data: integer and floating point. Within those classes it may understand some additional subclasses, such as different precisions.

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 (number of bus transfers each second). 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).

Languages

High-level languages (C, Fortran, Java, etc.) exist to save humans the bother of worrying about the precise details of the CPU in the computer they are using. They are converted to *machine code*, the binary representation of a specific CPU's instructions, by a program called a *compiler*.

Thus one can write a code once, and recompile for many different CPUs.

The language is a compromise between being easy for humans to understand, and easy for compilers to convert into efficient machine-code. Fortran excels at FORmula TRANslation (i.e. numeric work), whereas C is a more general-purpose language.

Diversity

Although most modern CPUs are conceptually similar, the details vary. Different CPUs will almost certainly be different physical shapes and expect different electrical signals, so need different motherboards. They will probably also have different instructions, numbers of registers, and ways of encoding instructions into binary.

There is no way that code written for one CPU will run on another. An instruction such as `fadd f16,f17,f18` would be nonsense for a CPU with just 16 floating-point registers, and, anyway, there is no reason for all CPU manufacturers to decide to encode `fadd f16,f17,f18` in the same manner.

If CPUs can be swapped between each others' sockets, they are said to be *socket-* or it plug-compatiable. If they can run each others' machine-code, they are said to be *binary compatible*.

Families

A family of CPUs is a series of CPUs which retains binary compatiblity with previous members, although instructions (and features) may be added so that the compatibility may not work in both directions.

IA32 Intel 386, 486, Pentium, Pentium II, Pentium III, Pentium 4, Pentium M; AMD Athlon, Athlon-64.

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

Alpha 21064 (EV4), 21164 (EV5), 21264 (EV6), 21364 (EV7)

Power Power, Power2, Power3, Power4

PowerPC 601, 603, 604, 620, 750 (=G3), 7400 (=G4), 970 (=G5)

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

Alpha vs IA32, Part I

The Alpha processor series was launched in 1992 as a legacy-free(?) RISC design, whereas the IA32 line appeared in 1985, and was proud to be merely a binary-compatible extension of the 8086 (1978).

	IA32	Alpha
Integer Regs	8 x 32 bit	32 x 64 bit
Integer Formats	8, 16 & 32 bit	32 & 64 bit
F.P. Regs	8 x 80 bit	32 x 64 bit
F.P. Formats	32, 64	32 & 64 bit IEEE
	& 80 bit IEEE	32 & 64 bit VAX
Instruction length	1 - c.14 bytes	4 bytes

The IA32 range has many complex instuctions absent on the Alpha range: trig functions, logarithms, scan or copy string in memory. It can also transfer data directly between arithmetic units and memory, without going via a register. All of these involve multiple functional units, often multiple times.

Registers

The existance of separate floating-point and integer registers has been discussed. There are also some dedicated registers.

There will always be an *instruction pointer*, a register dedicated to holding the address of the instruction currently being executed. There will usually be a *stack pointer*, which manages the stack (see later), and there may be a *frame pointer* (again, see later).

There may also be a zero register: a register which yields zero when read, and discards anything written to it.

Usually the integer registers are used for both addresses and integer data.

The Motorola 68000 series has separate registers for integer data and addresses.

The instruction pointer is sometimes also called the *program counter* (PC).

Alpha Registers

There are 32 integer registers, all 64 bit, conventionally called `r0` to `r31`, additional to the instruction pointer. A few of these are special purpose:

`r15`:	Frame pointer
`r26`:	Return address
`r30`:	Stack pointer
`r31`:	Zero register

Likewise there are 32 floating-point registers, all 64 bit, called `f0` to `f31`, and `f31` is the zero register.

x86 Registers

The IA32 processors have rather fewer registers, with an odd arrangement.

There are eight floating-point registers, all 80 bit. They cannot be addressed directly, but they form a stack, with the simplest floating point instructions, such as `fadd`, existing in a form which has no arguments, but which removes the top two items from the stack, and replaces them by their sum.

Most instructions can take forms such as
`fadd %st(3),%st`
(add to the third element on the stack the top), but one of the operands must be the stack top.

These are the only double-precision floating point registers on all IA32 CPUs up to and including the PentiumIII, and they were introduced in the 8087 in 1978.

IA32 Integer Registers: the beginning

The original 8086, and the 80286, had 16 bit integer registers. These were named `ax`, `bx`, `cx`, `dx`, `di`, `si`, `bp`, `sp` and `ip`. The first four could also be addressed as 8 bit registers, with `ah` being the 'high' (top) byte of `ax`, and `al` the 'low' (bottom) byte.

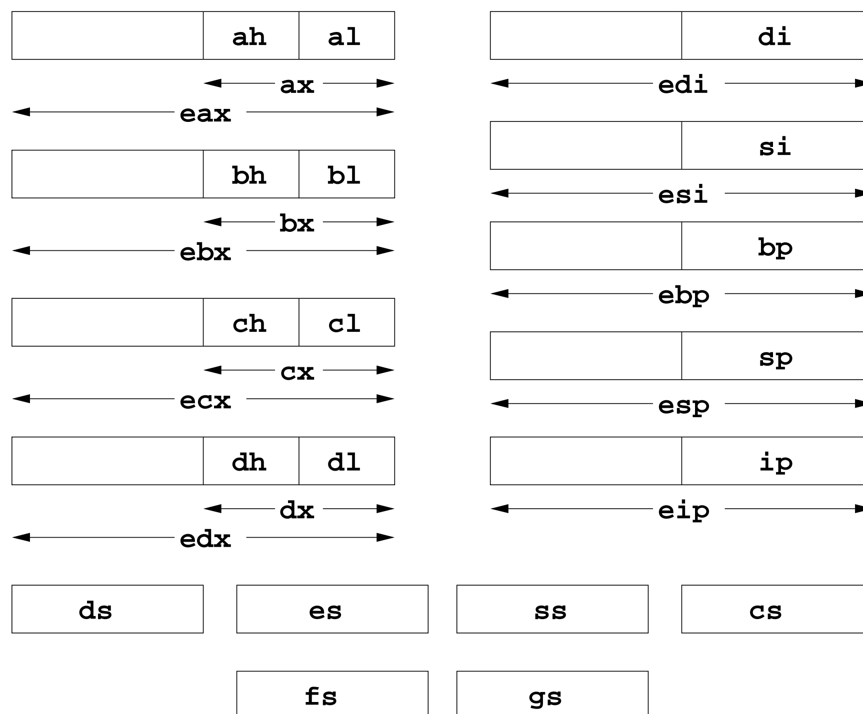
So that more than 64K could be addressed in 16 bits, each address was considered to be the combination of a *segement register* and one of the above integer registers. There were four 16 bit segment registers, `cs`, `ds`, `es`, and `ss`, and an address was written as a colon separated pair, e.g. `cs:ip`, and calculated as $16\times$ the segment register $+$ the offset, yielding effectively 20 bits (1MB).

There is no zero register, `cs:ip` is the instruction pointer, `ss:sp` the stack pointer and `ss:bp` the frame pointer.

The registers are not equivalent: integer multiplies *always* multiply by `ax` and place the result in `dx` and `ax`, string instructions *always* use `di` and `si`, loop instructions *always* use `cx`.

IA32: the 32 bit extension

The above 16 bit registers simply became the bottom two bytes of a set of extended registers, with the full names now being prefixed with an 'e'. The segment registers remained 16 bit, although two more, `fs` and `gs`, were added. Messy.



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.
- 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.

Streams

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

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, 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.

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, and a fifth is due in 2004.

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`)

Various Results, SPEC

Processor	MHz	SpecInt	SpecFP
Power 5	1900	–	2702
Itanium 2	1500	1243	2148
SPARC64 V	1890	1345	1803
Power 4	1700	1158	1776
Opteron 248	2200	1452	1691
Alpha 21364	1300	994	1684
Pentium 4	3600	1575	1630
Alpha 21264	1250	928	1365
UltraSPARC III	1200	905	1106
Pentium M	2000	1541	1088
Pentium III	1400	664	456

For each CPU, the best result (i.e. fastest motherboard / compiler / clock speed) as of 1/8/04 is given.

Note that the ordering by SpecInt would be rather different.

Regrettably Apple and SGI/MIPS do not publish SPEC scores.

SPARC64 V is Fujitsu's licenced clone of Sun's SPARC CPU.

Various Results, Streams and Linpack

				Linpack	
Machine	Year	CPU/MHz	Streams	code	`dgesv`
Pentium4	2002	P4/2400	1850	241	2750
Pentium4	2002	P4/1800	1140	140	1980
PentiumIII	1999	PIII/650	343	47	454
XP1000	2001	21264/667	990	125	965
XP1000	1999	21264/500	980	146	755
PW500au	1998	21164/500	233	42	590
AS500/500	1996	21164/500	170	32	505

The 'code' '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, whereas the two 21264s have identical memory subsystems, but different secondary cache speeds and core speeds. The 667MHz core appears to have a slower secondary cache.

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).

Eight bits is one bytes, therefore four bits must be one nybble.

Being Positive

This is tediously simple:

Bits	Base-10	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
...	...	
1001	9	9
1010	10	a
...	...	
1111	15	f

The obvious, universal, representation for positive integers: binary.

As 4 bits implies 2^4 combinations only 16 different numbers can be represented with 4 bits.

Hex is convenient as an eight bit byte can always be represented in just two digits, and converting binary to hex is trivial: each group of four bits can be converted independently.

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.

Two's complement

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

Chaos

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

Of these possibilities, two's complement is almost universally used, although offset and s-m are seen in floating point formats.

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

Remembering

One 8 bit byte is the smallest unit that can be extracted from the memory system of any current computer. However, one usually wishes to store larger values. Consider storing the value `0x1234` (4660 in base-10) as a two-byte value at address `0x1000`. Should one store `0x12` in the byte at address `0x1000`, and `0x34` at address `0x1001`, or vice-versa?

The former, which seems most logical, is called *big-endian* storage, and the latter *little endian*. The distinction applies to files as well as memory, so two computers must agree on endianness before they can exchange data succesfully.

Why little endian?

Consider storing 0x1234 as a four-byte value at address 0x1000.

Address	0x1000	0x1001	0x1002	0x1003
Little endian	0x34	0x12	0x00	0x00
Big endian	0x00	0x00	0x12	0x34

Now consider an idiot programmer reading the value from address 0x1000, and incorrectly assuming that it is a two-byte value, not a four-byte value. In the big-endian world, the value zero will be read. In the little-endian world, the value 0x1234 will be read, and the bug will escape detection.

Unfortunately, in the world of IA32, little-endian won. Tru64 on Alphas is also little-endian (although the CPU can operate in either mode). SPARC Solaris is big-endian.

Endian conversion is messy: single-byte values (characters) need no conversion, two-byte values need swapping, four byte values reversing in groups of four, and eight-byte values in groups of eight. This issue applies to (binary) files as well.

Adding up

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: $(-3) + 1 = 6$
- 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:

- 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.

Note that this does not work with multiplication or division.

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

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

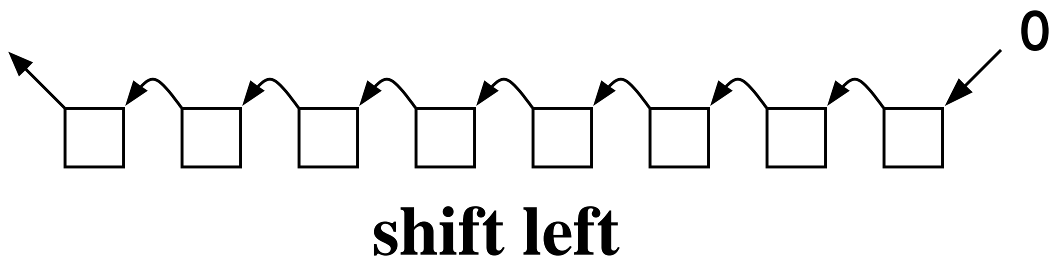
Some processors have expanding multiply instructions, e.g. for multiplying two 16 bit numbers, and keeping all 32 bits of the result. Other instructions simply truncate the result to the size of the operands.

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).

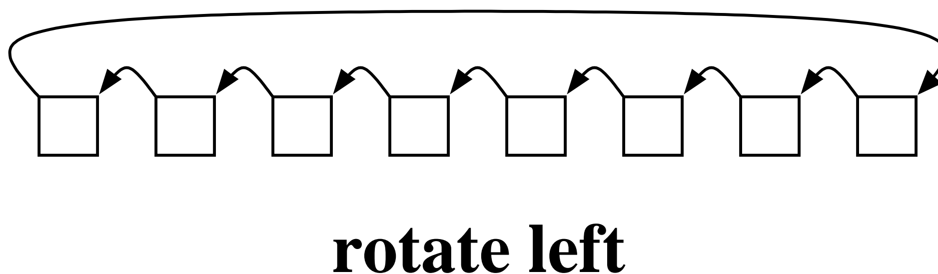
Integer division is yet more complex, and much rarer in code. Hence even fewer processors support it. Alpha does not.

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 $(a \text{ xor } b) \text{ xor } b \equiv a$.

C programmers will wish to distinguish between '3&&7', which is one (a logical operation), and '3&7', which is 3 (a bit-wise operation).

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 $\surd$.

'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

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.

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

e.g.

$$0.1000 \times 10^{1}$$

or

$$0.6626 \times 10^{-33}$$

(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

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

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

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

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

Algebra (2)

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

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

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

$$\begin{aligned} 0.6000 \times 10^1/0.7000 \times 10^1 &= 0.8571 \\ 0.6 \times 10^1 \times (1/0.7 \times 10^1) &= 0.6 \times 10^1 \times 0.1429 \\ &= 0.8574 \end{aligned}$$

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:

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

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!

Underflows, and Denormals

$0.1 \times 10^{-99}/0.1 \times 10^{10} = 0$ 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: $-0.1 \times 10^{-99}/0.1 \times 10^{10} = -0$.

$0.1 \times 10^{-99}/2$ 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.

More 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.

IEEE 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.

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.

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:

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

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.

Rounding & 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);
}
```

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 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.

Algorithms

Whereas in mathematics one reveres closed form solutions, and despises iterative methods, in computing the distinction is less clear.

Because floating point numbers are stored to finite precision, it is quite possible to converge an iterative method to the same accuracy as a closed-form method. It may be faster to do so, it can even be more accurate, depending on how complicated the expression for the closed form solution is, and how rapidly errors build up.

A few specific algorithms are considered below.

Scaling

One often worries about how the number of operations, and hence time, scales with the problem size. Answers for optimal methods for common problems are:

1	hashed search
$\ln n$	tree search
n	unstructured search
$n \ln n$	FFT (no large prime factors), sorting
n^2	general Fourier Transform
n^3	matrix inversion, matrix-matrix multiplication, determinants, orthogonalisation

But prefactors *do* matter, as seen from the Linpack benchmark results, where both the unblocked and library methods are n^3.

There are worse algorithms than the above, such as the 'Bubble' sort (n^2), and some $n!$ methods for finding matrix determinants.

Error propagation

The sensitivity of the logistic map to error accumulation has already been noted. Indeed, one can demonstrate that, for that equation, errors grow exponentially. Other iterative schemes exist in which errors decay exponentially, and there is a middle ground too.

In general, unless one's algorithm is insensitive to error accumulation (e.g. Newton-Raphson), it may be well to think about stability.

Iterative Improvement

If $\mathbf{B}$ is an approximation to the inverse of $\mathbf{A}$,

$$\mathbf{B} \rightarrow \mathbf{B} - \mathbf{B}(\mathbf{I} - \mathbf{A}\mathbf{B})$$

will be a better approximation.

This step involves matrix-matrix multiplications, so will be order n^3 operations. However, such techiques allow one to obtain accurate inverses, and estimates on one's error.

Note that solving a system of linear equations,

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

is not best done by explicitly inverting $\mathbf{A}$ and calculating $\mathbf{A}^{-1}\mathbf{b}$, and that an n^2 algorithm for iterative improvement exists, which is cheap compared to the n^3 cost of the initial approximation.

Random Numbers

Computers, being deterministic, are bad at generating random numbers.

Sometimes very good random numbers are required, e.g. for generating cryptographic keys. Sometimes much poorer randomness suffices.

When debugging, it is useful if the same 'random' numbers are used in each run. When generating results, this is usually unhelpful.

A common, simple, generator is the linear congruential algorithm:

$$x \to (ax + b) \bmod c$$

Where common choices are $a = 1103515245$, $b = 12345$, $c = 2^{32}$.

For $c = 2^{32}$, one need not take the modulus explicitly, but can rely on 32-bit integer arithmetic wrapping round.

Problems

The sequence alternates odd, even, odd, even. . .

The sequence mod 2^n has period n.

The whole sequence repeats exactly after 2^{32}.

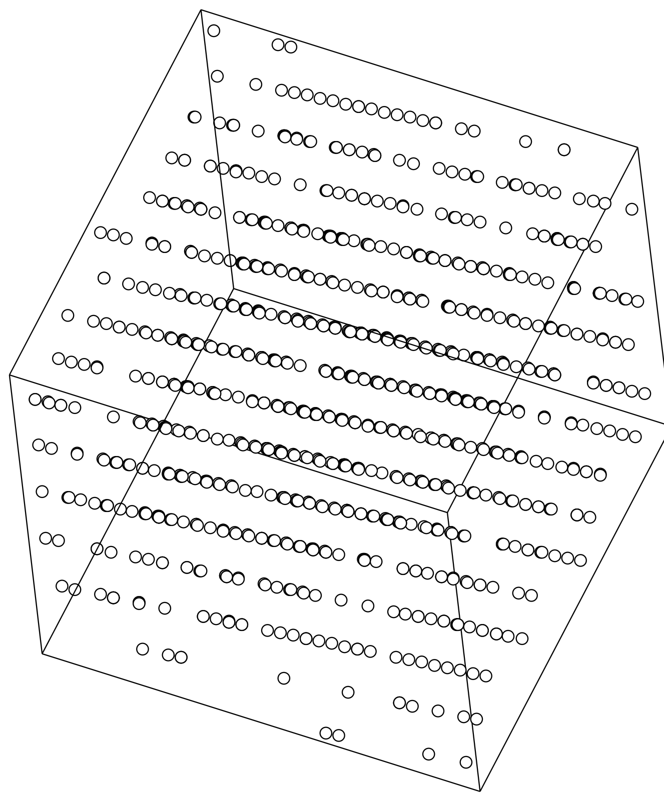
No number appears twice within this sequence.

(If one picks numbers randomly from a field of N, one expects to pick a repeat after $\approx 1.2\sqrt{N}$ goes.)

Cf. the infamous problem of calculating when two people in a set share the same birthday.

More problems

For sampling multidimensional spaces, the sequence is very poor, with points lying on a few discrete hyperplanes within the space.



1,000 triplets from a sequence with $c = 4096$.

Where 'a few' is no more than $c^{1/D}$ where D is the dimension of the space. Here no more than 16 2D planes are expected.

Ideas

Should the program use the same random sequence each time it is run?

Yes, if debugging, and probably not otherwise.

If parallel, should all nodes use the same sequence? Probably not.

If parallel, can one ensure that the precisely same result is obtained no matter how many nodes are used?

Often Physics requires a large number of mildly random numbers, whereas cryptography requires a small number of extremely unguessable numbers. Different generators suit different problems.

Pipelines

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.

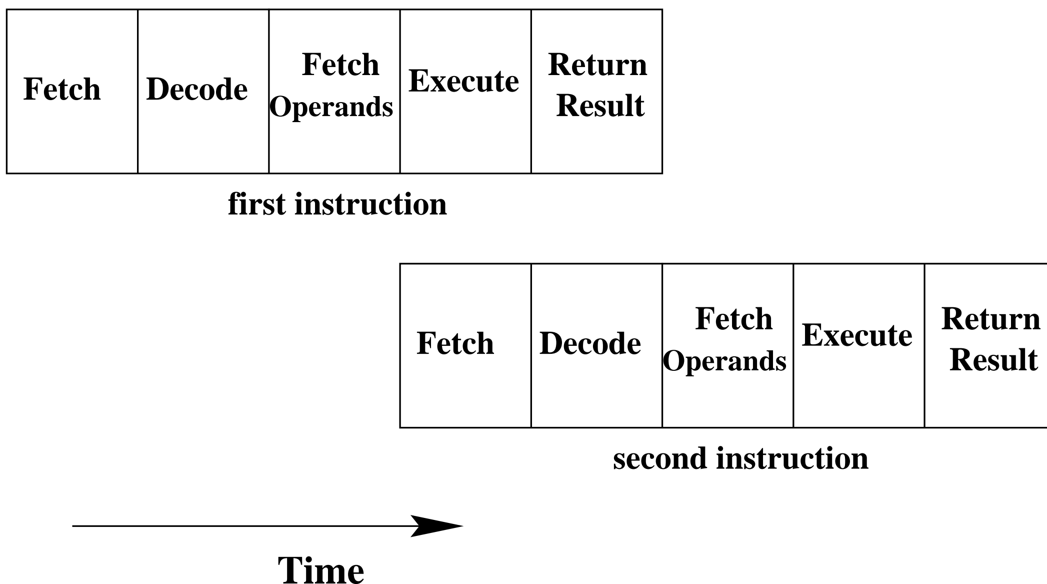
Always a *pipeline* of operations which must be performed sequentially.

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-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.

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:

$9.67 \times 10^5 + 4 \times 10^4$

First the exponents are adjusted so that they are equal:

$9.67 \times 10^5 + 0.4 \times 10^5$

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:
```

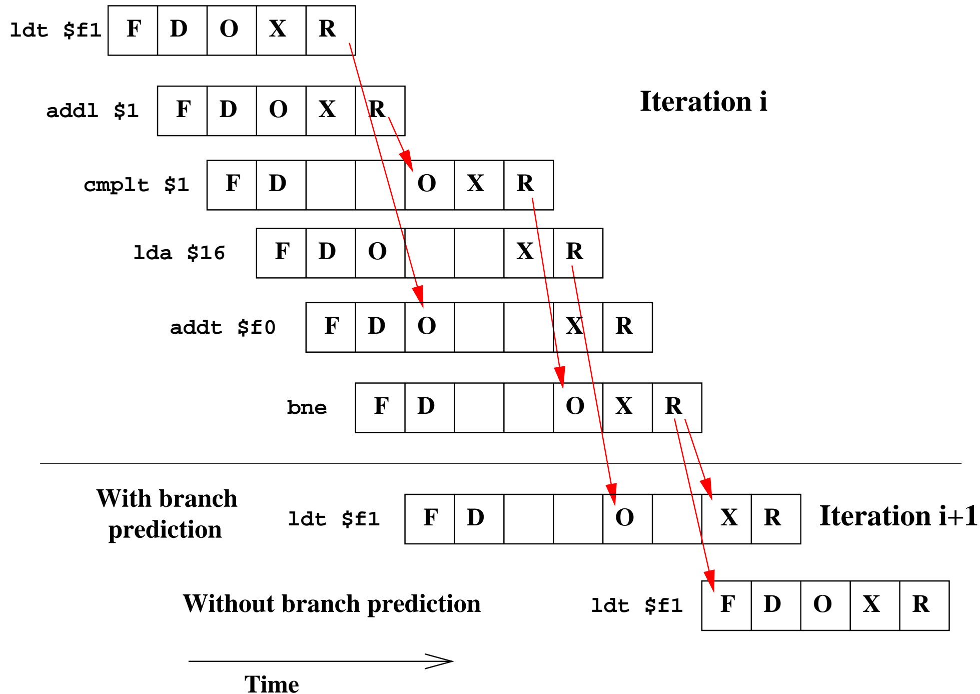
There will be a conditional jump or *branch* at the end of the loop. If the processor simply fetches and decodes the instructions following the branch, then when the branch is taken, the pipeline is suddenly empty.

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 ≠n, do another iteration
L$5:	

Predictions



With the simplistic pipeline model of page 78, 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;
```

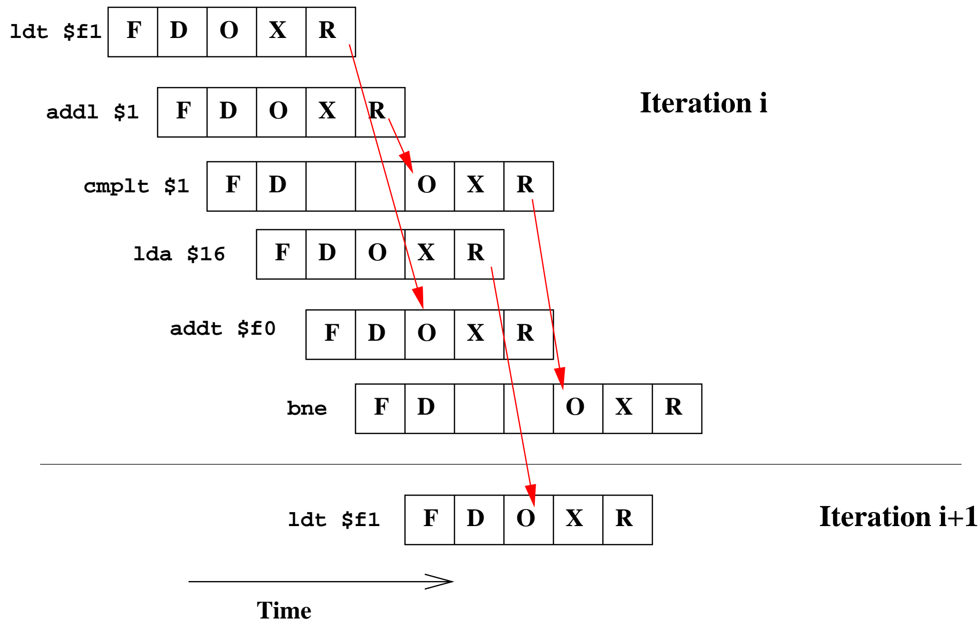
must be converted to

```
  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.

Memory

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

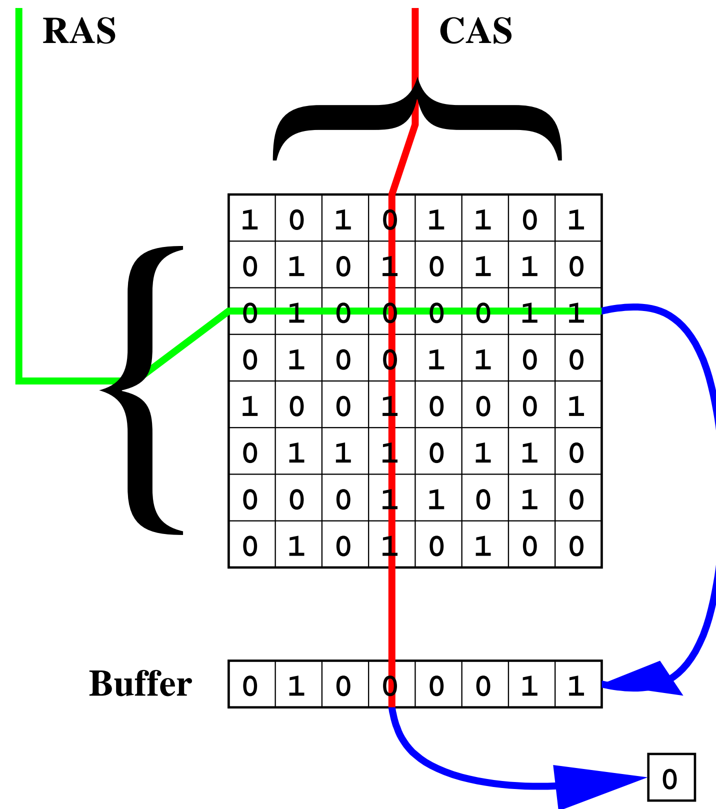
A typical DRAM cell consists of a single capacitor and field effect transistor. It stores a single bit, and has barely changed since 1974.

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 millions of 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 delay, $\mathrm{t_{RCD}}$, column placed on address lines, and Column Access Strobe pin signalled.
- After another delay, $\mathrm{t_{CAS}}$, 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, which takes $\mathrm{t_{RP}}$

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.

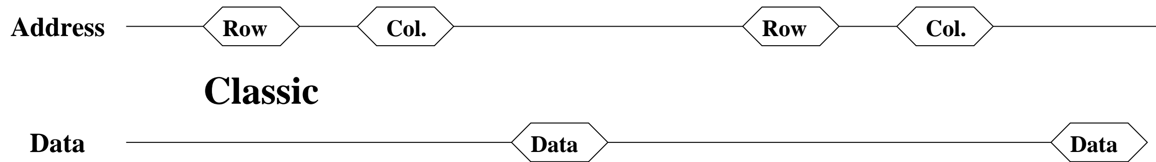
More Speed!

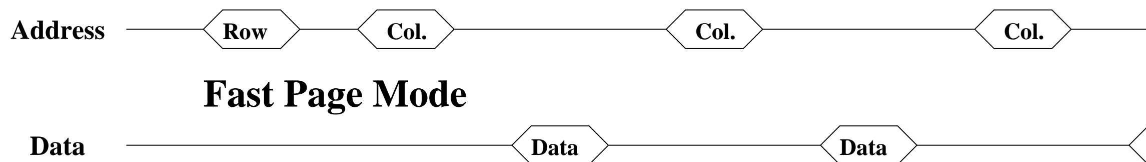
The above procedure is tediously slow. However, for reading consecutive addresses, one important improvement can be made.

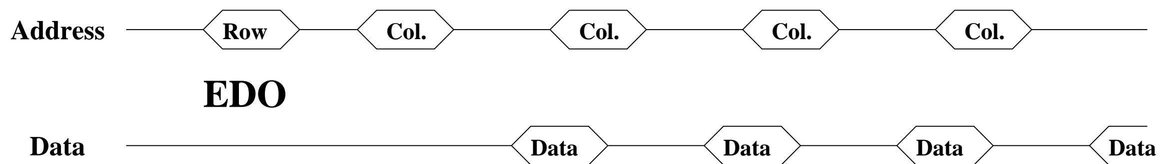
Having copied a whole row into the buffer (which is usually SRAM), if another bit from the same row is required, simply changing the column address whilst signalling the CAS pin is sufficient. There is no need to wait for the chip to write the row back, and then to rerequest the same row. Thus Fast Page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

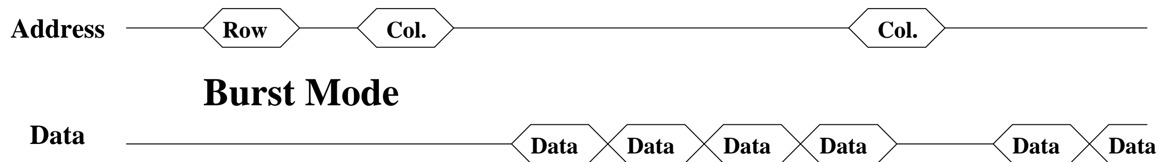
Extended Data Out (EDO) is similar, but allows the column address for the next read to be given whilst the data from the previous read are being read.

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. EDO overlaps the column addresses with the data, and finally, in Burst Mode, a *burst* of successive data items is produced without any address-bus activity.

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 and within the module are synchronised to the clock pulses.

SDRAM Timings

Three numbers usually describe the detailed timing of SDRAM. All are expressed in clock-cycles.

CL	CAS delay
t_{RCD}	RAS to CAS delay
t_{RP}	Row precharge

The data then appear one bus-width-full per clock-cycle.

There is significant granularity for SDRAM. For 133MHz SDRAM, the clock is 7.5ns, so to be CL2 it must have a CAS delay of under 15ns, whereas CL3 is under 22.5ns.

Actually, the timings need to be slightly faster than this to give buses time to settle

DDR-SDRAM

DDR-SDRAM transfers data on both the rising and falling clock edges: twice per cycle. This also enables it to have half-integer CAS delays. For DDR-SDRAM, the timings are expressed in terms of the undoubled clock rate, although the marketted clock rate is the doubled one.

In other words, '266MHz DDR-SDRAM' has a basic clock rate of 133MHz, and if run at CL2, that is the same CAS delay as a 133MHz SDRAM part running at CL2.

The fastest DDR-SDRAM currently (Summer 2004) is 400MHz 3-3-3. This means that random access takes at least six cycles of a 200MHz (5ns) bus, so is 30ns.

Only the data are sent at the doubled rate: commands are sent at the undoubled rate, so t_{RCD} must still be an integer.

DDR2 and RAMBUS

DDR2 is similar to DDR. It uses a lower voltage (1.8V, not 2.5V), so uses less power, and it runs the core memory at one quarter of the doubled data-bus speeds, so has higher latencies than DDR1 of the same bandwidth, but is likely to be able to achieve higher bus speeds (and hence bandwidths).

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 400MHz or 533MHz bus, over which data is transfered twice each clock cycle, but 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's latency is still around 30 to 40ns.

With an effective data rate of 1066MHz bus RDRAM issues of path length and termination become quite exciting, and *clock forwarding* (different clock pulses for data travelling in different directions) is used, because Einstein has difficulty synchronising things any other way. (The speed of light is just one foot per nanosecond in vacuum.)

PC Ratings

The ‘PC’ ratings often confuse, and that is probably the idea.

For SDRAM, the PC rating is the bus speed: PC133 SDRAM uses a 133MHz bus, can burst-transfer one bus-width (usually of 8 bytes) per clock cycle.

For RAMBUS, the PC rating is the clock-doubled speed: PC800 RDRAM uses a 400MHz bus, and can transfer one bus-width (usually of 2 bytes) per half clock cycle.

For DDR-SDRAM, the PC rating is the burst rate in MB/s.

So, for fair comparison, one should multiply SDRAM PC ratings by eight, and RAMBUS ones by two.

The rare 32 bit variant of RAMBUS uses burst rate in MB/s also.

Note that although PC ratings place limits on the other timing parameters, they do not specify them. Not all PC3200 DDR memory is equally fast.

Flash RAM

Flash RAM is found in USB storage devices, digital cameras, and it often stores computers' BIOSes. Unlike DRAM and SRAM, it has three distinct operations: read, erase and write. Writes are permitted only to an area which has been erased, and the erase operation acts on blocks (typically 16K), not bits or bytes.

Two types exist: one based on NOR gates, which yields 'low' read latencies (~ 70ns), but very slow erase times (<100KB/s), and one based on NAND gates, which has poor read latencies ($\sim 20\mu$s), but much faster erase and write times (around 10MB/s). Both have read bandwidths of over 20MB/s.

The NOR-based version is used for 'ROMs' holding programs to be executed: fast random reads are needed, and writing (e.g. BIOS upgrades) is rare. The NAND-based version is used for storage devices, where reads and writes occur at the block, not byte, level, and writes are common.

Flash devices claim data retention periods of over 10 years, and support for 10^4 (MLC) or 10^5 (SLC) erase cycles. All perform error detection and correction internally.

SLC: single level cell, one bit per cell. MLC: multi-level cell, four different charge levels recognised in a cell, giving two bits, and a much larger sensitivity to charge leakage.

But is it correct?

All forms of DRAM need refreshing as the charge leaks off 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. It cannot detect two-bit errors.

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. This standard level of ECC is sometimes known as SECDED: Single Error Corrected, Double Error Detected, and a common implementation is called the Hamming Code.

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%.

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 there would be more devastating, so most processors perform some form of error checking and/or correcting on their caches and registers, and data buses.

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.

Does it all matter?

TCM had 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-P4 PCs. . .

All Alphas and Suns have ECC as standard. 'Server-class' Pentium 4 systems have it as an option, but many 'desktop-class' PCs do not have it at all. All but one of TCM's P4-based PCs has ECC.

Speed Required

A typical CPU runs at, say, 1GHz. 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, 400MHz DDR-SDRAM, have a latency of 30 ns, and a peak bandwidth of 6.4GB/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 fails to address the latency issue, and fails for non-sequential access.

Bus widths of things currently in TCM:
Pentium 4 (RAMBUS): 32 bit
Pentium III: 64 bit
Alpha (XP900): 128 bit
Pentium 4 (DDR): 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.

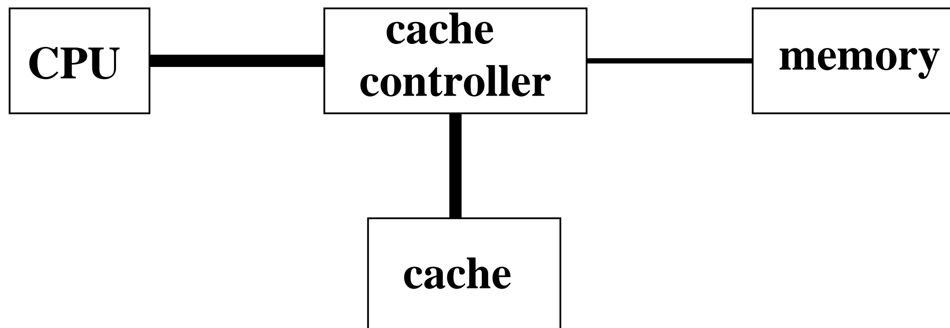
One assumes that data references have either *temporal locality* or *spatial locality*.

Temporal locality: if a location is referenced, it is likely to be referenced again soon.
Spatial locality: if a location is referenced, its neighbour is likely to be referenced soon.

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.

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

Computers use base-2, but humans tend not to like reading long base-2 numbers, and object to converting between base-2 and base-10. However, getting humans to work in base-16 and to 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.

```
0101 1101 0010 1010 1111 0001 1100 0011

 5    C    2    A    F    1    B    3
```

So

$$01011101001010101111000111000011_2$$
$$= 5C2AF1B3_{16} = 1546318259$$

As one hex digit is equivalent to four bits, 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.

Lines

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

We need to scan 65536 tags to determine whether something is in the cache. This will take far too long.

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

$$\begin{aligned} \text{64KB} &= 65536 \times 1 \text{ byte} \\ &= 4096 \times 16 \text{ bytes} \end{aligned}$$

We now need just 4096 tags, and 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.

We now have 64KB for real data, and 8KB for tags.

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.

A Further Compromise

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

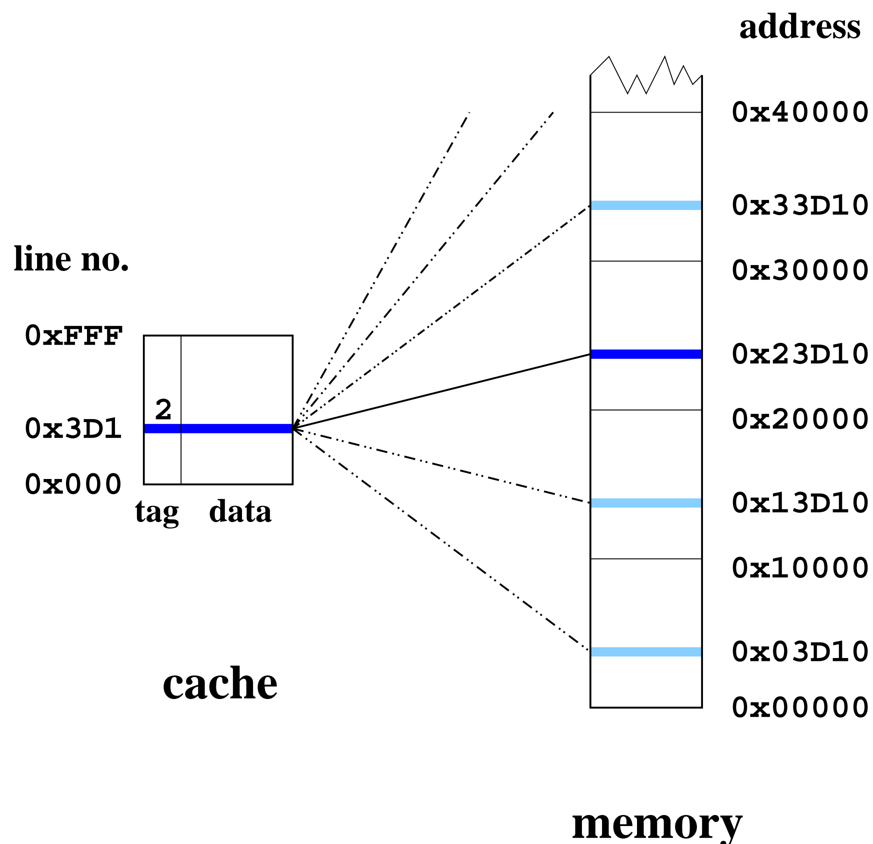
On seeing an address, e.g. `0x23D17`, we discard the last 4 bits, and scan all 4096 tags for the number `0x23D1`.

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

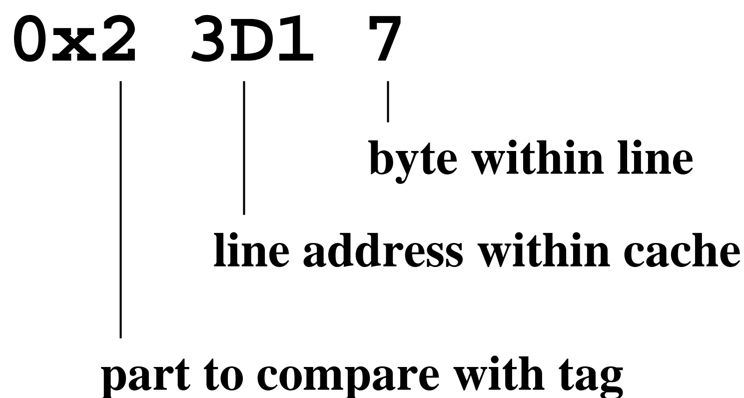
We have just developed a *direct mapped* cache. Each address in memory maps directly to a single location in cache, and each location in cache maps to multiple (here 16) locations in memory.



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

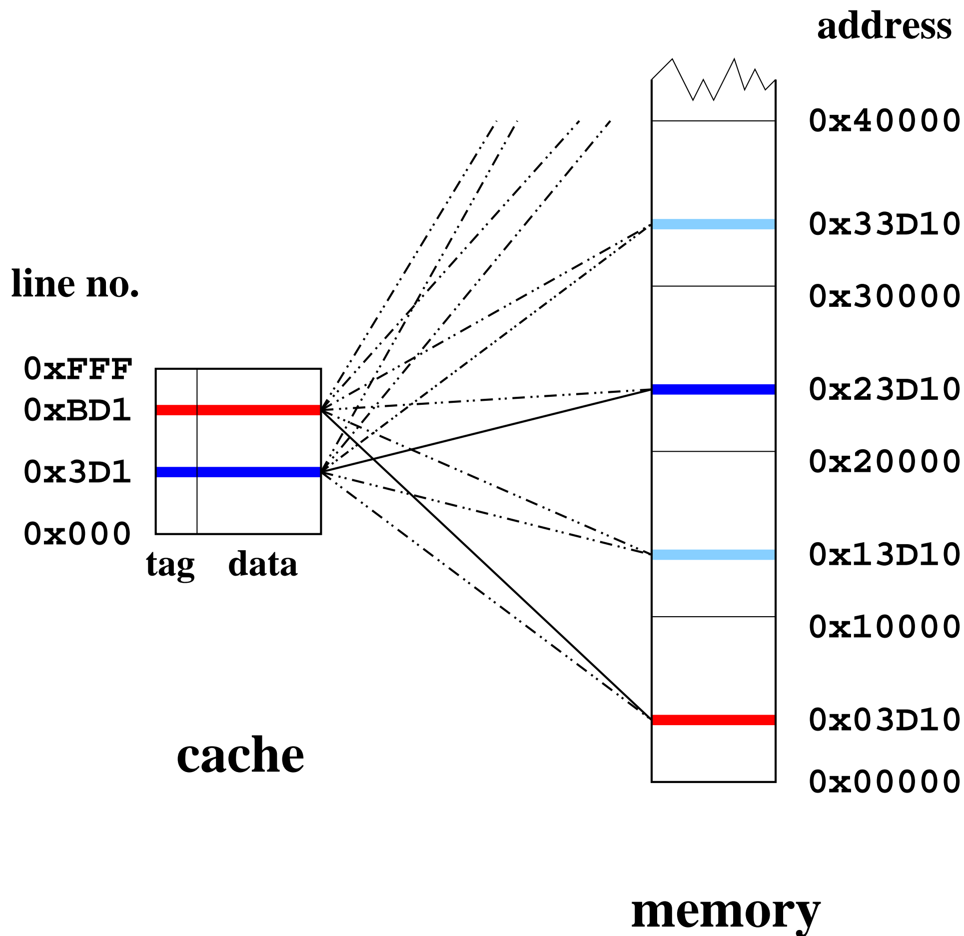
Even better, widely separated pieces of memory can be in cache simultaneously. E.g. 0x15674 in line 0x567 and 0xC4288 in line 0x428.

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.

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 49.

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

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, 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 (i.e. addresses which are a multiple of four), and 8 byte objects on 8 byte boundaries.

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.

The Relevance of Theory

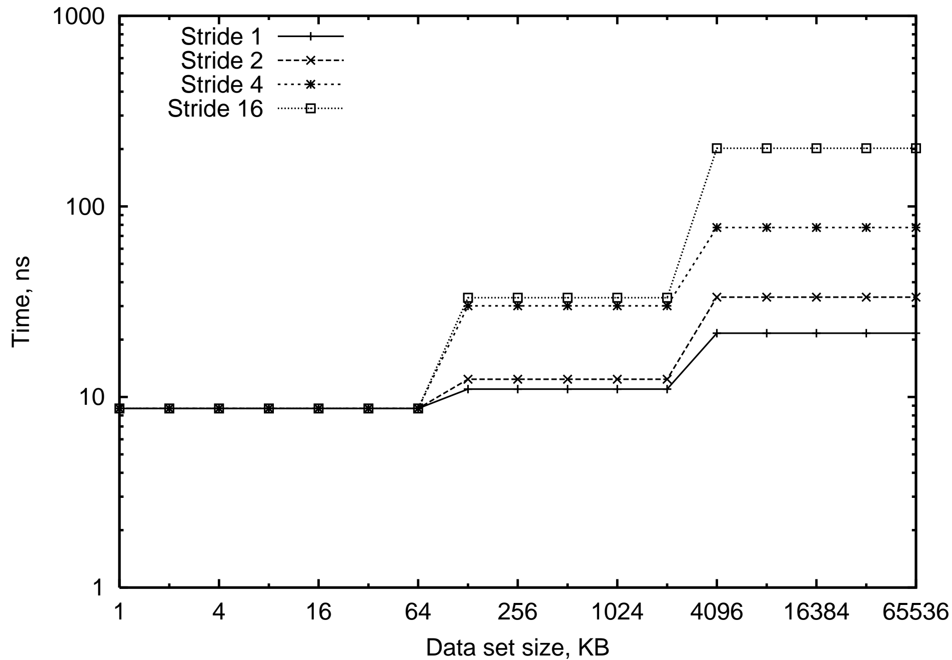
```
integer a(*),i,j

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

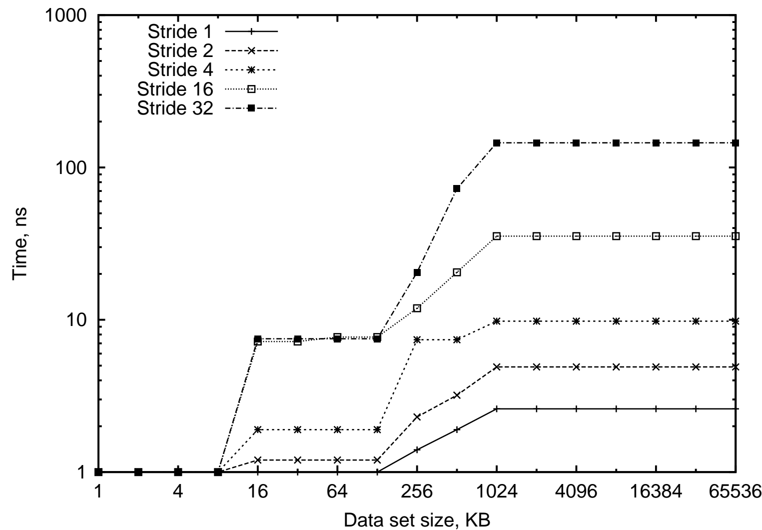


With a 16 element (64 bytes) stride, we see access times of 8.7ns for primary cache, 33ns for secondary, and 202ns for main memory. The cache sizes are clearly 64KB and 2MB.

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).

Common CPU Families

i386

The first member of the IA32 family was the 80386, released in 1985. It consisted of a significant set of extensions to the previous 80286, and had 32 bit integer registers, and supported virtual addressing, pre-emptive multi-tasking, and the isolation of the hardware from user code. It was certainly capable of running UNIX, and did run Linux and Solaris. It contained about 375,000 transistors, and ran at 5V at speeds of 20 to 33MHz.

Its maths co-processor was a separate chip, the 80387, and the interface between them was bizarre and slow, taking about 15 clock cycles to transfer an instruction from the 386 to the 387.

Memory accesses took at least two clock cycles: one to send out the address required, and one to receive the data (up to four bytes). The fastest integer instructions also took two clock cycles.

Going faster

This modest speed was still much faster than contemporary DRAM, which had an access time of around 80ns, compared to the CPU clock of 30 to 50ns. So an external cache controller and cache memory were normally placed on the motherboard to provide 32 to 64K of faster SRAM memory.

There was little point in the core going faster, as the average instruction required more than one memory reference, so was bound to take at over two clock cycles anyway.

The instruction itself needs to be fetched from memory, and it may involve none, one, or two futher memory references if it needs to fetch or store data from/to memory rather than registers.

The i486

The next processor in the IA32 line, the i486, contained all the functionality of the i386, the i387, the cache controller, and an 8KB 4-way associative write-through cache on the one chip. It could fetch data from its internal cache in one clock cycle, not two, and read a 16 byte cache line into itself in just five clock cycles. The time to interface to the internal maths coprocessor was about three clock cycles, and the integer unit was redesigned to complete the simplest instructions in just one clock cycle.

The result contained almost three times as many transistors as the 386, but still ran at 20 to 50MHz, and 5V.

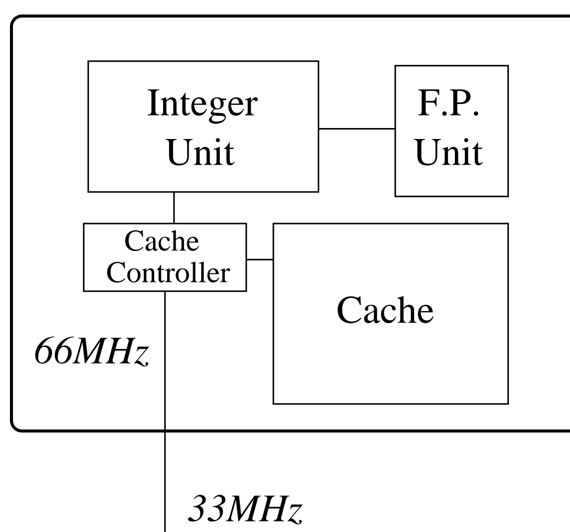
An external, second level of cache was usually present.

8KB of cache is 64,000 bits, and, at 6 transistors per SRAM cell, 384,000 transistors before one counts tag lines and control logic. Until it was possible to put around 1m transistors on a chip, one could not have a significant amount of cache on the chip.

Decoupling

Whereas the core of the 386 could compute as fast as one could get data into it, with 8KB of on-chip cache, the 486 can execute many instructions without making any reference to external memory. Thus, for certain small loops involving mainly reading from small areas of memory, the 486 was not constrained by the speed of its external interface, or bus.

Suddenly it is sensible to increase the speed of the chip without changing the speed of the external bus. Thus the 486DX2, with a core speed of 50 to 66MHz, twice the bus speed of 25 to 33MHz.



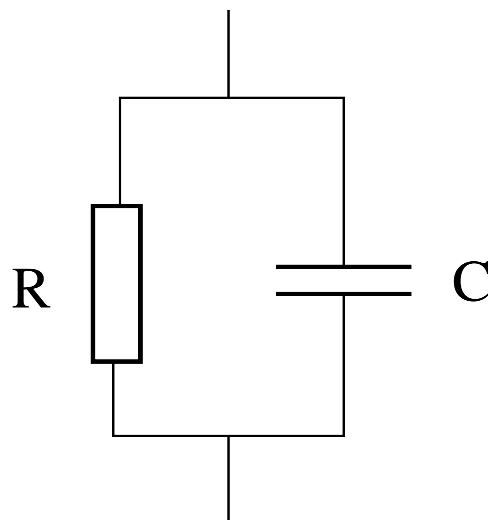
Speed limits

The first speed limit is provided by cache misses. The 486DX2 was typically just 70% faster than the corresponding 486, not 100% faster – the core was being idle whilst waiting for the external memory.

Cache improvements are the obvious answer, and the 486DX4, which ran its core at three times the external bus speed, had a internal 16KB cache, with some models having a write-back cache rather than write through.

Melt down

Just as important are thermal problems. A CPU can be modelled roughly as a capacitor in parallel with a resistor (representing leakage currents).



The power dissipated is simply

$$\frac{V^2}{R} + CV^2\nu$$

where ν is the frequency.

Keeping Cool

Technological advances keep on reducing the smallest feature size it is possible to place on a chip, from about 0.8μm for the 486 to 90nm today. This reduces C (in a slightly non-obvious fashion as C is proportional to area but inversely proportional to separation) decreases R, and permits one to decrease V whilst keeping electric fields unchanged.

In the days of the 486, leakage was negligible, and reducing V from 5V to 3.3V was the first big gain, which occured at 0.6μm, and enabled the 100MHz 486DX4 to exist.

The Pentium

The Pentium represented another major redesign of the core, but again almost no added instructions. The importance of improving the memory system to cope was understood, so the cache was split in two, 8KB for instructions, and 8KB write-back for data, ensuring that streaming large arrays into the processor would not disturb the instruction cache.

The speed of the external bus was increased to 60MHz and 66MHz (depending on model), and the width of that bus doubled to 64 bits. The Pentium had no 64 bit instructions, but it could fetch 8 bytes of data at once from memory.

The Pentium was also the first IA32 processor which could issue two instructions in a single clock-cycle, and which attempted branch prediction.

The original 60 and 66MHz Pentiums ran at 5V, and contained just over 3 million transistors. The later versions ran at 3.5V or less. Unlike the 486DX2 and DX4, the Pentiums could run at half-integer multiples of their bus speed, e.g. a 100MHz core with a 66MHz bus.

MMX

Intel's MultiMedia eXtensions were aimed at 2D bitmap image processing. The eight under-used floating point registers were re-used to hold 'vector' integer data: eight 8-bit values each, or four 16-bit values, or two 32-bit values, or one 64-bit value.

A range of operations was available for these data: add, subtract, shift, and, or, xor, compare and add with saturate. Mixing MMX instructions and normal floating-point instructions was not possible.

These improvements are irrelevant unless one recompiles one's code to use these extra instructions, and thus prevents it from running on IA32 processors which do not support MMX. However, the Pentium MMX also doubled the cache sizes to 16KB each, returned their associativity to 4-way from 2-way, and improved the branch predictor.

Add with saturate: using bytes, normally 200+200=144 (wrap-around). For add with saturate, 200+200=255 (largest possible value). This is very useful for certain photographic filters, and otherwise messy to code.

The Pentium MMX was manufactured at 0.35μm, so there was room for the extra transistors needed – about 1.2 million more. The voltage dropped again to just 2.8V.

The Pentium Pro, II and III

These processors, whose core is often refered to as the 'P6 core', are very similar, although the P II has MMX and the P III SSE extensions.

There are two major changes from the Pentium. Firstly the secondary cache and its controller move from the motherboard onto the CPU, further decoupling the CPU from the rest of the motherboard. Although the speed of the external bus increased modestly to 133MHz by the end of the P III line, the core speed reached over 1GHz.

Secondly, Intel gives up on CISC. Not quite, but these processors have a RISC core and an instruction decoder which takes the IA32 CISC instructions, and, for each one, issues one or more RISC 'μ-ops' to the core.

These processors had around 9 million transistors, excluding the secondary cache. Later versions ran at core voltages of under 2V.

SSE

When MMX was introduced, home computer users barely touched floating point. The rise of 3D animated graphics changed things dramatically.

However, 3D graphics do not need double precision arithmetic, but only single. So Intel decided to store two 32 bit floating point numbers in each of eight new floating point registers, and provide the basic FP operations operating on this 'vector' of two elements.

All useless for scientific work, and again requiring recompilation with a compiler capable of generating SSE code.

Real vector computers have typical vector lengths of 32 to 128 elements, not two or four.

When Intel's FP registers are used by MMX or SSE, the registers are addressed as eight conventional 'flat' registers, not as a stack.

The Pentium 4

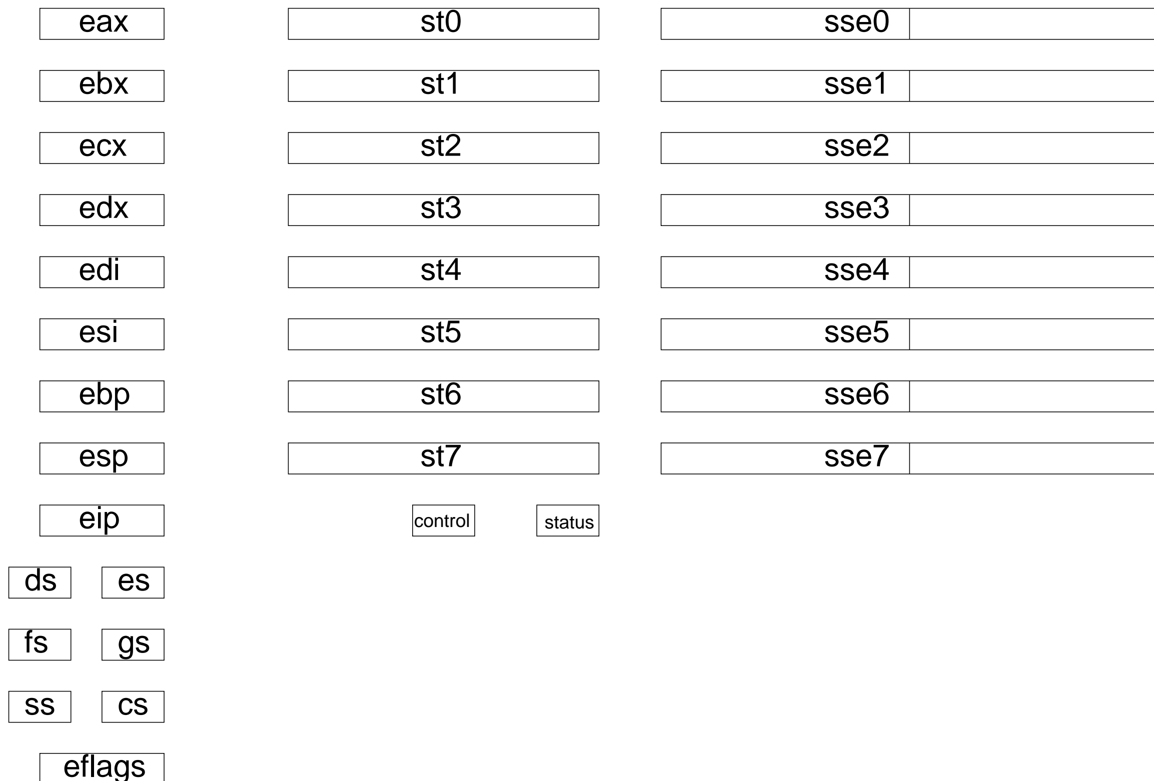
The Pentium 4 was a radical design change, though still using a RISC core fed with (different) 'μ-ops'. Unlike every previous generation change, where the new generation had run code faster than the old even at the same clock speed, the Pentium 4 ran code slower than a Pentium III at the same clock speed, but relied on being designed for very high clock-speeds.

Although the Pentium 4 has a clock speed about 50% higher than any of its rivals, its benchmark performance is similar. The 3.4GHz versions dissipate over 100W at 90nm, and it is rumoured that the power lost to leakage is approaching 50% of the total power. If true, this will make it hard to reduce power by shrinking the CPU further.

It introduced SSE2 – two double-precision floating point values held in each of eight 128 bit registers. At last scientfically useful!

Later versions included Hyperthreading.

IA32 Registers



This shows most of the registers present on a Pentium 4 or Pentium M: eight 32 bit 'general purpose' integer registers, six 16 bit segment registers, eight 80 bit floating-point registers arranged as a stack and also used by MMX integer instructions, and eight 128 bit SSE2 registers capable of holding two 64 bit doubles each. Also separate flags registers for integer and FP use, which have bits set by the results of comparisons.

x86-64

AMD's extension of the IA32 architecture to 64 bits, copied by Intel under the name EM64T, extends the existing integer registers to 64 bits, adds eight new integer registers, doubles the number of SSE2 floating-point registers, and adds a 'no execute' bit for memory management. It also removes many of the restrictions concerning which combinations of registers are permitted with certain instructions.

This extension is found in the Athlon64 and Opteron CPUs.

EM64T: Extended Memory 64 bit Technology

Alpha

The Alpha line started from a cleanish slate, and was designed as a simple RISC processor: 32 integer registers, all 64-bit, 32 fp registers, fixed length 32 bit instructions. and no support at all for 8 bit and 16 bit data types. The first version, the EV4 or 21064, appeared in 1992, with integral FPU and primary caches (8KB data, 8KB instruction, both direct mapped write through), and able to issue two instructions per clock-cycle (100MHz), at most one of them floating point.

In 1995, the next version (EV5 or 21164) added a 96KB 3-way associative write-back cache, and the ability to issue four instructions per clock-cycle, including simultaneous issue of FP add and multiply. The motherboard usually provided an additional external cache of 2 or 4MB.

Later Alphas

The next minor revision, the 21264A (EV5.6) added the 'BWX' extension, providing instructions for loading and storing single bytes and 16 bit values.

The final major revision of the core, the 21264 (EV6) supported out-of-order execution, as well as adding instructions for software prefetching of data, floating point square root, and bytewise minimum, maximum, and absolute difference of packed integer data (cf. Intel's MMX).

The cache architecture was improved to 64KB 2-way associative primary caches and secondary cache controller on the die, and a secondary cache of between 2MB and 16MB off-die but on the physical CPU module.

BWX: Byte Word eXtension

Clock speeds ranged from 460MHz to 1.25GHz for the EV6 series. The EV7 was little more than an EV6 with a smaller (1.75MB), faster, on-die secondary cache, an on-die memory controller interfacing directly to RAMBUS, and extra logic to make multiple-CPU machines readily.

The Others

Other CPUs follow the same sort of story. Sun's SPARC line has moved from being 32 bit RISC to 64 bit RISC (from the UltraSPARC III), and gained its multimedia extensions, which Sun calls 'VIS'. The PowerPC line of IBM, Motorola and Apple has followed a similar route, but calls its multimedia extensions 'Altivec', and they certainly include floating-point instructions. It became 64 bit with the PPC 970.

Alpha is unusual in not having made the transition from 32 bits to 64 bits: it has always been 64 bits. However, unfortunately it is also dead.

Most CPUs in the early 1980s had no cache on die, and any FPU was probably a separate chip. As feature sizes shrunk, and transistor counts increased, FPUs and small (16K) on-die caches were common by the early 1990s, and then larger, multi-level caches and even integrated memory controllers by the early 21st century.

The Future

Most CPU lines are now moving towards having multiple cores on a single die, sharing secondary cache. IBM's Power4 CPU was the first to do this, Sun's UltraSPARC IV is similar, and both AMD and Intel are expected to release 'multi-core' processors in 2005.

The argument for this is that the CPU core (even with primary cache) is much smaller the on-die secondary cache, so this is easily done. The argument against is that unless the CPU wishes to execute two programs at once, the second core is going to be idle.

However, more and more code is being written with parallelism in mind, and there is always that background MPEG decoder. . .

Permanent Storage

Disk Drives

Are remarkably boring, but worthy of mention because people do rely on them rather a lot. . .

Remarkably standard, with just three interfaces dominating the market for hard disks and CD ROMs: SCSI at the expensive end, EIDE (aka UDMA and ATA) at the cheap end, and SATA emerging in the middle.

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.
SATA: Serial ATA. Serial data bus supporting up to 127 devices using an IDE-like protocol.

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 120GB.

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.

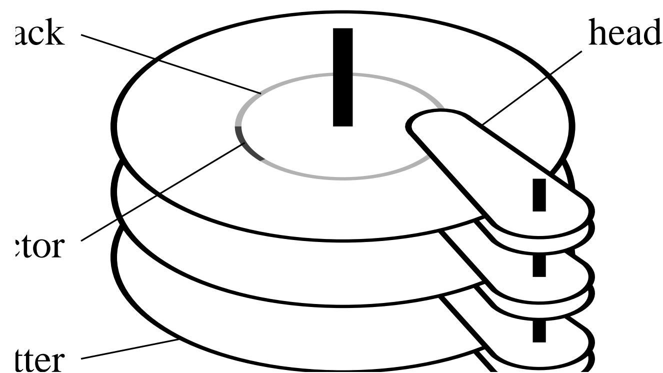
Although the heads move only radially, the air is dragged into tangential motion by the spinning platters, and in this air stream the heads fly.

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 $5\frac{1}{4}$ 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: $2 \times 80 \times 18 \times 512 = 1440$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 much larger than the tiny magnetic sensors used for hard drives. Thus seek times are around ten times higher at 80ms.

The data are written onto a single spiral track, starting at the centre. The capacity is around 700MB, or 65 minutes of uncompressed 16 bit stereo audio.

The transfer rate for an audio CD player is a constant 150KB/s, so audio CD players spin the disks at a constant linear velocity, 1.3m/s, corresponding to 200rpm when the head is at the edge of the disk, and 500rpm close to the centre.

Physical considerations

CD drives use an infra-red laser (780nm) with a spot size of 2.1μm. The tracks have a pitch of 1.6μm, and the pits a length of 0.8μm. For Al disks, the pit depth is $\lambda/4$, so that light reflected from the pit and the surrounding 'land' interferes destructively, and the pit appears dark. For recordable disks, dyes which permanently change their reflectivity on heating are used.

For synchronisation purposes, each 'pit' or 'land' region must be at least three bits long, and no more than eleven bits long. The eight-bit bytes are encoded into 17-bit objects which guarantee these properties.

Data are written in 2,353 byte sectors, each containing 2K of data, 276 bytes of error correction information, and a small amount of header information.

Physical limits

When used for data, there is no reason to spin the disk at a constant linear speed, so constant angular speed is used. However, above about 10,000 rpm the disk becomes unstable to cracks propagating from the centre, thus shattering the disk. This limits the rotation speed to about $52\times$ the original audio CD speed of 200rpm.

When spun at constant angular velocity, faster data transfer rates are achieved near the outer edge of the disk than near the inner edge.

The laser spot size is governed by the wavelength used, and the lens size, as an Airy diffraction disk will form.

So drives will not get any faster than $52\times$, and will only achieve that speed when the head is at the edge.

DVD drives

A DVD disk is physically the same size as a CD, but it uses a higher recording density giving 4.7GB per side of a disk. Some DVDs are double-sided, leaving almost no-where to write the label. . .

The laser used is red, 650nm, and the lens is wider, reducing the spot size to 1.3μm. When used for video the data transfer rate is 1.38MB/s, and the disk spins at 500 to 1500rpm.

The restriction on pit and land lengths is now $3 \leq l \leq 14$, and a byte is encoded to 16 bits. The data are still written in 2048 byte sectors, but the ECC data is placed in a full sector of its own, which occurs after every 16 data sectors.

The track pitch is 0.74μm and the pit length 0.4μm per bit.

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.

Thus the disk stores both the real data of the files, and data describing the structure of the files themselves. The latter is called *metadata*.

The metadata

A filesystem 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.
- a concept of file ownership.
- a concept of access permissions.

The last two are only really necessary for multiuser operating systems.

Consistency

There are various consistency rules that the metadata must obey. The number of blocks assigned to a file must be consistent with the length of the file. Each block must be free, or associated with a single file, or part of the metadata. Each subdirectory must have precisely one parent directory.

Consistency is often impossible to maintain during operations such as file deletion, and expensive (in terms of performance) at other times. Thus one should warn a computer than one intends to turn it off by shutting it down properly: it will then ensure that all metadata changes are sent to the disk so that one has a consistent filesystem.

If this is not done, it will wish to run some consistency checking program, which will automatically 'correct' errors using some highly artificial intelligence. Sometimes this works. . .

Windows uses `scandisk` for this, UNIX `fsck`.

Journals

Some filesystems keep *journals*: they write a log of operations they are about to do, then do them, and then remove those items from the log. If such a system loses power, when it next boots it can read the journal and complete any outstanding operations.

This slows down disk writes, as there is extra activity to the journal involved. For this reason usually only metadata are journalled, and the actual file data are not.

The FAT, ext2 and ufs filesystems do not journal, whereas NTFS, ext3, AdvFS, and Sun's logging extensions to ufs do.

As journalled filesystems are rarely checked, they can become amusingly corrupt.

Mirrors

A way of increasing reliability is for the OS or the disk controller 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.

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.

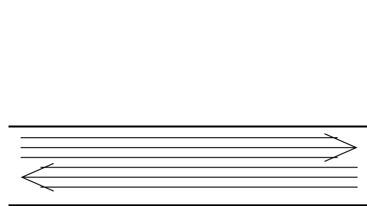
Tapes ideally store a single file each. Just data are stored, with no metadata (name, length, owner etc). The only metadata that a tape drive really understands are the 'end of file' mark and 'end of tape' mark. Thus it is possible to put several files on one tape, and then index the result by hand with 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.

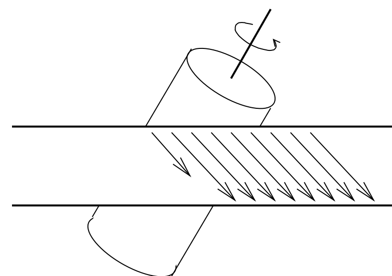
Tape Technologies

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.

Operating Systems

Operating Systems

Resource allocation (CPU time, memory)

Fair allocation between competing processes is good.

File system

Disks store raw data. File names and directories are an invention of the OS.

Hardware abstraction

A program wants to see a generic graphics device or keyboard, without needing to know the precise details of the model attached.

Security

Program A should be kept from program B's memory, and user A from user B's files.

A Process

A process is a single copy of a program which is running or, in some sense, active.

A process has resources, such as memory and open files, it is given time, *scheduling slots*, executing on a CPU with a certain priority, it has resource limits (maximum amounts of memory, CPU time, etc. it can claim), and it has an *environment*. Lastly, it has a parent. Each process is associated with a single user.

These resources are exclusive to each process, and no process can change another's resources. Processes are mostly independent.

Each process has a unique *PID*, its Process ID.

There are one or two simplifications above, some of which will be untangled later.

Process Trees

As each process has a sole parent, and may have no, one, or multiple children, on can draw a 'tree' of processes showing their relationships.

```
> ps -e --forest
PID TTY          TIME CMD
725 ?        00:00:00 xdm
736 ?        00:00:02  \_ X
737 ?        00:00:00  \_ xdm
768 ?        00:00:00      \_ fvwm2
819 ?        00:00:00          \_ FvwmButtons
821 ?        00:00:00          \_ FvwmIconMan
822 ?        00:00:00          \_ FvwmPager
823 ?        00:00:00          \_ xclock
824 ?        00:00:00          \_ xload
825 ?        00:00:00          \_ xterm
827 pts/0    00:00:00              \_ bash
836 pts/0    00:00:01                  \_ emacs
854 pts/0    00:00:00                  \_ ps
```

The `forest` option is found only on Gnu ps commands.

Address spaces

Each process has its own independent virtual address space for accessing memory. The dynamic mechanism for mapping this to real, physical memory will be considered later.

The result is that one process has no mechanism for accessing another process's memory, for a unique virtual address includes both the address *and* the PID, and identical virtual addresses with different PIDs will map to completely different physical addresses.

Thus with 32 bit Linux, the standard memory map leaves the code of a program starting at address `0x08000000` and the stack at `0xC000000`, and this is what each and every program sees each and every time it is executed, regardless of what else is going on in physical memory.

DOS uses only physical addressing. MacOS Classic and 16 bit versions of Windows use a single virtual address space for all processes. UNIX and WindowsNT use a separate VAS for each process as described here.

Kernels

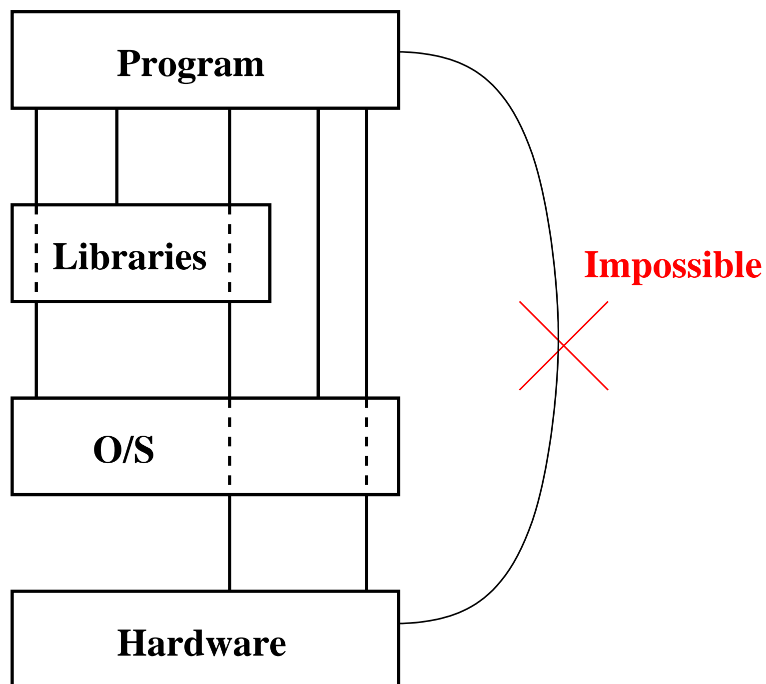
The OS kernel is very different from a process. There can only be one of it, and it can address physical memory directly, address any hardware directly, and do anything to any process. Indeed, one part of the kernel, the *scheduler*, is responsible for giving the processes any CPU time at all.

A process wishing to access some hardware device must do so via the kernel, and cannot do so directly. The kernel is able to ensure that when the CPU is not executing kernel code it is unable to execute certain privileged instructions which might allow a process direct access to the hardware.

This clearly requires some support from the CPU. CPUs of the early 1980s (8086 in the IBM PC, Z80 in the Sinclair Spectrum, 6502 in the BBC B, and others) simply did not have sufficient functionality. The i386 was the first PC processor really capable of running a modern OS.

Isolation

The kernel should form an inpenetrable layer between a process and the physical hardwaer of the machine, and thus ensure that the hardware resources are not abused and are fairly shared.



Kernel Functions

The Linux 2.6 kernel offers 273 different functions. Many of these are familiar:

Numeric	Name	C	Shell
1	exit	exit	exit
3	read	read	
4	write	write	
5	open	open	
6	close	close	
9	link	link	ln
10	unlink	unlink	rm
12	chdir	chdir	cd
15	chmod	chmod	chmod
21	mount	mount	mount
22	umount	umount	umount
34	nice	nice	nice
39	mkdir	mkdir	mkdir
40	rmdir	rmdir	rmdir
48	kill	kill	kill
52	umount2	umount2	umount -f
83	symlink	symlink	ln -s
88	reboot	reboot	reboot
95	fchown	fchown	chown

Whilst the Linux kernel maintains the ability to run binaries compiled for previous versions, this list can only grow. A binary program may call function 22 directly, expecting to find `umount`, and the fact that `umount2` has been added and would also do the job (and more) is irrelevant to it.

Some OSes discourage one from calling the kernel directly, supporting only calls via a supplied dynamically-linked library. Then it is the library interface which can only grow.

Disk Access

A process will usually access a disk drive in terms of files. The kernel will oblige, imposing any restrictions indicated by the filesystem as it does so.

The kernel also presents disk drives as *device files*. These can be used by a process to read and write raw data blocks directly from and to the disk without going via the filesystem. Any process which can do this can therefore bypass any access restrictions imposed by the filesystem.

This is still not the real, physical hardware. The process is still shielded from having to worry about whether it should be sending IDE, SCSI or floppy commands to the disk, about which PCI bus the controller is on, and which ID it has on that bus, etc. It is also prevented from sending commands other than reads and writes: not the harmless 'identify yourself' command, nor the harmful 'update your firmware from me' command.

Root Processes

A process run by root is little different from any other process. It still needs to call the kernel to access any hardware, and the access will still be indirect. The difference is that the kernel is more likely to say 'yes.' A root process can trivially read from, or write to, any regular file or device file, send a signal to any process, change any processes scheduling priority up or down, etc.

However, it still operates in its own virtual address space, and it will still die with a segmentation fault if it tries to access memory not allocated to it. It will also die if it tries to execute a CPU instruction reserved for kernel mode.

Accidents and Design

If a non-root process hits a bug and starts behaving randomly, it is extremely unlikely to have any adverse affect on anything, beyond perhaps wasting CPU time in an infinite loop, or filling a disk with an infinite file.

A root process is much more likely to cause trouble if it is buggy, but the expected outcome is still an uneventful death.

Triggering a bug in the kernel is very much more likely to cause trouble. A crash of the whole operating system is the expected outcome, and data loss is not unlikely.

Keeping the kernel small is therefore a good idea.

Malign Design

If a user process has malign intent and intelligence, it can probably crash the system, or at least make it unusable. Merely creating several dozen copies of itself, and then having each add zeros until they reach infinity should do the trick.

A malign root process can trivially do enormous damage: read and modify any files, intercept any data passing through the machine, install a new or modified OS, reboot the machine, etc.

Other Privilege Models

The UNIX privilege model is somewhere between simple and simplistic. There are pretty much three levels: unprivileged user, root, kernel, and the last two effectively have full control.

The world of VMS (and Windows NT) is different. It contains a long list of extra privileges a process might wish to have, such as read all files on local disks (a backup process), send 'interesting' network packets (see later), change user id, listen on privileged network ports, send signals to any process, etc.

This model is begining to creep into UNIX, particularly IRIX and Linux, in the form of 'capabilities.'

It may seem more sophisticated, and therefore superior, but it does have significant pitfalls. The capabilities overlap considerably, so it is much harder to work out how much privilege one really has given a user or a process. E.g. the privilege of writing to any file allows one to change any part of the OS, and thus gives one full control. So would giving full access to the raw disk device, *or* to the disk controller, *or* to the bus the controller is on. Writing to any file not owned by the system would probably be sufficient if one is cunning, sending signals *and* listening on privileged ports would surely be enough, etc.

The traditional UNIX model is so simple that the Board can almost understand it, so mistakes are less likely.

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.

When extra registers are added to a CPU architecture (e.g. SSE2), the OS must be aware of the need to save them, and the structure which stores register values for inactive processes must grow appropriately.

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.

Memory Management

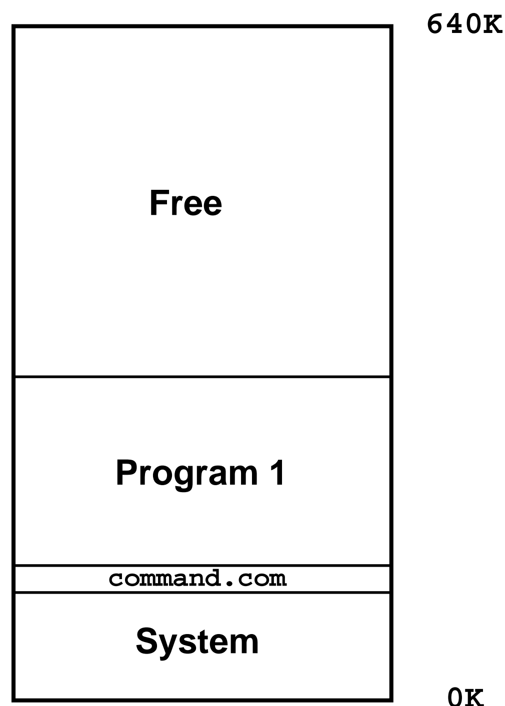
An operating system ought to have a mechanism for assigning memory to different processes on request.

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 used to have a memory manager with just these properties too.

What went wrong?

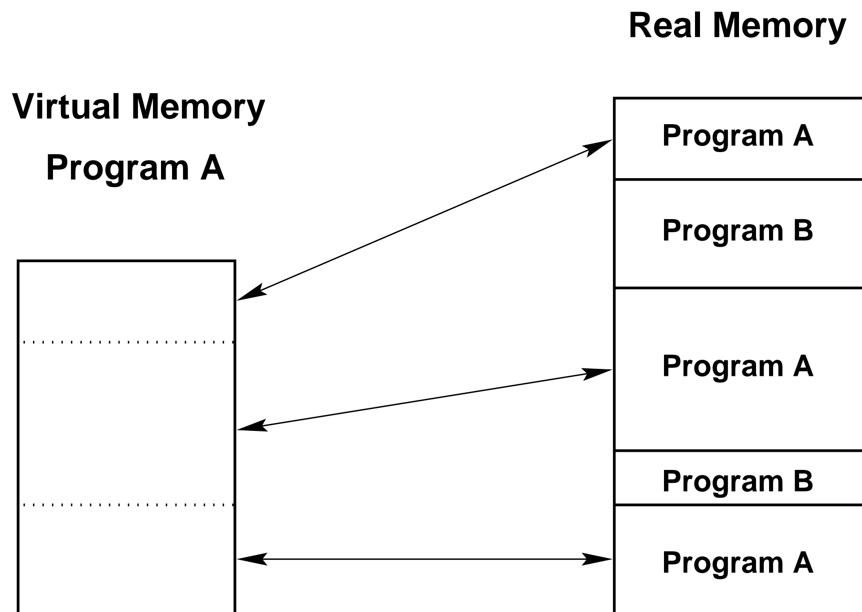
Actually, very little. The memory management of DOS or early versions of MacOS was about as good as one could achieve on the processors then available (8086 and 68000 respectively).

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 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.

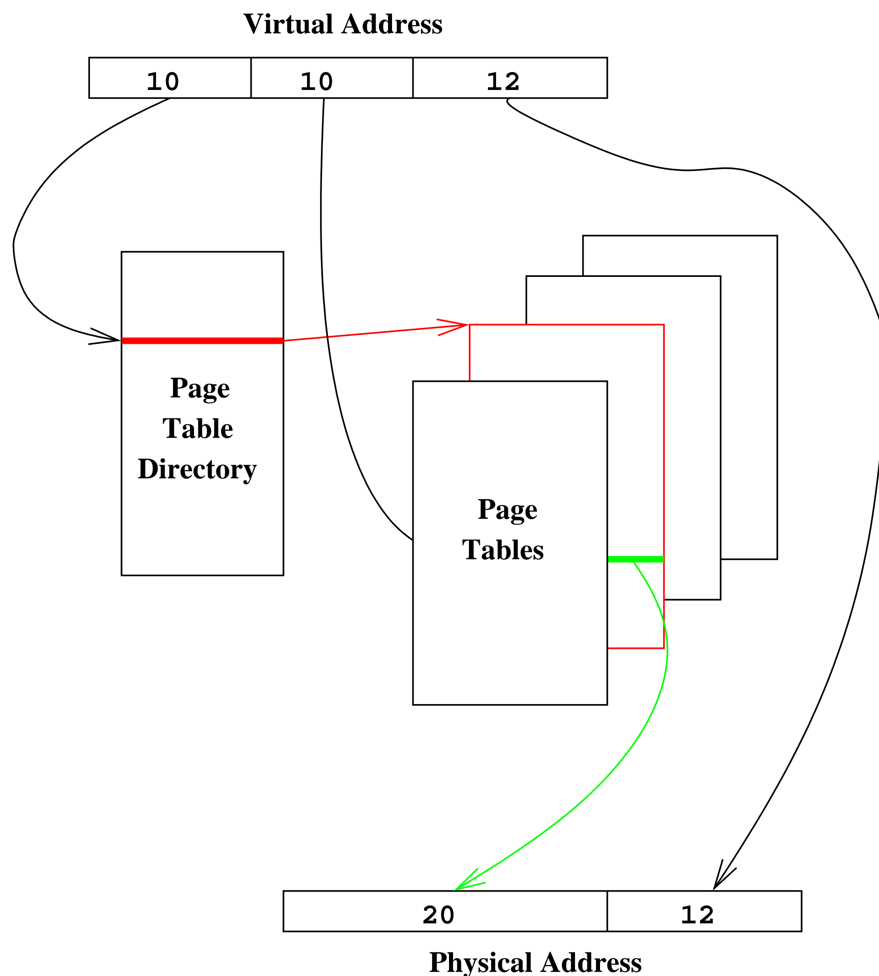
The first ten bits of an address are used as an index into the page table directory. If no virtual address with that starting sequence is valid, the directory will indicate this, and a *page fault* will occur. Otherwise, twenty bits indicating the position in physical memory of the page table for the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

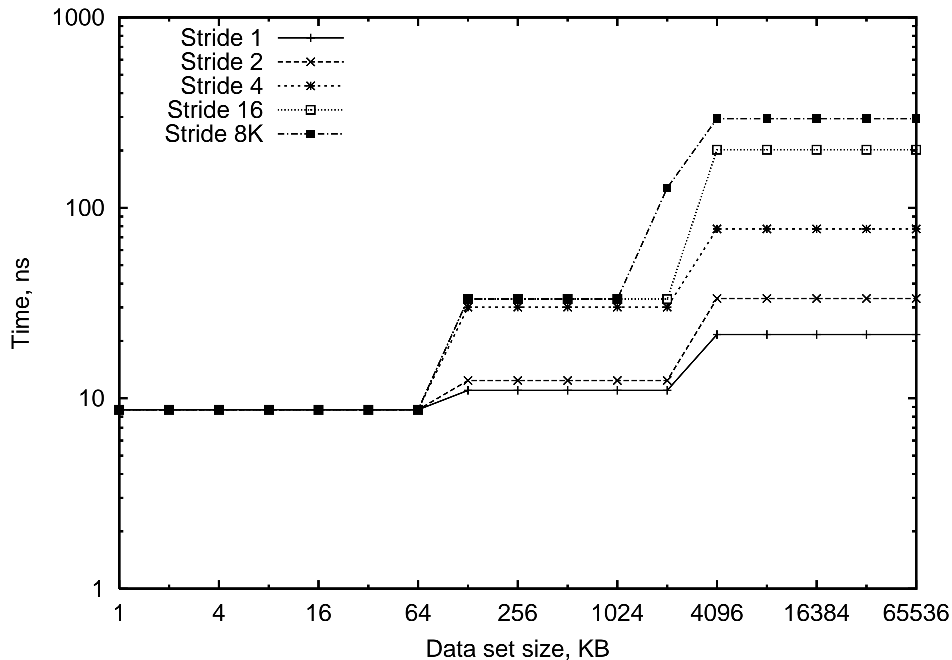
Fortunately, the CPU understands pages sufficiently well that it remembers where to find frequently-referenced pages using a special cache called a TLB. This means that it does not have to keep asking the operating system where a page has been placed.

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 131, 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.

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

Many CPUs support multiple page sizes, such as the Pentium which supports 4KB or 4MB, the UltraSPARC III which supports 8K, 64K, 512K and 4MB, and the EV6 Alpha allows a single TLB entry to refer to 1, 8, 64 or 512 consecutive pages. This reintroduces fragmentation problems: to allocate a 4MB page 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.

‘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.

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 deleted section on filesystems for a definition of an inode.

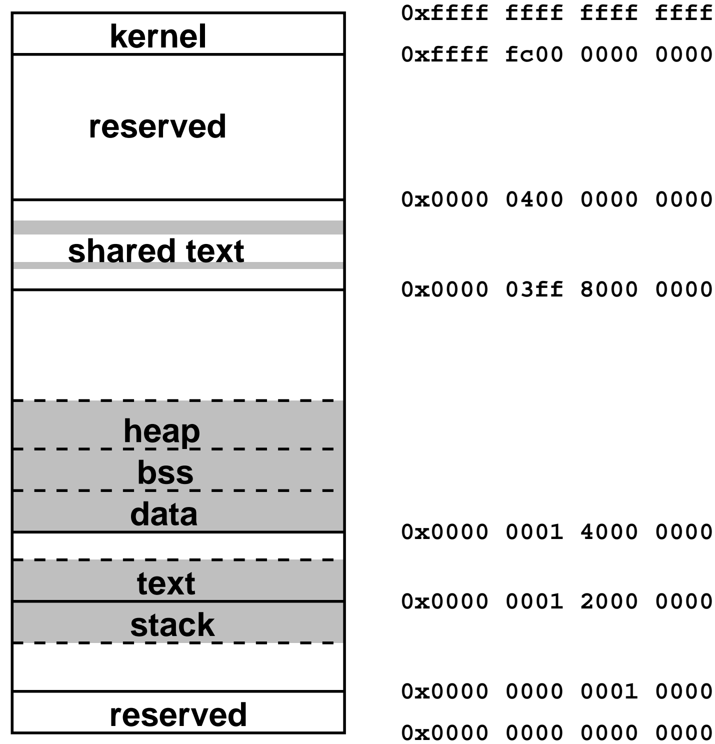
Enforcement

Most OSes can enforce the above restriction, denying attempts to modify read-only segments, or to execute code from data segments. This needs assistance from the CPU.

Regrettably Intel's IA32 line cannot distinguish between 'read' and 'execute' permissions.

Some compilers sensibly put constants into the text segment. As this segment does not have write permission, illicit attempts to modify such constants then fail with a segmentation fault.

A UNIX Memory Map

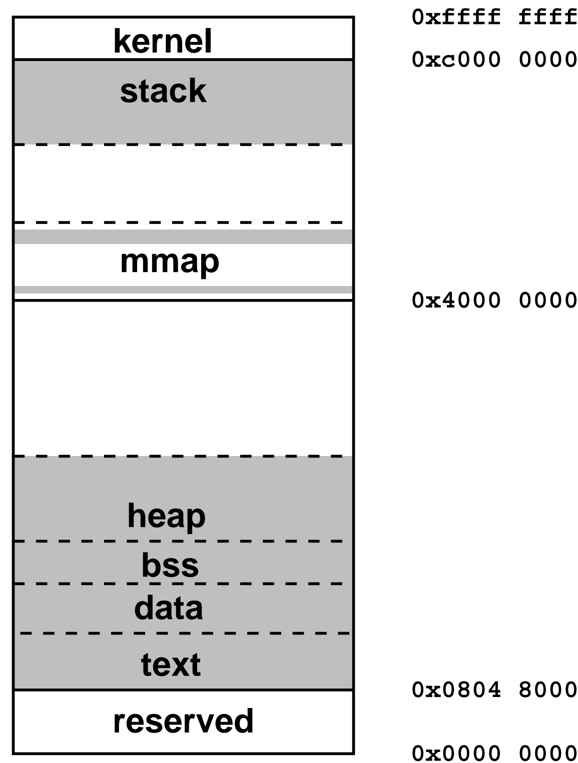


N.B. This view is per process, not for the whole machine.

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.

Parallel Computers

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:

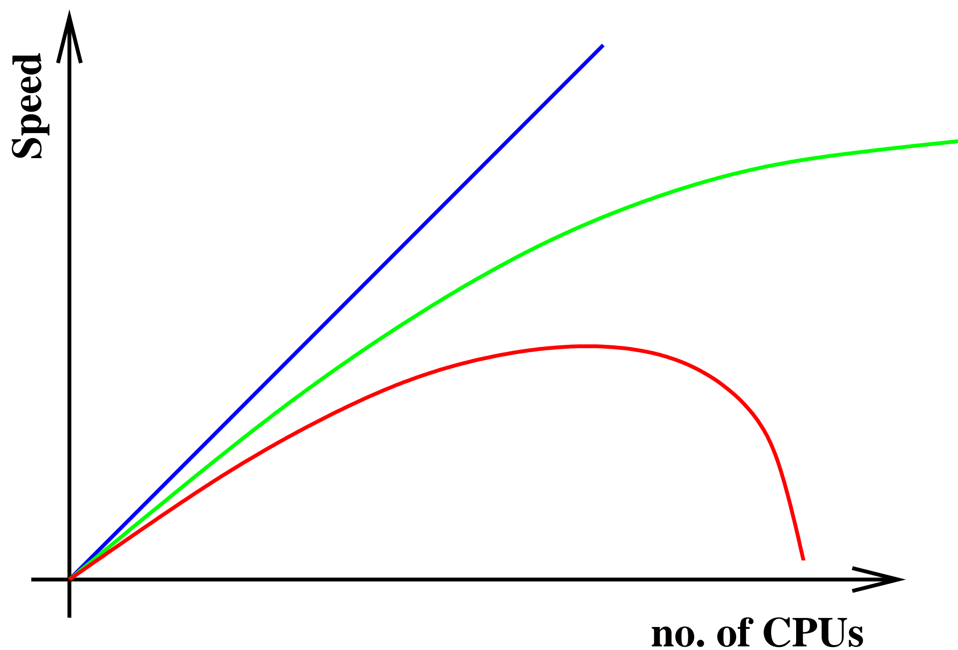
Integration of differential equation over very many timesteps. Clearly one cannot start the 5,000th timestep until the 4,999th has been finished. The process is fundamentally serial.

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%.

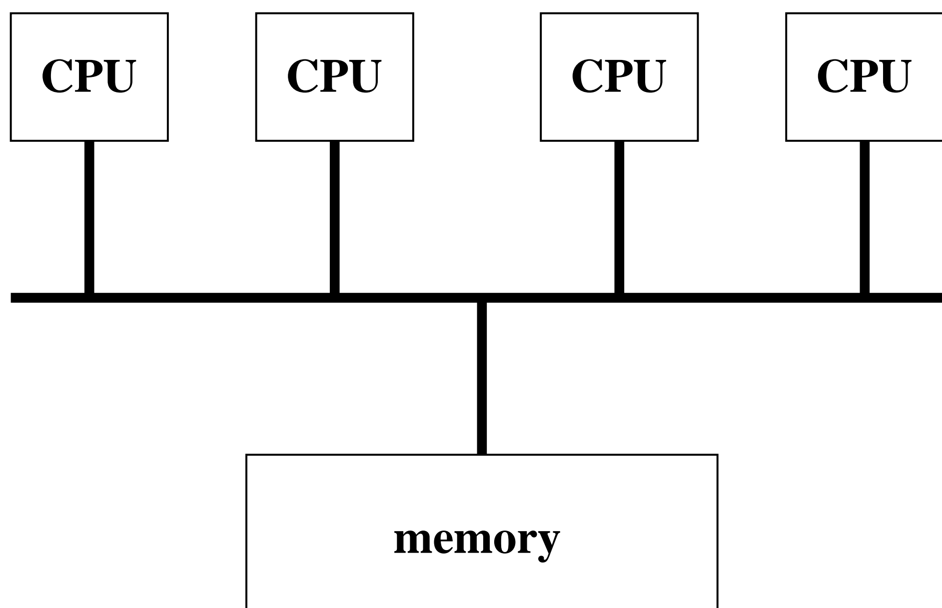
Supercomputers like *big* problems.

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.

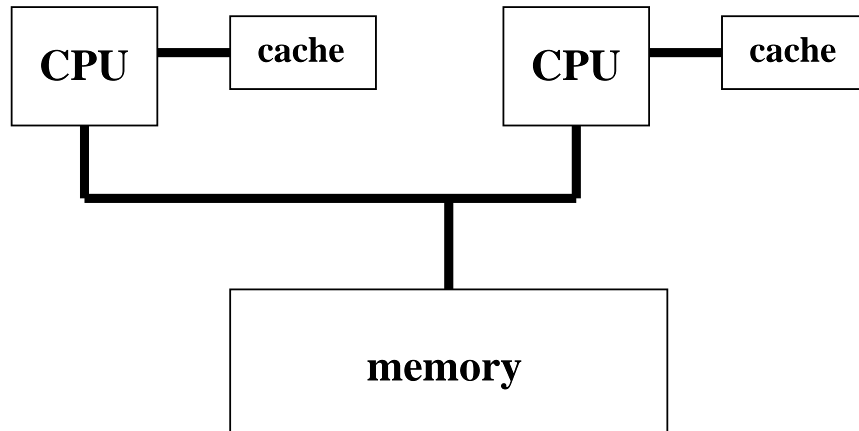
Shared memory

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

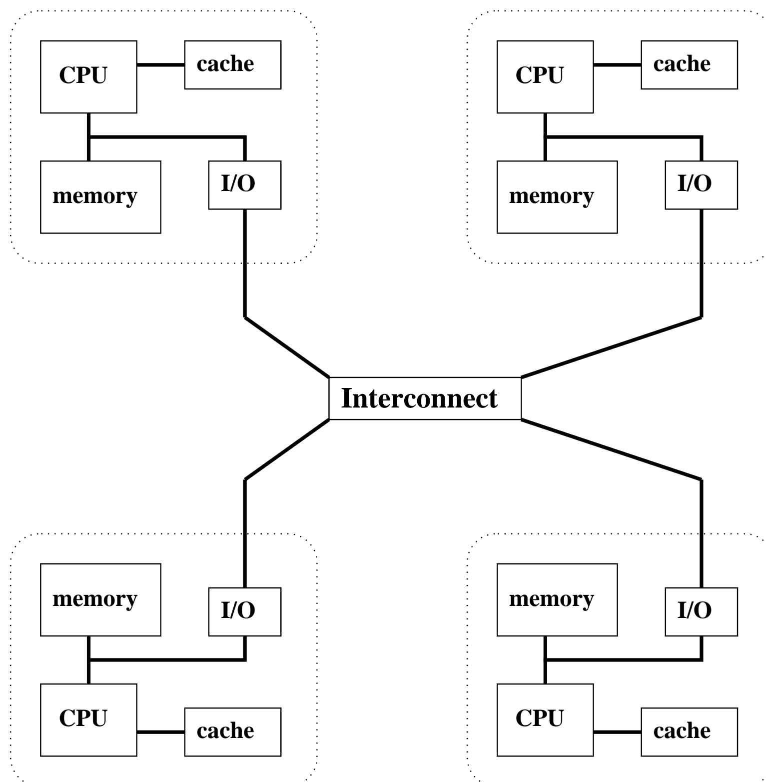
Two techniques are used for communicating with the other cache controllers. One is simply that details of all line fills are broadcast to all cache controllers, and the fill does not progress until the other controllers have had an opportunity to reveal that they held the line.

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.

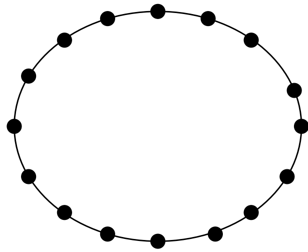
Two useful ways of characterising a network are the 'diameter', the maximum number of hops from one node to another, and the bisectional bandwidth, the bandwidth between two halves of the machine.

	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

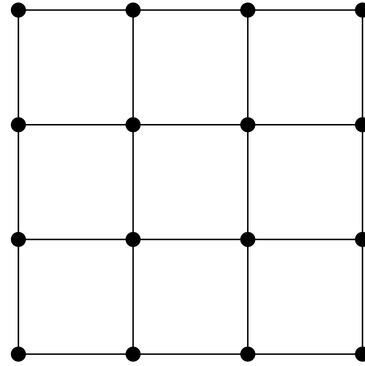
The Cray T3D is a 2D torus, the IBM SP2 a fat tree, the SGI Origin2000 a form of hypercube, and the Hitachi SR2201 a 3D X-bar.

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

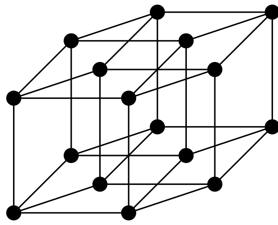
16 Nodes. . .



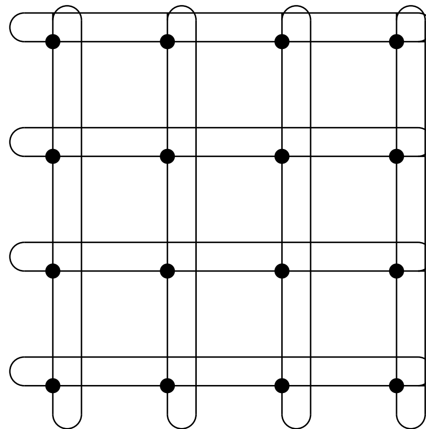
Ring (1D torus)



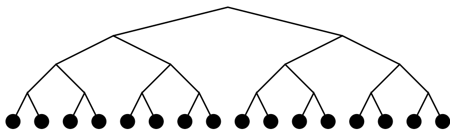
2D mesh



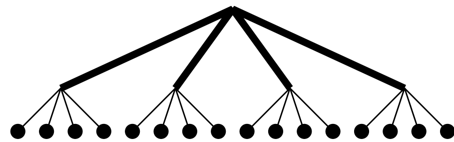
Hypercube



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-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

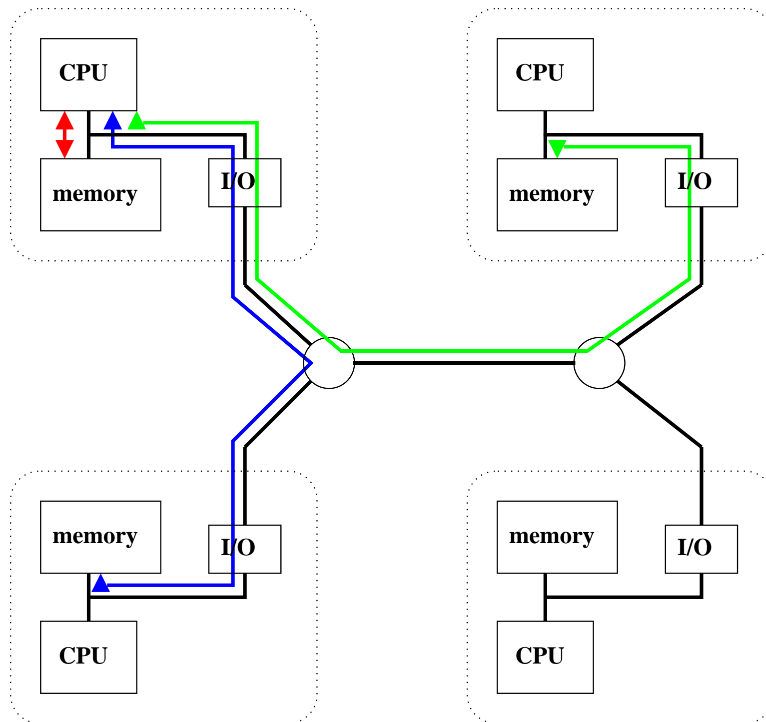
Most modern SMP machines are not bus based. Internally they are configured like MPPs, with the memory physically distributed amongst the processors. Much magic makes this distributed memory appear to be global.

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.

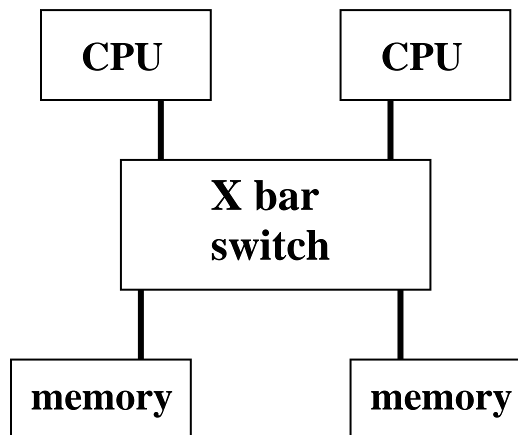
Whereas with MPP all memory accesses are guaranteed to be local, as one cannot access remote memory except by explicit requests at the program level, with SMP the compiler has many ways of getting things wrong.

```
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.

X-bar Switches

Some modern, small SMP machines, such as Compaq's DS25 or ES45, IBM's Sphinx, SGI's Octane or Sun's SunFire V480, use a rather different architecture.

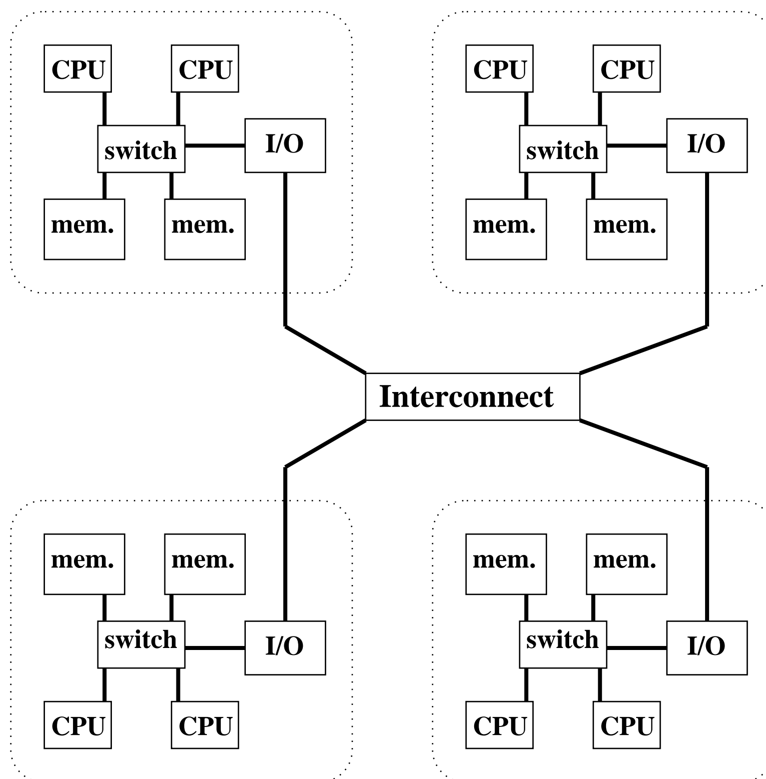


In this two CPU example, there are two distinct memory 'banks' and two CPUs joined by a switch which will let either CPU talk to either memory bank whilst the other pair also talk simultaneously. This fails when both CPUs wish to access data in the same memory bank.

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 SP, 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.

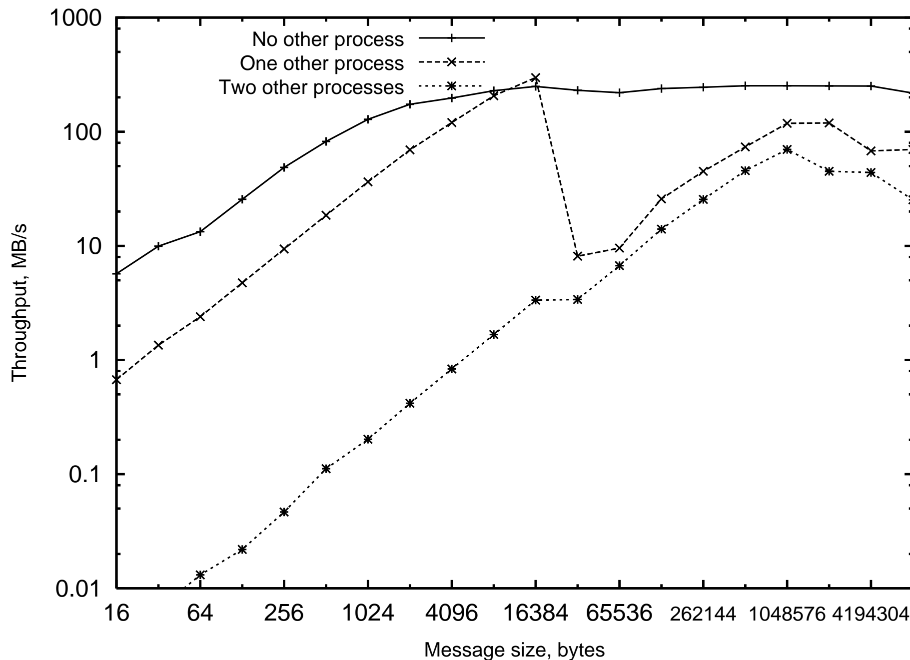
Practical Parallelism

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.

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.

SMT tends to reduce reproducibility in run-times: a head-ache for code tuning.

It also works poorly if the bottle-neck was throughput, either in terms of memory bandwidth, or for a single functional unit.

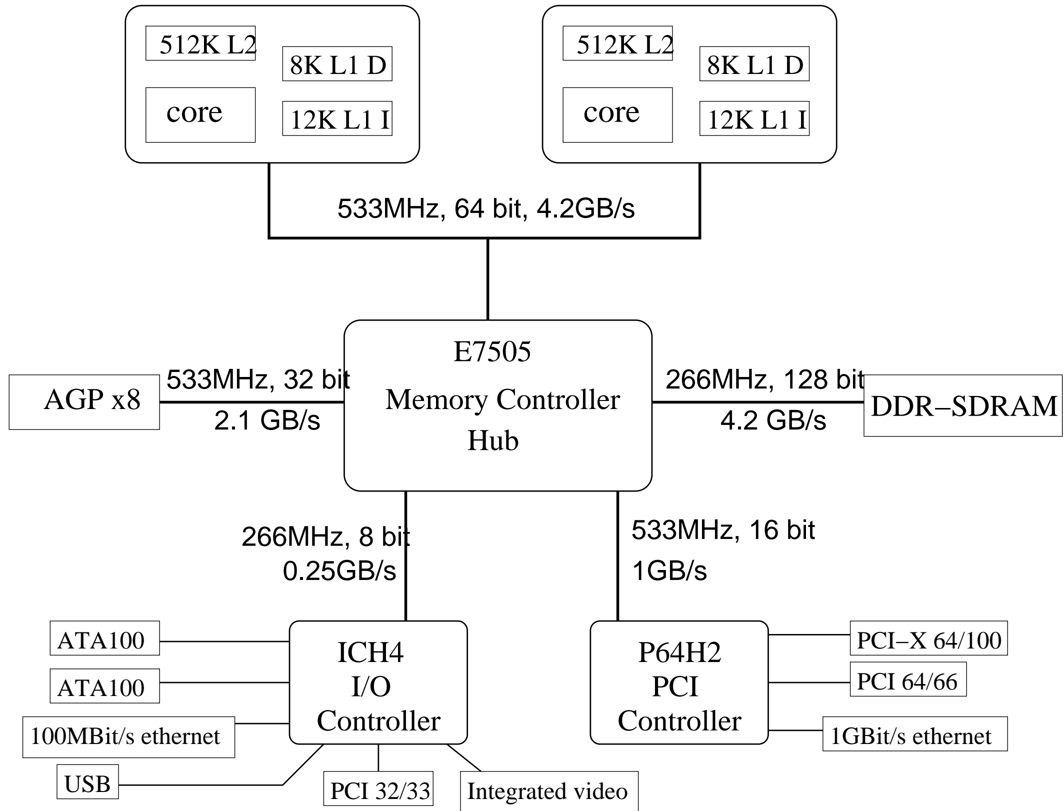
Dual Cores

The other approach to increasing the throughput of a single physical CPU is to place two (or more) complete CPUs on a single chip. This produces, instantly, the deprecated arrangement of two CPUs with shared memory bus discussed above.

There are many approaches to this, usually involving separate level 1 caches for each core, but possibly shared caches for level 2 (or level 3).

IBM's Power4 and Sun's UltraSPARC IV are already dual core, and many CPUs are expected to follow during 2005/6 (Opteron, Itanium, Pentium 4, PPC).

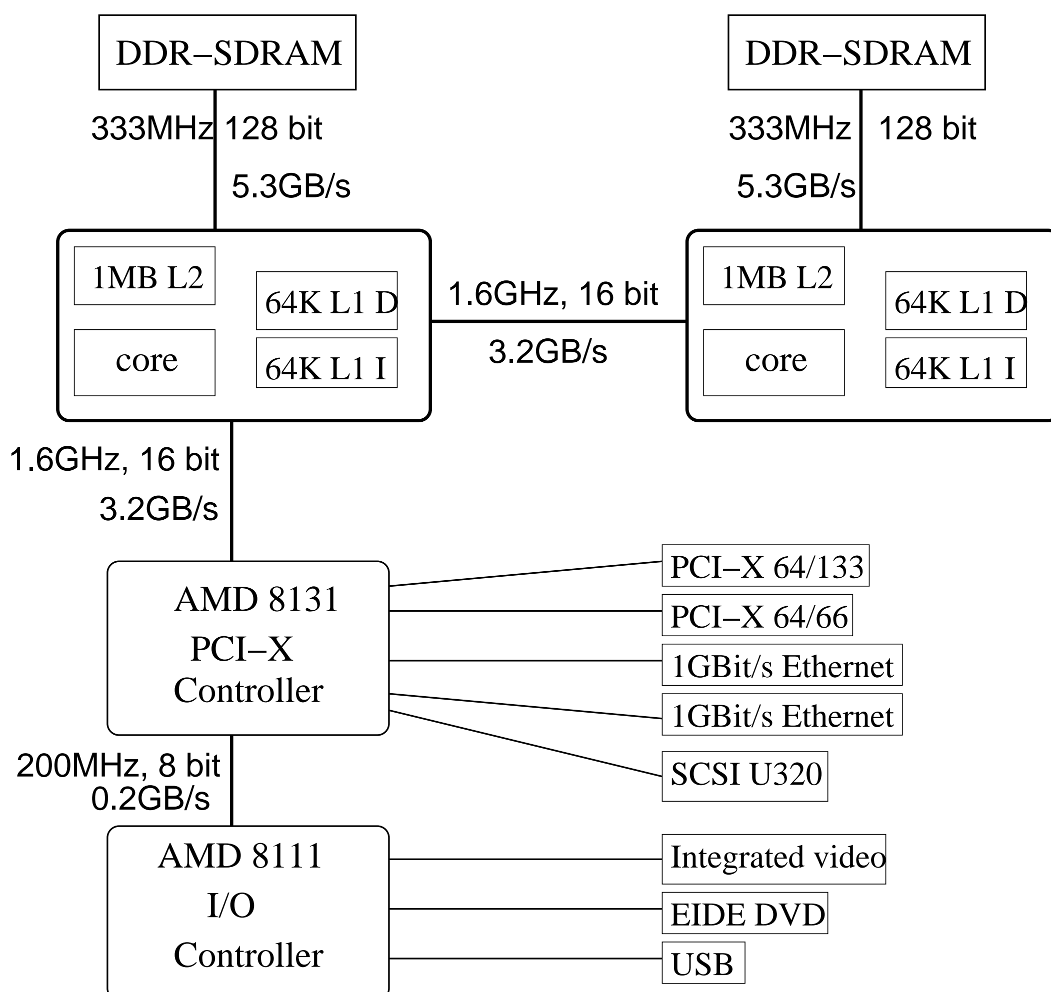
Dual Xeon



This shows the approximate architecture of a dual Pentium4 machine using Intel's 7505 chipset. The two processors, with their shared bus, are at the top.

For much of the life of this chipset, the corresponding uniprocessor chipset, the 875, used an 800MHz CPU bus, and a 400MHz, 128 bit bus to its DDR memory. Thus the single CPU board had greater memory bandwidth than both CPUs together on the dual board.

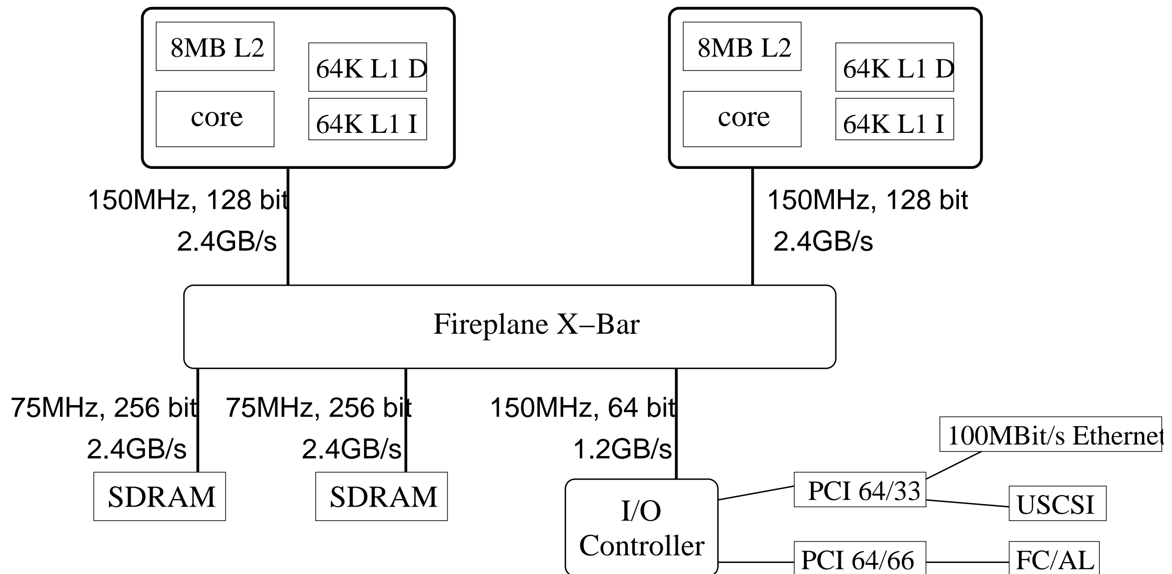
Dual Opteron



This diagram reflects the architecture of a dual Opteron machine, such as Sun's V20z. Notice that this is now a NUMA architecture, and that the total memory bandwidth is proportional to the number of CPUs.

Some dual Opteron machines, and most recent Althlon64 machines, run their memory at 400MHz.

Dual UltraSPARC III



This diagram reflects the architecture of a dual UltraSPARC III machine, such as Sun's 280R. This is a more expensive design than the preceeding two, but it avoids the problems of NUMA whilst keeping up the memory bandwidth. The (more expensive) DS25 from HP/Compaq is similar but faster, running its two 256 bit memory banks at 125MHz, and its CPU buses, whilst just 64 bits wide, run at 500MHz.

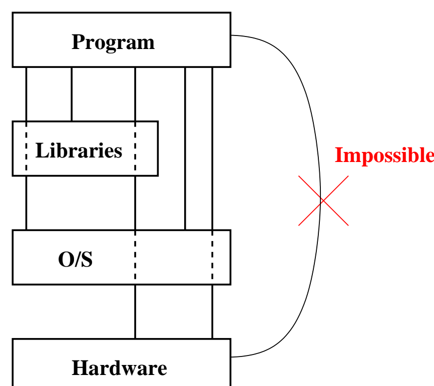
Programming

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`, 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))
`mkl`: Maths Kernel Library (Intel)

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.

Hello, World

The idea of a first example program being one to print the text “hello, world” is mainly due to Kernighan and Ritchie’s book “The C Programming Language.”

This section considers many ways of writing such a program, and, so that it is as clear as possible what is really happening, most of the examples are in assembler. The first does not even make use of the operating system to do more than act as a program loader.

!!WARNING!! Some of the examples in this section work only on very specific OS versions, although the concepts are much more general.

Direct Hardware Access

On an IBM PC, the default text video mode is 80 columns by 25 lines. The video memory is mapped starting at address 0xB8000 (top left of screen), with alternate bytes being the ASCII(-ish) representation of the character, and an attribute byte which specifies the colour.

MS DOS will execute a program with the extension `.COM` by loading it at an offset of 0x100 in a segment, setting all the segment registers to point to that segment, and starting to execute from address 0x100. Such a program may exit with the instruction `ret`.

As the screen usually scrolls by one line immediately after a command finishes, we shall print the string on the second line.

Thus `.COM` files give a simple, flat, 64K address space which contains all the text, data and stack. The stack is set up to grow down from the highest possible address in the segment, 0xffff.

Hello, World (1)

```
section .text
org 0x100

  mov ax,0xB800
  mov es,ax
  mov di,160

  lea si,[string]
  mov cx,12

next_ch:  movsb
  inc di
  loop next_ch

  ret

string db "Hello, World"
```

Hello, World (1)

Tell the assember that this is the text segment, and it starts at 0x100.

The address we wish to write the string to is 0xB800:160 – set this up in `es:di`. N.B. horrible mix of hex and decimal in that address!

Point `si` at the start of our string, and put the number of characters in `cx`.

The `movsb` instruction is a horrible CISCy thing. It reads a byte from `ds:si`, writes it to `es:di`, and adds one to both `si` and `di`.

We need to skip the attribute bytes in the video memory, so `di` is incremented again.

Finally `loop` is another CISCy thing. It decrements `cx`, and jumps to the label given if `cx` is not zero.

No instruction of the form `mov es,0xB800` exists.

Hello, World(1)

The above can be assembled (the syntax is NASM's) to give a remarkably short .COM file: just 32 bytes.

A disassembler interprets the resulting file as follows

```
D:\MJR\ASM\NASM>debug hello1.com
-u
0C80:0100 B800B8        MOV     AX,B800
0C80:0103 8EC0          MOV     ES,AX
0C80:0105 BFA000        MOV     DI,00A0
0C80:0108 8D361401      LEA     SI,[0114]
0C80:010C B90C00        MOV     CX,000C
0C80:010F A4            MOVSB
0C80:0110 47            INC     DI
0C80:0111 E2FC          LOOP    010F
0C80:0113 C3            RET
```

The second column is the binary representation of each instruction.

Note the variable instruction lengths, one to four bytes here, and the backward (little-endian) nature of the storage of immediate data: A000 for 00A0, 1401 for 0114, 0C00 for 000C, etc. In the loop instruction, the jump is -4 bytes, as the next instruction will be 10F (again) not 113. Converting -4 to two's complement, one gets FC. Label names have been lost – there is no occurance of 'next_ch' or 'string' in the .COM file.

Calling DOS

The above code has several disadvantages. It works in just one video mode. It always writes at the same location on the screen, regardless of what was there. It requires precise knowledge of the hardware. Its output does not obey the normal redirections ('>' and '|').

DOS provides a function for writing a string to the terminal, which works in whichever video mode is in use, which writes at the current cursor position, and which does obey redirections. DOS is called via a set of interrupt functions, normally interrupt number 0x21. The arguments to the function are passed in the CPU's registers. Most importantly, the `ah` register specifies which function one requires.

Function 9 prints a string from the address in `dx`. The string must be terminated by a '`$`'.

Function 0x4C exits, returning the contents of `al` as the exit code.

Hello, World (2)

```
section .text
org 0x100

  lea dx,[string]
  mov ah,9
  int 0x21

  mov ax,0x4C00
  int 0x21

string db "Hello, World$"
```

Now a mere 26 bytes!

Assembers often disagree about syntax. Some will insist that register names are prefixed by a %, and there are many conventions for labels.

Real Operating Systems

A real operating system would not allow direct hardware access as used in the first example above (indeed, in the presence of virtual addressing, the first example is nonsensical). It would insist on a coding style like the second.

However, like DOS it is called via an interrupt instruction, and again the required function and its arguments are placed in the CPU registers. Unlike DOS, the interrupt is always number 0x80.

Being C-based, UNIX tends to have functions similar to some of the C functions. Two of interest here are `write()` and `exit()`. In Linux these are function numbers four and one respectively.

Linux uses a more structured form for its binary files, called ELF.

Hello, Linux World

```
section .text
global _start
_start
  mov eax,4      ; NR_write is 4
  mov ebx,1      ; unit 1 is stdout
  lea ecx,[msg] ; pointer to message
  mov edx,13     ; length of message
  int 0x80
  mov eax,1      ; NR_exit is 1
  mov ebx,0      ; exit code of zero
  int 0x80

  msg db "Hello, World",10
```

This can be assembled with

```
> nasm -f elf  hello.asm
> ld hello.o
> ./a.out
Hello, World
```

Here `ld` is not linking, merely adding ELF magic to the object file. It likes the global symbol `_start` to specify where execution should commence.

Unportable

Those used to Gnu's assembler will be looking at the above code in disbelief. It would like:

```
.section .text
.global _start
_start:
   movl $4,%eax           # NR_write is 4
   movl $1,%ebx           # unit 1 is stdout
   lea msg,%ecx           # pointer to message
   movl $12,%edx          # length of message
   int $0x80
   movl $1,%eax           # NR_exit is 1
   movl $0,%ebx           # exit code of zero
   int $0x80

   msg:
     .ascii "Hello World\n"
```

Note the dollars before constants, %s before register names, the reversal of the order of the operands, and the change of the comment character.

The syntax used by NASM is that defined by Intel. That used by Gnu's assembler (gas) is defined by AT&T.

It's big!

The resulting executable is huge, about 770 bytes!

After using `strip` to remove the symbol table (for the label '`msg`' does now appear in the final executable) it is still 440 bytes.

Unlike DOS's `.COM` files, which contain just the raw code, Linux's ELF files (and even DOS's `.EXE` files) contain some degree of structure. The actual text segment is just 48 bytes, but the executable contains lengthy section headers and an overall header.

The overhead of the ELF format is not significant for a normal-sized program, and it does allow the file to be more precisely identified. This one firmly claims to be an i386 UNIX executable, for instance.

The text segment has grown slightly as all those constants which were 16 bits (two bytes) in DOS are now 32 bits.

The inclusion of an empty .bss section adds 44 bytes to the file size, and a .comment section giving the version of the assembler 80 bytes. The ELF header is 52 bytes, each section header a further 32 bytes. . .

For those suspicious of what `ld` is doing.

```
BITS 32
org  0x08048000
  db      0x7F,"ELF",1,1,1,0 ; magic
  dd      0,0                ; magic cont
  dw      2                  ; executable
  dw      3                  ; i386
  dd      1                  ; version
  dd      _start             ; entry point
  dd      52                 ; program header offset
  dd      0                  ; section header offset
  dd      0                  ; flags
  dw      52                 ; elf header size
  dw      32                 ; program header size
  dw      1                  ; number of program headers
  dw      0                  ; section header size
  dw      0                  ; number of section headers
  dw      0                  ; index of string table

  dd      1             ; loadable data
  dd      0             ; offset from start of file
  dd      $$            ; virtual address to load at
  dd      $$            ; ignored
  dd      filesize      ; size on disk
  dd      filesize      ; size in memory
  dd      5             ; memory segment perms (r-x)
  dd      0x1000        ; ignored

_start:
  mov eax,4     ; NR_write is 4
  mov ebx,1     ; unit 1 is stdout
  lea ecx,[msg] ; pointer to message
  mov edx,13    ; length of message
  int 0x80
  mov eax,1     ; NR_exit is 1
  mov ebx,0
  int 0x80
msg db "Hello, World",10

filesize      equ     $ - $$
```

This will assemble with

```
> nasm -f bin -o a.out hello.elf.asm; chmod +x a.out
```

and the resulting 132 byte executable can be run directly.

The `int` in detail

In the DOS example, we chose to call DOS via the conventional `int 0x21` call. However, the DOS 'kernel' ran with the same privileges as our own code, and we could have jumped into it by any route. Executing `int 0x21` merely places a return address on the stack, and jumps to the address given by entry number 0x21 in the interrupt vector table, which, for the 8086, starts at address zero, occupies the first 1K of memory, and is easily read or modified.

In Linux, `int 0x80` is rather different. The address it refers to is not modifiable by the user code, and when it is executed, a flag in the CPU is immediately set to indicate that the CPU is executing kernel code. When this flag is set, direct hardware access is possible. The flag gets reset as the execution returns to the original program. Any attempt to execute the privileged instructions which the kernel uses without this flag set will be denied by the CPU itself. There is a very clear distinction between 'kernel space' and 'user space'.

The 8086 interrupt table has 256 four-byte (segment then offset) entries.

The IA32 processors have several modes of operation. The default, used by DOS, has no concept of privileged instructions – everything is always acceptable. The mode which Linux (and WinNT and OS/2) use does enforce different privilege levels.

Using libraries

As a first example of using a library, we shall convert the above Linux code to call the `write()` and `_exit()` functions from libc, rather than using the kernel interface directly.

The most important UNIX library, `libc`, contains all the (non-maths) functions required by ANSI C and any extensions supported by the platform, as well as C wrapper to all kernel calls. Thus some of its fucntions, such as `strlen()`, do not call the kernel at all, some, such as `printf()` do considerable work before calling a more basic kernel function and others, such as `write()`, are trivial wrappers for kernel functions.

The last category is traditionally documented in section 2 of the manual pages, whereas the others are in section 3.

Some C functions call kernel functions occasionally, such as `malloc()`, which needs to provide any amount of memory that the program requests, but can only request memory from the kernel in multiples of the page size (typically 4K or 8K).

Using libraries: 2

Several changes are necessary to our code. The symbols `write` and `_exit` need to be declared to the assembler as *external* – they are not defined in the code given, and the linker will insert the relevant code.

The routines in libc can be invoked using the `call` instruction, but they do not expect their arguments to be in registers, but rather on the stack. The stack is a downwards-growing area of scratch memory whose next free address is given by the register `sp`. An instruction such as `push eax` puts a copy of the value in `eax` on the stack, and subtracts four from `esp` (as four bytes are needed to store the value in `eax`).

The `call` instruction also uses the stack for storing the *return address* – the address to return to after the function exits.

Hello, libc World

```
section .text
extern write
extern _exit
global _start
_start:
  mov eax,13    ; length of message
  push eax
  lea eax,[msg] ; pointer to message
  push eax
  mov eax,1     ; unit 1 is stdout
  push eax
  call write    ; write(fd,*buff,count)
  mov eax,0
  push eax
  call _exit    ; _exit(status)

  msg db "Hello, World",10
```

which will need to be linked with a line such as

```
> ld -static hello.libc.o -lc
```

The Linker

The linker searched for a file called 'libc.a', and looked in it for the symbols write and _exit. It has extracted the associated code and data, and any other parts of libc which those routines require, from the 16MB lump which is libc on the machine used, and added them to the code in the object file provided. The text sections are joined together, and likewise the data sections and comment sections. The final executable is still just 920 bytes long, of which 356 bytes is text.

.comment from library
.comment from object
.bss from library
.bss from object file
.text from library
.text from object file
ELF Headers

Relocation

It should be clear that none of the sections taken from the library will end up at any particular address. Their destination will depend on the size of the user-supplied program, and which other library functions have been included.

The linker performs the task of relocating the code, 'fixing' any absolute addresses within the library code (text and data) as required.

The virtual addresses at which the text, initialised data and uninitialised data (the fixed-szed segments) will be loaded is fixed at link time.

Moving to C

A C programmer would probably use the function `printf()` rather than `write()` in a ‘Hello World’ program.

```
section .text
extern printf
extern _exit
global _start
_start
  lea eax,[msg] ; pointer to message
  push eax
  call printf   ; printf(*buff)
  mov eax,0
  push eax
  call _exit    ; _exit(status)

msg db "Hello, World",10,0
```

Note that our string is now null-terminated, as a C library function would expect.

It’s HUGE

The resulting executable, even when stripped, is now over 370K.

The `printf()` function can do rather a lot. It can print integers of varying length in base 8, 10 and 16, and single or double precision floating point numbers in decimal or exponential notation with a specified precision, as well as strings. It can pad in various fashions, and returns the number of characters printed. It can parse widths and precisions from a format string or from its arguments. It can do all that Fortran’s `write` and `format` statements can, and more.

All this, just to print a short string. The price we have paid is about 370K of code in our executable which will not be used, but would have been used if the parameters passed to `printf` had been different.

Dynamic linking

Rather than store such large amounts of code in the executable, and needing to place the same in memory at run time, Linux, like most other OSes, can use *dynamic libraries* which are linked at run-time, and which need only be resident in memory once each. Linking dynamically reduces the executable size back to 1.4K.

When linking dynamically, the linker checks that all symbols can be resolved using the specified libraries, and it specifies the loader to be used at run-time to relink.

The run-time linking need not link against precisely the same libraries that were used at compile time – it may not even occur on the same computer. One hopes that the libraries found at run-time are compatible with those used at link time.

The dynamic link line is

```
> ld --dynamic-linker=/lib/ld-linux.so.2 hello.o -lc
```

Dynamic Advantages

Dynamic linking has many advantages. If one dynamically links against a maths library, one might use different versions at run-time depending on the precise processor in the computer – a Pentium 4 optimised library, or a Pentium M optimised library, for instance. One could also upgrade the maths library, and all applications dynamically linked against it immediately benefit from the upgrade with no need to recompile or relink.

A statically linked program interfaces directly with the kernel. In UNIX, usually the only library to do so is libc. If libc is dynamically linked, then the kernel interfaces can change and applications will still work if the libc interface is unchanged.

Dynamic Disadvantages

Sometimes it is not an advantage for programs to behave differently on different computers because they are picking up different libraries: it can make debugging much more interesting.

There is also a slight overhead on every function call to a dynamic library compared to a statically linked library. For trivial functions, this can result in a few percent performance loss. At link time, the address at which the dynamic library will be loaded at run time is not known. Any references to variables or functions in it are made via a look-up table. The run-time dynamic linker populates this look-up table, and places its address in a register. The code is compiled to perform lookups into this Global Offset Table relative to the pointer it is passed.

So referencing a dynamic symbol involves an extra indirection via the GOT, and the presence of the GOT wastes one of our few integer registers, (`ebx`).

However, many OSes are moving away from permitting wholly statically linked binaries.

Yet closer to C

A C program consists of a function called `main`. In order to make our assembler code the equivalent of the following C

```
#include<stdio.h>

int main(){
  printf("Hello, World\n");
  return(0);
}
```

we must define a function called `main()` and return a value from it. As the return value is simply passed back in the `eax` register, and the return instruction is `ret`, this is easily done.

The C version

```
section .text
extern printf
global main
main
  lea eax,[msg] ; pointer to message
  push eax
  call printf   ; printf(*buff)
  pop eax
  mov eax,0
  ret

msg db "Hello, World",10,0
```

This is best converted to an executable with

```
> nasm -f elf  hello.c.asm
> gcc hello.c.o
```

The gcc effect

The innocent line
```
> gcc hello.c.o
```
is equivalent to something like

```
> ld --dynamic-linker=/lib/ld-linux.so.2 \
  crt1.o crti.o crtbegin.o \
  hello.c.o \
  -lc crtend.o crtn.o
```

The five extra object files contain the magic `_start` symbol and the code required for then calling `main()` with any command-line arguments as C expects, then calling `_exit()` with the return code from `main()`. The C library (`libc`) is linked implicitly, and the dynamic linker added too.

The five extra object files should have full paths specified. Try
```
> gcc -v hello.c.o
```
to see the full detail.

If you think this is bad, look at what a Fortran compiler does!

The Stack

Because the `ret` instruction takes the return address from the stack, it is important that we keep careful track of what has been placed on the stack.

One value was placed their in order to call `printf()`, and that function will leave the stack unchanged when it exits. The calling program must clear up the stack, either with an instruction like

```
pop eax
```

which then gets discarded, or with something like

```
add esp,4
```

which is more efficient if there are many arguments / values to remove.

This program does not attempt to maintain the frame pointer (`ebp`) correctly. Usually the frame pointer should point to the top of the current stack frame, and the stack pointer to the bottom.

The Stack Frame

Address	Contents	Frame Owner
	. . .	calling
	2nd argument	function
ebp+8	1st argument	
ebp+4	return address	
ebp	previous ebp	
	local	current
	variables	function
	etc.	
esp	end of stack	

The Alpha uses register $30 as the stack pointer, $26 for the return address, and $15 for the frame pointer only if the frame is of variable size.

This use of the stack and frame pointers makes it easy for debuggers etc. to 'unwind' their way backwards up a call tree.

Using the Frame Pointer

At the start of the function, the value of the frame pointer, `ebp`, should be saved using

```
push ebp
mov ebp,esp
```

If room is required for local variables on the stack, one can simply subtract the required amount from `esp`.

At the end of the function, `esp` and `ebp` can be restored with

```
mov esp,ebp
pop ebp
```

assuming that `ebp` has not been modified by the function. This exit sequence can be done in a single, one-byte instruction, `leave`.

The C compiler

The C compiler operates in several distinct phases when executed as

```
> cc hello.c
```

First it preprocesses the source to a file in /tmp.
Second, it converts the C to assembler: the compilation stage. This will probably involve an intermediate language internally, and produces another file in /tmp.
Third, it assembles the result to an object file (in /tmp).
Finally, it links the object file.

The four stages are often done by four completely separate programs: cpp, cc, as and ld.

Fortran is similar, except it is usually not preprocessed.

Optimisation is performed both when converting to the intermediate language (front-end), and when converting that to assembler (back-end).

Why use an intermediate language? Consider maintaining a collection of compilers to convert three languages (e.g. C, C++, Fortran) for four different CPU families (IA32, x86-64, SPARC, PPC). Without an intermediate language that is twelve different compilers, with it is just three front-ends, and four back-ends.

C again

The compilation sequence can be stopped at any stage. If we stop at the assembler stage:

```
> cc -S hello.c
> less hello.s
        .section        .rodata
.LC0:
        .string "Hello, World"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $.LC0
        call    printf
        xorl    %eax,%eax
        leave
        ret
```

One may need to add '`-Os -fno-builtin`' to get the above result. . .

`xorl %eax,%eax` sets `%eax` to zero.

Nasty C

The statement

```
#include<stdio.h>
```

causes the file `/usr/include/stdio.h` to be inserted at that point by the preprocessor. Its purpose is to give a prototype for the `printf` function.

The resulting source file is increased from six lines to over 770 lines, as the header files define many other functions and constants. Those who would prefer to understand what is happening might prefer the code as

```
int printf (const char *fmt, ...);

int main(){
  printf("Hello, World\n");
  return(0);
}
```

Watching the action

Most versions of UNIX have a program which allows one to display system calls as they happen:

```
> strace ./a.out
execve("./a.out", ["./a.out"], [/* 22 vars */]) = 0
write(1, "Hello, World\n", 13Hello, World
)          = 13
_exit(0)                                 = ?
```

Note the system call was displayed, with arguments, followed by its return code after the '=' (13 for `write()`). The strace utility has intercepted both the entry to and exit from the kernel.

Unfortunately its output gets mixed with the program's own output.

More action

The above was the statically linked program from page 271. Considering the code from page 274 after dynamic linking, one can check which libraries it will load at run-time:

```
> ldd a.out
 linux-gate.so.1 =>  (0xffffe000)
 libc.so.6 => /lib/tls/libc.so.6 (0x4001f000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Here `ld-linux.so` is the helper program which actually performs the dynamic linking, and `libc.so` is the expected shared library. The `linux-gate` object is a consequence of version 2.6 of the Linux kernel which we shall ignore.

The `ldd` command is also found in Tru64 version 5 (and later, not version 4), and Solaris, amongst others.

All action

```
> strace ./a.out
execve("./a.out", ["./a.out"], [/* 21 vars */]) = 0
uname({sys="Linux", node="tcmpc34.phy.cam.ac.uk", ...}) = 0
brk(0)                                  = 0x804a000
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
             MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40017000
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)   = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=26693, ...}) = 0
old_mmap(NULL, 26693, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1348627, ...}) = 0
old_mmap(NULL, 1132940, PROT_READ|PROT_EXEC,
                            MAP_PRIVATE, 3, 0) = 0x4001f000
madvise(0x4001f000, 1132940, MADV_SEQUENTIAL|0x1) = 0
old_mmap(0x40129000, 32768, PROT_READ|PROT_WRITE,
           MAP_PRIVATE|MAP_FIXED, 3, 0x10a000) = 0x40129000
old_mmap(0x40131000, 10636, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40131000
close(3)                                   = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40018000
write(1, "Hello, World\n", 13Hello, World
)          = 13
exit_group(0)                              = ?
```

We don't see `ld-linux` itself loading, but we do see its actions in detail. It tries to open `/etc/ld.so.preload`, fails, then successfully opens `/etc/ld.so.cache`, which it then reads using `mmap`. It opens `libc.so.6` and reads the first 512 bytes. This includes enough header information that it then knows to `mmap` the first 1132940 bytes as readable and executable, so this must be `libc`'s text segment, and then 32K from `libc` as read/write but not executable – the data segment. Finally just over 10K is `mmaped` as uninitialised data.

Then the code really starts, with `printf()` appearing to require an `fstat()` and the reservation of an extra 4K of memory, as well as the expected `write()`.

Useful `strace`

One of the greatest uses of `strace` is in working out what configuration files a program is reading, or which one is causing it to die. The above output is rather verbose, but one can request that a subset of calls be traced, e.g.

```
> strace -e trace=file ./a.out
```

is often sufficient.

Output from `strace` can be redirected to a file using `-o`.

On TCM's Tru64 machines, `strace` is spelt with a capital 'S'. The other name is already taken by a less useful Tru64 utility.

There is no firm standard on what an strace-like utility should be called. The equivalent program on Solaris is called '`truss`' and on '`par`' on Irix. All have subtly different options.

Debugging

Two optional parts of an executable greatly assist debugging. The first gives the names of functions and variables. The second gives information about the mapping between line numbers in the source file and instructions in the text segment.

Neither is required for the program to run. Neither will be mapped into memory when the program is run. Both can be removed by using the '`strip`' command.

A debugger also usually requires that the frame pointer is maintained in the conventional fashion. It is possible to omit the frame pointer (`-fomit-frame-pointer` for gcc) but this tends to upset debuggers.

Debugging information is usually turned on by compiling with the -g flag, and it traditionally turns off all optimisation. Full optimisation with full debugging can be hard, as instructions may be reordered so that their correspondance to lines in the source code becomes hard to interpret.

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.

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.

Calling Compilers

Almost all UNIX commands never care about file names or extensions.

Compilers are very different. They do care greatly about file names, and they often use a strict left to right ordering of options.

Extension	File type
.a	static library
.c	C
.cc	C++
.cxx	C++
.C	C++
.f	Fixed format Fortran
.F	ditto, preprocess with cpp
.f90	Free format Fortran
.F90	ditto, preprocess with cpp
.i	C, do not preprocess
.o	object file
.s	assembler file

Consistency

It is usual to compile large programs by first compiling each separate source file to an object file, and then linking them together.

One must ensure that one's compilation options are consistent. In particular, one cannot compile some files in 32 bit mode, and others in 64 bit mode. It may not be possible to mix compilers either: certainly on our Linux machines one cannot link together things compiled with NAG's f95 compiler and Intel's ifc compiler.

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.

So that I am not accused of bias,

`http://www.tcm.phy.cam.ac.uk/~mjr/C/`

discusses why C is even worse. . .

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=3.141592653589793d0
```

(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.

Mind your language

There are many, many computer languages, and one must often choose an appropriate one, and use it in an appropriate fashion.

Of course one needs to balance one's own time against computer time, and everyone else's too. There is no need to spend as long worrying about something which is likely to take 5s of CPU as something which may take 5 years. Nor should one worry as much about a twenty line program for one's own use, as a thousand lines which you intend to pass on to other people.

Java: excellent for GUI interfaces, but slow and poor numerics.

Perl: excellent for string handling, but very slow for anything else.

C: general purpose language with poor optimisation potention.

C++: it will take you years to learn it.

FORTRAN: excellent numerical language, readily optimised if one ignores most of F90.

Timeo Danaos et Dona Ferentes

Beware of Geeks bearing gifts.

It would be nice to assume that all code you are given represents bug-free, robust, well documented, exemplary specimens of best practice. However, this is unlikely. Passing on trash is not helpful though: don't do it.

If you end up writing and passing on a thousand lines of code in the course of your work, then, whether you wanted to be or not, you are a professional programmer, and you need to show some sort of professional standards. Otherwise people may conclude that the sloppiness in your code infects your science too.

Comments, Indentation, Structure

Comments

Use them: they will help you if you need to read the code in a couple of years time!

Indentation

Again, should be used, in a consistent style.

Large code?

Consider the use of `make` for compilation, and `CVS` for version control. Try to keep individual source files and routines short.

Take care with `make`, for there are many implimentations and few standards. Either try a very basic subset of the syntax understood by all, or, perhaps, Gnu's `make`.

Numerics

What does your language do to the following two examples?

Example One

```
double x
integer i

x=10e20
i=x
```

Example Two

```
double x

x=-1
x=sqrt(x)

if (x<10) error "x too small"
```

Answers

Out of range conversion to integer

C: undefined. Java: `max_int`. Fortran: undefined

Square root of -1

C: implimentation defined, `errno` set. Java: NaN, comparison will be false. Fortran: undefined

Where the result is undefined above, the program may halt. Where it is defined, the program may not. . .

Standards

```
      integer i,c

      c=77
      do 10, i=1,0
        c=66
10    continue
c=90
      write(*,*)' Fortran version might be ',c
      end
```

Make sure that you, your compiler, and your co-workers, agree on what standard you are following.

In translation

```
#include<stdio.h>

int main(){
  float x=1./131072.;
  int i;
  unsigned j;
  long l;

  x=(1+x)*(1+x)-1-2*x;
  if (x>1e-12) printf("K&R style arithmetic\n");

  i=2//**/
     -2;
  if (i==-1) printf("K&R or C89 style comments\n");
  else printf("C++ or C99 style comments\n");

  if (sizeof('x')==1) printf("C++ compiler\n");
  else printf("C compiler\n");

  l=-1; j=1;
  if (j<l) printf("long is no longer than int\n");
  else printf("long is longer than int\n");

  return(0);
}
```


Yes, these are contrived examples in order to fit in a few lines. There are other surprises lurking though.

Know your language

When one starts using a computer language, one merely copies or modifies existing code, mostly following one's intuition, and relying on the compiler to point out where intuition failed. This is all perfectly fine.

When one needs to program more professionally, either because one is passing one's code on to other people, or because one is relying on substantial amounts of one's own code for one's work, it becomes necessary to have a more formal understanding of the language. It may be tedious to read a book on Fortran (or, worse, C), but it will stop people laughing at your code (and you), or cursing your code (and you) as much as otherwise would be the case.

Four Mistakes in ANSI89 C

```
#include<stdio.h>
#include<string.h>

int main(void){
  char message[13]="hello, world\n";
  char *hello="hello, world\n";
  int i=0,j,offset=0;

/* Print message in zig-zag */

  for(;++i<50;){
    offset=(++offset)%16;
    for(j=0;j<offset;j++) printf(" ");
    printf(message);
  }

  printf(hello);
  strcpy(hello,"Bonjour\n");
  printf(hello);
}
```

Four errors in F90

```
program integrate
! Integrate Gaussian from -inf to +inf
  use nag_f77_d_chapter
  implicit none
  integer, parameter :: lw=1000, liw=lw/4
  integer :: inf, ifail, iwrk(liw),i
  real(8) :: b,result,err, &
              epsabs,epsrel,wrk(lw)
  inf=2 ; b=0 ; ifail=1
  epsabs=1d-4 ; epsrel=1d-8

  call d01amf(gaussian,b,inf,epsabs, &
            epsrel,result,err,wrk,lw, &
            iwrk,liw,ifail)

  write(6,*)'Integral is ',result
  write(6,*)'sqrt(pi) is ',sqrt(2*asin(1.0))
contains
  function gaussian(x)
    real(8) :: gaussian, x
    gaussian=exp(-x**2)
  end function
end
```

Answers

Corrections to the following gratefully received.

C

Declarations are correct.

`offset=(++offset)%16;` – `offset` modified twice with no intervening sequence point.

`printf(message);` – `message` is not null terminated (although it would have been if the declaration had been `char message[14]=...`, and `message[12]` would have been an error (initialiser longer than array being initialised)).

`strcpy(hello,"Bonjour\n");` – attempts to modify read-only string.

`}` – no return value for `int` function.

F90

`real(8)` – compiler-specific. Should be `real (kind(1.0d0))`

`call d01amf` – passes name of internal function `gaussian`.

`write(6,*)` – unit 6 being stdout is compiler-specific.

`sqrt(2*asin(1.0))` – this is all default (i.e. single) precision

Appendix: ASCII codes

00	0x00	^@	32	0x20		64	0x40	@	96	0x60	‘
01	0x01	^A	33	0x21	!	65	0x41	A	97	0x61	a
02	0x02	^B	34	0x22	"	66	0x42	B	98	0x62	b
03	0x03	^C	35	0x23	#	67	0x43	C	99	0x63	c
04	0x04	^D	36	0x24	$	68	0x44	D	100	0x64	d
05	0x05	^E	37	0x25	%	69	0x45	E	101	0x65	e
06	0x06	^F	38	0x26	&	70	0x46	F	102	0x66	f
07	0x07	^G	39	0x27	’	71	0x47	G	103	0x67	g
08	0x08	^H	40	0x28	(	72	0x48	H	104	0x68	h
09	0x09	^I	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	^J	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	^K	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	^L	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	^M	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	^N	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	^O	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	^P	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	^Q	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	^R	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	^S	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	^T	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	^U	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	^V	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	^W	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	^X	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	^Y	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	^Z	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	^[	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	^\	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	^]	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	^^	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	^_	63	0x3F	?	95	0x5F	_	127	0x7F	

For reference, the ‘standard’ 7-bit ASCII map. Codes 0 to 31 are unprintable control codes, and code 127 is delete.

-r8, 318
.COM, 257, 266
.EXE, 266
/proc, 213
0x, 113

8087, 26

address lines, 93, 94
alignment, 127
allocate, 208
allocate on write, 124
Alpha, 23, 25, 148–150
Altivec, 150
Amdahl's law, 217, 232
and, 48
ANSI C, 65
ASCII, 45, 339
assembler, 85
ATE, 122

bandwidth, 97
bandwidth, hard disk, 155
bandwidth, interconnect, 230
big endian, 40
binary, 37
binary fractions, 54
BIOS, 103
bit flip, 104
BLAS, 252, 313
branch, 84, 88
branch prediction, 86
bss, 207
burst, 96, 97
bus, 18
byte, 29, 36

C, 319
cache
    associative, 121
    direct mapped, 118
    disk, 203, 204
    memory, 111
    primary, 126
    secondary, 126
    write back, 123, 124
    write through, 123
cache coherency
    broadcast, 225, 237
    directory, 225, 237
    snoopy, 123, 223, 237
cache controller, 112
cache hierarchy, 126
cache line, 115, 116
cache thrashing, 120
call, 270
cc-NUMA, 237
CD, 157–159
CISC, 81, 143
clock, 18, 98, 101
compiler, 20, 314–318
complex arithmetic, 63, 64
cpp, 285
crossbar, 228, 239
CSE, 296

DAT, 168, 169
data dependency, 79
data segment, 207
debugging, 293, 307, 317, 318
denormals, 57, 62
device file, 178
dirty bit, 123
disk thrashing, 199
distributed memory computer, 226
division
    integer, 46
DLT, 168, 169
DOS, 186–189, 257, 261

DRAM, 91, 92
DTLB, 197
DVD, 160

EBDIC, 45
ECC, 106
EDO, 95–97
EEPROM, 91
EIDE, 153
ELF, 263, 264, 266, 267
EM64T, 147
endian, 40, 41
EPROM, 91
exponent, 50
ext2, 164
ext3, 164

F90, 319–327
Flash RAM, 103
floppy disk, 156
FPM, 95–97
fragmentation, memory, 188
frame pointer, 282–284, 293
`fsck`, 163
function overloading, 255
functional unit, 17, 82

gas, 265
`gcc`, 281
GL, *see* OpenGL

Hamming Code, 106
hard disk, 153–155
Harvard architecture, 125
heap, 207, 208
helical scan, 168
hex, 37, 113
hit rate, 111, 122
HPF, 233
hypercube, 228
Hyperthreading, 145
hyperthreading, 244

IA32, 23, 26–28, 134–147
IBM 370, 60
IEEE 754, 55, 56, 58
in flight instructions, 86
infinity, 58
inlining, 305
instruction, 19
instruction decoder, 17, 80
instruction fetcher, 17
instruction pointer, 24
integers
    negative, 38
    positive, 37
issue rate, 82
ITLB, 197

journalling, 164
jump, *see* branch

K&R C, 65
kernel, 175, 178, 180, 268

language, 20
LAPACK, 252
latency, functional unit, 82
latency, hard disk, 155
latency, interconnect, 230
latency, memory, 97
`ld`, 264, 267, 271–273, 285
`ldd`, 289
`libc`, 269, 270, 281
libraries, 250–254
libraries, shared, 209
`limit`, 208
linking, 255, 317
    dynamic, 276–278
    static, 272, 273, 277

Linpack, 32
little endian, 40
load, 184
locked pages, 200
logistic map, 67
loop
  blocking, 311
  coda, 299
  elimination, 302
  fusion, 303
  interchange, 308
  invariant removal, 296
  pipelining, 306
  reduction, 300
  strength reduction, 304
  transformations, 312
  unrolling, 299

machine code, 20, 29
MacOS, 189
`main()`, 279
`make`, 330
`malloc`, 208, 212, 269
mantissa, 50
memory map
  Digital UNIX, 211
  DOS, 187, 188
  Linux, 212
memory refresh, 92
MESI, 224
metadata, 161–164
MFLOPS, 30
microcode, 61
MIPS, 30
mirror, 165
MLC, 103
MMX, 142
modulo scheduling, *see* loop pipelining
MPI, 227, 234, 235, 241, 242
MPP, 226
MTA, 244
multitasking, 183
  co-operative, 185
  pre-emptive, 185
multithreading, 243, 244

NAG, 252
name mangling, 255
NaN, 58, 59
nasm, 260, 265
`nice`, 184
non-blocking, 230
null pointer dereferencing, 212
NUMA, 237, 238
nybble, 36

offset, 38
OpenMP, 233
operating system, 184, 199, 250
Opteron, 147, 247
optimisation, 294–313
`or`, 48
out-of-order execution, 89
overflow, 43, 58, 59

page, 192
page fault, 194, 199
page table, 192–196
paging, 199
parallel computers, 215
parity, 105
PC ratings, 102
Pentium 4, 246
physical address, 191
PID, 172
pipeline, 78, 82
pipeline depth, 77
platter, 154, 155
`pmap`, 213

predication, 88
prefetching, 128, 129, 301
priority, 184
process, 172
process switch, 183
program counter, *see* instruction pointer
`ps`, 173, 184

quadratic formula, 65

RAID, 166
RAM, 91
RAMBUS, 101
random numbers, 72–75
ranges, IEEE 754, 60
ranges, integer, 44
RDRAM, 101
register, 17, 29
registers, 24–28
`renice`, 184
RISC, 81, 143
ROM, 91
rotate, 47

SATA, 153
scaling, 217, 218, 232
`scandisk`, 163
scheduler, 175, 184
SCSI, 153
SDRAM, 98
SECDED, 106
sector, 155
seek time, 155
segment, 207
segment register, 27
shared memory processor, 219
shift, 47
SIGBUS, 127
SIGFPE, 59
SIGILL, 29
sign-magnitude, 38
SIGSEGV, 194
SIMD, 215
`size`, 207
SMP, 219
SMT, 244
SPARC, 150
SPEC, 33
speculative execution, 87
spin wait, 241
SRAM, 91, 92, 111
SSE, 144
SSE2, 145
stack, 207, 208, 270, 282–284
stack frame, 283
stalls, 86
`strace`, 288, 290–292
streaming, 129
Streams, 31
`strip`, 293
sub-block, cache line, 116, 124
Sun 280R, 248
Sun V20z, 247
superscalar, 80
swap space, 201
swapping, 201

tag, 114, 115, 117–119, 121
tapes, 167, 168
text segment, 207
timeslice, 183
TLB, 197
topology, 228
track, 155
tree, 228
two's complement, 38

ufs, 164
`ulimit`, 208

UltraSPARC III, 248
underflow, 57
`uptime`, 184

vector computers, 109
victim cache, 122
virtual address, 191
virtual address space, 174
virtual disk, 165
virtual memory, 199
VIS, 150
`vmstat`, 203

word, 29

X-bar, *see* crossbar
x86-64, 147
`xor`, 48

zero, 39, 57, 58