Chapter 1 - Introduction

Computer architecture = computer aspects visible to the programmer, that have a direct impact on program execution. E.g. having a multiply instruction.

Computer organisation = operational units that realise the architectural specifications, like hardware details transparent to the programmer. E.g. the memory technology used.

Many computer models have the same architecture, but different organisations, resulting in different prices & performance.

A particular architecture can span many years, with its organisation changing every now and then.

With microcomputers, the relationship between architecture and organisation is very close. Because these small machines don't really need to be generation-to-generation compatible, changes in technology influence both organisation and architecture. E.g. RISC.

Structure = the way in which computer components are interrelated.

Function = how each component in the structure operates.

The computer hierarchy consists of different levels at which structure and function can be examined.

Function
There are four basic functions a computer can perform:

- Data processing
- Data storage
- Data movement
- Control

Structure
There are four main structural components:

- Central Processing Unit (CPU)
- Main memory
- I/O
- System interconnection

There are four main structural components of the CPU:

- Control unit
- Arithmetic & Logic Unit (ALU)
- Registers
- CPU interconnection

Chapter 2 - Computer evolution & performance

The stored-program concept = the idea of facilitating the programming process by storing the program in memory, alongside the data, so that a computer can get its instructions by reading them from memory, and you can alter a program by setting the values of a portion of memory.

John von Neumann began the design of a new stored-program computer, called the IAS computer, which is the prototype of all subsequent general-purpose computers.
All of today's computers have the same general structure and function, and are referred to as von Neumann machines.

Structure of the IAS computer:

- Main memory (containing data and instructions)
- ALU (which performs operations on binary data)
- Control unit (which interprets the instructions and causes them to be executed)
- I/O equipment

Computers are classified into generations based on the fundamental hardware technology used. Each new generation has greater processing performance, a larger memory capacity, and a smaller size than the previous one:

Generation	Technology	Typical speed (operations per second)
1	Vacuum tube	40 000
2	Transistor	200 000
3	Small- and medium-scale integration	1 000 000
4	Large-scale integration	10 000 000
5	Very-large-scale integration	100 000 000

Moore's law:
The number of transistors that can be put on a single chip doubles every 18 months.

Consequences of Moore's law:
- Lower cost of computer logic and memory circuitry
- Shorter electrical path lengths, which increase operating speed
- Smaller computers
- Reduced power and cooling requirements
- Fewer inter-chip connections (with more circuitry), making interconnections more reliable than solder connections

Characteristics that distinguish a family of computers:
- Similar or identical instruction set
- Similar or identical operating system
- Increasing speed
- Increasing number of I/O ports
- Increasing memory size
- Increasing cost

Computers are becoming faster and cheaper, but the basic building blocks are still the same as those of the IAS computer from over 50 years ago.

Microprocessor speed
The raw speed of the microprocessor won't achieve its potential unless it is fed a constant stream of work to do in the form of computer instructions. Some ways of exploiting the speed of the processor:
- Branch prediction - The processor looks ahead and predicts which instructions are likely to be processed next, prefetching and buffering them so that there's more work available.
- Data flow analysis - The processor analyses which instructions are dependent on each other's data to create an optimised schedule of instructions. Instructions are scheduled to be executed when ready, independent of the original order, preventing delays.
- Speculative execution - The processor uses branch prediction and data flow analysis to speculatively execute instructions ahead of their appearance in the program execution, holding the results in temporary locations. The processor is kept as busy as possible by executing instructions that are likely to be needed.

Performance balance = adjusting the organisation and architecture to compensate for the mismatch between the capabilities of various computer components.

Processor speed and memory capacity have grown rapidly, but the speed with which data can be transferred between main memory and the processor has lagged. The interface between processor and main memory is the most crucial pathway in the computer, because it is responsible for carrying a constant flow of program instructions and data between memory chips and the processor. If memory or the pathway can't keep up with the processor, valuable processing time is lost.
DRAM density is going up faster than the amount of main memory needed, which means that the number of DRAMs per system is actually going down, so there is less opportunity for parallel transfer of data.

Some ways of handling the DRAM density problem:

- Make DRAMs 'wider' rather than 'deeper', i.e. increase the number of bits that are retrieved at one time, and also use wide bus data paths
- Make the DRAM interface more efficient by including a cache
- Reduce the frequency of memory access by using cache structures between the processor and main memory
- Use higher-speed buses to increase the bandwidth between processors and memory, and use a hierarchy of buses to buffer and structure data flow

Handling of I/O devices:
The more sophisticated computers become, the more applications are developed that support the use of peripherals with intensive I/O demands.
Processors can handle the data pumped out by these devices, but the problem is in *moving* the data between processor and peripheral.

- Include caching and buffering schemes
- Use higher-speed interconnection buses and more elaborate bus structures
- Use multiple-processor configurations

Designers must strive to *balance* the throughput and processing demands of the processor, main memory, I/O devices, and the interconnection structures.
The design must cope with two evolving factors:

- The rate at which performance is changing differs from one type of element to another
- New applications / peripherals keep on changing the nature of the demand on the system

Hardware and software are generally **logically equivalent**, which means that they can often perform the same function. Designers have to decide which functions to implement in hardware and which in software. Cost usually plays a role.
Hardware offers speed, but not flexibility.
Software offers flexibility, but less speed.

Intel's Pentium
This is an example of CISC design.

Differences between some members of the Pentium family:

Pentium	Uses superscalar techniques, which allow multiple instructions to execute in parallel
Pentium Pro	Superscalar organisation, with aggressive use of register renaming, branch prediction, data flow analysis, and speculative execution
Pentium II	Incorporates Intel MMX technology to process video, audio, and graphics data efficiently
Pentium III	Incorporates additional floating-point instructions to support 3D graphics software
Pentium 4	Includes additional floating-point and other enhancements for multimedia
Itanium	Uses a 64-bit organisation with the IA-64 architecture

Evolution of the PowerPC
The 801 minicomputer project at IBM, together with the Berkeley RISC I processor, launched the RISC movement. IBM then developed a commercial RISC

workstation, the RT PC. IBM then produced a third system, which built on the 801 and RT PC, called the IBM RISC System/6000 and referred to as the POWER architecture. IBM then entered into alliance with Motorola and Apple, which resulted in a series of machines that implement the PowerPC architecture (derived from the POWER architecture).

The PowerPC architecture is a superscalar RISC system, and one of the most powerful and best-designed ones on the market.

Chapter 3 - Computer functions and interconnections

Computer components

Almost all computer designs are based on the **von Neumann architecture**, which is based on **three key concepts**:

- Data and instructions are stored in a *single read-write memory*
- The contents of this memory are addressable by *location*, without regard to the type of data stored there
- Execution occurs *sequentially* (unless explicitly modified) from one instruction to the next

The reasoning behind these concepts:
Previously programs were 'stored' by connecting logic gates together, which was difficult and cumbersome. Having the program stored in memory so that the control unit can interpret and execute the instructions is called the stored program concept. This is an improvement because the program can be changed or an entirely new one stored without having to perform changes to the hardware.

Hardwired program = a program that is not stored in memory, but forms part of the hardware. Logic components are physically and permanently connected, and contain the sequence of steps that the program should execute.

Memory - stores both instructions and data

- A set of locations (sequentially numbered addresses) contain binary numbers that can be interpreted as either instructions or data

CPU - exchanges data with memory and I/O equipment

- MAR (Memory address register) specifies the address in memory for the next read / write
- MBR (Memory buffer register) contains the data to be written into memory / receives the data from memory
- I/OAR (I/O address register) specifies a particular I/O device
- I/OBR (I/O buffer register) is used for exchanging data between an I/O module and the CPU

I/O - transfers data from external devices to CPU and memory and vice versa

- Internal buffers temporarily hold data until they can be sent on

Computer function

Instruction cycle = the processing required for a single instruction.
Two steps of instruction processing:

1. Fetch cycle: The processor reads instructions from memory one at a time
2. Execute cycle: The processor executes each instruction

Program execution consists of repeating the process of instruction fetch and instruction execution.
Program execution halts only if

- the machine is turned off,
- some sort of unrecoverable error occurs, or
- a program instruction halts the computer

Steps for program execution:

1. The address of the next instruction to be executed is determined

2. The processor fetches the instruction from memory (The PC holds the address of the instruction to be fetched next and is incremented after each fetch)
3. The fetched instruction is loaded into the IR (Instruction Register)
4. The control unit interprets (decodes) the instruction to see what is to be done
5. (If required, address(es) of the operand(s) are determined)
6. (If required, the operand(s) are fetched from memory)
7. The processor performs the required action (see * below)
8. The result is stored in memory, if required
9. Interrupt processing takes place if an interrupt has occurred (see **)

Types of actions that can be performed by the processor *

- Processor-memory (Data can be transferred from processor to memory or vice versa)
- Processor-I/O (Data can be transferred to / from a peripheral device by transferring between the processor and an I/O module)
- Data processing (The processor can perform arithmetic / logic operations on data)
- Control (An instruction can specify a jump to another instruction)

Interrupts = mechanisms by which other modules (like I/O or memory) may interrupt the normal processing of the processor.

Common classes of interrupts:

- Program interrupts - occur as a result of the execution of some instruction (like division by 0). Also known as traps.
- Timer interrupts - occur at regular intervals when the processor's built-in timer signals the OS to perform certain (regular) functions.
- I/O interrupts - caused by external hardware devices to inform the processor that an I/O operation is complete or that an error occurred.
- Interrupts caused by hardware failure - A power failure or memory parity error will cause a hardware interrupt.

Interrupts are provided mainly as a way to *improve processing efficiency*. Most external devices are much slower than the processor, so, with no interrupts, after each write operation the processor must pause and remain idle until the (e.g.) printer catches up.
With interrupts, the processor can execute other instructions while an I/O operation is in progress:

1. The program makes a WRITE call to an I/O program
2. A few instructions of the I/O program are executed (preparation code…)
3. Then control returns to the processor while the external device is busy
4. When the external device is ready to accept more data, it sends an *interrupt request signal* to the processor
5. The processor stops operation of the current program and services the I/O device, known as an *interrupt handler*
6. Afterwards the processor resumes the original execution

Interrupt cycle **
At the end of the instruction cycle, the processor checks to see if any interrupts have occurred. If not, the processor proceeds to the fetch cycle, otherwise:

- The processor suspends execution of the current program and saves its context (including the address of the next instruction to be executed)
- The program counter is set to the starting address of an *interrupt handler* routine

There is extra overhead with the interrupt handling process (Extra instructions must be executed to determine the nature of the interrupt and to decide on the appropriate action), but it still wastes less time than waiting for an I/O operation to finish.

Multiple interrupts
Two approaches can be taken to dealing with multiple interrupts:
1. Disable interrupts while an interrupt is being processed
When an interrupt occurs, interrupts are disabled and the processor ignores interrupt request signals. After the interrupt handler routine completes, interrupts are enabled and the processor checks to see if additional interrupts have occurred.
Advantage: Simple approach because interrupts are handled in sequential order.
Disadvantage: It doesn't take into account relative priority or time-critical needs.
2. Define priorities for interrupts
Interrupts of higher priority cause lower-priority interrupt handlers to be interrupted.

I/O function
An I/O module (like a disk controller) can exchange data directly with the processor. (The processor identifies a specific device that is controlled by a particular I/O module).
If necessary, the processor can grant an I/O module the authority to read from / write to memory, so that the I/O-memory transfer can occur without tying up the processor. (Direct memory access).

Interconnection structures

Interconnection structure = the collection of paths connecting the various modules (processor, memory, I/O).

Data to be exchanged:
Memory
A memory module consists of N words of equal length, with unique addresses. A word of data can be *read* from or *written* into the memory.
I/O module
I/O is functionally similar to memory (*read* & *write* operations). An I/O module can control more than one external device, each of which has a unique address. There are external data paths for the input & output of data with an external device. An I/O module may also be able to send *interrupt* signals to the processor.
Processor
The processor *reads* in instructions and data, *writes* out data after processing, and uses control signals to *control* the overall operation of the system. It also receives *interrupt signals*.

The interconnection structure must support the following types of transfers:
- Memory to processor
- Processor to memory
- I/O to processor
- Processor to I/O
- I/O to or from memory

Bus interconnection

Bus = a communication pathway connecting two or more devices.
Key characteristic: it is a *shared* transmission medium, and only one device can transmit signals at a time.
A bus consists of multiple communication pathways (lines), which can each transmit signals representing binary 1 / 0.
A sequence of digits can be transmitted across a single line.
Several lines can be used to transmit digits in parallel.
System bus = a bus that connects major computer components (processor, memory, I/O).

Bus structure

A system bus consists of about 50 → hundreds of separate lines, which each have a particular function.

Three functional groups of bus lines:

1. Data lines

Provide a path for moving data between system modules.
Collectively called the *data bus*.
The data bus can consist of 32 → hundreds of separate lines.
Bus width = the number of lines, which determines how many bits can be transferred at a time. (A key factor in determining system performance).

2. Address lines

Used to designate the source / destination of the data on the data bus.
The width of the address bus determines the max memory capacity.

3. Control lines

Used to control the access to and the use of the data and address lines.
(Data and address lines are shared by all components, so their use must be controlled).
Control signals transmit command and timing info between system modules:
* *Command signals* specify operations to be performed.
* *Timing signals* indicate the validity of data and address information.

Bus operation:

A module that wants to send data must
1. Obtain the use of the bus
2. Transfer data via the bus

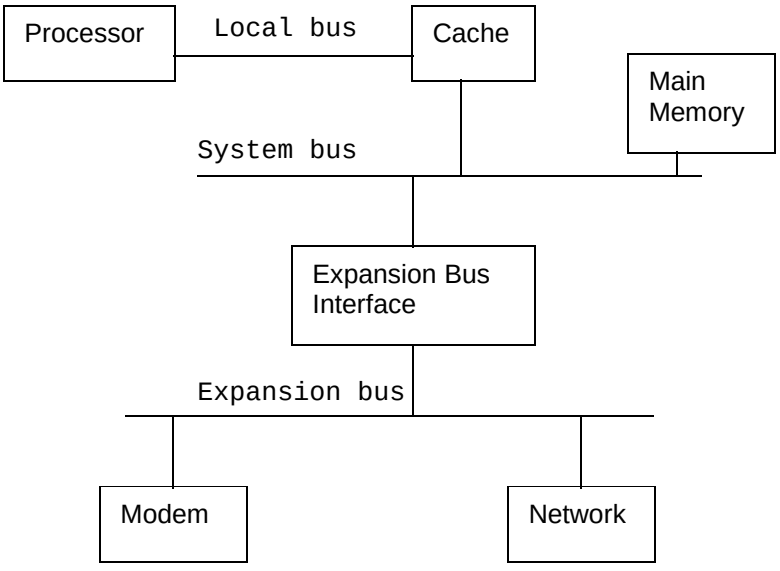
A module that wants to request data from another module must
1. Obtain the use of the bus
2. Transfer a request to the module over the appropriate control and address lines
3. Wait for the other module to send the data

An on-chip bus may connect the processor and cache memory.
An on-board bus may connect the processor to main memory and other components.

Multiple-bus hierarchies

Why connecting too many devices to the bus causes performance to suffer:
- The more devices attached, the greater the bus length, the greater the *transmission delay*.
- The bus may become a bottleneck as the *capacity* of the bus is reached.

How multiple buses can be used in a hierarchy:



This is the *traditional bus architecture*:

Main memory is moved off the local bus onto a system bus, so I/O transfers to and from memory don't interfere with the processor's activity.
You may connect I/O controllers directly onto the system bus, but it is more efficient to use an expansion bus (see diagram), because this allows support for a wide variety of I/O devices and insulates memory-to-processor traffic from I/O traffic.

Mezzanine architecture (high-performance architecture):
With increasing performance in I/O devices, a high-speed bus is needed between the expansion bus and system bus.
This bus supports connections to high-speed LANs, video & graphics workstation controllers, FireWire, etc.
Lower-speed devices are still supported off the expansion-bus, with an interface buffering traffic between the expansion bus and the high-speed bus.
Advantage: The high-speed bus brings high-demand devices into closer integration with the processor and is also independent of the processor.
Changes in processor architecture don't affect the high-speed bus, and vice versa.

Elements of bus design

1. Bus types
* *Dedicated*
  - Functional dedication:
  E.g. Using separate dedicated address and data lines.
  - Physical dedication:
  E.g. Using an I/O bus to interconnect all I/O modules.
  Advantage: High throughput because there is less bus contention.
  Disadvantage: Increased size and cost of the system.
* *Multiplexed*
Using the same lines for multiple purposes.
Advantage: The use of fewer lines saves space and cost.
Disadvantage: More complex circuitry is needed and there is a potential reduction in performance.

2. Method of arbitration
Arbitration is needed because only one unit at a time can transmit over the bus
* *Centralised*
A single hardware device (a bus controller / arbiter) is responsible for allocating time on the bus.
The device may be a separate module or part of the processor.
* *Distributed*
Each module contains access control logic and the modules act together to share the bus.

3. Timing
Timing refers to the way in which events are coordinated on the bus.
* *Synchronous*
The occurrence of events on the bus is determined by a clock. The bus has a clock line upon which a clock transmits a regular sequence of alternating 1s and 0s. A single 1-0 transmission is referred to as a clock cycle / bus cycle, and defines a time slot. All other devices on the bus can read the clock line, and all events start at the beginning of a clock cycle. Most events occupy a single clock cycle.
Advantage: Simpler to implement and test than asynchronous timing.
Disadvantage: Less flexible than asynchronous timing.
Disadvantage: Because all devices are tied to a fixed clock rate, the system can't take advantage of advances in device performance.
* *Asynchronous*
The occurrence of one event on a bus follows and depends on the occurrence of a previous event.
Advantage: A mixture of slow and fast devices, using older and newer technology, can share a bus.

4. Bus width
* *Address*
The wider the address bus, the greater the range of locations referrenced.
* *Data*
The wider the data bus, the greater the number of bits transferred at one time.

5. Data transfer type
* *Read*
* *Write*
* *Read-modify-write*
* *Read-after-write*
* *Block data transfer*

PCI (Peripheral Component Interconnect)

PCI is a popular high-bandwidth processor-independent bus that can function as a mezzanine or peripheral bus.
Intel worked on PCI for its Pentium-based systems and released the patents to the public domain so that PCI would be widely adopted.

Advantages:
- Delivers better system performance for high-speed I/O subsystems
- Requires very few chips to implement
- Supports other buses attached to it

Characteristics:
- Supports a variety of microprocessor-based configurations (including single- and multiple-processor systems)
- Provides a general-purpose set of functions
- Makes use of synchronous timing and a centralised arbitration scheme

Bus structure
PCI may be configured as a 32- or 64-bit bus.
The signal lines can be divided into the following functional groups:
Mandatory PCI signal lines
- System pins
- Address and data pins
- Interface control pins
- Arbitration pins
- Error reporting pins

Optional PCI signal lines
- Interrupt pins
- Cache support pins
- 64-bit bus extension pins
- JTAG/boundary scan pins

PCI commands
- Interrupt Acknowledge
- Special cycle
- I/O Read / Write
- Memory Read / Read Line / Read Multiple / Write / Write and Invalidate
- Configuration Read / Write
- Dual Address Cycle

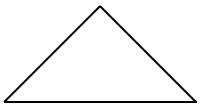
Chapter 4 - Cache memory

Characteristics of memory systems

The most important characteristics of memory are *capacity* and *performance*.
Location
Processor (local memory, in the form of registers)

Internal (main. E.g. Cache)
External (secondary. E.g. Peripheral storage devices)
Capacity
Word size (common word lengths are 8, 16, and 32 bits)
Number of words
Unit of transfer (= The no of bits read out of / written into memory at a time)
Word
Block (= a larger unit than a word; for external memory transfers)
Access Method
Sequential (Access is made in a *linear* sequence. E.g. Tapes)
Direct (Access is made by direct access to reach a *general vicinity* first)
Random (Any location can be selected at random and *directly* accessed)
Associative (A word is retrieved based on a portion of its *contents*)
Performance
Access time
(For RAM: The time it takes to perform a read / write operation)
(For non-RAM: the time it takes to position the read-write mechanism)
Cycle time (Access time + additional time needed before a 2nd access can begin)
Transfer rate (Rate at which data can be transferred in / out of a memory unit)
Physical type
Semiconductor
Magnetic (used for disk and tape)
Optical
Magneto-optical
Physical Characteristics
Volatile / non-volatile
Erasable / non-erasable
Organisation
(Organisation = the physical arrangement of bits to form words)

The memory hierarchy



Fast access time, but high cost and low capacity

Slow access time, but **low cost** and **high capacity**

The designer can't rely on a single memory component, but must employ a memory hierarchy for the combination of high capacity, low cost, and fast access.

The hierarchy:
Inboard memory - Registers → cache → main memory
Outboard storage - Magnetic disk, CD-ROM, CD-RW, DVD-RW, DVD-RAM
Off-line storage - Magnetic tape, MO, WORM

As you go down the hierarchy, the *frequency of access* of the memory by the processor decreases.

Locality of reference
During program execution, memory references by the processor tend to cluster. Once a loop / subroutine is entered, there are repeated references to a small set of instructions. Over a long period of time, the clusters in use change, but over a short period of time, the processor is mainly working with fixed clusters of memory references.
You can organise data across the hierarchy so that the percentage of accesses to the lower levels is less than the higher ones. E.g. you can temporarily place low-level clusters in higher levels, and later swap them back to make room for new clusters coming in.

Cache memory principles

Cache memory (small & fast) contains a copy of portions of main memory (slow & large). When the processor wants to read a word from memory, it first checks the cache. If it's not there, a block of main memory (consisting of some fixed

number of words) is read into the cache and then the word goes to the processor. Locality of reference means that it's likely there will be future references to other words in the same block.

Each line of cache contains a block of memory (with a fixed number of words). The number of cache lines is considerably less than the number of main memory blocks! At any time, some subset of the blocks of memory resides in lines in the cache. Because there are more blocks than lines, an individual line can't be uniquely and permanently dedicated to a particular block. So, each line includes a *tag* that identifies which particular memory block is currently being stored. (The tag is usually a portion of the main memory address).

Typical cache organisation:

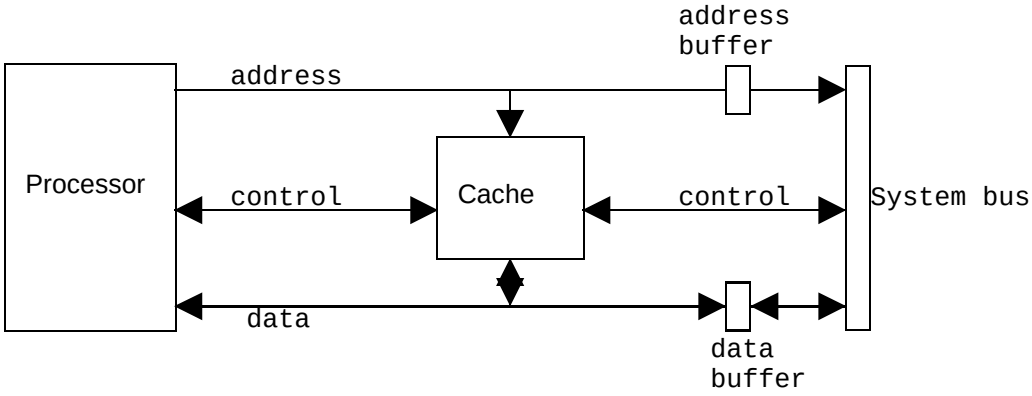


Diagram explanation:

- The cache connects to the processor via data, control, and address lines
- The data and address lines also attach to buffers, which attach to the system bus from which main memory is reached
- Cache hit: The data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic
- Cache miss: The address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor (In some organisations, the word is first read into the cache and *then* transferred from cache to processor)

Elements of cache design

Mapping function

Direct mapping
Each block of memory is mapped onto only one possible cache line. Only the address is used to determine the corresponding cache line. A tag is necessary to distinguish the data block from other data blocks that can fit into the relevant line
Advantage: Simple and inexpensive to implement.
Disadvantage: There is a fixed cache location for any given block.
Associative mapping
Each main memory block can be loaded into any line of cache. The cache control logic interprets each main memory address as a tag and a word field, where the tag field uniquely identifies a block of main memory. All lines have to be examined to determine whether a particular memory block is in the cache.
Advantage: Flexible
Disadvantage: Requires complex circuitry in order to examine all the tags of the cache in parallel.
Set-associative mapping
A compromise between direct and associative mapping. The cache is divided into sets, each of which consists of a number of lines. A block can be mapped into any of the lines of a particular set. The tag is only compared to the tags within a single set.

Advantages of both direct and associative methods, while their disadvantages are reduced.

Number of caches

Single and two level caches
The on-chip cache reduces the processor's external bus activity and therefore speeds up execution times and increases overall system performance. (The bus is then also free to support other transfers).
Two-level cache organisation: Internal on-chip cache (L1) and external cache (L2), which is the organisation found in most contemporary designs.
Without L2 cache, if the processor makes an access request for a memory location not in L1 cache, it would have to access DRAM or ROM memory across the bus, which results in poor performance because of slow bus speed.
Design features for multilevel caches:

- For an off-chip L2 cache, many designs don't use the system bus as the transfer path between L2 and processor, but use a separate *data path* to reduce the burden on the system bus.
- With the continued shrinkage of processor components, many processors now incorporate the *L2 cache on the processor chip*, improving performance.

Advantage of multilevel caches: Improved performance
Disadvantage of multilevel caches: Complicated design

Unified and split caches
It has become common to split the cache into two: One dedicated to instructions and one dedicated to data.
Advantages of a unified cache:

- *Higher hit rate* than split caches because it balances the load between instruction and data fetches automatically. (E.g. If there are many more instruction fetches than data fetches, the cache fills up with instructions rather than data)
- *Only one cache* needs to be designed and implemented.

Advantage of split caches:

- *Eliminates contention* for the cache between the instruction fetch/decode unit (which accesses instructions) and the execution unit (which accesses data). Parallel access is allowed.

Pentium 4 cache organisation

Two on-chip L1 caches, one for data and one for instructions.
The L1 data cache is 8K, using a line size of 64 bytes and a four-way set associative organisation.
An L2 cache feeds both of the L1 caches.
The L2 cache is 256K, using a line size of 128 bytes and an eight-way set associative organisation.

The processor core consists of four major components:

- Fetch / decode unit
  Fetches program instructions from L2 cache, decodes them, and stores the results in the L1 instruction cache
- Out-of-order execution logic
  Schedules execution of the micro-operations subject to data dependencies and resource availability (I.e. micro-operations may be scheduled for execution in a different order than they were fetched)
- Execution units
  Execute micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers
- Memory subsystem
  Includes the L2 cache and the system bus
  Accesses main memory when the L1 and L2 caches have a cache miss
  Also accesses the system I/O resources

Instruction cache
Unlike the organisation used in all previous Pentium models (and in most other processors), the Pentium 4 instruction cache sits between the instruction fetch / decode unit and the (out-of-order) execution core.
The reason:
Machine instructions are decoded into simple RISC-like instructions called micro-operations. (Using these simple micro-ops enhances performance). However, the Pentium machine instructions are cumbersome to decode, so performance is enhanced if this decoding is done independently of the scheduling and pipelining logic.

Data cache
The data cache employs a *write-back* policy: Data are written to main memory only when they are removed from the cache and there has been an update. (The write-back technique minimises memory writes because updates are only made in the cache).
The Pentium 4 processor can be dynamically configured to support write-through caching (A simple technique where all write operations are made to main memory as well as to cache, ensuring that main memory is always valid)
Two bits in control registers that control the data cache:

- CD (cache disable)
- NWT (not write-through)

Two instructions that control the data cache:

- INVD - invalidates / flushes the internal cache memory and signals the external cache (if any) to invalidate
- WBINVD - writes back and invalidates internal, then external cache

Chapter 5 - Internal memory

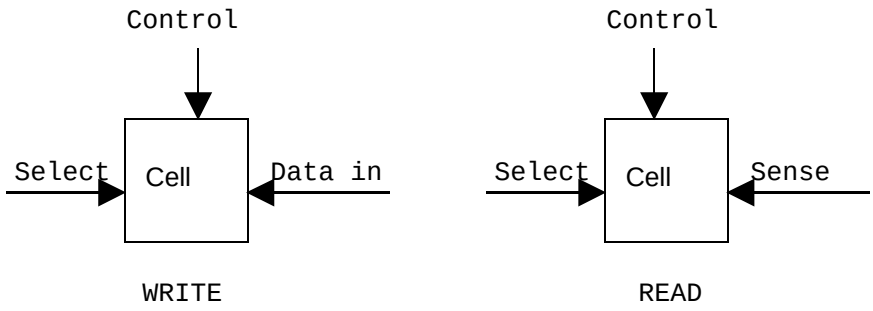
Semiconductor main memory

Semiconductor chips are universally used for main memory.
Memory cell = the basic element of a semiconductor memory.
All semiconductor memory cells share these properties:

- Two stable states, representing binary 1 and 0
- Capable of being written into (at least once), to set the state
- Capable of being read to sense that state

Operation of a memory cell:
The cell has 3 function terminals capable of carrying an electrical signal:

- Select terminal - selects a memory cell for a read or a write operation
- Control terminal - indicates read or write
- Third terminal -
  * For writing (the terminal provides an electrical signal that sets the cell's state to 1 or 0)
  * For reading (the terminal is used for output of the cell's state)



RAM
Characteristics:

- You can read & write to memory easily & rapidly
- Reading and writing are accomplished by using electrical signals
- Volatile (There must be a constant power supply)

Two traditional forms of RAM used in computers:

1. DRAM (dynamic)

The cells store data as charges, on capacitors. The presence or absence of a charge on a capacitor is interpreted as binary 1 or 0. DRAMs require periodic charge refreshing to maintain data storage, because capacitors have a natural tendency to discharge. 'Dynamic' refers to this tendency of the stored charge to leak away, even with power continuously applied.

Operation of DRAM:

Write operation

1. A voltage signal is applied to the bit line (high = 1, low = 0)
2. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor

Read operation

1. The address line is selected
2. The transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier
3. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains 1 or 0
4. The read out from the cell discharges the capacitor, which must be then restored to complete the operation

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analogue device. The capacitor can store any charge value within a range, but it is interpreted as 1 or 0.

2. SRAM (static)

Binary values are stored using traditional flip-flop logic-gate configurations. A static RAM will hold its data as long as power is supplied to it. No refresh is needed to retain data.

Operation of SRAM:

The address line (which controls transistors) is used to open or close a switch. When a signal is applied to the address line, the transistors are switched on, allowing a read / write operation.

DRAM	**SRAM**
Volatile (power must be continuously supplied to the memory to preserve the bit values)	
Analogue	Digital
Smaller and simpler memory cell	Larger and more complex memory cell
More dense and less expensive	Less dense and more expensive
Requires supporting refresh circuitry	
Favoured for large memory requirements	
Slower	Faster
Used for main memory	Used for cache memory (on & off chip)

Types of ROM

ROM contains a permanent pattern of data that can't be changed.
A ROM is non-volatile: no power is needed to maintain the bit values in memory.
You can't write to a ROM

PROM

The writing process ('programming') is performed electrically, with special equipment.

Advantages:

- A less expensive alternative when only a small number of ROMs with a particular memory content is needed
- Provide flexibility and convenience
- Good for high-volume production runs

Read-mostly memory:
(For when read operations are far more frequent than write operations)

	EPROM	Flash memory	EEPROM
Erasure required before a write	Yes	Yes	No

operation			
Method of erasure	Exposure to UV radiation	Electrical erasure	Electrical erasure
Erasure level	Chip level	Block level	Byte level
Density	High	High	Lower
Advantages	Can be updated multiple times	An entire flash memory can be erased in a few seconds	Flexible: can be updated in place
Cost and functionality	Lowest	Moderate	Highest

Error correction

Hard failure = a permanent physical defect so that the memory cell(s) affected can't reliably store data, but become stuck at 0 or 1 or switch erratically between 0 and 1.
Possible causes: harsh environmental abuse, manufacturing defects, and wear.

Soft error = a random, non-destructive event that alters the contents of one or more memory cells, without damaging the memory.
Possible causes: power supply problems or alpha particles resulting from radioactive decay.

The error-detection process:

1. Before data are read into memory, a calculation produces a code
2. Both the code and the data are stored
3. When the stored data word is read out, the code is used to detect errors
    - No errors: The fetched data bits are sent out
    - Error detected that can be corrected: The data bits + error correction bits are fed into a corrector
    - Error detected that can't be corrected: The condition is reported

Error-correcting codes = codes that operate as in point 3 above. A code is characterised by the number of bit errors in a word that it can detect and correct.
Hamming code = the simplest of the error-correcting codes. Parity bits can be checked. If there is an error, the appropriate bit is changed.

Advanced DRAM organisation

SDRAM
Exchanges data with the processor synchronised to an *external clock signal*.

1. The DRAM moves data in & out under control of the system clock
2. The processor (master) issues the instruction and address information, which is latched by the DRAM
3. The DRAM then responds after a number of clock cycles
4. Meanwhile, the master can do other tasks while the SDRAM is processing the request

An SDRAM's mode register specifies the burst length (= no of separate units of data synchronously fed onto the bus). It also allows the programmer to adjust the latency between receipt of a read request and the beginning of data transfer.
SDRAM performs best when it is transferring large blocks of data serially, like spreadsheet and multimedia applications.

DRAM	SDRAM
Asynchronous	Synchronous (external clock signal)
The processor must wait during the access-time delay, slowing down system performance.	The processor can do other tasks while the SDRAM is processing requests, improving system performance.

-	Mode register and associated control logic, for customisation.
-	Multiple-bank internal architecture that improves opportunities for on-chip parallelism.

Chapter 6 - External memory

Magnetic disk

A disk is a circular platter constructed of nonmagnetic material, called the substrate, coated with magnetisable material. Traditionally, the substrate was made from aluminium, but these days glass is used.
Advantages of using a glass substrate:

- Improves the uniformity of the magnetic film surface to increase disk reliability
- Reduces overall surface defects, which reduces read-write errors
- Supports lower fly heights
- Better stiffness reduces disk dynamics
- Greater ability to withstand shock and damage

Magnetic read & write mechanisms
Head = a conducting coil that is used to record and retrieve data. Some systems have two heads (one read and one write). During a read / write operation, the head is stationary while the platter rotates beneath it.
Write mechanism:
Electricity flowing through a coil produces a magnetic field. Pulses are sent to the write head and magnetic patterns are recorded on the surface below.
Read mechanism:
When the surface of the disk passes under the head, a magnetic field moving relative to a coil produces an electrical current in the coil.

Data organisation and formatting
Tracks = concentric rings on the platter, with the same width as the head. There are thousands of tracks per surface.
Gaps = separators between adjacent tracks, to minimise errors due to misalignment of the head or interference of magnetic fields.
Sectors = fixed- or variable-length divisions on a track, separated by inter-sector gaps. There are hundreds of sectors per track.

Bits near the centre of a rotating disk travel slower than bits on the outside, so it is necessary to compensate for the variation in speed so that the head can read all the bits at the same rate.

Two disk layout methods:
a) Constant Angular Velocity (CAV)
CAV = the fixed speed at which information can be scanned at the same rate.
Layout: The disk is divided into pie-shaped sectors and tracks, with increased spacing between the outer bits.
Advantage: Individual blocks of data can be directly addressed by track and sector. (It takes a short movement of the head to a specific track, and a short wait for the proper sector to spin under the head).
Disadvantage: The amount of data that can be stored on the outer tracks is the same as what can be stored on the short inner tracks.

b) Multiple zoned recording
Layout: The surface is divided into zones. Within a zone, the number of bits per track is constant. Zones further from the centre contain more bits (more sectors) than zones closer to the centre.
Advantage: Greater overall storage capacity.
Disadvantage: More complex circuitry.

Disks have control data recorded on them to indicate the starting point on the track and start & end of each sector.

Physical characteristics

Head motion

- *Fixed head disk*
  One read-write head per track.
  All of the heads are mounted on a rigid arm that extends across all tracks (rare).
- *Movable head disk*
  Only one read-write head.
  The head is mounted on an arm, which can extend and retract for the head to be positioned above any track.

Disk portability

- *Non-removable disk*
  Permanently mounted in the disk drive (E.g. the hard disk)
- *Removable disk*
  Can be removed and replaced with another disk, so unlimited amounts of data are available (E.g. floppy disks and ZIP cartridge disks)

Sides

- *Single sided*
- *Double sided*

Platters

- *Single platter*
- *Multiple platters*
  Multiple arms are provided. All of the heads are mechanically fixed so that they all are at the same distance from the centre of the disk (i.e. same track number) and move together. Cylinder = the set of all the tracks in the same relative position on the platter.

Head mechanism

- *Contact (floppy)*
  The head mechanism comes into physical contact with the medium
- *Fixed gap*
  The read-write head is positioned a fixed distance above the platter, allowing an air gap
- *Aerodynamic gap (Winchester)*
  A head must generate / sense an electromagnetic field of sufficient magnitude to write / read properly. The narrower the head, the closer it must be to the platter. (Narrower heads → narrower tracks → greater data density). However, the closer the head is to the disk, the greater the risk of error from impurities / imperfections.
  Winchester heads are used in sealed drive assemblies that are almost free of contaminants. They are designed to operate closer to the disk's surface, allowing greater data density. The head is an aerodynamic foil that rests lightly on the platter's surface when the disk is motionless. When the disk spins, the air pressure generated is enough to make the foil rise above the surface.

Disk performance parameters

Seek time = the time it takes to position the head at the track. (Consists of an initial start-up time, and the time taken to traverse the necessary tracks).
Rotational delay = the time it takes for the beginning of the sector to reach the head. (Floppy disks typically rotate between 300 & 600 rpm).
Access time = seek time + rotational delay = the time it takes to get into position to read / write.
Transfer time = the time required for the data transfer (read / write).

Other delays associated with a disk I/O operation:

- When a process issues an I/O request, it must first wait in a queue for the device to be available.
- If the device shares a single I/O channel, there may be an additional wait for the channel to be available.

Wait for device → Wait for channel → Seek → Rotational delay → Data transfer

RAID

RAID is a standardised scheme for multiple-disk database design.
The RAID strategy replaces large-capacity disk drives with multiple smaller-capacity drives and distributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental increases in capacity.
Allowing multiple heads to operate simultaneously achieves higher I/O and transfer rates, but increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

RAID levels 0 → 6 share three common characteristics:

- RAID is a set of physical disk drives viewed by the OS as a single logical drive
- Data are distributed across the physical drives of an array
- Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure (not RAID 0)

RAID level 0
Not true RAID because it doesn't include redundancy to improve performance.
Category: Striping
Striping = distributing data over multiple drives in round robin fashion.
Advantage of striping: If a single I/O request consists of multiple logically contiguous strips, then up to n (n = no of disks) strips for that request can be handled in parallel, greatly reducing the I/O transfer time.
Performance is excellent and the implementation is straightforward.
Disadvantage: Works worst with OS that mainly ask for data one sector at a time
Disadvantage: Reliability is potentially worse than having a single large disk.
Applications: Supercomputers, where performance and capacity are the main concern and low cost is more important than improved reliability.
RAID 0 for High Data Transfer Capacity:
Two requirements must be met for RAID 0 to achieve a high data transfer rate:
* A high transfer capacity must exist along the entire path between host memory and the individual disk drives. (Includes buses, I/O adapters, etc.)
* The application must make I/O requests that drive the disk array efficiently. (E.g. if large amounts of logically contiguous data are requested).
RAID 0 for High I/O Request Rate:
For an individual I/O request for a small amount of data, the I/O time is dominated by the motion of the disk heads (seek time) and the movement of the disk (rotational latency). A disk array can provide high I/O execution rates by balancing the I/O load across multiple disks. The larger the strip size, the more requests that can be handled in parallel.
(RAID 0 works best with large requests).

strip 0	strip 1	strip 2	strip 3
strip 4	strip 5	strip 6	strip 7
strip 8	strip 9	strip 10	strip 11

RAID level 1
Category: Mirroring
Differs from RAID levels 2 → 6 in the way in which redundancy is achieved. In these other RAID schemes, some form of parity calculation is used to introduce redundancy, but in RAID 1, redundancy is achieved by duplicating all the data.
Data striping (as in RAID 0) is used, but each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk with the same data.
Advantages:

- A read request can be serviced by either of the two disks that contains the requested data

- A write request requires that both corresponding strips be updated, but this can be done in parallel. (There are no parity bits to update as well)
- Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive (Excellent fault tolerance)

Disadvantage:

- Cost. Requires twice the disk space of the logical disk that it supports

RAID 1 can achieve high I/O request rates if the bulk of the requests are reads (doubling the performance of RAID 0).
Applications: System drives, critical files

strip 0	strip 1	strip 2	strip 3	strip 0	strip 1	strip 2	strip 3
strip 4	strip 5	strip 6	strip 7	strip 4	strip 5	strip 6	strip 7
strip 8	strip 9	strip 10	strip 11	strip 8	strip 9	strip 10	strip 11

RAID level 2
Category: Parallel access
RAID levels 2 & 3 make use of a parallel access technique. In a parallel access array, all member disks participate in the execution of every I/O request. The spindles of the individual drives are synchronised so that each disk head is in the same position on each disk at any given time.
Data striping is used (In RAID 2 & 3 the strips are as small as a byte or a word). An error-correcting code is calculated across corresponding bits in each data disk, and the bits of the code are stored in the corresponding bit positions on *multiple* parity disks. Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.
E.g. You could split bytes into two pairs of 4 bits, each one with three parity bits, forming 7-bit words.
Although RAID2 requires fewer disks than RAID1, it is still costly.
On a single read, all disks are simultaneously accessed. The requested data and the associated error-correcting code are delivered to the array controller. If there is a single-bit error, the controller can recognise and correct it instantly, so that the read access time is not slowed.
On a single write, all data disks and parity disks must be accessed for the write operation.
Disadvantage: all drives must be rotationally synchronised, and this only makes sense with a substantial number of drives.
Applications: none (environments in which many disk errors occur)

bit 0	bit 1	bit 2	bit 3	f(b)	g(b)	h(b)

RAID level 3
Category: Parallel access
The difference between RAID3 and RAID2 is that RAID3 requires only a single redundant disk, no matter how large the disk array.
RAID3 employs parallel access, with data distributed in small strips. Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.
Redundancy:
In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive and operation resumed.
In the event of a disk failure, all of the data are still available in what is referred to as *reduced mode*.
Reduced mode reads: The missing data are regenerated on the fly using an exclusive OR calculation.
Reduced mode writes: Consistency of the parity must be maintained for later regeneration.
Return to full operation requires that the failed disk be replaced and the entire contents of the failed disk be regenerated on the new disk.
Performance:

Because data are striped in very small strips, RAID3 can achieve very high data transfer rates. Any I/O request will involve the parallel transfer of data from all of the disks. For large transfers, the performance improvement is very noticeable. However, only one I/O request can be executed at a time, so in a transaction-oriented environment performance suffers.
Applications: Large I/O request size applications, like imaging and CAD

bit 0	bit 1	bit 2	bit 3	parity

RAID level 4
Category: Independent access
In an independent access array, each member disk operates independently, so that separate I/O requests can be satisfied in parallel.
Independent access arrays are better suited for high I/O request rates than high data transfer rates.
RAID 4 → 6: Striping is used (large strips)
RAID 4 has strip-for-strip parity written onto an extra device.
Disadvantage: If one sector is changed, it is necessary to read all the drives to recalculate the parity.
Applications: None

block 0	block 1	block 2	block 3	p(0-3)
block 4	block 5	block 6	block 7	p(4-7)
block 8	block 9	block 10	block 11	p(8-11)

RAID level 5
Category: Independent access
Eliminates the bottleneck of RAID 4 by distributing the parity bits uniformly over all the drives.
Disadvantage: in the event of a drive crash, reconstructing the contents of the failed drive is a complex process.
Applications: High request rate, read intensive, data lookup

block 0	block 1	block 2	block 3	p(0-3)
block 4	block 5	block 6	p(4-7)	block 7
block 8	block 9	p(8-11)	block 10	block 11

RAID level 6
Category: Independent access
Two different parity calculations are carried out and stored in separate blocks on different disks, making it possible to regenerate data even if two disks fail.
Advantage: Extremely high data availability.
Disadvantage: Substantial write penalty, because each write affects two parity blocks.
Applications: Those requiring extremely high availability

block 0	block 1	block 2	block 3	P(0-3)	Q(0-3)
block 4	block 5	block 6	P(4-7)	Q(4-7)	block 7
block 8	block 9	P(8-11)	Q(8-11)	block 10	block 11

Optical memory

CD-ROM
The audio CD and CD-ROM share a similar technology (both are made the same way). The main difference is that CD-ROM players are more rugged and have error correction devices to ensure that data are properly transferred from disk to computer. CDs can hold 60 minutes of music, and CD-ROMS can hold 650 Mbytes.
Appearance:

The disk is formed from a resin (like polycarbonate). Digital information is imprinted as a series of microscopic pits on the surface of the polycarbonate. (This is done firstly with a laser to create a master disk, which is used to make a die to stamp out copies onto the polycarbonate). The pitted surface is then coated with a highly reflective surface (aluminium / gold), which is covered with clear acrylic to protect it against dust and scratches. Finally, a label can be silk-screened onto the acrylic.
Polycarbonate → aluminium → acrylic → label.

Data retrieval:

Information is retrieved from a CD or CD-ROM by a laser in the player / drive unit. The laser shines through the clear polycarbonate while the disk spins. The intensity of the reflected light of the laser changes as it encounters pits and lands: Pits → a low intensity is reflected back to the source; lands → a higher intensity is reflected back. The change between pits and lands is detected by a photosensor and converted into a digital signal. (Beginning / end of a pit = 1, no change in elevation = 0).

Data organisation:

Information is organised on a single spiral track, spiralling outwards (Greater capacity than concentric tracks!) Sectors near the outside are the same length as those near the inside, so information is packed evenly. The information is scanned at the same rate by rotating the disk at variable speed (Slower for accesses near the outer edge and faster for the centre). The pits are read by the laser at a *constant linear velocity* (CLV).
Data are organised as a sequence of blocks, with the following fields:

- *Sync*
  Identifies the beginning of a block
  Consists of a byte of all 0s, 10 bytes of all 1s, and a byte of all 0s
- *Header*
  Contains the block address and the mode byte
  Mode 0 = blank data field
  Mode 1 = use of an error-correcting code and 2048 bytes of data
  Mode 2 = no error-correcting code and 2236 bytes of data
- *Data*
  User data
- *Auxiliary*
  Additional user data in mode 2
  Error-correcting code in mode 3

With the use of CLV, random access becomes more difficult. The head moves to the general area, and then tries to find and access the specific sector.

Applications:

CD-ROM is appropriate for the distribution of large amounts of data to a large number of users.
Not appropriate for individualised applications, because of the expense of the initial writing process.

Advantages:

- The optical disk and info on it can be mass replicated inexpensively (unlike a magnetic disk).
- The optical disk is removable, allowing the disk itself to be used for archival storage. (Most magnetic disks are not removable).

Disadvantages:

- Read-only, so can't be updated
- Access time is much longer than that of a magnetic disk drive

CD Recordable

The disk can be written to once with a moderately intensive laser beam. Instead of pitting the surface to change reflectivity, there is a dye layer which does the same thing, after being activated by a laser.
The disk can be read on a CD-R / CD-ROM drive.
The CD-R optical disk is good for archival storage of documents and files, providing a permanent record of large volumes of data.

CD Rewritable
Phase change: A disk uses a material that has two different reflectivities in two different phase states.
Amorphous state: The molecules exhibit a random orientation; reflects light poorly
Crystalline state: Smooth surface that reflects light well.
A beam of laser light can change the material from one phase to another.
Disadvantage of phase change optical disks: The material eventually loses its properties, so you can have only up to 1 000 000 erase cycles.
Advantage over CD-ROM & CD-R: Can be rewritten and used as true secondary storage.

DVD
Can store 7 times the amount of data as CD-ROMs.
Come in writeable (DVD-R, DVD-RW: one-sided) as well as read-only (DVD-ROM: one- or two-sided) versions.
Three differences from CDs:

- Bits are packed more closely on a DVD, and a laser with shorter wavelength is used, increasing the capacity to 4.7 GB
- The DVD has a second layer of pits and lands on top of the first layer, doubling the capacity to about 8.5 GB. (By adjusting focus, the lasers in DVD drives can read each layer separately)
- The DVD-ROM can be two sided, bringing total capacity up to 17 GB

Magnetic tape

Tape systems use the same reading and recording techniques as disk systems. A flexible polyester tape is coated with magnetisable material and is housed in a cartridge.
Parallel recording:
Data are structured as a number of parallel tracks running lengthwise. Tapes used to have 9 tracks (to store one byte at a time + a parity bit), but now use 18 (word) or 36 (double word) tracks.
Serial recording:
Data are laid out as a sequence of bits along each track, as is done with magnetic disks. Most modern systems use this method.
As with the disk, data are read and written in contiguous blocks, called *physical records*, on a tape. Blocks on the tape are separated by gaps called *inter-record gaps*. As with the disk, the tape is formatted to assist in locating physical records.
Serpentine recording (for serial tapes):
When data are being recorded, the first set of bits is recorded along the whole length of the tape. When the end of the tape is reached, the heads are repositioned to record a new track, and the tape is again recorded on its whole length, this time in the other direction. The process continues until the tape is full. To increase speed, the read-write head is capable of reading and writing a number of adjacent tracks simultaneously. Although data are recorded *serially* along individual tracks, blocks in sequence are stored on *adjacent* tracks.

A tape drive is a sequential-access (as opposed to direct-access) device. Unlike the disk, the tape is in motion only during a read / write operation. Magnetic tape was the first kind of secondary memory and is still widely used as the lowest-cost, slowest-speed member of the memory hierarchy.

Chapter 7 - Input / Output

Reasons why peripherals are not connected directly to the system bus:

- There are a wide variety of peripherals with various methods of operation. It would be impractical to incorporate the necessary logic within the processor to control a range of devices

- The data transfer rate of some peripherals is much slower than that of memory or processor. It is impractical to use the high-speed system bus to communicate directly with the peripheral
- The data transfer rate of some peripherals is faster than that of the memory or processor
- Peripherals often use different data formats and word lengths than the computer to which they are attached

Functions of an I/O module:

- Interface to the processor and memory via the system bus / central switch
- Interface to one or more peripheral devices by tailored data links

External devices

An external device attaches to the computer by a link to an I/O module. This link is used to exchange control, status, and data between the I/O module and the external device.
Three categories of external devices:

- Human readable (for communicating with the user. E.g. printers)
- Machine readable (for communicating with equipment. E.g. tape systems)
- Communication (for communicating with remote devices E.g. another PC)

An external device has an interface to the I/O module with:

- Control signals (that determine the function the device will perform)
- Data (in the form of a set of bits to be sent/received from the module)
- Status signals (to indicate the state of the device)

Control logic controls the device's operation in response to the I/O module.
The *transducer* converts data from electrical to other forms of energy.
A *buffer*, associated with the transducer, temporarily holds data for transfer

Keyboard / monitor
Each character is associated with a code (7 or 8 bits). IRA (International Reference Alphabet) is the most commonly used text code, with 128 different characters.
Printable characters: alphabetic, numeric, and special characters
Control characters: can have to do with controlling printing / displaying (E.g. carriage return) or with communications procedures
Keyboard input:

1. When you press a key, an electronic signal is generated
2. The signal is interpreted by the transducer in the keyboard and translated into the bit pattern of the corresponding IRA code
3. This bit pattern is then transmitted to the I/O module

Output:

1. IRA code characters are transmitted to an external device from the I/O module
2. The transducer at the device interprets this code and sends the required electronic signals to the output device (for e.g. display)

Disk drive
A disk drive contains electronics for exchanging data, control, and status signals with an I/O module plus the electronics for controlling the disk read/write mechanism.
Fixed-head disk:
The transducer can convert between the magnetic patterns on the moving disk surface and bits in the device's buffer.
Moving-head disk:
Must additionally be able to cause the disk arm to move radially in and out across the disk's surface.

I/O modules

Module function
The major functions for an I/O module fall into these categories:

1. Control and timing

Co-ordinates the flow of traffic between internal resources and external devices.

Steps for transferring data from an external device to the processor:

1. The processor interrogates the I/O module to check the status of the attached device
2. The I/O module returns the device status
3. If the device is ready, the processor requests the transfer of data
4. The I/O module obtains a unit of data from the external device
5. The data are transferred from the I/O module to the processor

2. Processor communication

Involves the following:

- *Command decoding*

  The I/O module accepts commands from the processor, sent as signals on the control bus.

- *Data*

  Data are exchanged between processor and I/O module over the data bus.

- *Status reporting*

  Because peripherals are so slow, it is important to know the status of the I/O module, which can be reported with a status signal.

- *Address recognition*

  An I/O module must recognise one unique address for each peripheral it controls.

3. Device communication

This communication involves commands, status information, and data.

4. Data buffering

Data travelling from main memory to the I/O module:

Data coming from main memory are sent to an I/O module in a rapid burst, because of the high transfer rate. The data are buffered in the I/O module and then sent to the peripheral device at its data rate.

Data travelling from the device to the I/O module:

Data must be buffered so as not to tie up the memory in a slow transfer operation.

The I/O module must be able to operate at both device and memory speeds.

5. Error detection

Classes of errors:

- Mechanical and electrical malfunctions reported by the device (e.g. paper jam)
- Unintentional changes to the bit pattern as it is transmitted from device to I/O module

Some form of error-detecting code is often used to detect transmission errors, e.g. the use of a parity bit on each character of data.

Module structure

Data registers buffer data transferred to and from the module.

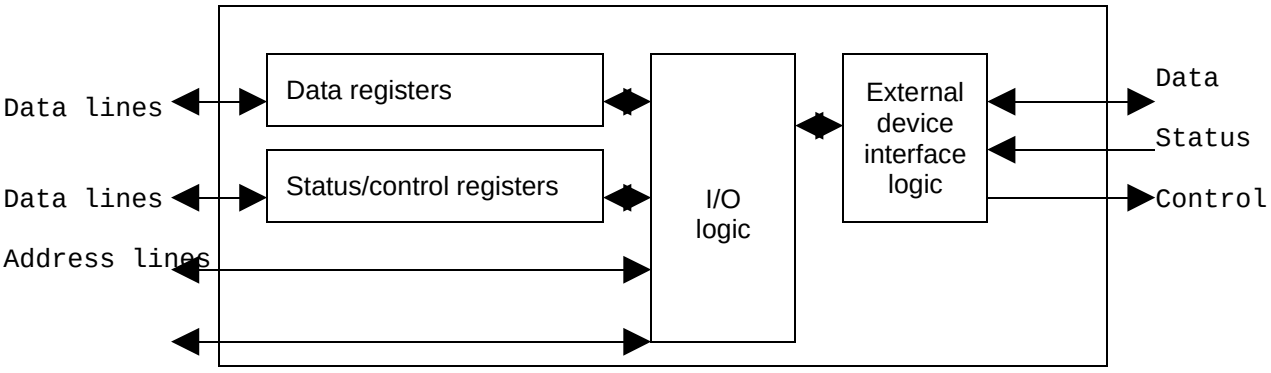
A status register provides current status info, and may also function as a control register, to accept detailed control information from the processor.

Control lines are used by the processor to issue commands to the module.

Address lines are for generating the module's unique address.

The module contains I/O logic specific to the interface with each device that it controls.

Diagram of an I/O module



I/O channel / I/O processor = an I/O module that takes on most of the processing burden, presenting a high-level interface to the processor. (Commonly seen on mainframes).
I/O controller / device controller = a primitive I/O module that requires detailed control. (Commonly seen on microcomputers).

1. Programmed I/O

1. When the processor encounters a program instruction relating to I/O, it issues a command to the appropriate I/O module
2. The I/O module performs the requested action and sets the appropriate bits in the I/O status register
3. (The I/O module takes no further action to alert the processor)
4. The processor periodically checks the status of the I/O module until it finds that the operation is complete

Disadvantage: Programmed I/O is a time-consuming process that keeps the processor busy needlessly. As a result, the level of the performance of the entire system is severely degraded.

I/O commands (issued by the processor)
There are four types of I/O commands that an I/O module may receive:

- *Control*
  Used to activate a peripheral and tell it what to do (like rewind)
- *Test*
  Used to test various status conditions (like if the peripheral is available)
- *Read*
  Causes the I/O module to obtain an item of data from the peripheral and place it in the internal buffer. The processor can then obtain the data item by requesting that the I/O module place it on the data bus.
- *Write*
  Causes the I/O module to take an item of data from the data bus and transmit it to the peripheral.

I/O instructions (executed by the processor)
There is a close correspondence between I/O instructions that the processor fetches from memory and the I/O commands that the processor issues to an I/O module to execute the instructions.
Each device is given a unique address. When the processor issues an I/O command, the I/O module must interpret the address lines to determine if the command is for itself.
When the processor, main memory, and I/O share a common bus, two modes of addressing are possible:
a) Memory-mapped I/O
There is a *single address space* for memory locations and I/O devices.
The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both memory and I/O devices.
A single read line and a single write line are needed on the bus.
Advantage: Larger repertoire of instructions allows more efficient programming.
Disadvantage: Valuable memory address space is used up.
b) Isolated I/O
The address space for I/O is *isolated* from that for memory.
The bus has memory read & write plus input & output command lines. The command line specifies whether the address refers to a memory location or an I/O device.

2. Interrupt-driven I/O

1. The processor issues an I/O command to a module and then goes on to do some other useful work
2. The I/O module will interrupt the processor to request service when it is ready to exchange data with the processor
3. The processor then executes the data transfer, and resumes its former processing

Advantage: More efficient than programmed I/O because it eliminates needless waiting.
Disadvantage: Still consumes a lot of processor time, because every word of data that moves between memory and I/O module must pass through the processor.

Interrupt processing
When an I/O device completes an I/O operation, the following sequence of events occur:

1. The device issues an interrupt signal to the processor
2. The processor finishes execution of the current instruction
3. The processor tests for an interrupt, and sends an acknowledgement signal to the device that issued the interrupt
4. The processor prepares transferring control to the interrupt routine by saving information needed to resume the current program at the point of the interrupt. (The PSW and PC are pushed onto the control stack)
5. The processor now loads the new PC value based on the interrupt
6. The rest of the 'state' information of the interrupted program must be saved (like the contents of the processor registers) on the stack
7. The interrupt handler then processes the interrupt
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers
9. Finally the PSW and PC values from the stack are restored, so the next instruction to be executed will be from the previously interrupted program

Design issues
Device identification

- *Multiple interrupt lines*

It is impractical to dedicate more than a few bus lines to interrupt lines, so even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it, so one of the other techniques must be used on each line.

- *Software poll*
  1. When the processor detects an interrupt, it branches to an interrupt-service routine which polls each I/O module to determine which module caused the interrupt
  2. Once the correct module is identified, the processor branches to a device-service routine specific to that device.

Disadvantage: time-consuming.

- *Daisy chain (hardware poll, vectored interrupt)*

All I/O modules share a common interrupt request line. The interrupt acknowledge line is daisy chained through the modules.

1. When the processor senses an interrupt, it sends out an interrupt acknowledge
2. This signal passes through a series of I/O modules until it gets to a requesting module
3. The requesting module responds by placing a word on the data lines
4. (The word (vector) is a unique id, like the I/O module's address)
5. The processor uses the vector as a pointer to the appropriate device-service routine

Advantages: more efficient than software polls, and avoids the need to execute a general interrupt-service routine first.

- *Bus arbitration (vectored)*

An I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the line at a time.

1. When the processor detects the interrupt, it responds on the interrupt acknowledge line
2. The requesting module then places its vector on the data lines

Order of processing

The above techniques also provide a way of assigning priorities when more than one device is requesting interrupt service.
With multiple lines, the processor just picks the interrupt line with the highest priority.
With software polling, the order in which the modules are polled determines their priority.
Bus arbitration can also employ a priority scheme.

3. Direct memory access

Drawbacks of programmed and interrupt-driven I/O:

1. The I/O transfer rate is *limited by the speed* with which the processor can test and service a device
2. The *processor is tied up* in managing an I/O transfer (a number of instructions must be executed for each I/O transfer)

When large volumes of data need to be moved, DMA is more efficient.

DMA function

A DMA module is added to the system bus, which is capable of mimicking the processor and taking over control of the system for the processor. It needs to do this to transfer data to and from memory over the system bus. The DMA module must either use the bus only when the processor doesn't need it, or force the processor to suspend operation temporarily (= cycle stealing).

1. When the processor wants to read / write a block of data, it issues a command to the DMA module, with the following info:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read from / write to
- The number of words to be read / written

2. (The processor can then continue with other work because it has delegated this I/O operation to the DMA module)
3. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.
4. When the transfer is complete, the DMA module sends an interrupt signal to the processor

The processor is involved only at the beginning and end of the transfer.

I/O channels and processors

Evolution of the I/O function

1. The CPU directly controls a peripheral device
2. A controller or I/O module is added (no interrupts yet)
3. Same as step 2, but with interrupts now
4. The I/O module is given direct access to memory via DMA
5. The I/O module is enhanced to become a processor in its own right
6. The I/O module has a local memory and is now a computer in its own right

As you go down the evolutionary path, more and more of the I/O function is performed without CPU involvement, improving performance.

Characteristics of I/O channels

I/O channel = an I/O module as in steps 5 and 6 above.
I/O channels can execute I/O instructions and have complete control over I/O operations. The CPU just instructs the I/O channel to execute a program in memory, giving it the device, area of memory, priority, etc.
Two types of I/O channels:

1. Selector channel
Controls multiple high-speed devices and, at any one time, is dedicated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and effects the data transfer. Each device is handled by a *controller* (I/O module). Thus, the I/O channel serves in place of the CPU in controlling these I/O modules (controllers).
2. Multiplexor channel
Can handle I/O with multiple devices at the same time. For low-speed devices, a *byte multiplexor* accepts / transmits characters as fast as possible to multiple devices. For high-speed devices, a *block multiplexor* interleaves blocks of data from several devices.

The external interface: FireWire and InfiniBand

Types of interfaces
The interface to a peripheral from an I/O module must be tailored to the nature and operation of the peripheral.
Parallel interface:
There are multiple lines connecting the I/O module and the peripheral, and multiple bits are transferred simultaneously.
Traditionally used for higher-speed peripherals (like tape and disk).
Serial interface:
There is only one line used to transmit data, and bits must be transmitted one at a time.
Traditionally used for printers and terminals.

Dialogue between an I/O module and peripheral, for a write operation:
1. The I/O module sends a control signal requesting permission to send data
2. The peripheral acknowledges the request
3. The I/O module transfers data (one word or block, depending on the peripheral)
4. The peripheral acknowledges receipt of the data

An I/O module has a buffer that can store data being passed between the peripheral and the rest of the system. The buffer allows the I/O module to compensate for the differences in speed between the system bus and its external lines.

Point-to-point and multipoint configurations
Point-to-point:
A point-to-point interface provides a dedicated line between the I/O module and the external device.
On small systems, typical point-to-point links include those to the keyboard, printer, and external modem.
Multipoint:
Multipoint external interfaces support external mass storage devices (disk and tape drives) and multimedia devices (CD-ROMs, video, audio).
These multipoint interfaces are in effect external buses.
E.g. FireWire and InfiniBand.

FireWire serial bus
The high-speed I/O channel technologies developed for mainframe and supercomputer systems are too expensive and bulky for use on smaller systems. FireWire (a high-performance serial bus) was developed as a high-speed alternative to SCSI and other small-system I/O interfaces.

Advantages of FireWire over older I/O interfaces:
- Very high speed
- Low cost
- Easy to implement

FireWire is finding favour not only for computer systems, but also in digital cameras, VCRs etc. where it is used to transport images.

One of FireWire's strengths is that it uses serial transmission rather than parallel. (Parallel interfaces, like SCSI, require more wires so it's more expensive and synchronisation between wires can be a problem).

Computers are getting physically smaller. Handheld computers have little room for connectors, yet need high data rates to handle images and video.

The intent of FireWire is to *provide a single I/O interface* with a simple connector that can handle numerous devices through a single port, so that the mouse, laser printer, external disk drive, LAN hookups, etc. can be replaced with this single connector.

FireWire configurations

FireWire uses a daisy-chain configuration, with up to 63 devices connected off a single port. Up to 1022 FireWire buses can be interconnected using bridges, enabling a system to support as many peripherals as required.
FireWire provides for *hot plugging* (You can connect peripherals without switching the computer off).
FireWire provides for *automatic configuration* (You don't need to manually set device IDs).

The FireWire standard specifies a set of 3 layers of protocols to standardise the way in which the host system interacts with the peripheral devices over the serial bus:
1. Physical layer
Defines the transmission media that are permissible under FireWire and the electrical & signalling characteristics of each.
2. Link layer
Describes the transmission of data in the form of packets.
3. Transaction layer
Defines a request-response protocol that hides the lower-layer details of FireWire from applications.

InfiniBand

InfiniBand is a recent I/O specification aimed at the high-end server market. The main purpose of InfiniBand is to *improve data flow* between processors and intelligent I/O devices.
InfiniBand is intended to:

- Replace the PCI bus in servers
- Provide greater capacity
- Provide increased expandability
- Provide enhanced flexibility in server design

In essence, InfiniBand eables servers, remote storage, and other network devices to be attached in a central fabric of switches and links. The switch-based architecture can connect up to 64 000 servers, storage systems, and networking devices.

Infiniband architecture

PCI is a limited architecture compared to InfiniBand. With InfiniBand, you don't have to have the basic I/O interface hardware inside the server chassis - remote storage, networking, and connections between servers are accomplished by attaching all devices to a central fabric of switches and links. Removing I/O from the server chassis allows greater server density and allows for a more flexible and scalable data centre, as independent nodes may be added as needed.

Infiniband operation

Each physical link between a switch and an attached interface can support up to 16 logical channels, called *virtual lanes*. One lane is reserved for fabric management and the other lanes for data transport. Data are sent in the form of a stream of packets, with each packet containing a portion of the total data to be transferred, plus addressing and control information. A set of communications protocols is used to manage the transfer of data. A virtual lane is temporarily dedicated to the transfer of data from one end node to another

over the InfiniBand fabric. The InfiniBand switch maps traffic from an incoming lane to an outgoing lane to route the data between the desired end points.

Chapter 9 - Computer arithmetic

ALU inputs:
The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.
The registers present data to the ALU.
ALU outputs:
The results of an operation are stored in registers.
The ALU may set flags as the result of an operation.

Integer representation
Sign-Magnitude representation
The left-most bit represents the sign (1 = -, 0 = +).
Disadvantages:

- Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes
- There are two representations of 0

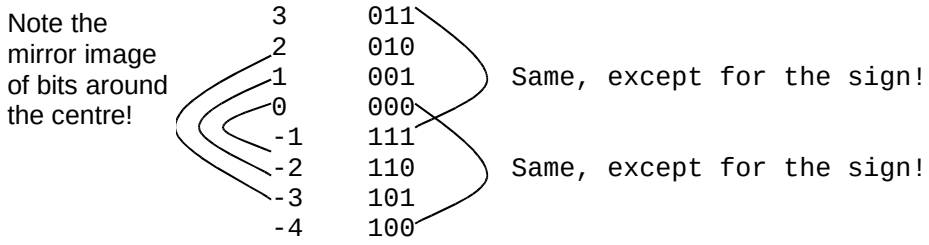
When changing a number to a greater bit length, just move the sign-bit to the new leftmost position and fill in with zeros.
E.g. 1010 → 10000010

Twos complement representation
The left-most bit is a sign-bit (1 = -, 0 = +), but it is *part* of the number, which makes arithmetic easy. There is also only one representation of 0.
When changing a number to a greater bit length, move the sign bit to the new leftmost position and fill in with copies of the sign bit.
E.g. 1010 → 11111010

Fixed-point representation
The radix point (period) is fixed and assumed to be to the right of the rightmost digit. You can use the same number representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at another location.

Integer arithmetic
Negation
For sign-magnitude representation, just invert the sign bit.

For twos complement representation, do the following:
1. Invert all the bits (i.e. take the Boolean complement)
2. Add 1



The reason why you have to add 1 when you negate a number is because the numbers aren't centred around zero.
Note that the numbers range from -2^{n-1} to $2^{n-1}-1$, where n is the number of bits in use (3 in the above example). In total, there are 2^n different numbers.

Addition
In twos complement representation, just add the two numbers and ignore the carry bit (if any).

Overflow = when the result is larger than can be held in the word size. You know there is an overflow if you add two numbers with the same sign and the result has the opposite sign.

Subtraction
In two's complement representation, just take the twos complement of the subtrahend, and add both numbers together.

Floating-point representation

Floating-point numbers are expressed as a significand (fractional number) multiplied by the base B (i.e. 2) to the power E. This B^E is what moves the decimal point right (if E is positive) or left (if E is negative).

General format:

Sign of significand (0/1)	Biased exponent (e)	Significand (S) or Mantissa

Note: the exponent E is given in a *biased* representation: a fixed value, called the bias, is subtracted from the field (e) to get the true exponent value.
The bias is worked out from the number of bits allocated to the biased exponent (this varies with the different standards).
Advantage of biased representation: Nonnegative floating-point numbers can be treated as integers for comparison purposes.

Floating-point numbers must be *normalised*, which just means putting the number in a different format. This is done by shifting the radix point and adjusting the exponent accordingly. (This varies with the different standards).

Note: The more bits allocated to the exponent, the larger the *range* of expressible numbers (but the number of different values isn't increased!). The more bits allocated to the significand, the greater the *precision*.

Examples:

Give the representation of -6.6875 in floating-point format

Step 1. Convert the number into binary.
The integer part can be worked out as follows:
Repeatedly *divide* the number by 2, keeping track of the remainder (ignore the sign for the moment).
6 / 2 = 3 remainder 0 ↑
3 / 2 = 1 remainder 1 |
1 / 0 = 0 remainder 1
The number (from left to right) is the concatenation of the remainders (from bottom to top), i.e. 110.

The fractional part can be worked out as follows:
Repeatedly *multiply* the fraction by 2 (until you get nothing to the right of the decimal point), keeping track of the number that is found to the left of the decimal point, and subtracting it with each multiplication.
0.6875 * 2 = **1**.375 |
0.375 * 2 = **0**.75 ↓
0.75 * 2 = **1**.5
0.5 * 2 = **1**.0
The number (from left to right) is the concatenation of the digits to the left of the decimal point (from top to bottom), i.e. 1011.

So, 6.6875 in binary is 110.1011

Step 2. Normalise the number.
IEEE format:
Shift the radix point until there is a **1** to the **left** of it.

110.1011 = 1.101011 * 2^2 (The decimal point was shifted left 2 places)
Remember to *exclude* the bit to the left of the decimal when working out f!

IBM 360/370 format:
Similar to the IEEE format, but you have to shift the binary point by **4** places at a time because **base 16** is used (0000 → 1111 = 0 → 15)
110.1011 = 0.01101011 * 16^1

DEC PDP 11/VAX format:
Similar to the IEEE format, but shift the radix point until there is a **1** to the **right** of it.
110.1011 = 0.1101011 * 2^3
Remember to *exclude* the bit to the right of the decimal when working out f!

Step 3. Get the value of e.
e (the biased exponent) is a large number, from which you have to subtract the bias to get the true exponent. To work out e, just add the exponent from step 2 to the bias!

IEEE format:

sign bit s	e (8 bits)	fraction f(23 bits)

The bias = 2^{k-1} - 1, where k is the number of bits in the binary exponent.
Because the standard reserves 8 bits for the exponent, the bias = **127**.

The exponent from step 2 was 2, so e = 127 + 2 = 129.
i.e. the number equals 1.101011 * $2^{129-127}$

IBM 360/370 format:

sign bit s	e (7 bits)	fraction f(24 bits)

The bias = 2^{k-1}, where k is the number of bits in the binary exponent.
Because the standard reserves 7 bits for the exponent, the bias = **64**.

The exponent from step 2 was 1, so e = 64 + 1 = 65.
i.e. the number equals 0.01101011 * 16^{65-64}

DEC PDP 11/VAX format:

sign bit s	e (7 bits)	fraction f(24 bits)

The bias = 2^{k-1}, where k is the number of bits in the binary exponent.
Because the standard reserves 8 bits for the exponent, the bias = **128**.

The exponent from step 2 was 3, so e = 128 + 3 = 131
i.e. the number equals 0.1101011 * $2^{131-128}$

Step 4. convert e to binary.
IEEE format:
129 = 10000001

IBM 360/370 format:
65 = 1000001

DEC PDP 11/VAX format:
131 = 10000011

Step 5. Use the sign, exponent, and fraction to represent the number.
IEEE format:

1 10000001 10101100000000000000000

IBM 360/370 format:
1 1000001 01101011000000000000000

DEC PDP 11/VAX format:
1 10000011 10101100000000000000000

Chapter 10 - Instruction sets

Machine instruction characteristics

Instruction set = the collection of different instructions that can be executed.

Elements of a machine instruction

- *Operation code (opcode)*
  Specifies the operation to be performed. E.g. ADD
- *Source operand reference*
  The input operand for the operation
- *Result operand reference*
  The operation may produce a result
- *Next instruction reference*
  This tells the CPU where to fetch the next instruction

Usually, if there is no explicit reference to the next instruction, the following one is fetched.
Three areas for the *source* and *result* operands:

- Main / virtual memory
- CPU register
- I/O device

Instruction representation
Each instruction is represented by a sequence of bits, and is divided into fields, e.g:

Opcode 4 bits **MOV*	Operand reference 6 bits **BX*	Operand reference 6 bits **AX*

The instruction is read into an instruction register (IR) in the CPU. The CPU extracts the data from the various instruction fields to perform the required operation.

Instruction types

- Data processing (Arithmetic and logic instructions)
- Data storage (Memory instructions)
- Data movement (I/O instructions)
- Control (Test and branch instructions)

Number of addresses
Most instructions have one or two operand addresses, with the address of the next instruction being implicit (obtained from the program counter).
With one-address instructions, the second address is implicit (the accumulator). Zero-address instructions reference the stack.

Fewer operand addresses → more primitive instructions → less complex CPU → shorter instruction length → more total instructions → longer execution times → longer, more complex programs

With one-address instructions, you only have one general-purpose register (the accumulator), but with multiple-address instructions, you have multiple general-purpose registers. This allows some operations to be performed solely on registers. Because *register references are faster* than memory references, this speeds up execution.

Instruction set design
Fundamental design issues:

- Operation repertoire (How many and which operations to provide)
- Data types (Different types of data upon which operations are performed)
- Instruction format (Length, number of addresses, size of fields, etc.)
- Registers (Number of CPU registers that can be referenced)
- Addressing (The mode(s) by which the address of an operand is specified)

Types of operands

Machine instructions operate on data. The most important general categories of data are:

1. Addresses
Addresses can be considered as unsigned integers, and can be used in calculations E.g. to determine the main memory address.

2. Numbers
Three types of numerical data are common:

- Integer / fixed point
- Floating point
- Decimal

Human users deal with decimal numbers, so there is a need to convert from decimal to binary & vice versa. If there is a lot of I/O, and not much computation, it is preferable to store and operate on the numbers in decimal form. Packed decimal representation stores each digit as a 4-bit binary number, and the 4-bit codes are strung together (1111 on the left = negative).

3. Characters
The most commonly-used character code is ASCII, which represents 128 different characters, in 7 bits each.
EBCDIC is used in IBM S/390 machines, and is an 8-bit code.

4. Logical data
It can be useful to see data as a string of bits, with the values 0 and 1.
Advantages of the bit-oriented view:

- You can store an array of Boolean data items (true & false)
- You can manipulate the bits of a data item (like shift the bits)

Note: The same data can be treated as sometimes logical and other times as numerical or text. The ‘type’ of a unit of data is determined by the operation being performed on it.

Data types

The Pentium can deal with data types of 8 (byte), 16 (word), 32 (doubleword), and 64 (quadword) bits in length. To allow maximum flexibility in data structures and efficient memory utilisation, words need not be aligned at even-numbered adddresses.
The Pentium uses the little-endian style: the least significant type is stored in the lowest address.

Types of operations

The same general types of operations are found on all machines:

1. Data transfer

- The location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack.
- The length of data to be transferred must also be indicated.
- The mode of addressing for each operand must also be specified

If both source and destination are *registers*, then the CPU simply causes data to be transferred from one register to another; this is an operation internal to the CPU.
If one or both operands are in *memory*, then the CPU must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode
2. If the address refers to virtual memory, translate from virtual to actual memory address
3. Determine whether the addressed item is in cache
4. If not, issue a command to the memory module

2. Arithmetic
Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. Other possible operations include: Absolute, negate, increment, and decrement. The ALU portion of the CPU performs the desired operation.

3. Logical
NOT, AND, OR, and XOR are the most common logical functions.
Bit shifting:

- With a logical shift, the bits of a word are shifted left or right. On one end, the bit shifted out is lost.
- With an arithmetic shift, the data is treated as a signed integer so the sign bit isn't shifted. On a right shift, the sign bit is replicated, and on a left shift, the shift is performed on all bits but the sign bit.
- With a rotate, all the bits being operated on are preserved because the shift is cyclic.

4. Conversion
Conversion instructions are those that change the format or operate on the format of data. (E.g. converting from decimal to binary or converting one 8-bit code to another).

5. I/O
Variety of approaches: isolated programmed I/O, memory-mapped programmed I/O, DMA, and the use of an I/O processor.

6. System control
These instructions can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the use of the OS. (E.g. a system control instruction may read / alter a control register).

7. Transfer of control
These instructions change the sequence of instruction execution. The CPU must update the program counter to contain the address of some instruction in memory.
Some reasons why transfer-of-control instructions are required:

- Some instructions need to be executed multiple times
- Virtually all programs involve some decision making
- Programming is made simpler if you can break up a task into procedures

The most common transfer-of control operations:
Branch instructions
Conditional or unconditional branches (jne / jmp) can be used to create a repeating loop of instructions.
Skip instructions
The skip instruction includes an implied address (Typically the next address is skipped).
Procedure call instructions
The two main reasons for using procedures are economy and modularity.
Two basic instructions are involved: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called.

There are three common places for storing the return address:

- Register
- Start of called procedure
- Top of stack

Parameters can be passed in registers, or be stored in memory just after the CALL instruction. The best way to pass parameters is by using the stack.

Operation types

Call/return instructions
The Pentium provides four instructions to support procedure call/return: CALL, ENTER, LEAVE, RETURN.

Memory management
A set of specialised instructions deals with memory segmentation. These are privileged instructions that can only be executed from the OS.

Condition codes
Condition codes are bits in special registers that may be set by certain operations and used in conditional branch instructions. These conditions are set by arithmetic and compare operations.

Assembly language

Programs written in assembly language are translated into machine language by an assembler.
'db' and 'dw' are *assembler directives* that don't form part of the program code that will be executed, but just tell the assembler where and how to reserve memory and how it should be initialised.

Stacks

Stack pointer: Contains the address of the top of the stack
Stack base: Contains the address of the bottom of the stack
Stack limit: Contains the address of the end of the reserved block

The stack grows from *higher* addresses to *lower* addresses.
To speed up stack operations, the top two stack elements are often stored in registers, and the stack pointer contains the address of the third element.

Expression evaluation
Postfix / reverse Polish notation:
a + b = ab+
(a + b) * c = ab + c *
a + (b * c) = abc * +
How it works: When reading a postfix expression from left to right, as soon as you have two variables, followed by an operator, do the calculation and replace those three items with the result, then proceed.
(You can do the same with the stack by pushing variables and popping the top two when you get an operator and then pushing the result back on).

Little- big- and bi-endian

Byte ordering
Big endian: Bytes are stored from left to right.
Little endian: Bytes are stored from right to left.

E.g. 12345678

Address	**100**	**101**	**102**	**103**
Big-endian value	12	34	56	78
Little-endian value	78	56	34	12

Advantages of the big-endian style:

- A big-endian processor is faster in comparing character strings
- Values can be printed left to right without causing confusion
- Big-endian processors store their integers and character strings in the same order (most significant byte comes first).

Advantages of the little-endian style:

- A big-endian processor has to perform addition when converting a 32-bit integer address to a 16-bit integer address, to use the least significant bytes
- It is easier to perform higher-precision arithmetic with the little-endian style because you don't have to find the least-significant byte and move backwards

The bi-endian architecture enables software developers to choose either mode when migrating operating systems and applications from other machines.

Chapter 11 - Addressing modes and instruction formats

Addressing

Immediate addressing
The operand (a value)is present in the instruction.
E.g. mov ax,1
This mode can be used to define and use constants or set initial values of variables.
Advantage: No memory reference other than the instruction fetch is required to obtain the operand, saving one memory cycle.
Disadvantage: The size of the number is restricted to the size of the address field, (which is small compared with word length).

Direct addressing
The address field contains the effective address of the operand.
E.g. mov ax,[102h], or mov [110h],bx
Not common on contemporary architectures.
Advantage: Requires only one memory reference and no special calculation.
Disadvantage: Provides only a limited address space.

Indirect addressing
The address field refers to the address of a word in memory, which contains a full-length address of the operand.
Advantage: For a word length of N, an address space of 2^N is now available.
Disadvantage: Instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

Register addressing
Similar to direct addressing, but the address field refers to a register rather than a main memory address.
E.g. mov ax,bx
Advantage: Only a small address field is needed in the instruction.
Advantage: No memory references are required.
Disadvantage: The address space is very limited.

Register indirect addressing
Similar to indirect addressing, but the address field refers to a register rather than a memory location.
E.g. mov dl,[bx]
Same advantages and disadvantages as for indirect addressing.
On the Pentium, register indirect addressing is only possible with SI, DI, BP, and BX.
Two forms of register indirect addressing:

Indexed addressing
You can use SI to index array elements.

```
E.g.  mov si,array
      add si,3          ; now si indexes the fourth element
      mov al,[si]       ; (extract the contents pointed to by si)
```

Base-indexed addressing
This can also be used to index array elements, but here BP or BX are used as base registers. The base register is normally set to the start of the array and the index is used as an offset into the array.
E.g. add al,[bx+si] or mov dx,[m_addr+si]

Displacement addressing
Combines the capabilities of direct addressing and register indirect addressing. The instruction must have two address fields, at least one of which is explicit. The value contained in one *address field* is used directly. The other address field refers to a *register* whose contents are added to the value to produce the effective address.
Three common uses of displacement addressing:
Relative addressing
The *implicitly referenced register* is the program counter (PC). I.e. The current instruction address is added to the *address field* to produce the effective address.
Base-register addressing
The *referenced register* contains a memory address, and the *address field* contains a displacement from that address. (The register reference may be explicit or implicit).
Indexing
The *address field* references a main memory address, and the *referenced register* contains a positive displacement from that address. (This is the opposite of base-register addressing).

Stack addressing
A pointer is associated with the stack whose value is the address of the top of the stack. If the top two elements of the stack are in CPU registers, the stack pointer references the third element. The stack pointer is maintained in a register, so references to stack locations in memory are in fact *register indirect addresses.* The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.
E.g. pop ax
Note: The stack ‘grows backwards’ in memory, so after a push, SP is decremented, and after a pop, SP is incremented. (E.g. if SP = FFFC, after PUSH AX, SP = FFFA)

Pentium addressing modes

The Pentium has a variety of addressing modes to allow efficient execution of high-level languages.
The segment register (there are 6 of them, including: DS - data, ES - extra, SS - stack, and CS - code) determines the segment that is the subject of the reference.
The base register (BX) and index register (SI / DI) may be used in constructing an address.

Immediate mode
The operand (byte, word, or doubleword of data) is included in the instruction.

Register operand mode
The operand is located in a register (16-bit general registers, like AX, BX... or 8-bit registers, like ah, al...).

The following addressing modes reference locations in *memory*:

Displacement mode
The operand's offset is contained as part of the instruction. With segmentation, all addresses in instructions refer merely to an offset in a segment.

The remaining addressing modes are indirect:
Base mode
Specifies that one of the registers contain the effective address (Equivalent to register indirect addressing).

Base with displacement mode
The instruction includes a displacement to be added to a base register, which may be any of the general-purpose registers.

Scaled index with displacement mode
The instruction includes a displacement to be added to an index register.

Base with index and displacement mode
Sums the contents of the base register, the index register, and a displacement to form the effective address.

Based scaled index with displacement mode
Sums the contents of the index register multiplied by a scaling factor, the contents of the base register, and the displacement.

Relative addressing
A displacement is added to the value of the program counter, which points to the next instruction. The displacement is treated as a signed byte, word, or doubleword value, and that value either increases or decreases the address in the program counter.

Instruction formats

An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. The format must, implicitly or explicitly, indicate the addressing mode for each operand.

Instruction length
1. There is a trade-off between the desire for a powerful instruction repertoire and a need to save space:
The more opcodes, operands, addressing modes, and greater address range, the easier it is for the programmer because shorter programs can be written with more flexibility. However, all these things lead to longer instruction lengths, which can be wasteful.
2. The instruction length should be equal to the memory-transfer length (bus length) or one should be a multiple of the other.
3. The instruction length should be a multiple of the character length, which is usually 8 bits, and the length of fixed-point numbers.

Allocation of bits
There is a trade-off between the number of opcodes and the power of the addressing capability:
The more opcodes, the more bits in the opcode field.
For an instruction format of a given length, this reduces the number of bits available for addressing. Using variable-length opcodes means there is a minimum opcode length, but for some opcodes, additional operations may be specified by using additional bits in the instruction. (For a fixed-length instruction, this leaves fewer bits for addressing).

Factors that go into determining the use of the addressing bits:

- Number of addressing modes
  Explicit addressing modes require more bits than implicit modes.
- Number of operands
  Each operand in the instruction might require its own mode indicator.

- Register versus memory
  With a single user-visible register (the accumulator), one operand address is implicit and consumes no instruction bits.
  Even with multiple registers, only a few bits are needed to specify the register. (The more that registers can be used for operand references, the fewer bits that are needed).
- Number of register sets
  Most machines have one set of general-purpose registers, with 32 / more registers in the set. The Pentium has several specialised sets.
  Advantage: For certain registers, a functional split requires fewer bits to be used in the instruction.
- Address range
  For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used.
- Address granularity
  For addresses that reference memory rather than registers, another factor is the granularity of addressing. An address can reference a word or a byte (with byte-addressing requiring more address bits).

Variable-length instructions
If the designer provides a variety of instruction formats of different lengths, it makes it easier to provide a large repertoire of opcodes, with different opcode lengths. Addressing can also be made more flexible, with various combinations of register and memory references plus addressing modes.
Disadvantage: Increased CPU complexity.
Because the CPU doesn't know the length of the next instruction to be fetched, it fetches a number of bytes / words equal to at least the longest possible instruction. This means that sometimes multiple instructions are fetched.

Chapter 12 - CPU structure and function

Processor organisation

The CPU must: *Fetch instructions* from memory, *interpret* them, *fetch data* from memory or an I/O module, *process* the data, and then *write* data to memory or an I/O module.
The ALU does the actual computation / processing of data.
The control unit controls the movement of data and instructions into and out of the CPU and controls the operation of the ALU.
The registers serve as a minimal internal memory.
Within the CPU, an *internal CPU bus* transfers data between the registers and the ALU.

Register organisation

User-visible registers
Some design issues:

- Should you use completely general-purpose registers, or specialise their use?
  Advantage: Specialised registers can have implicit opcodes, saving bits.
  Disadvantage: Specialisation limits the programmer's flexibility.
- How many registers (general purpose or data + address) should be provided?
  Disadvantage: Fewer registers results in more memory references.
  Disadvantage: More registers require more operand specifier bits.
- How long should the registers be?
  Registers that hold addresses must be long enough to hold the largest address.
  Registers that hold data must be able to hold values of most data types.

These are registers that may be referenced by means of machine language.

General purpose registers

These can be assigned to a variety of functions by the programmer.

AX	Primary accumulator Mainly used for operations involving data movement, I/O and arithmetic MUL assumes that AX contains the multiplicand DIV assumes that AX contains the dividend
BX	Base register This is the only general-purpose register that can be used as a pointer Also used for arithmetic
CX	Count register Used to control the number of times loops are to be executed or the number of shifts to perform Also used for arithmetic
DX	Data register Some I/O operations like In and OUT use the DX register Multiply & divide operations involving 16-bit registers use DX:AX

Data registers

These may be used only to hold data and can't be employed in the calculation of an operand address.

Address registers

These may be general-purpose, or may be devoted to a particular addressing mode, e.g:

** Segment pointers*

A segment register holds the address of the base of the segment.
Each segment in memory is 64K bytes long. A segment begins on a paragraph boundary that is a multiple of 16 (i.e. 10h). Since the start address of a segment always ends with 0 in hex, it is unnecessary to store the last digit. (E.g. The address of a segment starting at 18A30h is stored as 18A3h and can be written as 18A3[0]h).
A program running under DOS is divided into three primary segments:

- Code segment (CS)
  Contains the machine instructions of the program.
  All references to memory locations that contain instructions are *relative* to the start of a segment specified by the CS register.
  segment:offset references a byte of memory (offset = 0→FFFF / 64K-1).
  The IP register contains the offset (relative to the start of the segment) of the next instruction to be executed, so CS:IP forms the actual 20-bit address of the next instruction (= effective address).
  E.g. BCEF:0123 (CS contains BCEFh and IP contains 123h)
  Add the segment address BCEF**0** and the offset 0123 to get the actual (effective) address BD013.
  The IP register can't be referenced directly by a programmer, but it can be changed indirectly with JMP instructions.
- Data segment (DS)
  Contains the variables, constants and work areas of a program.
  With 'MOV AL,[120h]', the instruction at location 120h *relative* to the contents of the DS register is fetched.
  To work out the actual address from where the byte of data will be moved, do the same calculation as above (i.e. append 0 to the data segment address and add it to the given address).
- Stack segment (SS)
  Contains the program stack, which is used to save data and addresses that need to be temporarily stored.

** Index registers*

These are used for indexed addressing and may be autoindexed.
Index registers contain the offset, relative to the start of the segment, for variables.
SI (source index) usually contains an offset value from the DS register, but it can address any variable.
DI (destination index) usually contains an offset from the ES register but can address any variable.

SI and DI registers are available for extended addressing and for use in addition and subtraction. They are required for some string operations.

** Stack pointer*

If there is user-visible stack addressing, then the stack is in memory and there is a dedicated register that points to the top of the stack. This allows implicit addressing (i.e. push and pop don't need to contain an explicit stack operand).

The stack is located in the stack segment (SS) and the stack pointer (SP) register holds the address of the last element that was pushed on. (SP contains the *offset* from the beginning of the stack to the top of the stack).

SS:SP contains the address of the top-of-the-stack.

The BP (base pointer) register contains an offset from the SS register, and facilitates the referencing of parameters (data & addresses passed via the stack). Normally the only word in the stack that is accessed is the one on top. However, the BP register can also keep an offset in the SS and be used in procedure calls, especially when parameters are passed to subroutines. SS:BP contains the address of the current word being processed in the stack.

Condition codes (flags)

Condition codes are bits set by the CPU as a result of operations.

The code may be tested as part of a conditional branch operation, but cannot be altered by the programmer.

Flag	**Debug representation**	**Description**
CF: Carry Flag	CY = CarrY NC = No Carry	Contains 'carries' from the high-order bit following arithmetic operations and some shift & rotate operations.
PF: Parity Flag	PE = Parity Even PO = Parity Odd	Checks the low-order eight bits of data operations. Odd = 0, Even = 1
AF: Auxiliary Flag	AC = Auxilliary Carry NA = No Auxilliary	Set to 1 if arithmetic causes a carry.
ZF: Zero Flag	NZ = Not Zero ZR = ZeRo	Set as a result of arithmetic / compare operations. 0 = no, 1 = yes (= zero result). JE and JZ test this flag.
SF: Sign Flag	PL = PLus NG = NeGative	Set according to the sign after an arithmetic operation (0=+, 1=-). JG and JL test this flag.
TF: Trap Flag	(Not shown in Debug)	Debug sets the trap flag to 1 so that you can step through execution one instruction at a time. (Use the 't' command).
IF: Interrupt Flag	EI = Enable Interrupts DI = Disable Interrupt	Indicates if interrupts are disabled. 0 = disabled, 1 = enabled.
DF: Direction Flag	UP = UP (right) DN = DowN (left)	Used by string operations to determine the direction of data transfer. 0: left-to-right data transfer 1: right-to-left data transfer
OF: Overflow Flag	NV = No Overflow OV = OVerflow	Indicates a carry into and out of the high-order sign bit following a signed arithmetic operation.

Control and status registers

Some design issues:

- Certain types of control information are of specific utility to the OS, so the designer must understand the OS and tailor the register organisation to it
- The allocation of control information between registers and memory: It is common to dedicate the first (lowest) few hundred words of memory for control purposes. The designer must decide how much control info should be in registers and how much in memory

Program counter (PC) - contains the address of an instruction to be fetched
Instruction register (IR) - contains the instruction most recently fetched
Memory address register (MAR) - contains the address of a location in memory
Memory buffer register (MBR) - contains a word of data to be written to memory or the word most recently read
The CPU updates the PC after each instruction fetch so the PC always points to the next instruction to be executed. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analysed. Data are exchanged with memory using the MAR and MBR. The MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers exchange data with the MBR.
Within the CPU, data must be presented to the ALU for processing. (The ALU may have direct access to the MBR and user-visible registers, or there may be additional buffering registers at the boundary to the ALU).

Program status word (PSW) = a set of registers that contain status information
Common flags include the ones mentioned in the table above.

Example microprocessor register organisation

<table>
<tr><th colspan="2">Intel 8086</th><th>Motororola MC6800</th></tr>
<tr><td colspan="2"><table>
<tr><th>General registers</th><th>Pointer & index</th></tr>
<tr><td>AX - accumulator</td><td>SP -stack pointer</td></tr>
<tr><td>BX - base</td><td>BP - base pointer</td></tr>
<tr><td>CX - count</td><td>SI - source index</td></tr>
<tr><td>DX - data</td><td>DI - dest index</td></tr>
<tr><th>Segment</th><th>Program status</th></tr>
<tr><td>CS - code</td><td>Instr Ptr</td></tr>
<tr><td>DS - data</td><td>Flags</td></tr>
<tr><td>SS - stack</td><td></td></tr>
<tr><td>ES - extra</td><td></td></tr>
</table></td><td>8 data registers
9 address registers
Program status registers (PC & status)</td></tr>
<tr><td colspan="2">Every register is special-purpose, but some can be used for general purposes</td><td>Regular instruction set, with no special-purpose registers</td></tr>
</table>

Instruction cycle

The main subcycles of the instruction cycle are: Fetch, Execute, Interrupt.

The indirect cycle
This is a subcycle for when indirect addressing is used.
The revised instruction cycle:

1. The address of the next instruction to be executed is determined
2. The processor fetches the instruction from memory
3. The fetched instruction is loaded into the IR
4. The control unit interprets the instruction
5. The address(es) of the operand(s) are determined
6. The operand(s) are fetched from memory *USING INDIRECT ADDRESSING*
7. The processor performs the required operation
8. The result is stored in memory, if required, *USING INDIRECT ADDRESSING*
9. Interrupt processing takes place if an interrupt has occurred

Data flow
Fetch cycle:

1. The PC contains the address of the next instruction to be fetched
2. This address is moved to the MAR and placed on the address bus
3. The control unit requests a memory read and the result is placed on the data bus and copied into the MBR and then moved to the IR
4. The PC is incremented by 1, preparing for the next fetch

Indirect cycle:

1. The control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing
2. If it does, the right-most n bits of the MBR (which contain the address reference) are transferred to the MAR

3. The control unit requests a memory read, to get the desired address of the operand into the MBR

Execute cycle:

- This cycle takes many forms, depending on which instruction is in the IR

Interrupt cycle:

1. The current contents of the PC must be saved (transferred to the MBR) so that the CPU can resume normal activity after the interrupt
2. The PC is loaded with the address of the interrupt routine so the next instruction cycle will begin by fetching the appropriate instruction

Instruction pipelining

Instruction prefetch / fetch overlap

A two-stage pipeline:

1. The first stage fetches an instruction and buffers it
2. When the second stage is free, the first stage passes it the buffered instruction
3. While the second stage is executing the instruction, the first stage fetches and buffers the next instruction

Some problems:

- The execution time will generally be longer than the fetch time, so the fetch stage may have to wait before it can empty its buffer
- A conditional branch instruction makes the address of the next instruction to be fetched unknown (but the next instruction can be fetched anyway, just in case)

A multi-stage pipeline:

Instruction processing can be decomposed into these stages:

Fetch Instruction	FI	DI	CO	FO	EI	WO					
Decode Instruction		FI	DI	CO	FO	EI	WO				
Calculate Operands			FI	DI	CO	FO	EI	WO			
Fetch Operands				FI	DI	CO	FO	EI	WO		
Execute Instruction					FI	DI	CO	FO	EI	WO	
Write Operand											

With this decomposition, the various stages will be of more or less equal duration. However, not all instructions will go through all stages.

Some problems (as for 2-stage pipelines):

- If the stages are not of equal duration, there will be some waiting involved
- The conditional branch instruction can invalidate several instruction fetches

Additional problems (not in 2-stage pipelines):

- Register and memory conflicts: E.g. the CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline (so you need control logic)

It may seem that the greater the number of stages in the pipeline, the faster the execution rate, but there are factors that slow things down:

- At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer
- The amount of control logic required to handle memory and register dependencies and to optimise the use of the pipeline increases enormously with the number of pipeline stages

The Pentium processor

Register type	Number	Purpose
Integer unit:		
General	8	General-purpose user registers, like AX, BX
Segment	6	Contain segment selectors, like CS, SS
Flags	1	Status and control bits, like CF (Carry Flag)
Instruction pointer	1	Instruction pointer, IP
Floating-point unit:		
Numeric	8	Hold floating-point numbers
Control	1	Control bits
Status	1	Status bits
Tag word	1	Specifies contents of numeric registers
Instruction pointer	1	Points to instruction interrupted by execution
Data pointer	1	Points to operand interrupted by exception

Interrupt processing

Interrupts and exceptions

Interrupts	Exceptions
Generated by a signal from *hardware*	Generated from *software*
May occur at random times during the execution of a program, e.g. if you press a key on the keyboard	Provoked by the execution of an instruction, e.g. INT 21h
1. Maskable interrupts: The processor doesn't recognise a maksable interrupt unless the interrupt enable flag is set	1. Processor-detected exceptions: Result when the processor encounters an error while attempting to execute an instruction
2. Non-maskable interrupts: Recognition of such interrupts can't be prevented	2. Programmed exceptions: Instructions that generate an exception

Interrupt vector table
Every type of interrupt is assigned a number, which is used to index into the Pentium's interrupt vector table.

Interrupt number	Address	Contents of address
	10	Code segment address
2	8	Offset address
	6	Code segment address
1	4	Offset address
	2	Code segment address
0	0	Offset address

For interrupt type n, the instruction offset is stored in the word at address 4*n and the code segment address in the word at address (4*n)+2.
Each code segment and offset points to its own interrupt handler (interrupt service routine), which is a block of code that executes if that particular interrupt occurs.
If more than one exception or interrupt is pending, the processor services them in a predictable order.
(Priority is not determined by the location of vector numbers within the table)

Interrupt handling
A transfer to an interrupt-handling routine uses the system stack to store the processor state. When an interrupt occurs, the following sequence of events takes place:

1. The stack segment register and extended stack pointer register are pushed onto the stack (if the transfer involves a change of privilege level)
2. The current value of the EFLAGS register is pushed onto the stack
3. The interrupt and trap flags are cleared
4. The current code segment pointer (CS) and IP are pushed onto the stack
5. If the interrupt is accompanied by an error code, the error code is pushed onto the stack

6. The interrupt vector contents are fetched and loaded into the CS and IP registers. Execution continues from the interrupt service routine

To return from an interrupt, all the saved values are restored and execution resumes from the point of the interrupt.

Reverse byte order in memory

The CPU expects numeric data in memory (not registers) to be stored in reverse byte order. E.g. 0015h is stored as: 15h 00
The CPU reverses the bytes again when loading the data from memory into registers.

Chapter 13 - Reduced Instruction Set Computers

The key elements of most RISC designs:

- A large number of general-purpose registers
- A limited and simple instruction set
- An emphasis on optimising the instruction pipeline

Instruction set characteristics

Because of the amount of bugs in programs, high-level programming languages (HLLs) have been developed to allow the programmer to express algorithms more concisely. However, there is a big semantic gap between HLL operations and those provided in computer architecture.
Studies have been conducted to determine the characteristics of machine instructions generated from HLL programs, so now researchers are trying to make the architecture that supports the HLL simpler, rather than more complex.
The studies show that conditional statements (if, loop), references to scalars and operands, and procedure calls and returns are all frequent or time-consuming operations that are good candidates for performance optimisation.
In RISC designs,

- The large number of registers optimises *operand referencing*
- The high proportion of *conditional branching* and *procedure call* instructions makes a straightforward instruction pipeline inefficient
- A simplified instruction set is indicated

The use of a large register file

The large proportion of assignment statements and operand accesses indicate heavy reliance on register storage. Register storage is the fastest storage device (faster than both main memory and cache), so you should let the most frequently accessed operands be kept in registers and minimise register-memory operations.
The software approach:
Rely on the compiler to maximise register usage.
This approach requires the use of sophisticated program-analysis algorithms.
The hardware approach:
Simply use more registers so that more variables can be held in registers for longer periods of time.

Register windows

The problem with procedure calls is that local variables change with each call and return, and parameters must be passed too.
Studies show that procedures employ only a few passed parameters and local variables. To exploit this, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor to use a different fixed-size window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.
At any one time, only one window of registers is visible and is addressable as if it were the only set of registers. The window is divided into three fixed-sized areas:

- Parameter registers - hold parameters for procedure calls
- Local registers - used for local variables, as assigned by the compiler
- Temporary registers - used to exchange parameters and results with the next lower level (procedure called by current procedure). They are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data.

The actual organisation of the register file is as a circular buffer of overlapping windows. (The register windows hold the most recent procedure activations, while older activations are saved in memory, to be restored later).

Global variables

The window scheme provides an efficient organisation for storing local scalar variables in registers, but doesn't address the need to store global variables (i.e. those accessed by more than one procedure).
There are two alternatives:

1. Variables declared as global in an HLL can be assigned memory locations by the compiler, and all machine instructions that reference these variables will use memory-reference operands. (This is straightforward, but inefficient for frequently accessed global variables).
2. Incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures. (Disadvantages: hardware burden to accommodate the split in register addressing, and the compiler must decide which global variables should be assigned to registers).

Large register file versus cache

The register file, organised into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used the most heavily, acting like a faster cache memory.

Large register file	Cache
Holds *all* the local scalar variables of the most recent N-1 procedure activations	Holds a *selection* of recently used scalar variables
Efficient use of *time*, because all local scalar variables are retained	Efficient use of *space*, because it is reacting to the situation dynamically
Disadvantage: Inefficient use of space, because not all procedures will need the full window space allotted to them	Disadvantage: Data are read into the cache in *blocks*, so some of the data will not be used
Can hold some global scalars, but it is difficult for a compiler to determine which ones will be heavily used	Can handle global as well as local variables
Infrequent use of memory	Set associative memories with a small size, so data might overwrite frequently used variables
To reference a local scalar, a 'virtual' register number and a window number are used. These can pass through a simple decoder to select one of the physical registers	A full-width memory address must be generated. The access time is much longer

From the point of view of performance, the window-based register file is superior to cache for local scalars.

Compiler-based register optimisation

You could have a small number of registers and let the compiler optimise their usage. Graph colouring is the most commonly used technique.

In general, there is a trade-off between the use of a large set of registers and compiler-based optimisation. (The larger the number of registers, the smaller the benefit of register optimisation).

Reduced instruction set architecture

CISC

There is a trend to richer instruction sets, which include a larger number of instructions and more complex instructions. This is because architects attempted to design machines that provided better support for HLLs.

Reasons for CISC:

- Compiler simplification - If there are machine instructions that resemble HLL statements, compiler writing is simplified
  Problem: Complex machine instructions are hard to exploit because the compiler must find those cases that exactly fit the construct
- Smaller programs - Programs take up less memory and have fewer instructions, so fewer instruction bytes need to be fetched
  Problem: The program may be shorter, but the number of bits of memory occupied may not be smaller
- Faster programs - Complex HLL operations execute more quickly as a single machine instruction rather than as a series of more primitive instructions
  Problem: The entire control unit must be made more complex or the microprogram control store must be made larger, to accommodate a richer instruction set. Either factor increases the execution time of the instructions

Characteristics of reduced instruction set architectures

1. One machine instruction per machine cycle

Machine cycle = the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.
With simple, one-cycle instructions, there is little or no need for microcode; the machine instructions can be hardwired. Such instructions should execute faster because it isn't necessary to access a microprogram control store during instruction execution.

2. Register-to-register operations

With only simple LOAD and STORE operations accessing memory, you simplify the instruction set and therefore the control unit. Such an architecture encourages the optimisation of register use, so that frequently accessed operands remain in high-speed storage.

3. Simple addressing modes

Almost all instructions use simple register addressing. Several additional modes, like displacement and PC-relative, may be included. Other, more complex modes may be synthesised in software from the simple ones. This design feature simplifies the instruction set and the control unit.

4. Simple instruction formats

Instruction length is fixed and aligned on word boundaries.
Field locations, especially the opcode, are fixed.
Advantages:

- With fixed fields, opcode decoding and register operand accessing can occur simultaneously
- Simplified formats simplify the control unit
- Instruction fetching is optimised because word-length units are fetched
- Alignment on a word boundary also means that a single instruction doesn't cross page boundaries

Benefits of the RISC approach

Performance benefits

- More effective *optimising compilers* can be developed (With more primitive instructions there are more opportunities for code efficiency)

- Most instructions generated by a compiler are relatively *simple* anyway (so a control unit built specifically for those instructions could execute them faster than a comparable CISC)
- *Instruction pipelining* can be applied much more effectively with a reduced instruction set
- RISC processors are more responsive to *interrupts* because interrupts are checked between rather elementary operations

VLSI implementation benefits

With the advent of VLSI, it is possible to put an entire processor on a single chip. For a single-chip processor, there are two motivations for following a RISC strategy:

- *Performance* issue (On-chip delays are of much shorter duration than inter-chip delays)
- *Design-and-implementation* time (A RISC processor is far easier to develop than a VLSI processor)

CISC versus RISC characteristics

RISC designs may benefit from the inclusion of some CISC features and vice versa.

The PowerPC uses a RISC design and the Pentium II uses a CISC design, but neither is pure.