1 Computer hardware

Most computers are organized as shown in Figure 1.1. A computer contains several major subsystems --- such as the Central Processing Unit (CPU), memory, and peripheral device controllers. These components all plug into a "Bus". The bus is essentially a communications highway; all the other components work together by transferring data over the bus.

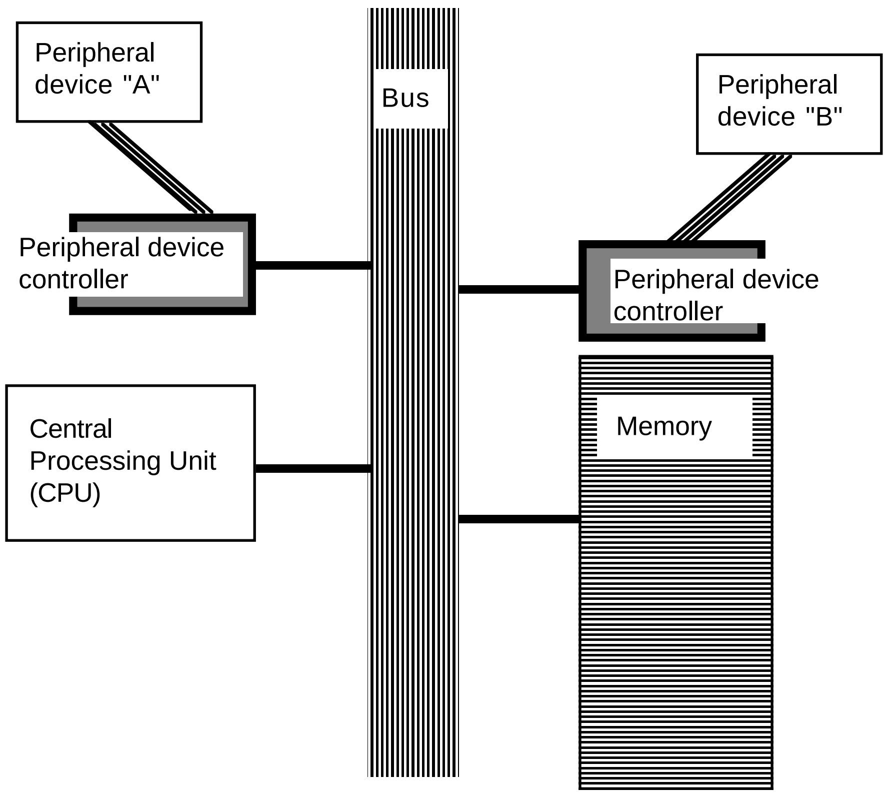

Peripheral device "A"
Bus
Peripheral device "B"
Peripheral device controller
Peripheral device controller
Central Processing Unit (CPU)
Memory

Figure 1.1 Schematic diagram of major parts of a simple computer.

The active part of the computer, the part that does calculations and controls all the other parts is the "Central Processing Unit" (CPU). The Central Processing Unit (CPU) contains electronic clocks that control the timing of all operations; electronic circuits that carry out arithmetic operations like addition and multiplication; circuits that identify and execute the instructions that make up a program; and circuits that fetch the data from memory.

Instructions and data are stored in main memory. The CPU fetches them as needed.

Peripheral device controllers look after input devices, like keyboards and mice, output devices, like printers and graphics displays, and storage devices like disks. The CPU and peripheral controllers work together to transfer information between the computer and its users. Sometimes, the CPU will arrange for data be taken from an input device, transfer through the controller, move over the bus and get loaded directly into the CPU. Data being output follows the same route in reverse – moving from the CPU, over the bus, through a controller and out to a device. In other cases, the CPU may get a device controller to move data directly into, or out of, main memory.

1.1 CPU AND INSTRUCTIONS

The CPU of a modern small computer is physically implemented as single silicon "chip". This chip will have engraved on it the million or more transistors and the interconnecting wiring that define the CPU's circuits. The chip will have one hundred or more pins around its rim --- some of these pins are connection points for the signal lines from the bus, others will be the points where electrical power is supplied to the chip.

Although physically a single component, the CPU is logically made up from a number of subparts. The three most important, which will be present in every CPU, are shown schematically in Figure 1.2.

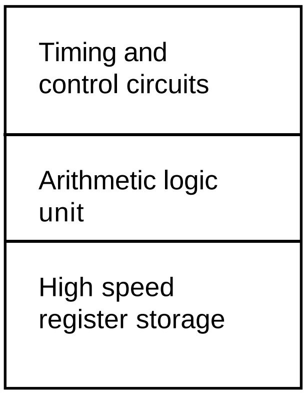


Figure 1.2 Principal components of a CPU.

Timing and control circuits

The timing and control circuits are the heart of the system. A controlling circuit defines the computer's basic processing cycle:

fetch, decode, execute cycle

```
repeat
    fetch next instruction from memory
    decode instruction (i.e. determine which data
           manipulation circuit is to be activated)
    fetch from memory any additional data that are needed
    execute the instruction (feed the data to the
           appropriate manipulation circuit)
until "halt" instruction has been executed;
```

Along with the controlling "fetch-decode-execute" circuit, the timing and control component of the CPU contains the circuits for decoding instructions and decoding addresses (i.e. working out the location in memory of required data elements).

Arithmetic logic unit

The arithmetic logic unit (ALU) contains the circuits that manipulate data. There will be circuits for arithmetic operations like addition and multiplication. Often there will be different versions of such circuits – one version for integer numbers and a second for real numbers. Other circuits will implement comparison operations that permit a program check whether one data value is greater than or less than some other value. There will also be "logic" circuits that directly manipulate bit pattern data.

High speed register storage

While most data are kept in memory, CPUs are designed to hold a small amount of data in "registers" (data stores) in the CPU itself. It is normal for main memory to be large enough to hold millions of data values; the CPU may only have space for something like 16 values. A CPU register will hold as many bits as a "word" in the computer's memory. Bits, bytes, words etc are described more in section 1.2. Most current CPUs have registers that each store 32 bits of data.

The circuits in the ALU often are organized so that some or all of their inputs and outputs must come from, or go to, CPU registers. Data values have to be fetched from memory and stored temporarily in CPU registers. Only then can they be combined using an ALU circuit, with the result again going to a register. If the result is from the final step in a calculation, it gets stored back into main memory.

While some of the CPU registers are used for data values that are being manipulated, others may be reserved for calculations that the CPU has to do when it is working out where in memory particular data values are to be stored.

"Data registers" and "address registers"

CPU designs vary with respect to their use of registers. But, commonly, a CPU will have 8 or more "data" registers and another 8 "address" registers. Programmers who write in low-level "assembly languages" (Chapter 2) will be aware of these data and address registers in the CPU. Assembly language code defines details such as how data should be moved to specific data registers and how addresses are to be calculated and saved temporarily in address registers. Generally, programmers working with high level languages (Chapter 4) don't have to be concerned about such details; but, when necessary, a programmer can find out how the CPU registers are used in their code.

In addition to the main data and address registers, the CPU contains many other registers, see Figure 1.3. The ALU will contain numerous registers for holding temporary values that are generated as arithmetic operations are performed. The timing and control component contains a number of registers that hold control information.

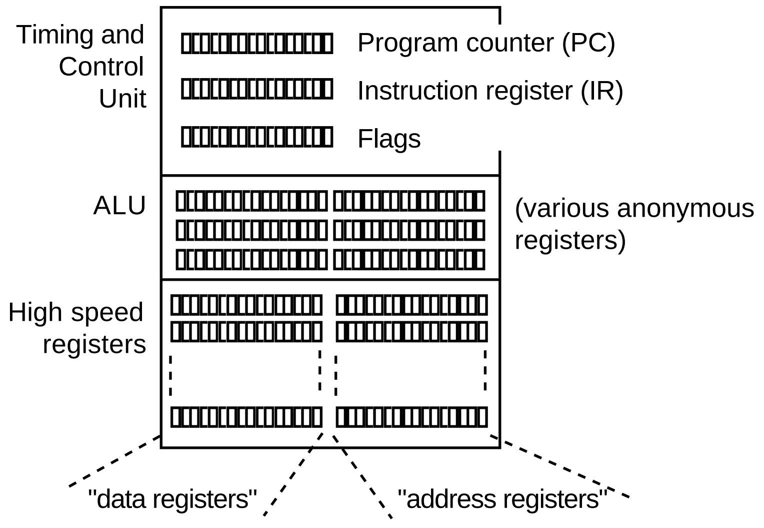


Figure 1.3 CPU registers.

Program Counter and Instruction Register

The Program Counter (PC) holds the address of the memory location containing the next instruction to be executed. The Instruction Register (IR) holds the bit pattern that represents the current instruction; different parts of the bit pattern are used to encode the "operation code" and the address of any data required.

Flags register

Most CPUs have a "flags" register. The individual bits in this register record various status data. Typically, one bit is used to indicate whether the CPU is executing code from an ordinary program or code that forms part of the controlling Operating Systems (OS) program. (The OS code has privileges; it can do things, which ordinary programs can not do, like change settings of peripheral device controllers. When the OS-mode bit is not set, these privileged instructions can not be executed.) Commonly, another group of bits in the flags register will be used to record the result of comparison instructions performed by the ALU. One bit in the flags register would be set if the comparator circuits found two values to be equal; a different bit would be set if the first of the two values was greater than the second.

Programs and instructions

Ultimately, a program has to be represented as a sequence of instructions in memory. Each instruction specifies either one data manipulation step or a control action. Normally, instructions are executed in sequence. The machine is initialized with the program counter holding the memory location of the first instruction from the program and then the fetch-decode-execute cycle is started. The CPU sends a fetch request to memory specifying the location specified by the PC (program counter); it receives back the instruction and stores this in the IR. The PC is then updated so that it holds the address of the next instruction in sequence. The instruction in the IR is then decoded and executed. Sometimes, execution of the instruction will change the contents of the PC. This can happen when one gets a "branch" instruction (these instructions allow a program to do things like skip over processing steps that aren't required for particular data, or go back to the start of some code that must be repeated many times).

Instruction repertoire

A CPU is characterized by its instruction repertoire – the set of instructions that can be interpreted by the circuits in the timing and control unit and executed using the arithmetic logic unit.. The Motorola 68000 CPU chip can serve as an example. The 68000 (and its variants like 68030 and 68040) were popular CPU chips in the 1980s being used in the Macintosh computers and the Sun3 workstations. The chip had, as part of its instruction repertoire, the following instructions:

ADD	Add two integer values
AND	Perform an AND operation on two bit patterns
Bcc	Test a condition flag, and possibly branch to another instruction (variants like BEQ testing equality, BLT testing less than)
CLR	Clear, i.e. set to 0
CMP	Compare two values
JMP	Jump or goto
JSR	Call a subroutine
SUB	Subtract second value from first
RTS	Return from subroutine

mnemonic instruction names

(Instructions are usually given short "mnemonic" names – names that have been chosen to remind one of the effect achieved by the instruction, like ADD and CLeaR.)

Different CPU architectures, e.g. the Motorola 68000 and Intel-086 architectures, have different instruction sets. There will be lots of instructions that are common – ADD, SUB, etc. But each architecture will have its own special instructions that are not present on the other. Even when both architectures have similar instructions, e.g. the compare and conditional branch instructions, there may be differences in how these work.

Figure 1.4 is a simplified illustration of how instructions are represented inside a computer.

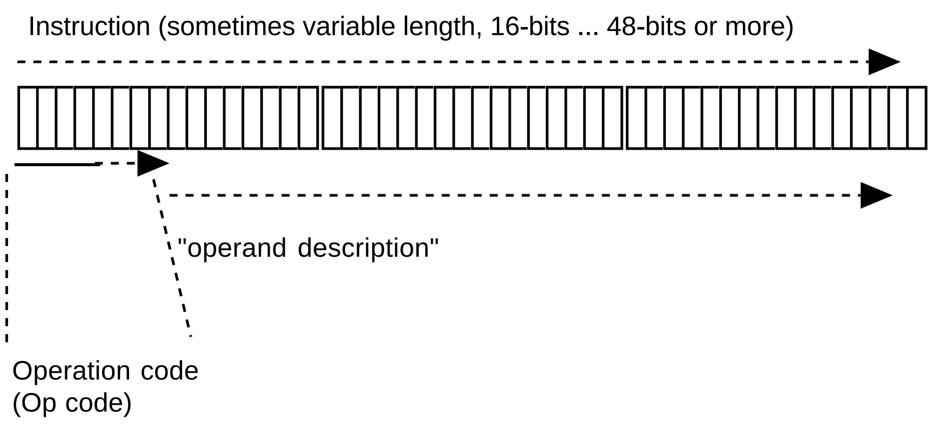

Instruction (sometimes variable length, 16-bits ... 48-bits or more)
"operand description"
Operation code
(Op code)

Figure 1.4 Simplified representation of an instruction inside a computer.

An instruction is represented by a set of bits. A few CPUs have fixed size instructions; on such machines, every instruction is 16-bits, or 32-bits or whatever. Most CPUs allow for different sizes of instructions. An instruction will be at least 16-bits in size, but may have an additional 16, 32, or more bits.

Op-code

The first few bits of an instruction form the "Op-code" (operation code). These bits identify the data manipulation or control operation required. Again CPUs vary; some use a fixed size op-code, most have several different layouts for instructions with differing numbers of bits allocated for the op-code. If a CPU uses a fixed size op-code, decoding is simple. The timing and control component will implement a form of multiway switch. Thus, if one had a 4-bit op-code, one could have a decoding scheme something like the following:

```
0000      Do an addition
0001      Do a subtraction
0010      Copy (move) some data
0011      Do an AND operation
...
...
```

The meaning of the remaining bits of an instruction depends on the actual instruction.

Operand(s)

Many instructions require that data be specified. Thus, an ADD instruction needs to identify which two values are to be summed, and must also specify a place where the result should be stored. Often some of this information can be implicit. An ADD instruction can be arranged so that the sum of the two specified values always replaces the first value wherever this was stored.

Although some data locations can be implicit, it is necessary to define either the source or destination locations for the other data. Sometimes a program will need to add numbers that are already held in data registers in the CPU; at other times, the program may need to fetch additional data from memory. So sometimes the "operand description" part of an add instruction will need to identify the two CPU data registers that are to be used; other times, the "operand description" will have to identify one CPU register and one memory location. Occasionally, the "operand description" part might be used to identify a CPU register and the *value* that is to added to that register's existing contents.

Addressing modes

It is here that things get a bit complex. CPUs have many different ways of encoding information about the registers to be used, the addresses of memory locations, and the use of explicit data values. A particular machine architecture will have a set of "addressing modes" – each mode specifies a different way of using the bits of the operand description to encode details concerning the location of data values. Different architectures have quite different sets of addressing modes.

Some instructions don't need any data. For example, the "Bcc" (conditional branch) group instructions use only information recorded in the CPU's Flags register. These instructions have different ways of using the operand bits of instruction word. Often, as with the Bcc instructions, the operand bits encode an address of an instruction that is to be used to replace the current contents of the program counter. Replacing the contents of the PC changes the next instruction executed.

1.2 MEMORY AND DATA

Computers have two types of memory:

ROM – Read Only Memory

RAM - normal Read Write Memory

The acronym RAM instead of RWM is standard. It actually standards for "Random Access Memory". Its origin is very old, it was used to distinguish main memory (where data values can be accessed in any order – hence "randomly") from secondary storage like tapes (where data can only be accessed in sequential order).

ROM memory

ROM memory is generally used to hold parts of the code of the computer's operating system. Some computers have small ROM memories that contain only a minimal amount of code just sufficient to load the operating system from a disk storage unit. Other machines have larger ROM memories that store substantial parts of the operating system code along with other code, such as code for generating graphics displays.

RAM memory

Most of the memory on a computer will be RAM. RAM memory is used to hold the rest of the code and data for the operating system, and the code and data for the program(s) being run on the computer.

Bits, bytes, and words

Memory sizes may be quoted in bits, bytes, or words:

Bit	a single 0 or 1 data value
Byte	a group of 8 bits
Word	the width of the primary data paths between memory and the CPU, maybe 16-bit (two byte), 32-bit (four byte) or larger.

Memory sizes are most commonly given in terms of bytes. (The other units are less useful for comparative purposes. Bits are too small a unit of storage. Word sizes vary between machines and on some machines aren't really defined.) The larger memory units like bytes and words are just made up from groups of bits.

All storage devices require simple two-state components to store individual bits. Many different technologies have been used Some early computers distinguished 0 and 1 bit values by the presence or absence of a pulse of energy moving through a tube of mercury; external storage was provided using paper media like cards or tapes where the presence or absence of a punched hole distinguished the 0/1 bit setting. Later, the most popular technology for a computer's main memory used small loops of magnetic oxide ("cores") that could be set with differing North/South polarity to distinguish the 0/1 bit state. Disks (and tapes) still use magnetic encoding – 0/1 bit values are distinguished by the magnetic state of spots of oxide on the disk's surface.

The main memories of modern computers are made from integrated circuits. One basic circuit is a "flip-flop". This uses four transistors wired together; it can be set in an on or an off state and so can hold one bit. A more elaborate circuit, with eight flip-flops, can hold one byte. Repeated again and again, these can be built up into integrated circuits that hold millions of bytes. Individual memory chips with as much as 4 million bytes of storage capacity can now be purchased. A computer's memory will be made up out of several of these chips.

Figure 1.5 is a simplified illustration of memory for a machine with a 16-bit (two byte) word size.

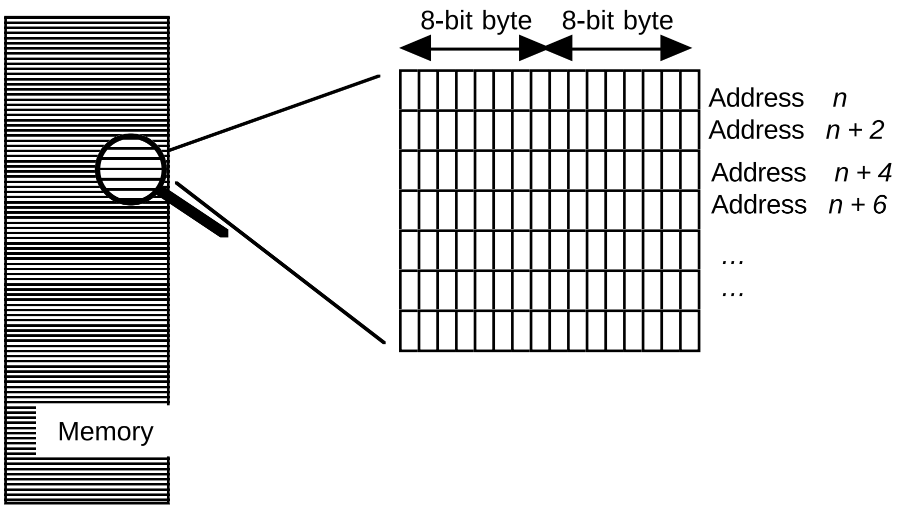


Figure 1.5 Memory organized in words with integer addresses.

Memory addresses

Memory can be viewed as a vector (one dimensional array) of words. The words can be identified by their positions (index number) in this vector – the integer index of a word is its "address". Most computers are designed to allow addressing of individual bytes. (If a request is made for a specific byte, the memory unit may return the entire word, leaving it to the CPU circuits to select the required byte). Because the individual bytes are addressable, word addresses increase in 2s (or in 4s if it is a machine with 4 byte words).

The amount of memory available on a computer has increased rapidly over the last few years. Most current personal computers now have around 8 million bytes of storage (8 megabyte, 8MB); more powerful workstations have from 32MB to 256MB and large time shared systems may have 1000 MB (or 1gigabyte).

Cache memories

"Cache" memories are increasingly common ("cache – a hiding place for provisions, treasures etc"). Cache memories are essentially hidden from the applications programmer; the cache belongs to the computer hardware and its controlling operating system. These work together using a cache to increase performance. Currently, a typical cache memory would be up to 256 KB in size. The cache may form a part of the circuitry of the CPU chip itself, or may be a separate chip. Either way, the system will be designed so that information in the cache can be accessed much more quickly than information in main storage.

The OS and CPU hardware arrange to copy blocks of bytes ("pages") from main memory into the cache. The selected pages could be those with the instructions currently being executed. Most programs involve loops where particular sets of instructions are executed repeatedly. If the instructions forming a loop are in the cache, the CPU's instruction-fetch operation is greatly speeded up. Sometimes it is worth copying pages with data from main memory to the cache – then subsequent data accesses are faster (though data that get changed do have to be copied back to main memory eventually). The operations shifting pages, or individual data elements, between cache and memory are entirely the concern of the CPU hardware and the operating system. The only way that a programmer should be able to detect a cache is by noticing increased system's performance.

Data as bit patterns in memory

All data manipulated by computers are represented by bit patterns. A byte, with 8 individual bits, can represent any of 256 different patterns; some are shown in Figure 1.6.

Character data

A set of 256 patterns is large enough to have a different pattern for each letter of the alphabet, the digits, punctuation characters, and a whole variety of special characters. If a program has to work with textual data, composed of lots of individual characters, then each character can be encoded in a single byte. Of course there have to be conventions that assign a specific pattern to each different character. At one time, different computer manufacturers specified their own character encoding schemes. Now, most use a standard character encoding scheme known as ASCII (for American Standard Code for Information Interchange). Although standardized, the assignments of patterns to characters is essentially arbitrary; Figure 1.6 shows the characters for some of the illustrated bit patterns.

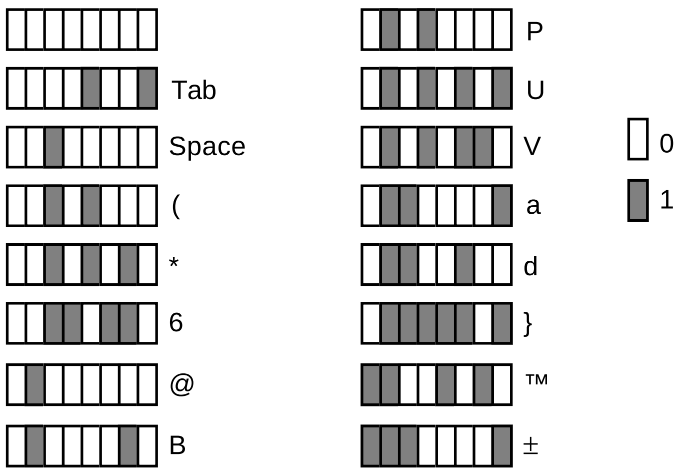


Figure 1.6 Some of the 256 possible bit patterns encodable in a single byte and the (printable) characters usually encoded by these patterns.

ASCII character codes

This ASCII scheme is mandated by an international standard. It specifies the bit patterns that should be used to encode 128 different characters including all the letters of the Roman alphabet, digits, punctuation marks and a few special control characters like "Tab". The remaining 128 possible patterns (those starting with a 1 in the leftmost bit of the byte) are not assigned in the standard. Some computer systems may have these patterns assigned to additional characters like ™, ±, ¢, ‡.

Numeric data

Bit patterns can also be used to represent numbers. Computers work with integer numbers and "floating point" numbers. Floating point numbers are used to approximate the real numbers of mathematics.

Integer values

A single byte can only be used to encode 256 different values. Obviously, arithmetic calculations are going to work with wider ranges – like -2,000,000,000 to +2,000,000,000. Many more bits are needed to represent all those different possible values. All integer values are represent using several bytes. Commonly, CPUs are designed to work efficiently with both two-byte integers and four-byte integers (the CPU will have two slightly different versions of each of the arithmetic instructions). Two-byte integers are sufficient if a program is working with numbers in the range from about minus thirty thousand to plus thirty thousand; the four-byte integers cover the range from minus to plus two thousand million.

Figure 1.7 shows the common representations of a few integers when using two bytes. The number representations have an obvious regular pattern. Unlike the case of character data, the patterns used to represent integers can not be arbitrary. They have to follow regular patterns in order to make it practical to design electronic circuitry that can combine patterns and achieve effects equivalent to arithmetic operations. The code scheme that provides the rules for representing numbers is known as "two's complement notation"; this scheme is covered in introductory courses on computer hardware.

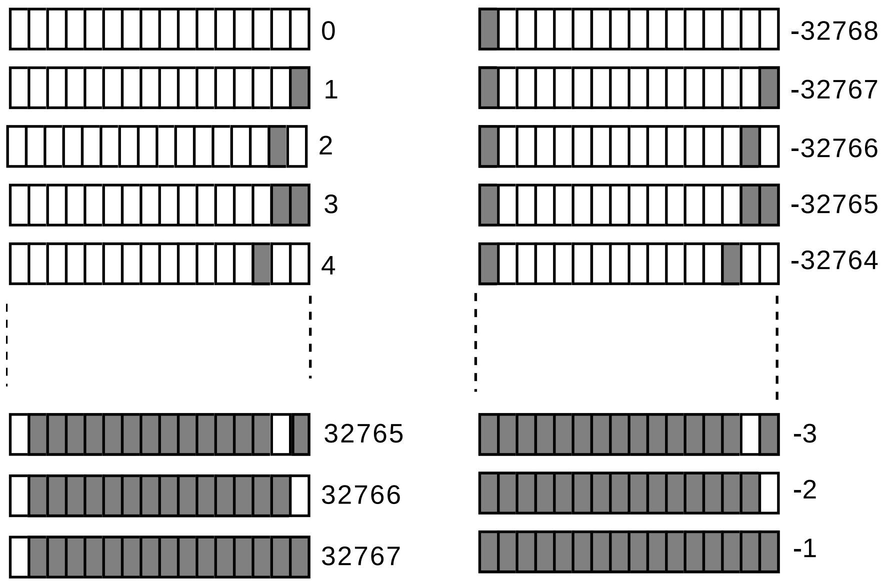


Figure 1.7 Representing integers in two byte (16-bit) bit patterns in accord with the "two's complement notation scheme".

There are other coding schemes for integers but "two's complement notation" is the most commonly used. The actual coding scheme used to represent integers, and the resulting bit patterns, is not often of interest to programmers.

While a computer requires special bit patterns representations of numbers, strings of 0 and 1 characters are not appropriate for either output or input. Humans require numbers as sequences of digit characters. Every time a number is input to a program, or is printed by a program, some code has to be executed to translate between the binary computer representation and the digit string representation used by humans.

The electronic circuits in the CPU can process the binary patterns and correctly reproduce the effects of arithmetic operations. There is just one catch with integers – in the computer they are limited to fixed ranges.

Integer overflow

Two-bytes are sufficient to represent numbers from -32767 to +32768 – and it is an error if a program generates a value outside this range. Mistakes are possible. Consider for example a program that is specified as only having to work with values in the range 0 to 25000; a programmer might reasonably choose to use two-byte integers to represent such values. However, a calculation like "work out 85% of 24760" could cause problems, even though the result (21046) is in range. If the calculation is done by multiplying 85 and 24760, the intermediate result 2104600 is out of range.

Arithmetic operations that involve unrepresentable (out of range) numbers can be detected by the hardware – i.e. the circuits in the ALU. Such operations leave incorrect bit patterns in the result, but provide a warning by setting an "overflow" bit in the CPU's flags register. Commonly, computer systems are organized so that setting of the overflow bit will result in the operating system stopping the program with the error. The operating system will provide some error message. (The most common cause of "overflow" is division by zero – usually the result of a careless programming error or, sometimes, due to incorrect data entry.)

Floating point numbers

Floating point numbers are used to approximate real numbers. They are mainly used with engineering and scientific calculations. The computer schemes for floating point numbers are closely similar to normalized scientific notation:

Number	Normalized scientific representation
`17.95`	`+1.795 E+01`
`-0.002116`	`-2.116 E-03`
`1.5`	`+1.5 E+00`
`31267489.2`	`+3.12674892 E+07`

The normalized scientific notation has a sign, a "mantissa", and a signed "exponent" – like '+', 1.795, and E+01. Floating point representations work in much the same way (you can think of floating point as meaning that the exponent specifies how far the "decimal" point need to be moved, or floated, to the left or the right).

A floating point number will be allocated several bytes (at least four bytes, usually more). One bit in the first byte is used for the sign of the number. Another group of bits encode the (signed) exponent. The remaining bits encode the mantissa. Of course, both exponent and mantissa are encoded using a binary system rather than a decimal system.

Since the parts of a floating point number are defined by regular encoding rules, it is again possible to implement "floating point" arithmetic circuits in the ALU that manipulate them appropriately. Like integers, floating point numbers can overflow. If the number has an exponent that exceeds the range allowed, then overflow occurs. As with integers, this is easy to detect (and is most often due to division by zero). But with floating point numbers, there is another catch – a rather more pervasive one than the "overflow" problem.

Round-off errors

The allocation of a fixed number of bits for the mantissa means that only certain numbers are accurately represented. When four bytes are used, the mantissas are accurate to about eight decimal digits. So, while it might be possible to represent numbers like 0.81246675 and 0.81246676 exactly, all the values in the range 0.812466755.. to 0.812466764... have to be approximated by the nearest number that can be represented exactly i.e. 0.81246676. Any remaining digits, in the 9th and subsequent places in the fraction, are lost in this rounding off process.

Each floating point operation that combines two values will finish by rounding off the result to the nearest representable value. It might seem that loss of one part in a hundred million is not important. Unfortunately, this is not true. Each calculation step can introduce such errors – and a complete calculation can involve millions of steps in which the errors *may* combine and grow. Further, it is quite common for scientists and engineers to be trying to calculate the small difference between two large values – and in these cases the "rounded off" parts may be comparable to the final result.

While integer overflow errors are easily detected by hardware and are obvious errors, round off errors can not be dealt with so simply. Those needing to work extensively with floating point numbers really need to take a numerical analysis course that covers the correct way of organizing calculations so as to minimize the effects of cumulative round off errors.

Instructions – just another bit pattern

As explained in section 1.1, instructions are represented as bit patterns – maybe 16 bits (two bytes) in length, possibly longer. In memory, instructions are stored in a sequence of successive bytes.

All just bit patterns

A few experimental computers have been built where every word in memory had something like 2 or 3 extra bits that tagged the type of data stored in that word. These "tagged memory architecture" machines might have used code 00 to mark a word containing an instruction, 01 if it contained integer data etc. Such computers are atypical. On most computers, bit patterns in memory have no "type", no intrinsic meaning. The meaning of a bit pattern is determined by the circuit of the CPU that interprets it; so if it ends up in the IR (instruction register) it gets interpreted as representing an instruction, while if it goes to an (integer) addition circuit in the ALU it is interpreted as an integer. It is possible (though uncommon) to make programming errors so that data values fetched from memory to be interpreted as instructions or, alternatively, for a program to start storing results of calculations in those parts of its memory that hold the instruction sequence. If a program contains such gross errors, it usually soon attempts an illegal operation (like attempting to execute a bit pattern that can not be recognized as a valid instruction) and so is stopped by the computer hardware and operating system.

1.3 BUS

A computer's bus can be viewed as consisting of about one hundred parallel wires; see Figure 1.8. Some of these wires carry timing signals, others will have control signals, another group will have a bit pattern code that identifies the component (CPU, memory, peripheral controller) that is to deal with the data, and other wires carry signals encoding the data.

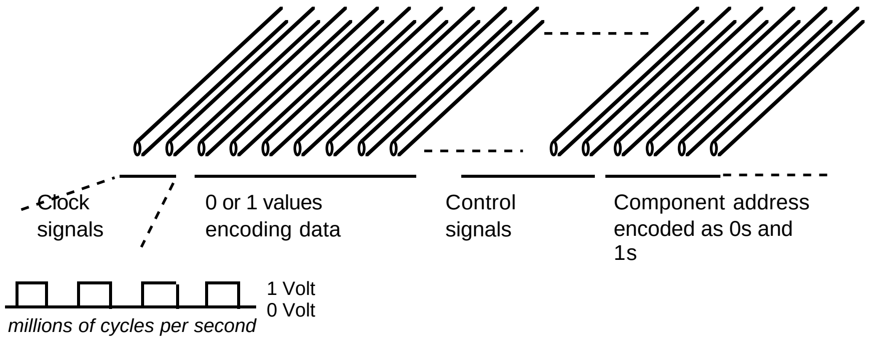

Clock signals
0 or 1 values encoding data
Control signals
Component address encoded as 0s and 1s
1 Volt
0 Volt
millions of cycles per second

Figure 1.8 Bus.

Signals are sent over the bus by setting voltages on the different wires (the voltages are small, like 0-volts and 1-volt). When a voltage is applied to a wire the effect propagates along that wire at close to the speed of light; since the bus is only a few inches long, the signals are detectable essentially instantaneously by all attached components. Transmission of information is controlled by clocks that put timing signals on some of the wires. Information signals are encoded on to the bus, held for a few clock ticks to give all components a chance to recognize and if appropriate take action, then the signals are cleared. The clock that controls the bus may be "ticking" at more than one hundred million ticks per second

The "plugs" that attach components to the bus incorporate quite sophisticated circuits. These circuits interpret the patterns of 0/1 voltages set on the control and address lines – thus memory can recognize a signal as "saying" something like "store the data at address xxx", while a disk control unit can recognize a message like "get ready to write to disk block identified by these data bits". In addition, these circuits deal with "bus arbitration". Sometimes, two or more components may want to put signals on the bus at exactly the same time – the bus arbitration circuitry resolves such conflicts giving one component precedence (the other component waits a few hundred millionths of a second and then gets the next chance to send its data).

1.4 PERIPHERALS

There are two important groups of input/output (i/o) devices. There are devices that provide data storage, like disks and tapes, and there are devices that connect the computer system to the external world (keyboards, printers, displays, sensors).

The storage devices record data using the same bit pattern encodings as used in the memory and CPU. These devices work with blocks of thousands of bytes. Storage space is allocated in these large units. Data transfers are in units of "blocks".

The other i/o devices transfer only one, or sometimes two, bytes of data at a time. Their controllers have two parts. There is a part that attaches to the bus and has some temporary storage registers where data are represented as bit patterns. A second part of the controller has to convert between the internal bit representation of data and its external representation. External representations vary – sensors and effectors (used to monitor and control machinery in factories) use voltage levels, devices like simple keyboards and printers may work with timed pulses of current, some devices use flashes of light.

1.4.1 Disks and tapes

Most personal computers have two or three different types of disk storage unit. There will be some form of permanently attached disk (the main "hard disk"), some form of exchangeable disk storage (a "floppy disk" or possibly some kind of cartridge-style hard disk), and there may be a CD-ROM drive for read-only CD disks.

Optical disks CD disks encode 0 and 1 data bits as spots with different reflectivity. The data can be read by a laser beam that is either reflected or not reflected according to the setting of each bit of data; the reflected light gets converted into a voltage pulse and hence the recorded 0/1 data values gets back into the form needed in the computer circuits. Currently, optical storage is essentially read-only – once data have been recorded they can't be changed. Read-write optical storage is just beginning to become available at reasonable prices.

Magnetic disks Most disks use magnetic recording. The disks themselves may be made of thin plastic sheets (floppy disks), or ceramics or steel (hard disks). Their surfaces are covered in a thin layer of magnetic oxide. Spots of this magnetic oxide can be magnetically polarized. If a suitably designed wire coil is moved across the surface, the polarized spots induce different currents in the coil – allowing data to be read back from the disk. New data can be written by moving a coil across the surface with a sufficiently strong current flowing to induce a new magnetic spot with a required polarity. There is no limit on the number of times that data can be rewritten on magnetic disks.

Tracks The bits are recorded in "tracks" – these form concentric rings on the surface of the disk, see Figure 1.9. Disks have hundreds of these tracks. (On simple disk units, all tracks hold the same number of bits; since the outermost tracks are slightly longer, their bits are spaced further apart than those on the innermost tracks. More elaborate disks can arrange to have the bits recorded at the same density; the outer tracks then hold more bits than the inner tracks.)

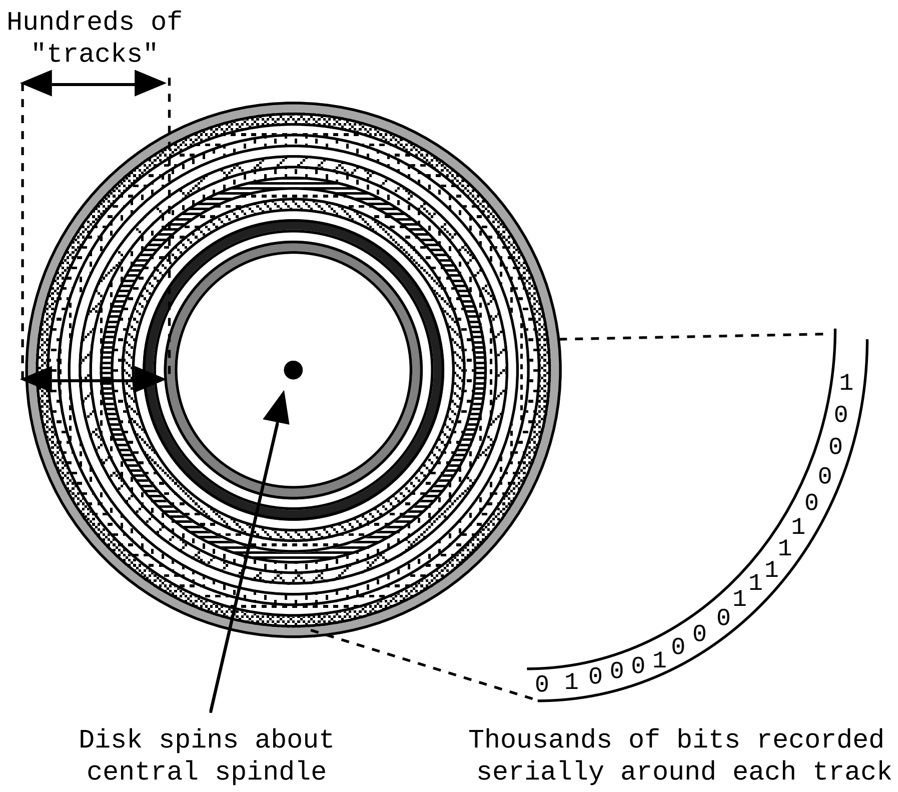

Hundreds of "tracks"
1 0 0 0 0 1 1 1 1 0 0 0 1 0 0 0 0 1 0
Disk spins about central spindle
Thousands of bits recorded serially around each track

Figure 1.9 Bits are stored in concentric "tracks" on disk.

Blocks (or sectors)

Tracks are too large a unit of storage – they can hold tens of thousands of bits. Storage on a track is normally broken down into "blocks" or sectors. At one time, programmers could select the size of blocks. A programmer would ask for some tracks of disk space and then arrange the breakdown into blocks for themselves. Nowadays, the operating system program that controls most of the operations of a computer will mandate a particular block size. This is typically in the range 512 bytes to 4096 bytes (sometimes more).

The blocks of bytes written to the disk will contain these data bytes along with a few control bytes added by the disk controller circuitry. These control bytes are handled entirely by the disk controller circuits. They provide housekeeping information used to identify blocks and include extra checking information that can be used, when reading data, to verify that the data bits are still the same as when originally recorded.

The disk controller may identify blocks by block number and track number, see Figure 1.10., or may number all blocks sequentially. If the disk shown in Figure 1.10 used sequential numbering, track 1 would contain blocks 16, 17, ….

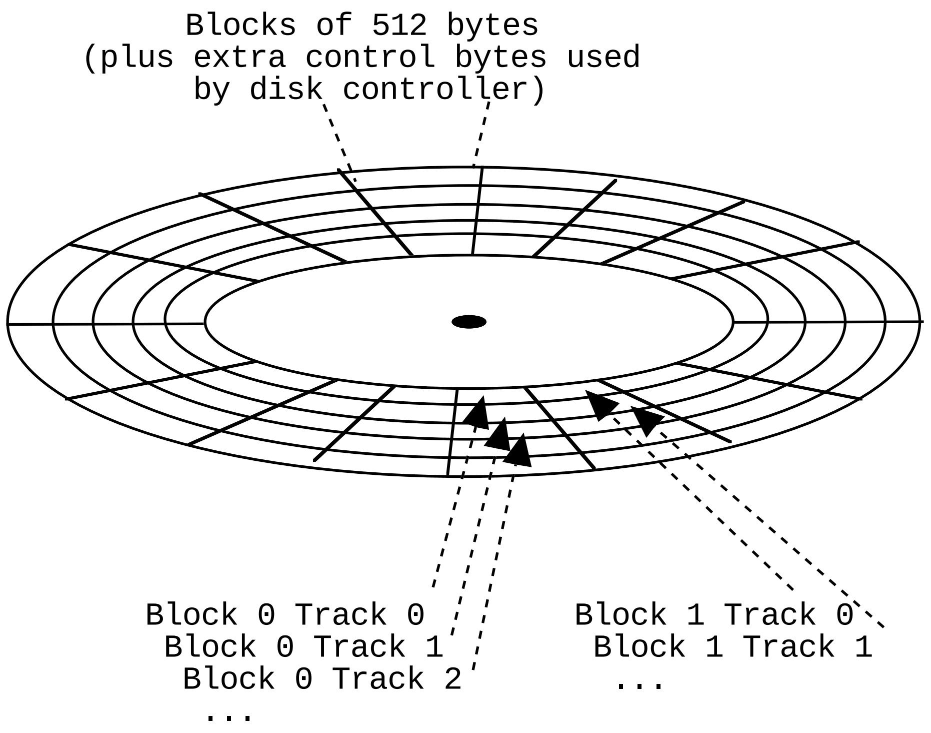


Figure 1.10 Tracks divided into storage blocks.

"Seeking" for tracks

Before data blocks can be read or written, the read/write head mechanism must be moved to the correct track. The read/write head contains the coil that detects or induces magnetism. It is moved by a stepping motor that can align it accurately over a specific track. Movements of the read/write heads are, in computer terms, relatively slow – it can take a hundredth of a second to adjust the position of the read/write heads. (The operation of moving the heads to the required track is called "seeking"; details of disk performance commonly include information on "average seek times".) Once the head is aligned above the required track, it is still necessary for the spinning disk to bring the required block under the read/write head (the disk controller reads its control information from the blocks as they pass under the head and so "knows" when the required block is arriving). When the block arrives under the read/write head, the recorded 0/1 bit values can be read and copied to wherever else they are needed.

Disk cache memory

The read circuitry in the disk reassembles the bits into bytes. These then get transferred over the bus to main memory (or, sometimes, into a CPU register). Disks may have their own private cache memories. Again, these are "hidden" stores where commonly accessed data can be kept for faster access. A disk may have cache storage sufficient to hold the contents of a few disk blocks (i.e. several thousand bytes). As well as being sent across the bus to memory, all the bytes of a block being read can be stored in the local disk cache. If a program asks the disk to read a block of data that is in the cache, the disk unit doesn't need to seek for the data. The required bytes can be read from the cache and sent to main memory.

Commonly, hard disks have several disk platters mounted on a single central spindle. There are read/write heads for each disk platter. Data can be recorded on both sides of the disk platters (though often the topmost and bottommost surfaces are unused). The read/write heads are all mounted on the same stepping motor mechanism and move together between the disk platters, see Figure 1.11.

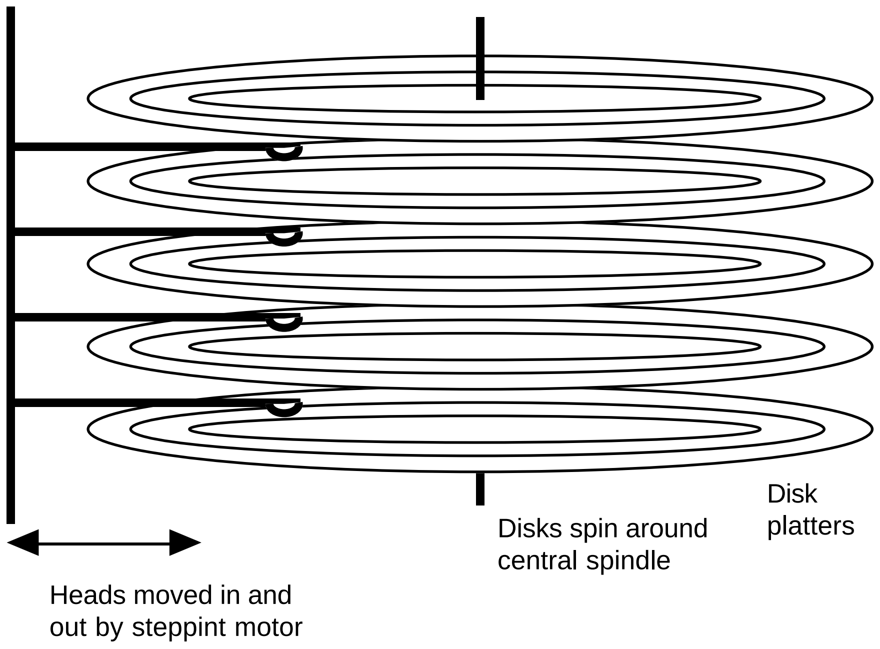


Figure 1.11 Read/write head assemble and multiplatter disks.

Disk controller

A controller for a disk is quite elaborate, see Figure 1.12. The controller will have several registers (similar to CPU registers) and circuitry for performing simple additions on (binary) integer numbers. (The cache memory shown is optional; currently, most disk controllers don't have these caches.) One register (or group of registers) will hold the disk address or "block number" of the data block that must be transferred. Another register holds a byte count; this is initialized to the block size and decremented as each byte is transferred. The disk controller stops trying to read bits when this counter reaches zero. The controller will have some special register used for grouping bits into bytes before they get sent over the bus. Yet another register holds the address of the (byte) location in memory that is to hold the next byte read from the disk (or the next byte to be written to disk).

Errors can occur with disk transfers. The magnetic oxide surface may have been damaged. The read process may fail to retrieve the data. The circuits in the disk can detect this but need some way of passing this information to the program that wanted the data. This is where the Flags register in the disk controller gets used. If something goes wrong, bits are set in the flags register to identify the error. A program doing a disk transfer will check the contents of the flags register when the transfer is completed and can attempt some recovery action if the data transfer was erroneous.

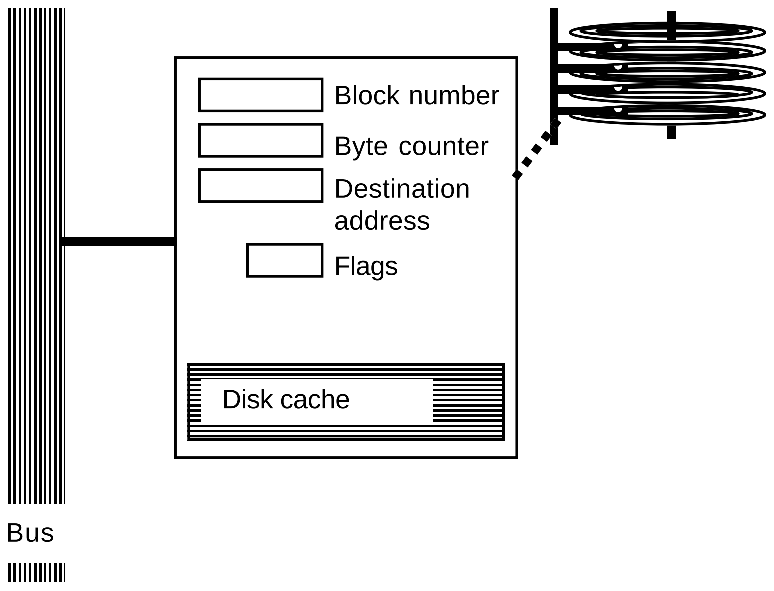


Figure 1.12 Disk controller.

This elaborate circuitry and register setup allows a disk controller to work with a fair degree of autonomy. Figure 1.13 illustrates the steps involved in a data transfer using such a disk and controller.

The data transfer process will start with the CPU sending a request over the bus to the disk controller; the request will cause the disk unit to load its block number register and to start its heads seeking to the appropriate track (step 1 in Figure 1.13).

It may take the disk a hundredth of a second to get its heads positioned (step 2 in Figure 1.13). During this time, the CPU can execute tens of thousands of instructions. Ideally, the CPU will be able to get on with other work, which it can do provided that it can work with other data that have been read earlier. At one time, programmers were responsible for trying to organize data transfers so that the CPU would be working on one block of data while the next block was being read. Nowadays, this is largely the responsibility of the controlling OS program.

When the disk finds the block it can inform the CPU which will respond by providing details of where the data are to be stored in memory (steps 3 and 4 in Figure 1.13).

Direct Memory Access

The disk controller can then transfer successive bytes read from the disk into successive locations in memory. A transfer that works like this is said to be using "direct memory access".

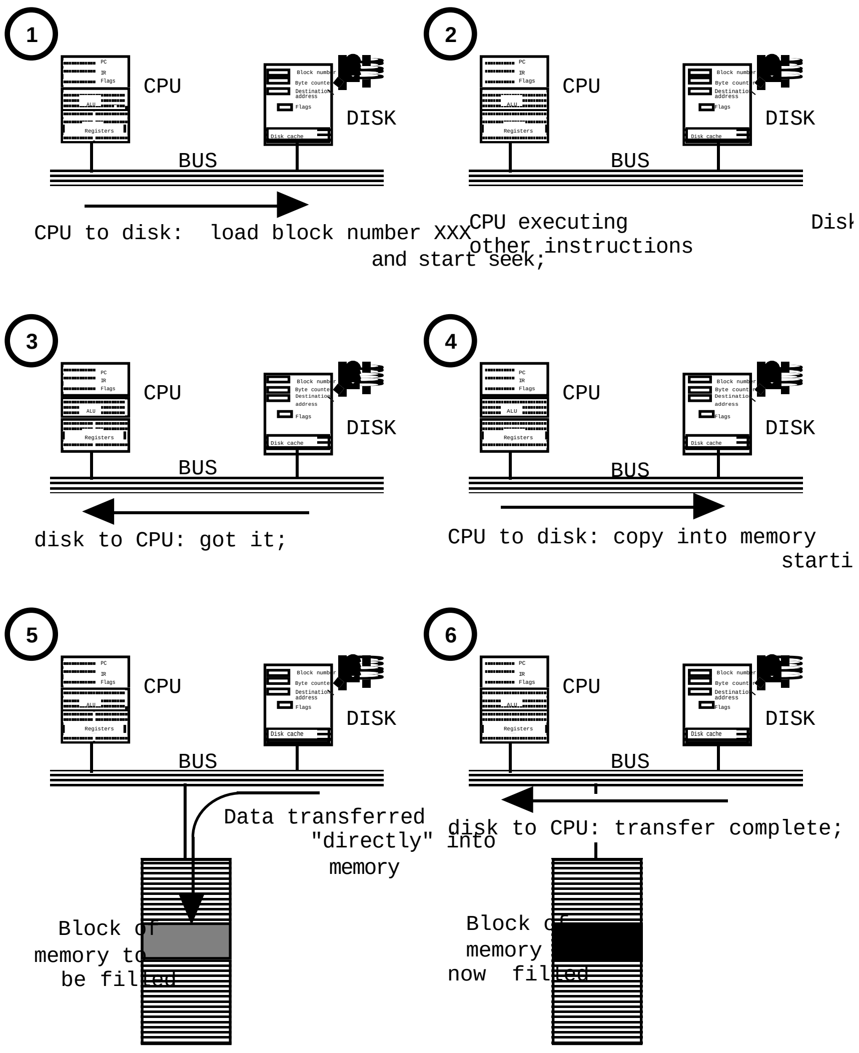

1
CPU
DISK
BUS
CPU to disk: load block number XXX and start seek;
2
CPU executing other instructions
Disl
3
disk to CPU: got it;
4
CPU to disk: copy into memory starti
5
Data transferred "directly" into memory
Block of memory to be filled
6
disk to CPU: transfer complete;
Block of memory now filled

Figure 1.13 A disk data transfer using direct memory access.

The data transfer, step 5 in Figure 1.13, will take a little time (though very much less than the seek step). Once again, the CPU will be able to get on with other work.

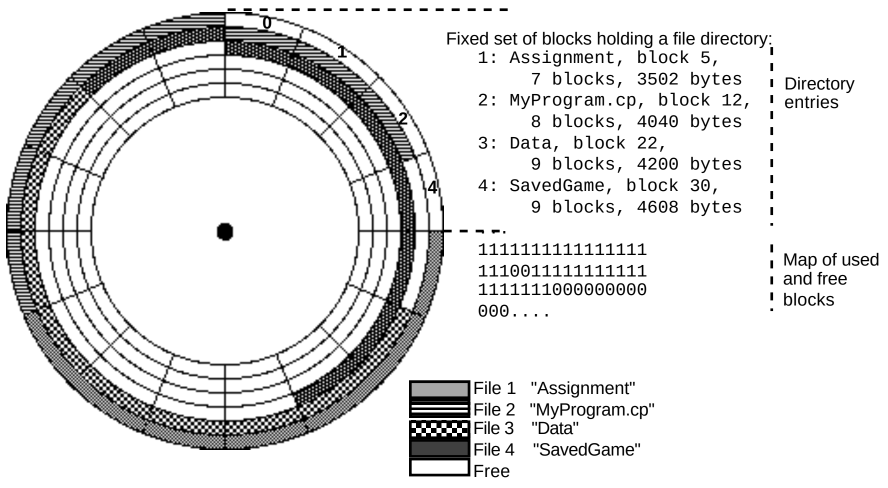
When the transfer is complete, the disk controller will send another signal to the CPU, (step 6).

Files

Data files on disk are made up out of blocks. For example, a text file with twelve thousand characters would need twenty four 512-byte blocks. (The last block would only contain a few characters from the file, it would be filled out with either space characters or just random characters). Programmers don't chose the blocks used for their files. The operating system is responsible for choosing the blocks used for each file, and for recording details for future reference.

Figure 1.14 illustrates the simplest scheme used for block allocation. This scheme was sometimes used for small disks on early personal computers (late 1970s and early 1980s). All modern systems use more sophisticated schemes but the essence is the same.

File directory As shown in Figure 1.14, a few blocks of the disk are reserved for a "file directory". The data in these blocks form a table of entries with each entry specifying a file name, file size (in bytes actually used and complete blocks allocated), and some record of which blocks are allocated. The allocation scheme shown in Figure 1.14 uses a group of contiguous blocks to make up each individual file. This makes it easy to record details of allocated blocks, the directory need only record the file size and the first block number.



```
1: Assignment, block 5,
    7 blocks, 3502 bytes
2: MyProgram.cp, block 12,
    8 blocks, 4040 bytes
3: Data, block 22,
    9 blocks, 4200 bytes
4: SavedGame, block 30,
    9 blocks, 4608 bytes

1111111111111111
1110011111111111
1111111000000000
000....
```

Figure 1.14 Simple file directory and file allocation scheme.

In addition to the table of entries describing allocated files, the directory structure would contain a record of which blocks were allocated and which were free and therefore available for use if another file had to be created. One simple scheme uses a map with one bit for each block; the bit is set if the block is allocated.

Tapes Tapes are now of minor importance as storage devices for users' files. Mostly they are used for "archival" storage – recording data that are no longer of active interest but may be required again later. There are many different requirements for archival data. For example, government taxation offices typically stipulate that companies keep full financial record data for the past seven years; but only the current year's data will be of interest to a company. So, a company will have its current data on disk and hold the data for the other six years on tapes. Apart from archival storage, the main use of tapes is for backup of disk units. All the data on a computer's disks will be copied to tape each night (or, maybe just weekly). The tapes can be stored somewhere safe, remote from the main computer site. If there is a major accident destroying the disks, the essential data can be retrieved from tape and loaded on some other computer system.

The tape units used for most of the last 45 years are physically a bit like large reel-to-reel tape recorders. The tapes are about half an inch wide and two thousand feet in length and are run from their reel, through tensioning devices, across read-write heads, to a take up reel. The read write heads record 9 separate data tracks; these 9 tracks are used to record the 8-bits of a byte along with an extra check bit. Successive bytes are written along the length of the tape; an inch of tape could pack in as much as a few thousand bytes. Data are written to tape in blocks of hundreds, or thousands, of bytes. (On disks, the block sizes are now usually chosen by the operating system, the size of tape blocks is program selectable.) Blocks have to be separated by gaps where no data are recorded – these "inter record gaps" have to be large (i.e. half an inch or so) and they tend to reduce the storage capacity of a tape. Files are written to tape as sequences of blocks. Special "end of file" patterns can be recorded on tape to delimit different files.

A tape unit can find a file (identified by number) by counting end of file marks and then can read its successive data blocks. Data transfers are inherently sequential, block 0 of a file must be read before the tape unit can find block 1. Files cannot usually be rewritten to the same bit of tape – writing to a tape effectively destroys all data previously recorded further along the tape (the physical lengths of data blocks, interrecord gaps, file marks etc vary a little with the tension on the tape so there is no guarantee that subsequent data won't be overwritten). All the processes using tapes, like skipping to file marks, sequential reads etc, are slow.

Modern "streamer" tape units used for backing up the data on disks use slightly different approaches but again they are essentially a sequential medium. Although transfer rates can be high, the time taken to search for files is considerable. Transfers of individual files are inconvenient; these streamer tapes are most effective when used to save (and, if necessary restore) all the data on disk.

1.4.2 Other I/O devices

Simple keyboards and printers

A keyboard and a printer are representative of simple Input/Output (I/O) peripheral devices. Such devices transfer a single data character, encoded as an 8-bit pattern, to/from the computer. When a key is pressed on the keyboard, internal electronics identifies which key was pressed and hence identifies the appropriate bit pattern to send to the computer. When a printer receives a bit pattern from the computer, its circuitry works out how to type or print the appropriate character.

Figure 1.15 shows, in simplified form, the general structure of a controller for one of these simple i/o devices. The controller will have a 1-bit register for a "ready flag". This flag is set when the controller is ready to transfer data. There will be an 8-bit data register (or "buffer" register) that will hold the data byte that is to be transferred. The controller will incorporate whatever circuits are needed to convert the bit pattern data value into output voltages/light pulses/…. The controller will be connected to the actual device by some cable. This will have at least two wires; if there are only two wires, the bits of a byte are sent serially. Many personal computers have controllers for "parallel ports"; these have a group of 9 or more wires which can carry a reference voltage and eight signal voltages (and so can effectively transmit all the bits of a byte at the same time).

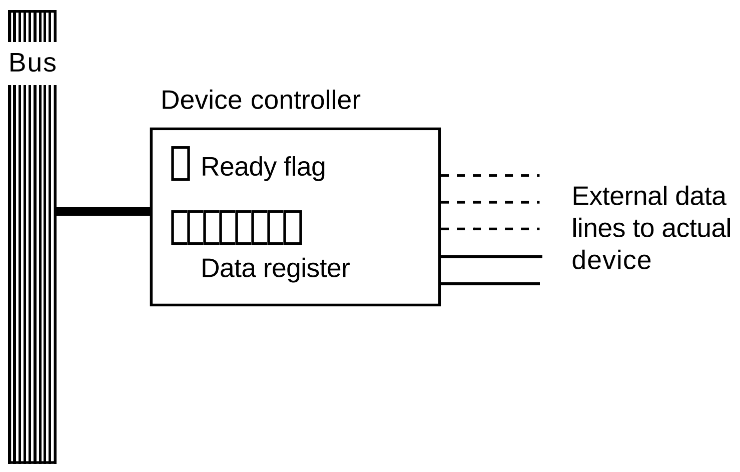


Figure 1.15 Controller for a simple i/o device such as a keyboard.

A simple approach to handling input from such a device is illustrated in Figure 1.16. The mechanism illustrated uses a wait loop – the program code makes the CPU wait until data arrive from the device. The program code would be:

```
repeat
    ask the device its status
until device replies "ready"

read data
```

The "repeat" loop would be encoded using three instructions. The first would send a message on the bus requesting to read the status of the device's ready flag. The instruction would cause the CPU to wait until the reply signal was received from the device. The second instruction would test the status data retrieved. The third instruction would be a conditional jump going back to the start of the loop. The loop corresponds to panes 1 and 2 in Figure 1.16; these two panes just show the exchange of signals caused by execution of the instruction.

When a key is pressed on the keyboard (pane 3 in Figure 1.16) the hardware in the keyboard identifies the key and sends its ASCII code as a sequence of voltage pulses. These pulses are interpreted by the controller which assembles the correct bit pattern in the controller's data register. When all 8 bits have been obtained, the controller will set the ready flag.

The next request to the device for its status will get a 1 reply (step 4). The program can continue with a "read data register" request which would copy the contents of the device's data register into a CPU register. If the character was to be stored in memory, another sequence of instructions would have to be executed to determine the memory address and then copy the bits from the CPU register into memory.

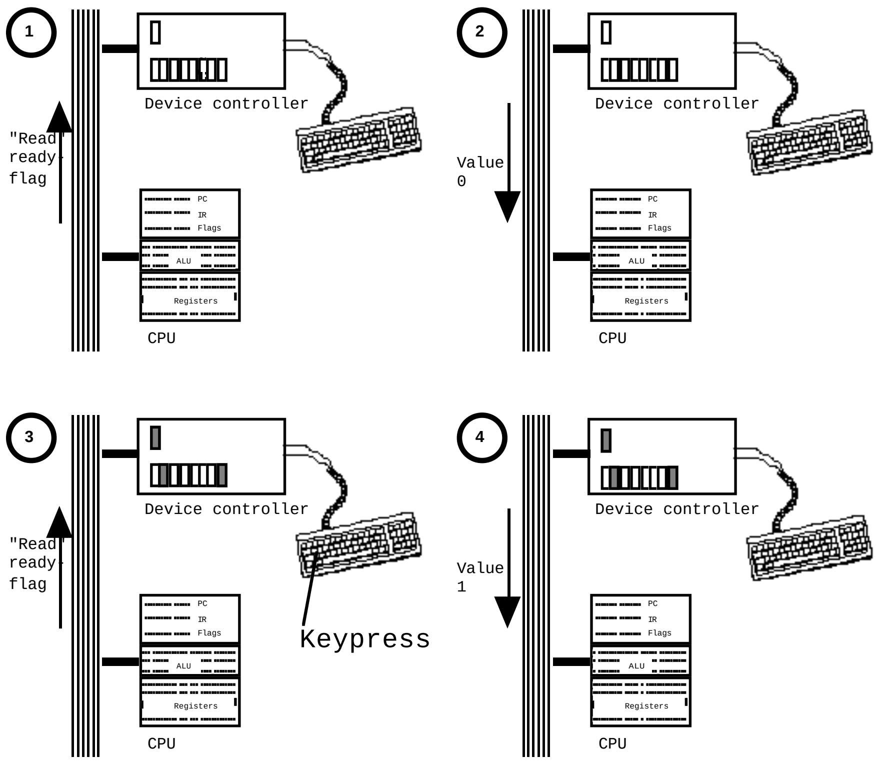


Figure 1.16 Input from a simple character-oriented device like a keyboard.

A wait loop like this is easy to code, but makes very poor use of CPU power. The CPU would spend almost all its time waiting for input characters (most computer users type very few characters per second). There are other approaches to organizing input from these low-speed character oriented devices. These alternative approaches are considerably more complex. They rely on the device sending an "interrupt signal" at the same time as it sets its ready flag. The program running on the CPU has to be organized so that it gets on with other work; when the interrupt signal is received this other work is temporarily suspended while the character data is read.

Visual display devices

Visual displays used for computer output have an array of data elements, one element for each pixel on the screen. If the screen is black and white, a single bit data element will suffice for each pixel. The pixels of colour screens require at least one byte of storage each. The memory used for the visual display may be part of the main memory of the computer, or may be a separate memory unit. Programs get information displayed on a screen by setting the appropriate data elements in the memory used by the visual display. Programs can access the data element for each pixel. Setting individual pixels is laborious. Usually, the operating system of the computer will help by providing routines that can be used to draw lines, colour in rectangles, and show images of letters and digits.

Clocks

There are many other input and output devices that can be attached to computers. Computers have numerous clock devices. Apart from the high frequency clocks that control the internal operations of the CPU and the bus, there will be clocks that record the time of day and, possibly, serve as a form of "alarm clock" timer. The time of day clock will tick at about 60-times per second; at each tick, a counter gets incremented. An alarm clock time can be told to send a signal when a particular amount of time has elapsed.

A-to-D converters

"Analog-to-Digital" (A-to-D) converters change external voltages ("analog" data) into bit patterns that represent numbers ("digital" data). A-to-Ds allow computers to work with all kinds of input. The input voltage can come from a photo-multiplier/detector system (allowing light intensities to be measured), or from a thermocouple (measurements of temperature), a pressure transducer, or anything else that can generate a voltage. This allows computers to monitor all kinds of external devices – everything from signals in the nerves of frog's leg to neutron fluxes in a nuclear reactor. Joystick control devices may incorporate simple forms of A-to-D converters. (Controllers for mice are simpler. Movement of a mouse pointer causes wheels to turn inside the mouse assembly. On each complete revolution, these wheels send a single voltage pulse to the mouse controller. This counts the pulses and stores the counts in its data registers; the CPU can read these data registers and find how far the mouse has moved in x and y directions since last checked.)

The controller for an A-to-D will be similar to that shown in Figure 1.15, except that the data register will have more bits. A one-byte data register can only represent numbers in the range 0 to 255; usually an accuracy of one part in 250 is insufficient. Larger data registers are used, e.g. 12-bits for measurements that need to be accurate to one part in four thousand, or 16-bits for an accuracy of one part in thirty thousand. On a 12-bit register, the value 0 would obviously encode a minimum (zero) input, while 4095 would represent the upper limit of the measured range. The external interface parts of the A-to-D will allow different measurement ranges to be set, e.g. 0 to 1 volt, 0 to 5 volt, -5 to 5 volt. An A-to-D unit will often have several inputs; instructions from the CPU will direct the controller to select a particular input for the next measurement.

D-to-A converters

A "Digital-to-Analog" (D-to-A) converter is the output device equivalent to an A-to-D input. A D-to-A has a data register that can be loaded with some binary number by the CPU. The D-to-A converts the number into a voltage. The voltage can then be used to control power to a motor. Using an A-to-D for input and a D-to-A for output, a computer program can do things like monitor temperatures in reactor vessels in a chemical plant and control the heaters so that the temperature remains within a required range.

Relays

Often, there is a need for a computer to monitor, or possibly control, simple two-state devices – door locks (open or locked), valves (open or shut), on/off control lights etc. There are various forms of input devices where the data register has each bit wired so that it indicates the state of one of the monitored devices. Similarly, in the corresponding output device, the CPU can load a data register with the on/off state for the controlled devices. A change of the setting of a bit in the control register causes actuators to open or close the corresponding physical device.

EXERCISES

1 A computer's CPU chip is limited in the number of circuits that it can contain. CPU designers have the equivalent of a million transistors from which to build their circuits. A CPU designer can implement circuits that perform many different kinds of data manipulation. Alternatively, a designer can chose to implement fewer distinct operations, instead using their circuits to duplicate elements (allowing two or more operations to proceed simultaneously) or so as to have more high speed registers. One approach gains speed by the program needing to execute relatively few instructions to achieve complex data manipulations; the other approach gains speed by having much faster, though simpler operations. Designers dispute which approach is best.

   Research these two approaches – they are known by the acronyms CISC and RISC. Write a brief report summarizing the information that you were able to obtain.

   The CPU used in the machine that you will be using for your course will be based on one of these design approaches. Which one?

2. Produce a table containing summary statistics on the machine that you will be using for your course. How much main memory? Does the video display use a separate memory, if so how much? How many colours can the video display use? What is the hard disk capacity? What I/O devices are attached? What is the CPU? ...

3. Charles Babbage, an English mathematician of the early nineteenth century, designed a number of calculating engines. The first, the "difference engine", was a special purpose calculator used to compute tables of functions (e.g. tables giving sines of angles); although Babbage never completed his model, a working version of the difference engine was marketed by a Swedish company in the mid nineteenth century.

   Babbage's other machine was the "Analytical Engine". This was a general purpose programmable computing device – a real computer that was to be constructed entirely from cog wheels, gears, pistons and similar mechanical components. Some computer historians suggest that had it been built the Analytical Engine would have been a fully working computer analogous to modern machines, albeit very much slower.

   Attempts at building the Analytical Engine failed; the technology available in the 1830s didn't permit construction of mechanical components with the precisions required.

   Suppose it had been possible to build these machines. Write a "science fantasy" essay based on the assumption that the Babbage Computer Mark 1 entered commercial production in January 1840. (Possible ideas – a system manager's manual for steam driven computing devices, Charles Dicken's Mr. Scrooge as a database administrator, or a maybe a more serious essay exploring differences in how society might have evolved.)