

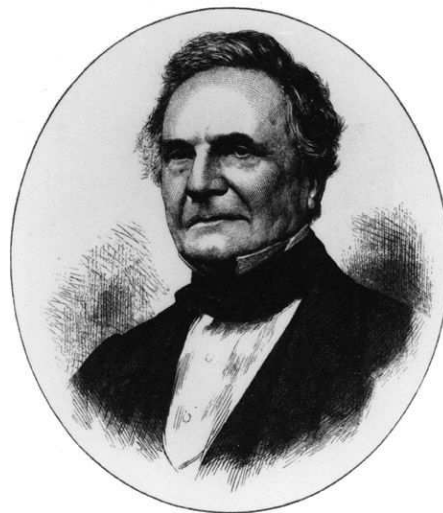
An Introduction to Computing

MJ Rutter
mjr19@cam.ac.uk

Michaelmas 2001

The Beginnings of Digital Computing

All good things come from Cambridge, and computing is no exception. Babbage is generally regarded as the father of modern computers. He was educated at Trinity, and became the Lucasian Professor of Mathematics. More importantly, in 1821 he invented the 'Difference Engine', a mechanical computer for calculating mathematical tables. Although Babbage's version never progressed beyond a prototype, in 1854 George Scheutz constructed a version which saw successful and useful service.



Charles Babbage (1791-1871)

The Beginnings of Programming

The first programmer is also English: Ada Countess Lovelace. Ada, a daughter of Lord Byron, met Babbage in 1834, and wrote an article in 1843 which described how to use computers such as Babbage had invented for solving mathematical problems.



Ada Countess Lovelace (1815-52)

Electronic Computing

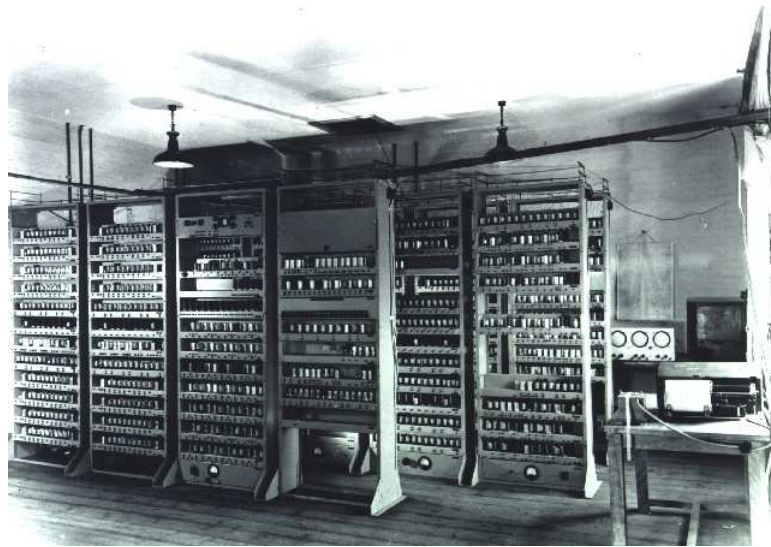
The next major advance in computing was the move from mechanical computers to electronic computers. This occurred at Bletchley Park in 1944 when Colossus was built. Again, many ex-Cambridge people were involved (Turing, Newman, Tutte and others).

Colossus used thermionic valves, not transistors, so was rather large. It was also not programmable: it was built for one specific task (decrypting a certain class of German codes).

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

The Beginning of Computational Science

EDSAC ran its first program in May 1949. It was run by the forerunner of the University's Computing Service, and 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.



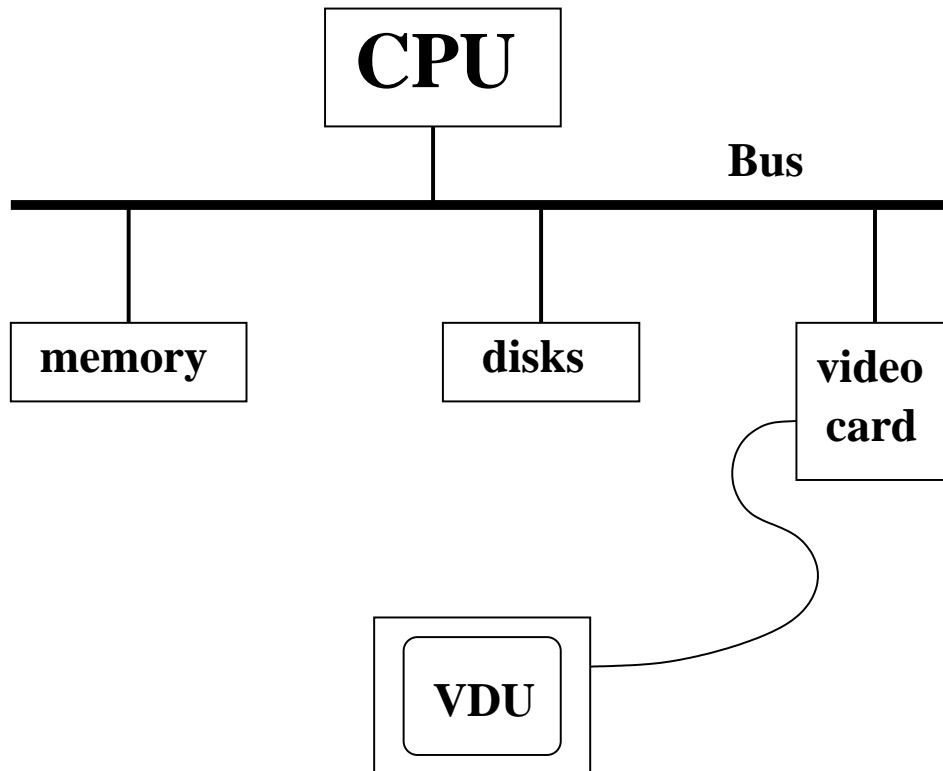
EDSAC-1

Further advances

The 1950s saw the introduction of transistors to computing, and the 1960s the introduction of integrated circuits. Companies such as IBM and DEC were very active at this time.

The 1970s saw the introduction of the VLSI (very large scale integration) chips that all modern computers use, as well as the rise of a company called Intel.

Inside the Computer



Storage and execution

One part of a computer, the CPU, is responsible for actually performing operations. Another, loosely called 'memory', is responsible for storing the data being processed, any intermediate results, and the instructions which the CPU is executing.

Memory has taken many forms over the years: paper tape, punched cards, magnetised ferrite beads, steel wire, magnetic tape, magnetic disks, and semiconductor-based memory. The last two are relevant for the computers used in this course!

Standard semiconductor memory (RAM) costs about £150 for a gigabyte (8×10^9 bits), whereas disk drives are about 20x cheaper. RAM is much faster than disk drives, but is *volatile* – its contents are lost when the computer is turned off.

von Neumann

The Hungarian John von Neumann has his name attached to the class of computer which uses the same memory for both programs and data. This is the form every modern computer takes.



von Neumann (1903–1957)

The Heart of the Computer

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

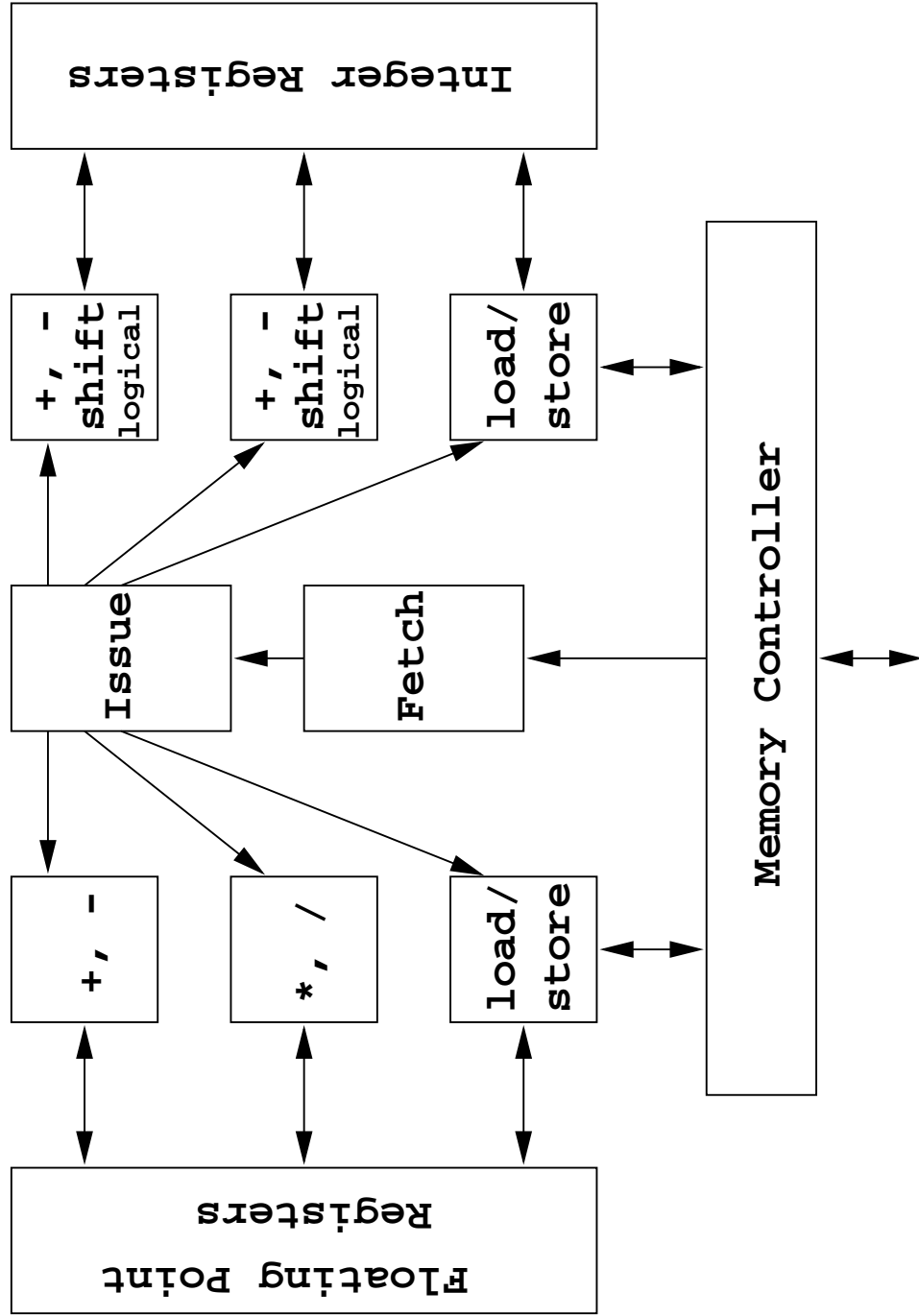
A typical modern CPU understands two main classes of data: integer and floating point. Within those classes it may understand some additional subclasses, such as different precisions.

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

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

Typical instructions

Integer:

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

Floating point:

- arithmetic: $+$, $-$, $*$, $/$, $\sqrt{\quad}$, negate
- comparison
- load / store (copy between register and memory)

Control:

- (conditional) jump

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 send to floating point addition unit
- wait for add to complete
- retrieve result and place in f6

Exact sequence varies from processor to processor.

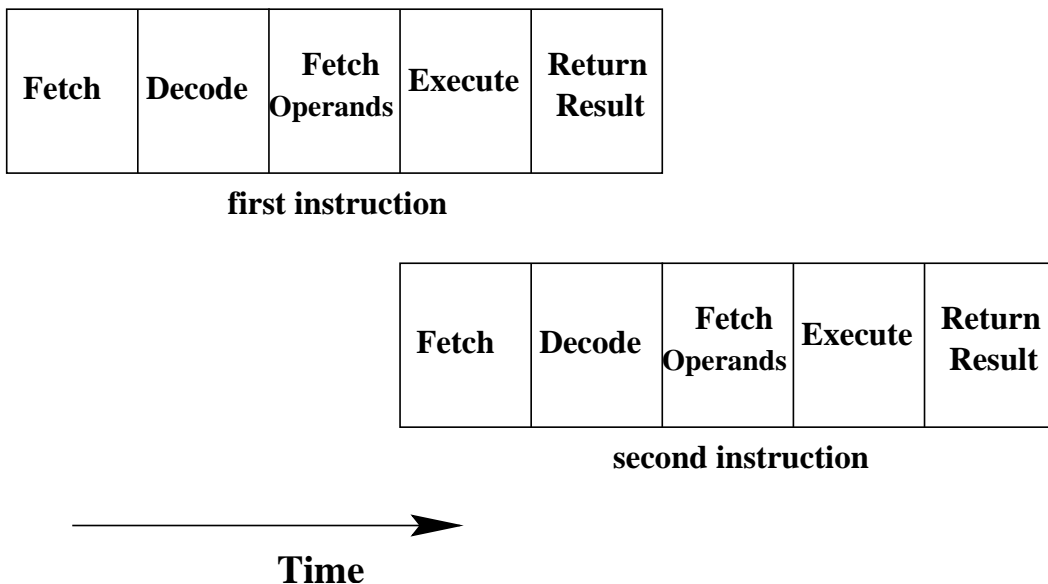
Always a *pipeline* of operations which must be performed sequentially.

The number of *stages* in the pipeline, or *pipeline depth*, can be between about 5 and 10 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 jump 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.

Within a functional unit

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

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

First the exponents are adjusted so that they are equal:

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

only then can the mantissas be added

$$10.01 * 10^5$$

then one may have to readjust the exponent

$$1.001 * 10^6$$

So floating point addition usually takes at least three clock cycles. But the adder may be able to start a new addition every 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.

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	1-2
imul	3-15	3-15
fadd	3	1
fmul	3	1
fdiv	15-25	15-25
fsqrt	15-25	15-25

In general, most things take 1 or 2 clock cycles, except integer multiplication, and floating point division and square root.

Meaningless Indicators of Performance

- MHz: the silliest: some CPUs take 4 clock cycles to perform one operation, others perform four operations in one clock cycle. Only any use when comparing otherwise identical CPUs.
- MIPS: Millions of Instructions Per Second. Theoretical peak speed of decode/issue logic.
- MTOPS: Millions of Theoretical Operations Per Second. Current favourite of the US Government.
- FLOPS: Floating Point Operations Per Second. Theoretical peak issue rate for floating point instructions. Loads and stores usually excluded. Ratio of + to * is usually fixed (often 1 : 1).
- M, G and T FLOPS: 10^6 , 10^9 , 10^{12} FLOPS.

Representing Integers

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

For historical reasons bits are processed in groups of eight, called *bytes*.

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

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

Being Positive

This is tediously simple:

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

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

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

Being Negative

Sign-magnitude

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

Offset

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

One's complement

Reverse all the bits to represent negative numbers.

Two's complement

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

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

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 2's complement gives:

$$5 + (-4) = 1$$

Ranges

bits	unsigned	2's comp.
8	0 to 255	-128 to 127
16	0 to 65535	-32768 to 32767
32	0 to 4294967296	-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
1000000-1011001	A to Z
1100000-1111001	a to z

Punctuation fills in the gaps.

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

Other main mapping is EBDIC, used (mainly) by old IBM mainframes.

Typical timings

For most modern processors the standard integer operations take just one clock cycle, excepting just multiplication and division (if available). Multiplication typically takes around 6 cycles, and division often 30 to 60 cycles.

This leads to the bizarre position that floating point multiplication and division are typically faster than the corresponding integer operations!

CPU Families

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

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

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

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

Common Families

8086 Intel 8086, 80286, and all IA32

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

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

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

Power Power, Power2, Power3

PowerPC 601, 603, 604, 620, 7400

MIPS64 R8000, R10000, R12000

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

Intra-family differences

Within a family differences are usually slight. The instruction set usually stays the same, sometimes with minor additions (e.g. MMX for IA32, AltiVec for PPC), as does the number and size of registers.

More readily changed are things like the number of functional units, the number of instructions which can be issued per clock cycle, and the execution time of instructions. Away from the CPU core, the size of any on-chip cache might change, as might the memory bus speed and width.

Using the start of the IA32 family as an example:

	386	486	Pentium
max issue rate / clk	0.5	1	2
on chip cache	-	8K	8K + 8K
bus width (bits)	32	32	64
core / bus speed	1	1-3	1-3
core speed (MHz)	16-33	25-100	60-200
fadd (clock cycles)	32	8	2

Disk Drives

Disk drives are the main form of non-volatile storage on a modern computer. They are also one of the few moving parts left. Their mechanical nature can make them very slow compared to semiconductor memory. Whereas RAM will take about 100 ns to fulfill a request for any given piece of data it stores, a disk drive might take around 10 ms, for it may need to move its read heads across the surface of the disk.

Once the heads are in place, disk drives are only about one order of magnitude slower than RAM, not five!

Disk drives tend to store data in blocks of 512 bytes.

Files

Files, and their limitations, are a feature of the *operating system*. Disks merely store blocks of data, the operating system interprets some of those blocks as describing the structure of files and directories, and others as being the contents of those files.

Thus the same physical disk drive, placed in different computers running different operating systems, will have different limits on the length of file names, the length of files, the sort of characters allowed in file names, the type of access control associated with a file, and even the precise amount of disk space available for files. One operating system will not, in general, be able to read a disk written by a different OS.

The Operating System

The operating system is a piece of software which has full control of the computer hardware. It has many tasks, including:

- Conjuring a concept of files and directories out of a dumb disk drive.
- Apportioning memory and CPU time between different processes.
- Enforcing access restrictions

DOS, MacOS, Windows and UNIX are all examples of operating systems.

The modern computer

A modern computer can run multiple programs simultaneously, with multiple users using the machine at once too. It can prevent any one user from taking an unfair share of the machine, or from reading data private to another user.

That, at least, is the theory.

Note that 'simultaneously' means only 'simultaneously as perceived by a human.' The computer will actually run one program exclusively for a few ms, then switch to running another.

Note too that most 'home' computers cannot cope with multiple simultaneous users.

However, the system used for this course, Linux, is close to fulfilling these ideals.