Chapter 2.2 The computer

The Internet depends on computers

The Internet is a network of computer networks, thus an Internet doesn't seem to make much sense without computers. But there is a much deeper relation between the Internet and computers.

One defining characteristic of the Internet is that data is “dumped” into the net in a format that includes the address of the sender and recipient. No previous attempt has been made to establish a link to the recipient, if it even exists. Instead, the delivery of the data relies on a series of specialized computers on the Internet, routers, to make decisions and take actions in order to deliver the data. As the message moves from one link to another, there are often delays of up to hundreds of milliseconds, since the traffic load and capacity of the links can be very different. A computer is needed at these nodes to store and then retransmit the message. Computers record the delivery times of messages through different routes in order to make routing decisions for future traffic.

A second characteristic of the Internet is its ability to transmit data through different types of networks in the process of getting the message from sender to recipient. The networks typically use different formats, and computers are needed to transform the data from old into the new formats.

Some Internet applications require messages to be retransmitted if delivery acknowledgement is not received in an appropriate time. Computers are needed to implement this kind of procedure. On the Internet computers are everywhere doing everything.

At the time the Internet was being developed and tested most of the communication engineers that designed and maintained the telephone network thought the scheme for the Internet was so complicated, and relied so heavily on computer processing, that it had no chance of actually working. Only a computer nut could believe it could function, and some of them were not so sure either. In the next chapter we will explore some of the details of the work that computers have to accomplish to get the Internet to work.

What is a computer?

This is actually not such an easy question to answer. A computer is a device that can make logical decisions, but how complex does the logical task need to be for the device to be called a computer? The problem is that there is a continuum of devices that might be called computers, from a thermostat that closes a switch when the temperature drops below a set point to a supercomputer solving partial differential equations to predict the weather.

It is a little easier to describe a desktop, or personal computer (PC[1]). The PC must be able to run programs that do certain common tasks: act as a word processor, make spreadsheets, make and manipulate data bases, process and store pictures and music, play games, etc. The PC also has familiar components; a display, a keyboard, devices to read and probably write to removable disks, e.g. floppy disks, CD-ROMs.

However, the hidden, or "embedded" computer in your car may be as or more powerful as your PC. This computer receives data from sensors attached to the engine, does all sorts of complex calculations, and sends commands to the engine to keep it running smoothly as the temperature and other driving parameters change. It doesn't need a disk drive because it doesn't need to store a large amount of data. It runs only one program, and that program was loaded into the computer memory when the car was made. It needs to store only a small amount of data. It doesn't need a keyboard and display because it uses the "instruments" in the dashboard as its display (although some high end cars do have displays). However, it saves some information and can send it to the computer that is attached when the car is serviced.

Many of the computers that run the Internet are like the embedded computers. They don't need a keyboard or display because they can be accessed from the Internet using another computer. They aren't usually on the top of a desk because it's more convenient to run them in a closet or special room that has plenty of air conditioning, electrical power, and sound insulation. However, they are still computers.

Basis of the general purpose computer

The concept of the modern general purpose digital computer was first clearly articulated by John von Neumann in the 1940s. At that time there already existed very complicated and ingenious machines for processing digital data. IBM had created the data processing industry using machines that could record data by punching holes into cards, calculate totals and other functions from this data, and sort cards into categories. However, the tasks these machines were to perform were specified before the operation by human operators that moved switches or levers that defined the task for that run. Von Neumann's insight was that the "task" was just another kind of "data". Thus, the operation that was to be preformed on a deck of punched cards could be specified by information coded on another deck of punched cards. This turned out to be a revolution, as it enabled far more complicated sets of instructions to be used and created the field of computer programming.

Logic and the computer

A computer must be able to perform operations on specified data in response to commands contained in a computer program. In an actual computer the data may be in the form of large words, say 32 bits long. Some of the commands may be quite complicated, and there can be hundreds of different commands. However, all these commands can be implemented on data of any size by a collection of only three types of basic logical operations acting on one bit of the data at a time. The three basic

[1] Of course I am using PC in a generic way here, and not implying that it is a descendant of the IBM PC which uses an x86 CPU and a of Microsoft Windows operating system

operations implemented by these devices are AND, OR, and NOT, illustrated in Figure LogicGates.

Figure LogicGates

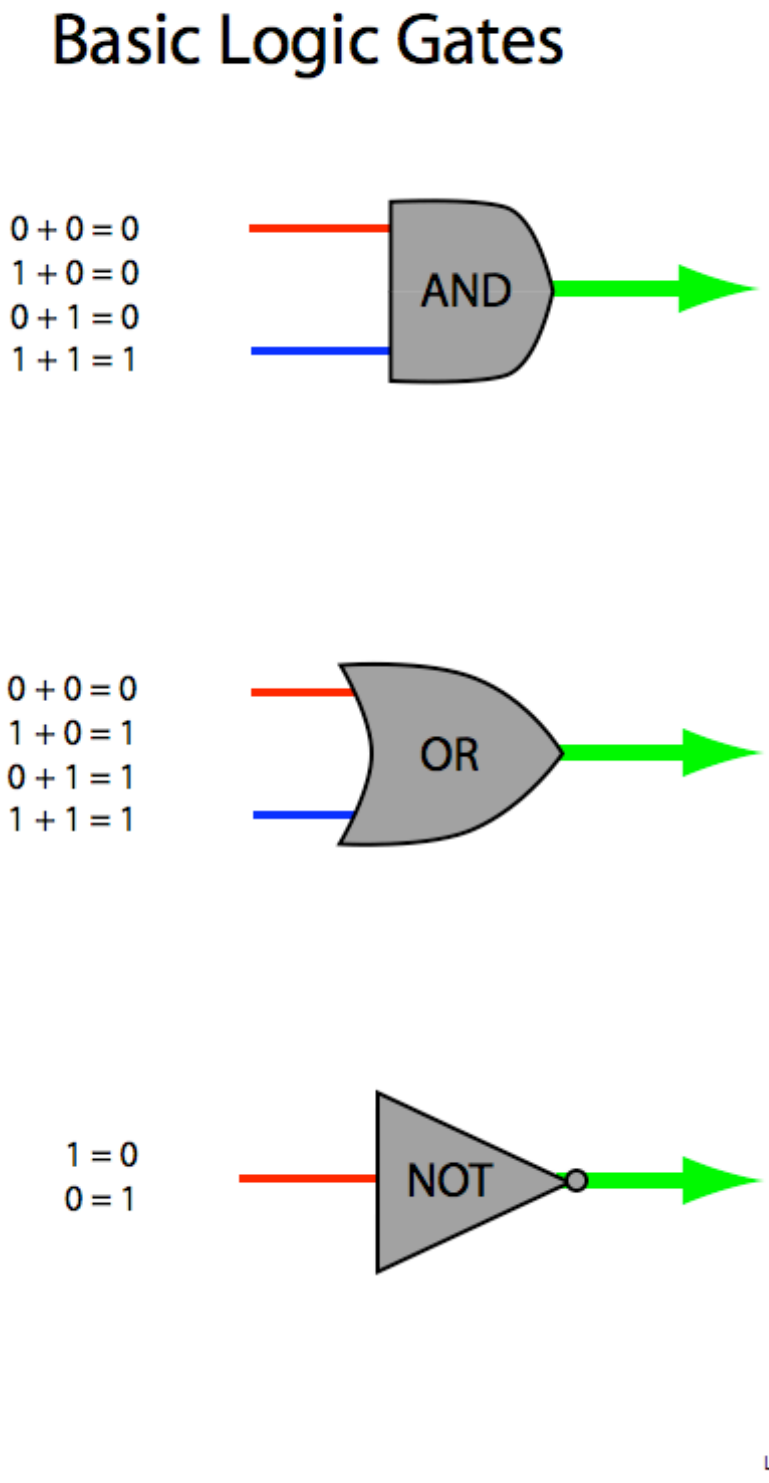


Figure LogicGates. The three basic logical operations are: AND, OR, NOT. The input data lines are the red and blue lines on the left, while the output in the green arrow on the right. AND requires 1 on both inputs to output 1 while OR requires 1 on one or both inputs to output a 1. The output of NOT is the inverse of the input. The symbols are those used by hardware engineers to design logic chips that can do complicated operations.

These devices are called gates because they can be visualized as letting a signal pass if the inputs are appropriate, although that analogy doesn't really apply to the NOT gate. The three basic gates can be combined to make a layer of slightly more complicated devices that are needed so often that they have their own symbols, such as the NOR, which is a combination of OR and NOT. These devices can be combined again to form the next layer of complexity, such as an ADD unit. Layer by layer the designer creates a chip that may contain several million individual gates.

Logic gates are made using semiconductors

Logic gates can be (and have been) constructed using clock-like mechanical devices, electrical or pneumatic relays, and vacuum tubes. However, today logic is constructed using semiconductors, because they are many orders of magnitude smaller and cheaper, and use less energy. Semiconductors act as electrically controlled switches due to the quantum mechanics of electrons in crystals.

The manufacture of semiconductor devices is big business, since computers, automobiles, cameras, cell phones, and on and on, use them. Thus, a huge amount of intellectual and financial resources have been focused on the development and manufacture of these devices.

When semiconductor devices were first introduced, they contained a single transistor in a physical container about 5x5x5 mm in size, and three leads. These units were about 10 times smaller in each dimension than vacuum tubes, and used orders of magnitudes less power, and so were a vast improvement. However, individual transistors had to be mounted on a board and the terminals connected by wires or stripes of metal to make a complete device.

Since the dimension of the actual transistor was much smaller than the device, and it did not produce significant heat compared to a vacuum tube, it was possible to place several transistors on a single small semiconductor crystal and make all the electrical connections between then using thin strips of metal on the surface of the crystal. Thus the physical units, the chips, used to make a computer, TV receiver, or radio began to carry out complex functions that previously required ten or a hundred individual transistors. The most complex circuits are needed in computers, and the devices used in them became known as large scale integrated devices (LSI).

The technology used in the manufacture of large scale integrated (LSI) devices is as least as impressive as the devices themselves, but that's another story. The basic process, photolithography, consists of focusing several optical images of components of desired circuit onto the surface of the crystal which has been coated with a light sensitive material. In a complex series of chemical processes the images become a conducting layer of metal on the surface of the crystal. Repeated application of this process finally produce a device that can contain several millions of individual gates (or transistors) per square cm. Since semiconductor technology is so important, it should be interesting to explore the physics that makes it possible.

Electrons in a crystal

A crystal is a regular array of atoms, with a simple example being diagramed in Figure xtal.

Figure xtal

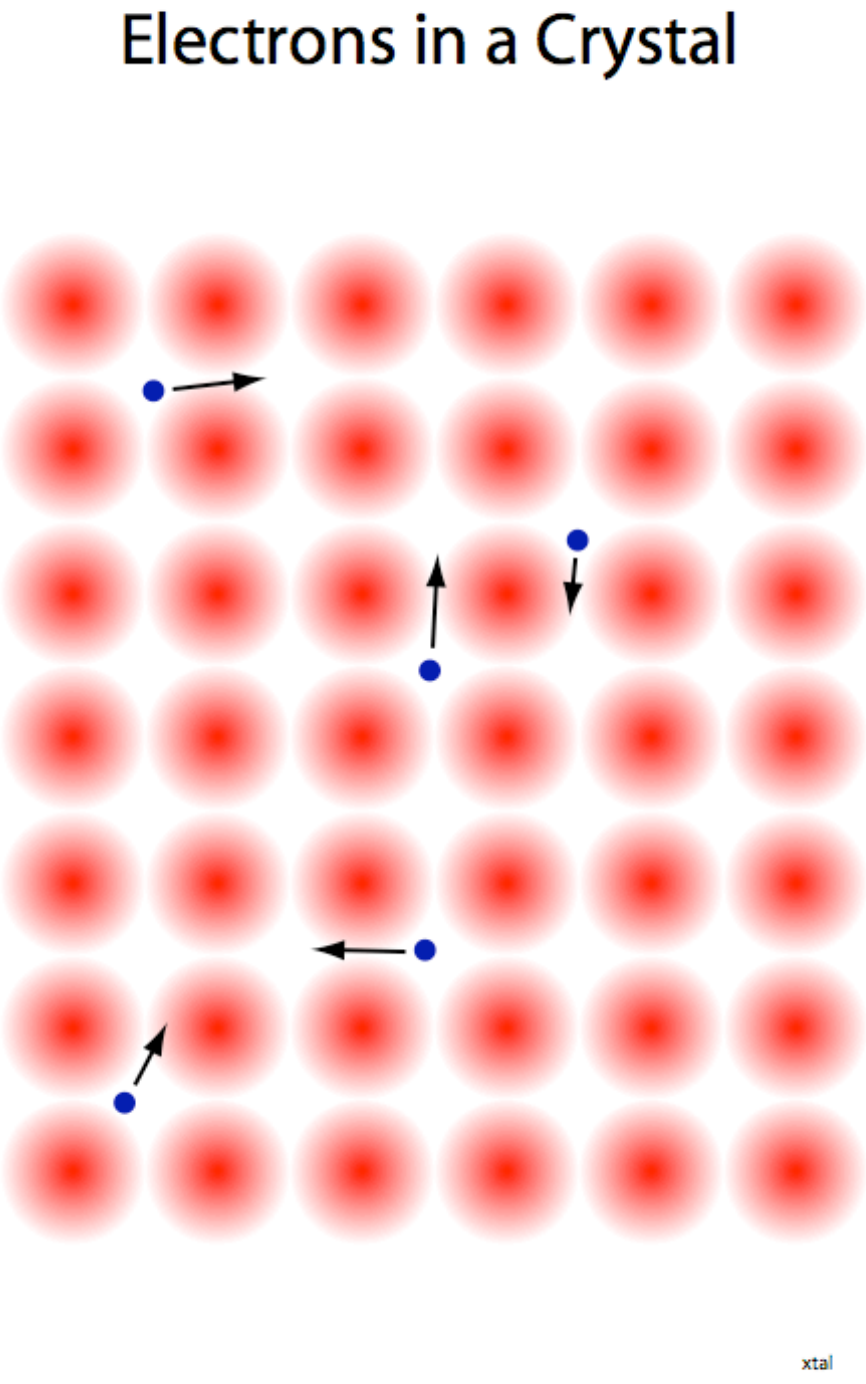


Figure xtal. This crystal is a simple cubic array of atoms. The majority of the electrons are closely associated with a specific atom, and hold the atoms together in the crystal matrix. However, a few electrons can wander around the crystal, and thus generate an electric current if a voltage is applied.

Most of the electrons in the crystal are tightly associated with the atoms. These are called valence electrons because they are responsible for the valence forces that hold the atoms of the crystal together. However, there may also be loosely bound electrons, the number depends on the chemistry of the crystal, which can move under the influence of an electric field to produce a current. If a small electric field produces a large current of electrons the crystal is a good conductor.

There are three complications to this picture and all are the result of the fact that objects as small as electrons can be understood only using quantum mechanics. Quantum mechanics says that in a crystal (or anywhere) electrons (or any particles) can only have certain, discrete energies which are related to each other as an integer series, i.e. 1, 2, 3, …. That is the first complication.

Electrons are Fermions[2], which means that only two electrons can have the same energy. That is the second complication. This means that as you add electrons to a crystal they "fill up" the discrete energy levels starting from the lowest energy level. It's like filling up a glass of water, the first water molecules go to the bottom of the glass and then gradually fill the glass. Since the bottom of the glass represents the lowest energy (due to gravity) you could say that the lower energy levels are filling up just like electrons fill up a crystal.

The third complication is that, due to quantum mechanics, certain bands of energy are not available to electrons. The size and energy of these forbidden bands, or gaps, depend on the nature of the crystal. As you might imagine, these gaps can have a profound influence on the electronic properties of the crystal. However, before looking at several types of crystals, we need to explicitly define the properties a crystal must have in order to conduct electricity.

To conduct an electrical current there must be free electrons and vacant nearby energy levels (holes) for the electrons to move into. In Figure ElectronBands we see five crystal types that have very different gap structures and corresponding differences in electrical conductivity.

[2]Named after the physicist Enrico Fermi, who received the Nobel Prize in 1938. The other type of particle is the Boson, named after the physicist Satyendra Bose. Any number of Bosons can occupy a single energy level.

Figure ElectronBands

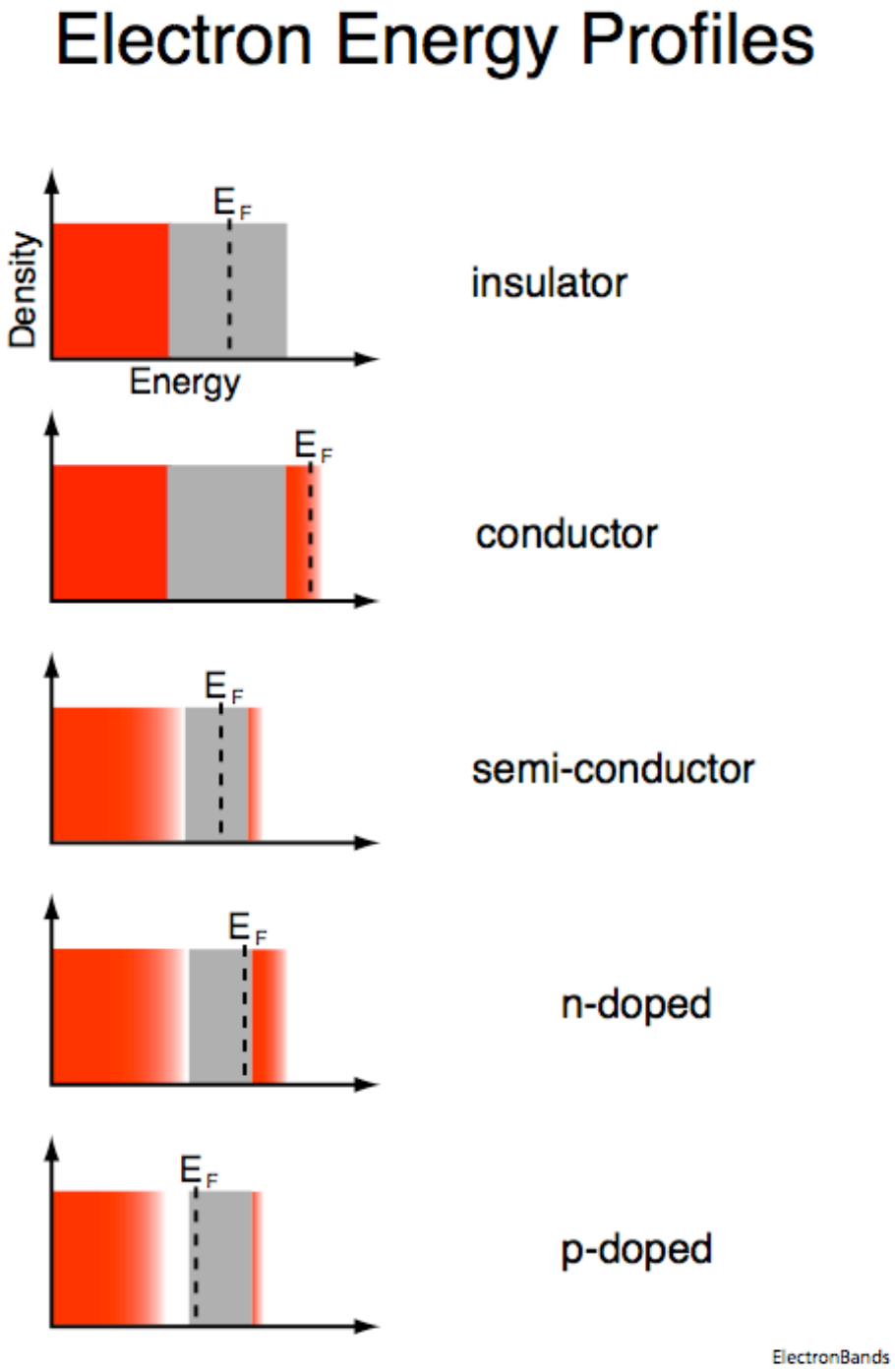


Figure ElectronBands. The distribution of electron energies is confined by the position of the gap and E_F, the Fermi energy. There are no free electrons in the insulator, but the conductor has both free electrons and adjacent empty levels. The semi-conductor has fewer free electrons than the conductor because the electrons must jump higher in energy to get above the gap. Conductivity can be improved by either adding trace metals that increase the number of free negative electrons (n–doped) or free positive holes (p–doped).

At room temperature electrons have sufficient thermal energy to jump to energies slightly above E_f, however, in the crystal represented by the first panel of Figure ElectronBands the levels below the gap are full, and E_f is too high to permit electrons to jump above the gap. No electrons and no holes means no conduction; this crystal is an insulator.

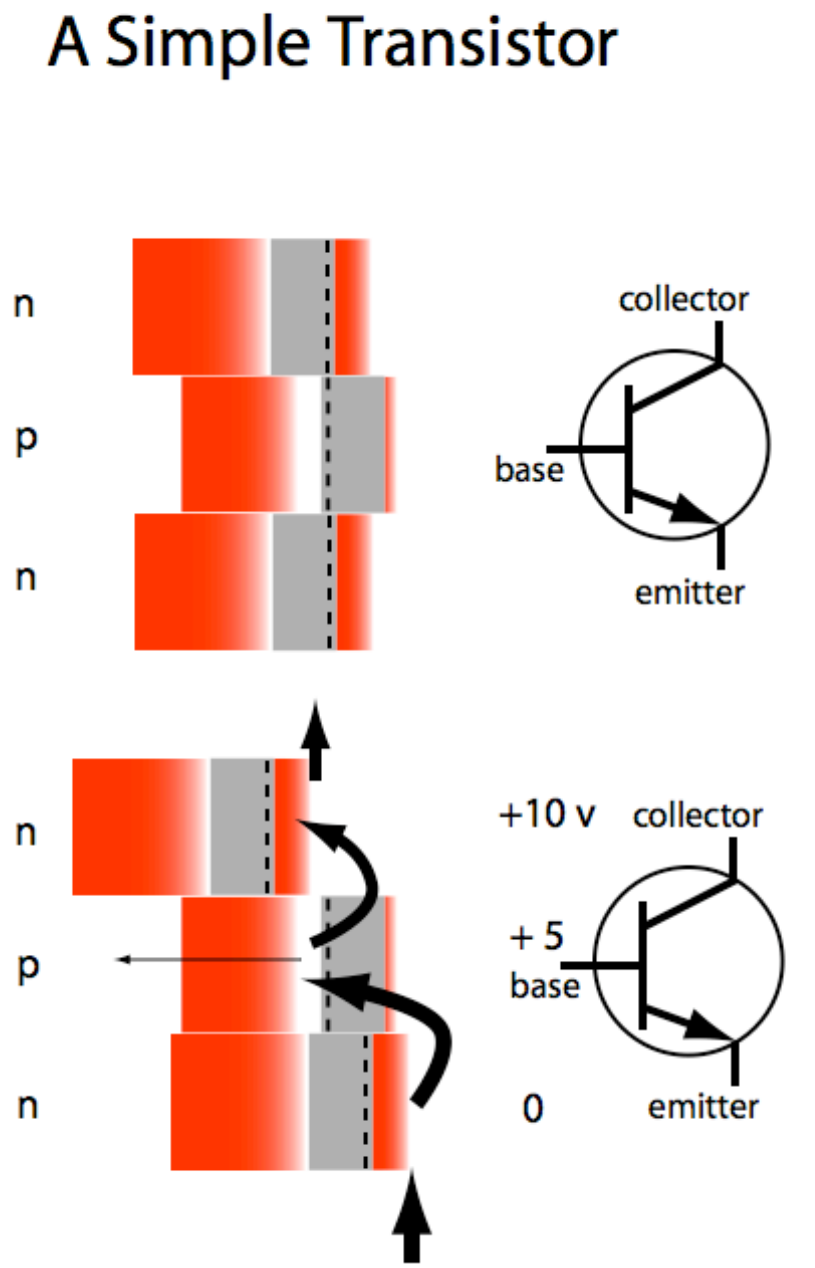
In the next crystal there are sufficient electrons to fill levels above the gap, which makes all the difference in the world. Near the top of the electrons distribution there are plenty of free electrons and holes so the crystal is a good conductor.

The semi-conductor crystal is intermediate between the first two. The energy gap is smaller than the gap in the insulator, and so some electrons now have enough thermal energy to get above the gap. These electrons, and the holes they left behind, can then conduct a current, but not nearly as well as the conductor. The conduction of the semi-conductor can be improved either by adding a small amount of a metal that donates negative charged electrons (n-doped) or by adding a small amount of a metal that accepts electrons (p-doped), creating positively charged holes.

Transistors

The doped semi-conductors, when sandwiched together, make a composite conductor that can be controlled by a small electrical current. This is the magic, the payoff for wading through descriptions of gaps and Fermi energies. One type of sandwich is illustrated in Figure Transistor.

Figure Transistor



Transistor

Figure Transistor. This transistor is a sandwich of n, p, and n doped silicon. When the three materials are in electrical contact the three E_f must be the same, and thus the positions of the bands are shifted. If a small current is allowed to flow from emitter to base the relative positions of the three E_f shift to greatly increase the flow of current through the transistor. Thus the transistor amplifies this small control current. The symbols for the transistor are in the right hand column. Note that electrons flow from negative to positive.

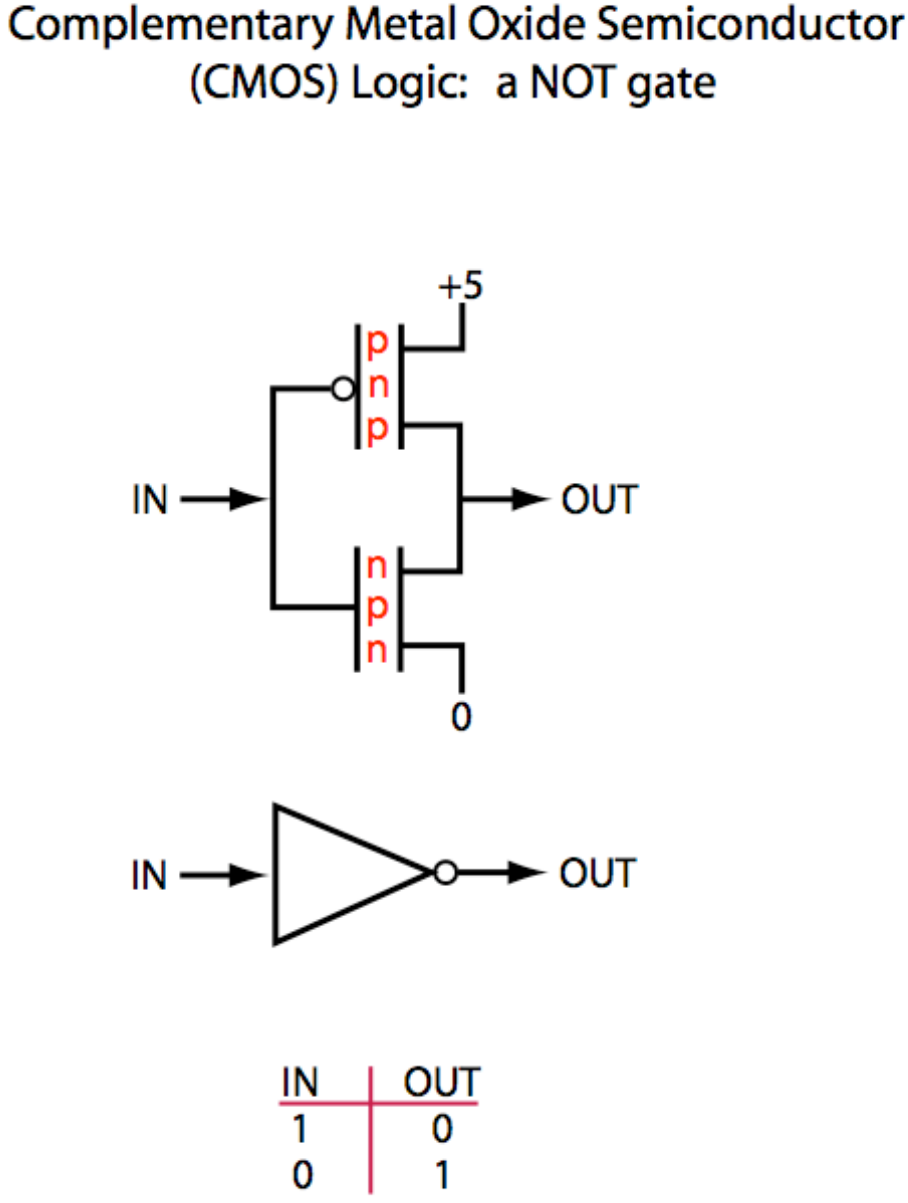
A three crystal sandwich and the resulting band structure is shown in the top panel of Figure Transistor. When the crystals with different band structures are pressed together so they connect electrically, a small current flows for an instant which establishes a voltage gradient at the boundary. After this happens the E_F of the crystals are equalized, which requires the band structures to shift.

In the lower panel of Figure Transistor a voltage is applied between the emitter and collector. A current will flow only if the voltage on the base is intermediate between the other two. Thus the base controls a current valve. Moreover, when the base has turned on the valve, only a relatively small current flows through the base. Thus this semiconductor sandwich is an amplifier! It is necessary for the valves (or relays, or gates, or whatever you call them) to also be amplifiers because there will always be losses as a signal proceeds through a complex series of gates. Without amplification the signal would be progressively decrease until it was ineffective. In addition, in many cases the output of one gate must drive two or more subsequent gates. This can only be done if there is amplification. Amplification could always be obtained by adding special amplifiers, but it's much more efficient to have the gates themselves amplify the signal.

The gates typically used in computers are slightly different than shown in Figure Transistor. First, current flowing from emitter to collector is controlled with an electric field, not an electric current, across the base segment of the transistor. This improvement is called a field effect transistor (FET). In this transistor a layer of silicon oxide, a good insulator, is created between the surface of the base segment and its electrode.

Secondly, a pair of npn and pnp transistors are used to provide a symmetrical response to a positive and a negative input. The combination of these two changes is called a Complementary Metal Oxide Semiconductor, CMOS. The use of this technology to create a NOT gate is illustrated in Figure CMOS.

Figure CMOS



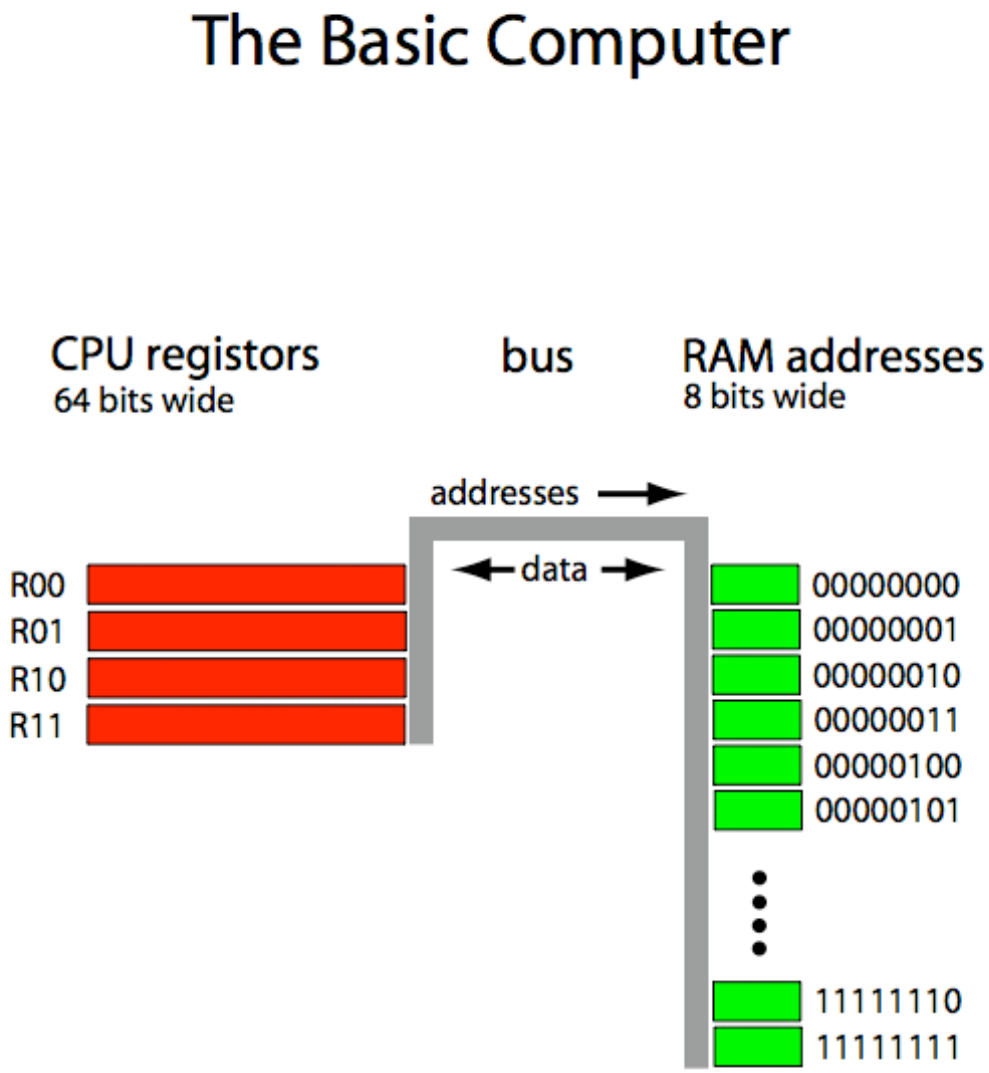
CMOS

Figure CMOS. The first diagram shows that the INPUT is insulated from the transistors by a layer of metal oxide, and thus only an electric field controls the flow of current through the devices. A pnp and a npn device are connected in series to produce a unit that responds symmetrically to a positive and negative input. This particular device produces the inverse of the input, it is a NOT gate. The second panel shows the standard icon used by engineers to represent this gate and the last panel shows the logic matrix, i.e. the INPUT vs OUTPUT.

The basic computer

The basic components of a general purpose computer are the central processor unit (CPU) and random access memory (RAM). Both of these devices contain addressable memory which means that data can be stored and retrieved using an address.

Figure Computer1



Computer1

Figure Computer1. The central processor unit (CPU) registers in this computer hold 64 bits. Some segments of these 64 bits may contain data and some instructions. Data is stored to and retrieved from random access memory (RAM) by the CPU. Data is typically stored in RAM in byte (8 bit) segments, each with a unique address. However, a data item may be several bytes long and segments of multiple bytes may be transferred to and from RAM by a single command.

Central processing unit (CPU)

A CPU has several addressable registers with each storing 8 to 64 bits of data, depending on the type and model of computer. However, the registers in a CPU can be rather complicated since there are often several types with different sizes and purposes. In addition (no pun intended), the purpose of registers is to implement logical and arithmetic operations, not just store data. The functioning of a typical CPU will be more obvious when we describe a simple computer program in a later section of this chapter.

Random access memory (RAM)

Random access memory (RAM) consists of a large number of identical storage locations which hold most of the data that is being used by the computer during the execution of a program. Data is stored in RAM by sending an address along with the data, and data is also retrieved by address. The “random” in RAM doesn’t mean its operation is not predictable, but rather that data can be retrieved from any location in one operation, as contrasted to sequential memory, in which data must be stored and retrieved sequentially. Retrieving data in the middle of a sequential device requires reading all the preceding data.

The basic unit of memory is a capacitor, two metallic plates separated by an insulator. When a voltage is applied across the plates a transient current flows as an electric field forms between the plates; the capacitor has been charged. If the two plates are now isolated from the voltage, the energy stored in the field and the voltage between the plates remain. Thus, if a voltage detector is applied between the plates at a later time, the fact that the capacitor has been charged can be detected. This is a device to store one bit of information, as seen in the top of Figure RAM1.

Figure RAM1

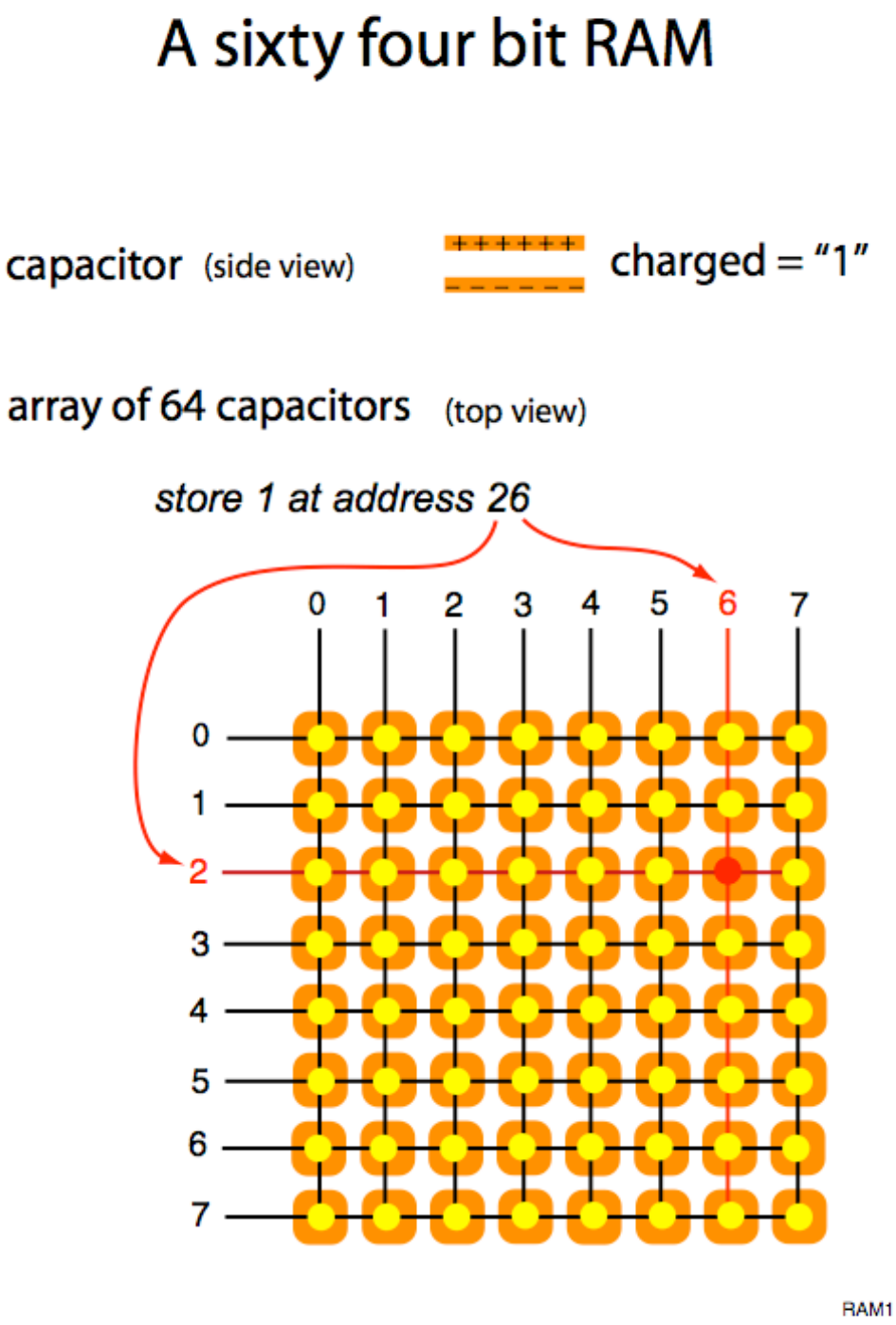

A sixty four bit RAM
capacitor (side view)
charged = "1"
array of 64 capacitors (top view)
store 1 at address 26
0 1 2 3 4 5 6 7
0
1
2
3
4
5
6
7
RAM1

Figure RAM1. Information is stored in arrays of capacitors as an electric charge (or voltage). A single memory chip contains an array of capacitors along with connecting address lines and supporting circuits. A binary address is used efficiently when the array has 2^n by 2^n capacitors. Thus commercial RAM chips will typically store 16, 64, 256, or 1024 Mb (where a Mb is actually 1024 x 1024 bits).

A hypothetical small RAM is diagrammed in Figure RAM1. It can store 64 bits of information, with each bit having a unique address. The key to efficient addressing is to arrange the 64 capacitors in an 8 x 8 array, so that the address to any bit can be described as a row and column index. In this example a voltage is applied to row 2 and column 6. All the address lines are connected to the individual capacitors by an AND gate, so only the capacitor at row 2 and column 6 is charged, storing a “1” at that address.

In this small RAM the advantage of using of rows and columns to address a memory location may not seem impressive; 16 lines versus 64 lines if the capacitors were arranged linearly. However, the advantage increases dramatically as the size of the memory module increases. A one Mb RAM requires 1,000 rows and columns, as contrasted to 1,000,000 lines if it were organized in a linear fashion.

However, computers store and retrieve at least one byte of information in one command, and thus each byte, not each bit, is assigned an address. To make a RAM module which stores bytes, eight one bit RAM chips are assembled into a module, as shown in Figure RAM2.

Figure RAM2

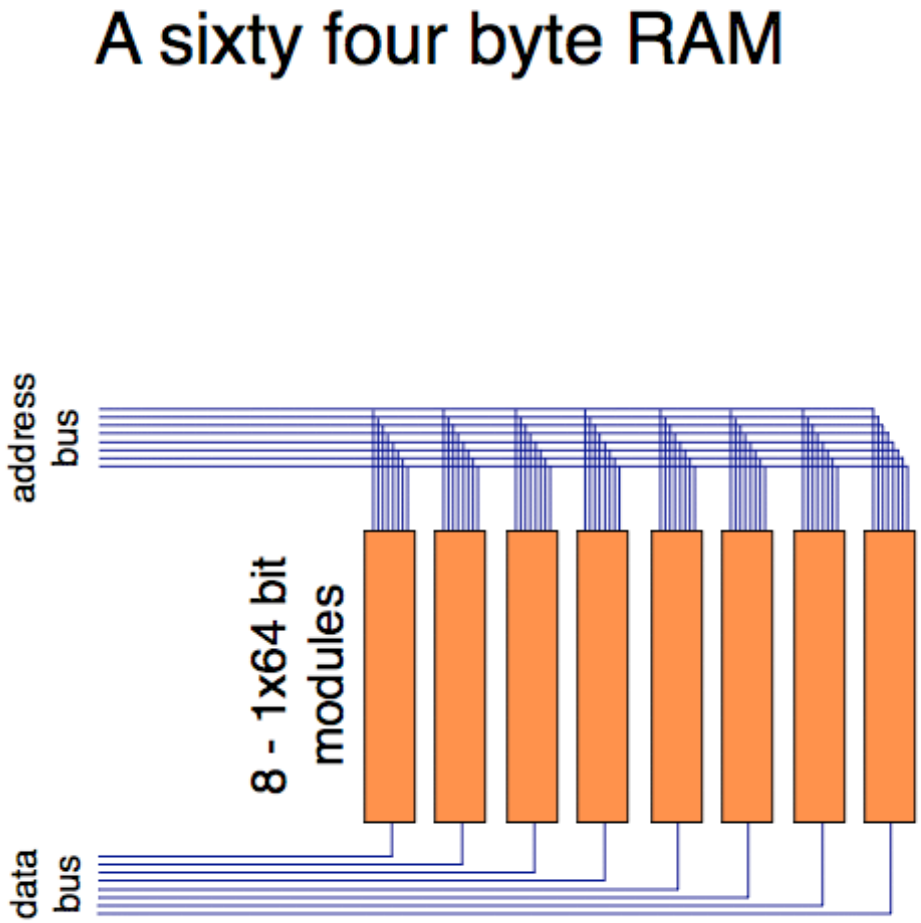


Figure RAM2. The eight one-bit RAM chips that make up this module share row and column address lines. The top 4 lines of the address bus specify the row while the bottom 4 lines specify the column . The row and column data from the bus is decoded on each chip to select the bit at row 2 and column 6 (see Figure RAM1). The data sent on the data bus is not decoded, it just sets the status of the corresponding bits on each chip.

The memory module is sent an address and a data signal. The address permits the corresponding capacitor in each chip to be set by the data signal. Thus, the address signal must be decoded into a row and column index by a decoder . Since the corresponding capacitor in each chip is to be enabled, all eight RAM chips receive the same address.

The data signal does not need to be decoded, rather the bits need only to be routed to the appropriate RAM chips. The first bit sets the capacitor in the first chip, the second the second, and so on.

There are actually several types of RAM, each appropriate to a specific use. The memory we have been talking about is usually implemented using dynamic (DRAM). This type is fairly fast and cheap, but as you suspect there must be a cost of some kind to pay or there would only be DRAM. One cost is that it “remembers” the data for only a microsecond or so. Thus, it must be refreshed very frequently. While refreshing requires additional components on the chip, DRAM is still a good bargain for this type of use.

While the CPU and RAM are the core of a computer, there are other components that are often needed.

Figure Computer2

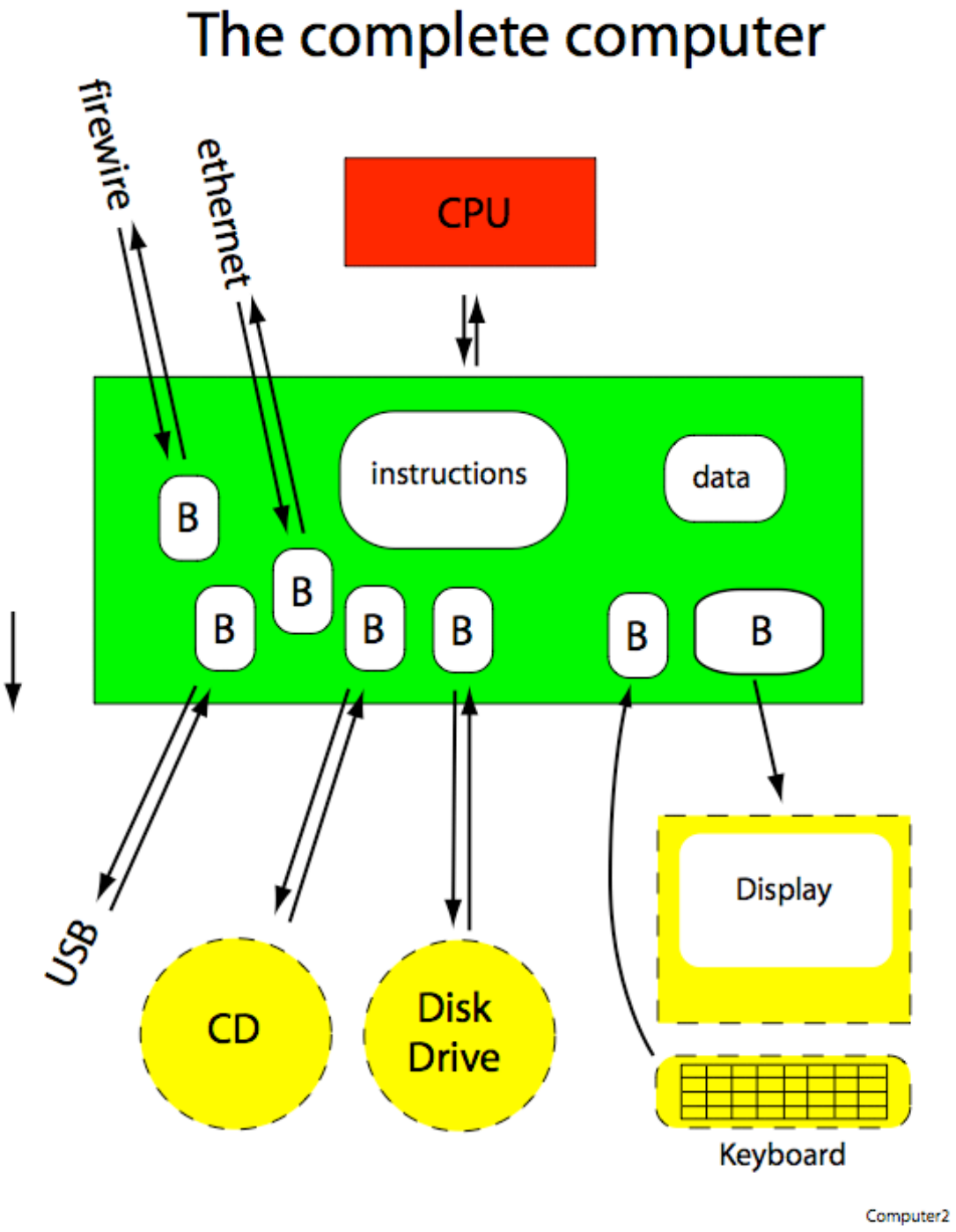


Figure Computer2. The CPU (red) performs operations on the data. The RAM (green) stores several types of data, including the instructions that constitute the computer program. A display, keyboard, disk drive and CD reader are needed for most computers. Universal Serial Bus (USB), Firewire, and Ethernet connections are required to communicate to the outside world. All of these devices send and received data, which is stored in buffers in RAM. In addition, special circuits are required to control the devices and format data.

Input and output

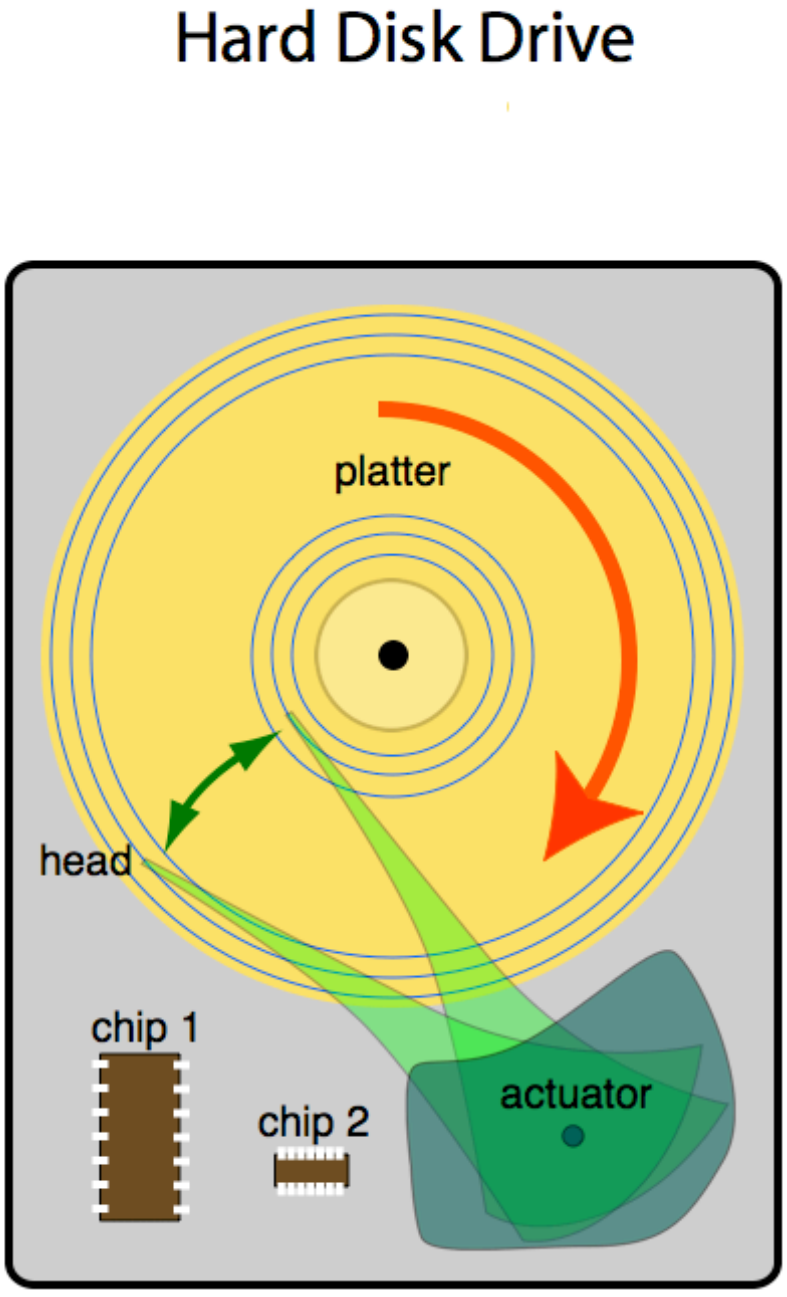
Most computers have connections that receive and send data from and to the external world. Of course the whole purpose of the computers that run the Internet is to receive and send data. Many of these machines (routers) must accept data from several lines and send data out on several lines. However, one CPU can only do one instruction at a time. Thus, there must be little memories, or buffers, attached to each input and output (in Figure aComputer the buffers are a small segment of RAM, but they can also be separate small memory units). An input buffer stores data that the CPU can read when it is not busy. The CPU then processes the data in that buffer. When the CPU has data to send, it doesn't wait for the output line to be free, the CPU dumps the data into a buffer and then works on another job. When the output line is free the buffer sends the data on its way.

If we had a high speed movie of the computer working we would see that the CPU and RAM are a very fast team. They work together to process chunks of data far faster than the data comes in or out. Thus, they can jump around to look at input buffers, and if there is data , it is processed by the CPU using the instructions in RAM and then dumped into an output buffer. It looks to a slow human like smooth, simultaneous processing of several streams of data, but it's actually a jumpy, rapid sequence of different activities.

Disk drives

A disk drive provides storage for a large amount of data at modest cost, on the order of 1 TB for $1000 (in the year 2005). However, writing and reading data to a disk is slower than accessing RAM, several msec compared to less than a μsec for RAM.

Figure HardDrive



HardDrive

Figure HardDrive. The thin magnetic disk (yellow) rotates clockwise and a read-write head at the end of an arm (green) flies along the surface. The data is written as patterns of magnetized spots along circular tracks (blue circles, but of course the tracks can not be seen by eye). To read data from the disk, the head is positioned over the appropriate track and the magnetized spots induce currents which can be decoded to recover the data. There are several chips (brown) which provide the logic to drive the heads to the correct tracks and process the current pulses.

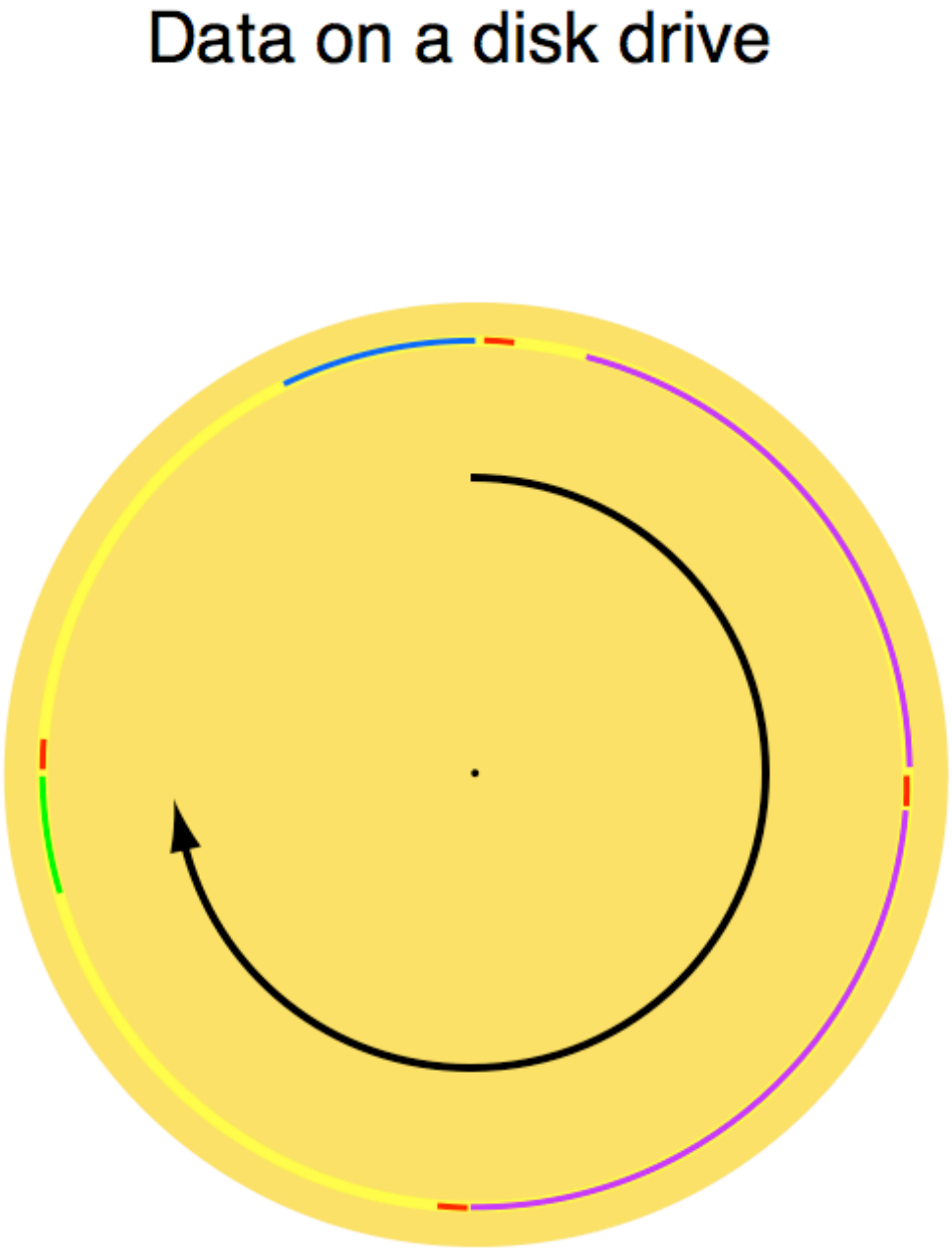
The hard disk drive[3] consists of a set of disks, that a motor rotates at about 10,000 rpm in a sealed box. Each disk is coated with a very thin layer of an iron-nickel alloy that is easily magnetized. Information is written on the surface of the platters by electrical currents flowing through tiny heads on the ends of arms. The resulting series of magnetic domains form concentric circles, or tracks, on the platters.

The position of each arm is controlled by an actuator, a coil on the end of the arm which moves between sets of magnets. Information written on the disk is recovered by a read head that is on the same arm as the write head. The arms gently push the heads down on the platters, but the rotation of the platters drags along air which floats the heads above the platter surface. The rotating air causes the heads to "fly" a fraction of a micron above the platter surface. The disk drive has heads for each disk surface, a disk drive that has 8 platters will have 16 sets of write-read heads on 16 arms.

The typical hard disk drive for a desk top compute might store 120 GB of data, on 3 platters with a diameter of 95 mm. This high density of information is achieved by using many circular tracks on each platter, separated by less than 1 μm, with magnetic domains of length 0.2μm. In order to write and read data with such small dimensions, the heads must be very small and their positions must be precisely controlled.

[3]The "hard" disk has rigid platters, as contrasted to the "floppy" disk, commonly used in desktop computers before 1990. The floppy disk, a flexible plastic disk in a paper envelope, could be inserted and removed through a slot in the front of the computer. In a later version of the floppy disk the paper envelope was replaced by a rigid plastic shell. However, the structure of these drives was not compatible with the extreme mechanical precision required for high density storage. The early hard disk drives also had platters that could be removed. However, these systems, like the floppy drives, did not have the mechanical reproducibility to achieve high information density. Thus, all modern hard disk drives are sealed units.

Figure DiskTracks



DiskTracks

Figure DiskTracks. Each circular track is divided into several equal sized sectors by a small servo block (red) followed by a data block. If the data (purple, green, blue) does not fill a data block, the remainder is unused. If the data requires more than one data block, more are used.

The data stored on a disk drive is located by using information stored on the directory file on that disk. This special file is essentially a table, indexed by the name of the file, giving the servo block addresses that correspond to the data blocks containing the file. The directory also contains a list of all empty data blocks. When a new file is written on the disk, the disk controller checks the directory before writing data. When a file is deleted, only the entry in the directory corresponding to that file is deleted, the file itself is not altered[4]. If a disk is fairly full, and has been used for some time, the files will tend to be fragmented because files that are deleted will not be the exact same size as the new files that written into the reclaimed space. The new files must thus be fragmented into segments that fit the empty data blocks. There are utility applications that will de-fragment a disk by collecting fragment of a file to create a contiguous segment that can be accessed more quickly.

Computer programs, software

A computer program is a list of commands that the computer is to execute sequentially. However, some of these commands may be branches. One type of branch command may instruct the computer to go back to an earlier point in the program and start execution again, forming a loop. Another type of branch command is conditional, instructing the computer to execute one or another set of commands depending on the value of a variable which in term has been computed by another segment of instructions. Thus, the actual program executed by the computer may be very complex, is not usually linear, and may not even be defined until the program runs and processes the data.

There is no universal set of commands that a computer executes, each model can have a different set. However, each set of instructions has the property that it can do any thing the other sets of instructions can do; the instruction sets are said to be "complete". Thus, a large instruction set can't do any tasks that a small set can't. However, large instruction sets can typically do tasks using a smaller number of instructions, because they contain complex instructions that are equivalent to two or more instructions from the small set. Larger sets are not necessarily better, because simple instructions often can be executed more rapidly than the complex ones.

The CPU is the chip that actually executes the commands, which are called OPCODES. An OPCODE is typically defined in terms of an operation and several memory locations, which may be in the CPU or RAM. Some CPU registers have special functions, e.g. the program counter register contains the location of the next instruction. A typical OPCODE might be: ADD 1,3. This might mean "add the contents of register 1 to the contents of register 3 and store the result in register 3". Doing this operation may not seem earth-shaking, but it's the stuff that computer programs are made of. “ADD 1,3” is of course a mnemonic for a machine instruction which must be

[4]If you want to actually delete the information you need to write over the file with a nonsense pattern. Many operating systems and utility applications can do this.

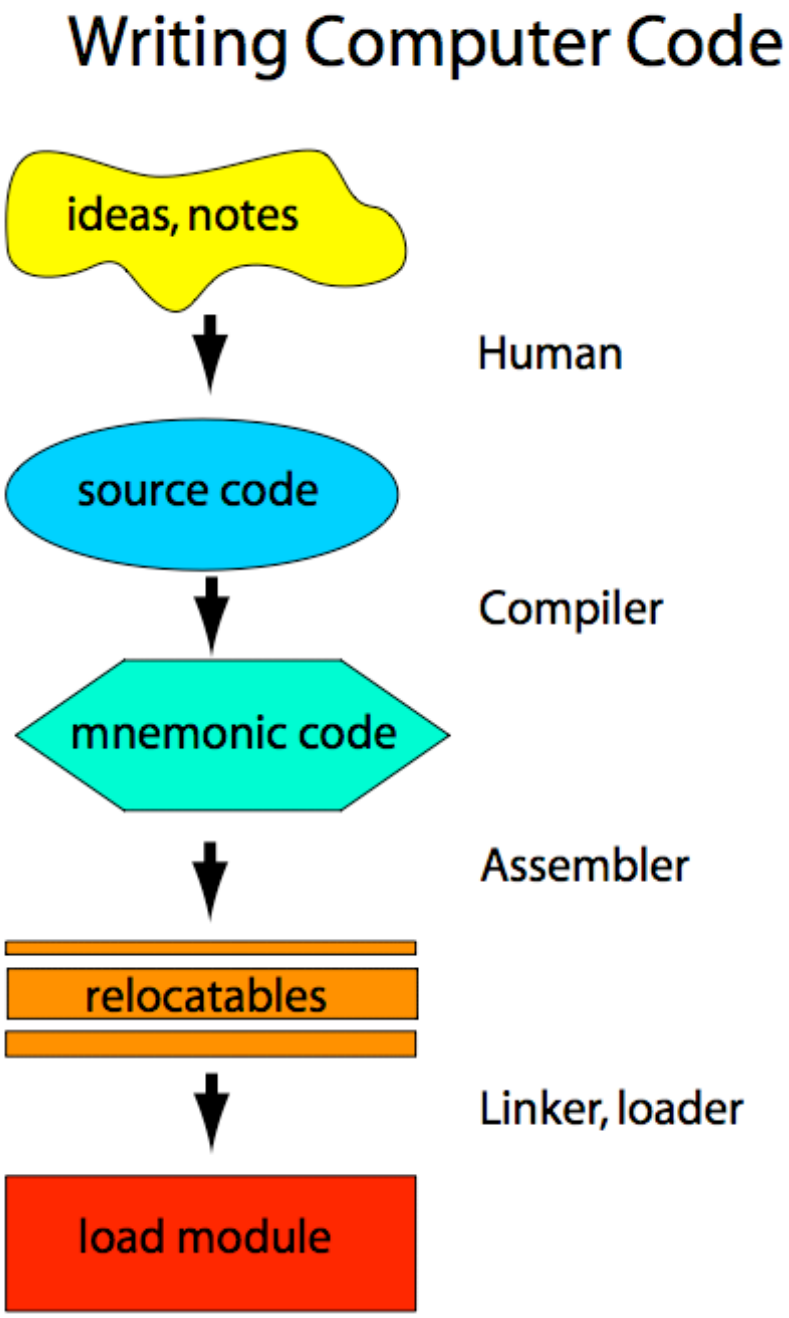
expressed in a binary format for it to be used by the CPU, e.g. “11010011010000111101010100001011”

Compilers and high level languages

It takes a human programmer considerable time and effort to learn the mnemonics for OPCODES of a specific CPU and become proficient in translating typical tasks into a series of these OPCODES. This overhead to writing code becomes more even more onerous since a programmer is unlikely to be able to base a career on a specific CPU; new CPUs are introduced every year. If, in addition to learning the latest “hot” OPCODE set the human programmer was required to write code in binary form it would be a crushing job.

However, computers themselves are at their best in doing huge, but repetitive and well defined tasks. Thus, the ability to produce complex computer programs was vastly improved as computers themselves were used in the process of writing code. As a first step programs called assemblers translated mnemonic code into binary and allowed programmers to designate memory locations as named variables instead of a binary addresses. They still were chained to the OPCODES of a CPU, but at least the details were handled by the assembler program.

Figure WriteCode



WriteCode

Figure WriteCode. The human programmer writes source code in a high level language. This language is translated by the compiler program into assembler code where each line of the code represents one machine instruction. The assembler program translates this code to machine readable code. Since all real programs consist of several packages of code the linker program is needed to substitute symbolic addresses where ever one package calls another. Finally the loader program substitutes machine addresses and loads the linked program into computer memory.

The next step in program development was the invention of "high level" languages which described operations without reference to any specific computer. As mentioned in the previous section, any computer can do anything that any other computer can do, thus we don't have to worry that an operation described by a high level language can't be implemented by a specific computer. The code in the high level language was then translated into OPCODES by a compiler program. The first high level language was FORTRAN (FORmula TRANslation), designed to enable even simple minded engineers and scientists to write computer code that could solve algebraic formulas. At first the "real" computer programmers whined that no mere compiler program could generate code that was as efficient and elegant as the code that they could write. Actually, this is sometimes true, but using FORTRAN is so much easier, and the code usually isn't that much slower that human code. As computers have become cheaper and faster, the limit is not usually the speed of the computer to run the code but the speed at which the programmer can write code, particularly good code that doesn't crash.

Computer programmers still work for their money. Writing code in a high level language is much easier than writing assembler code, but the programs have become much bigger and more complicated, so the total work remains about the same. FORTRAN seemed great when it was invented, but then computer engineers and scientists invented new high level languages. Every few years a new miracle language is invented and the old miracle language becomes passe.

In Figure Code a very simple program that adds two numbers is first presented in C, a high level language (the high level language code is often called source code, because it is the source of data to be processed by the compiler). The program is to be run on a UNIX type operating system (actually a Apple Macintosh G5 running operating system X).

Figure Code

C source code: written by programmer

CODE	COMMENT
int x,y,z;	declare x,y,z to be of type int
x=2;	put “2” into location x
y=4;	put “4” into location y
z=x+y;	add x to y and put result in location z

Assembler code: generated by complier

CODE	COMMENT	
mr r30,r1	move:	put (r1) into r30
li r0,2	load immediate:	put “2” into r0
stw r0,32(r30)	store word:	put (r0) into memory location (32 + (r30))
li r0,4	load immediate:	put “4” into r0
stw r0,36(r30)	store word:	put (r0) into memory location (36 + (r30))
lwz r2,32(r30)	load wd & zero:	put (32 + (r30)) into r2 and zero low half
lwz r0,36(r30)	load wd & zero:	put (36 + (r30)) into r0 and zero low half
add r0,r2,r0	add:	add (r0) to (r2) and put into r0
stw r0,40(r30)	store word:	put (r0) into memory location (40 + (r30))

Figure Code. The left column contains code, the right column comments. The top segment is a program written in the high level language C while the bottom segment is the translation into assembler language.

C source code

The first step in the C source code is to define all the variables we plan to use in the program. These variables are similar to those used in algebra, they are symbols that represent numbers, text, audio, etc. In the final machine code they will be the computer memory address for the first byte of the space that will store the data. Fortunately we never have to know the actual machine addresses for these variables since those assignments will be handled by the compiler, assembler, linker, and loader. However we do need to tell the compiler what type of data these variables will represent because that will determine how much memory space is allocated and how the data will be processed. In this example the three variables x, y, and z will all be integers, i.e. fixed point numbers[5]. They are declared to be "int" variables, which for this machine means they can be positive or negative integers that can be stored in one word which is 4 bytes; thus they must have an absolute value no larger than about 10^9.

The next two lines assign the values 2 to x and 4 to y, i.e. stores 2 and 4 at the memory addresses starting at x and y. The next and final line of this program assigns the value x+y to z, i.e. stores the sum of data in memory addresses x and y in the address z [6]. That's the end, I said it was a simple program. In a real program we would do something with z, at least display it on the screen, but here we just end.

Assembler code

There are more lines in assembler versus source code for several reasons. First, tasks necessary to implement the instructions in the source code must be explicitly listed. Second, a single line of C code, since it is like a line of an algebraic equation, can imply many basic operations, although in this very simple example that is not the case. Third, the complier may add some instructions that in general would be necessary, even though they may not be in a specific example.

The first line of the assembler code puts the contents of register 1 into register 30. We will see later what this data must be. However, note that we are now dealing with absolute machine addresses for the registers, although the external memory addresses will still only be implied.

In the next line we put the integer "2" into register 0, and in the next line we store the contents of this register into the external memory location "32" plus the contents of register 30. This external memory location is where all values of x will be stored. Now we know what was going on in the first line of the code; the starting external memory address for our data will be stored in register 1 by the loader and we have transferred

[5] Fixed point means that the user (or programmer) needs to keep track of the position and insert the decimal point. It is really an integer mode, and if the number is very small or very large the "extra" zeros use up valuable space in the registors. Only "floating point" format actually keeps track of the decimal point for you. However, floating point operations are slower, and rounding errors can produce unexpected results. Banks use "int" variables for most of their calculations.

[6] The similarity of C code to algebra can be misleading. These "equations" actually represent operations, not relationships. Thus, z only equals x + y after that operation has been executed. If in this program we had added the line "x = 8", z would retain the value 6 until the command "z = x + y;" had been executed again.

the address to register 30 because the complier may want to use register 1 for some other purpose. But why is the compiler transferring “2” to external memory? There are 32 registers in this machine, far more than will be needed to finish our simple task. The answer is that the complier does not have a global view of the program and our grand computational plan. It makes the assumption that we may use x at some later time, and thus it is best to store it in a safe place. Compliers may get points for being fast, but failing to implement source code even once is a fatal flaw, thus sometimes they must play it safe.

The next two lines of code are almost a repetition, the integer “4” is put into register 0 and then is transferred to external memory at the location “36” plus the contents of register 30. Note that this starting address is four plus the address of x, i.e. the compiler is allowing 4 bytes to store x. It does this because we have declared x to be an “int” in the first line of the source code.

The next two lines of code transfer x and y from external memory into registers 2 and 0. All registers in this machine are 8 bytes long, and x and y, each only 4 bytes long, are stored in the high half of each register. Since the low half of each register could contain junk, this portion must be zeroed.

The next line adds the contents of registers 2 and 0 and puts the sum in register 0. The “add” OPCODE is only appropriate for integers, and it is used by the complier because x, y, and z have been declared integers in the first line of the source code.

Finally, the sum, z, is stored in external memory location “40” plus the contents of register 30. Again, 4 bytes are allocated to store z.

Subroutines, layers, and crashes

The two biggest problems in writing computer code are just the total time required to do the job, and avoiding code that causes the computer to crash. It doesn’t take much imagination to appreciate the time required in writing a program of a million lines, but it may not be so obvious that a program can cause a computer to crash.

As in other professions, it is important for programmers to avoid "reinventing the wheel". Thus, good computer programs are built using subroutines which carry out common tasks[7]. This practice saves programmers time by reducing the total amount of code, and it allows the time to be focused on making sure the subroutines are well written. Complex tasks may be split into several programs, and the programs may grouped into levels. The problem of dividing a complex problem into smaller sub-problems is arguably the most creative part of programming.

The lowest layers of programs running on a computer comprise the "operating system". These programs handles all the housekeeping tasks of the computer, so that the programmers that write the "applications", e.g. a word processor, don't have to. Housekeeping includes keeping track of memory use, so two programs don't try to use the same address, organizing files on the hard disk so the users can find them, etc. One layer in most operating systems is the Graphical User Interface (GUI). This set of

[7] Program units have different names and structures, e.g. objects, subroutines, functions, drivers, but for this general discussion I call them all subroutines.

subroutines generates the windows and icons that allow the user to interact graphically with the computer using a mouse. Programmers writing a new spreadsheet program for a PC don't need to write the code to actually draw a window. Instead they insert an instruction in their code that specifies only that they want a window to appear on the display, and identifies the data that will be displayed in that window. The GUI layer of the operating system contains the code that actually draws the window and puts the text in it.

A sad aspect of computer programming is that it is possible to write code that causes a computer to crash. It is not only possible, but it is difficult to avoid. The frequency of crashes is a function of the skill and discipline of the programmers, the computer language being used, and the compiler that translates the high level language into assembly code. The errors I am talking about here are far less obvious than misspellings, since the compiler can check for these. A typical error is the miss-calculation, due to an error in the code, of a memory location while the program is running. In many cases the location doesn't exist, so it is clear that this is an error, but what should the computer program do when this happens? It is almost impossible for the program to guess what the correct location is, i.e. what the programmer really meant to do, and so the only option is to stop, which is a crash.

It might seem that testing could reveal code that had errors and could thus crash. Computer code that is designed for critical tasks or will be sold in large numbers is certainly tested extensively, and many errors are detected and then corrected. However, complex, large computer programs usually have so many variables which can take on so many values that it is impossible to test each combination. The only feasible approach is to check a large number of random parameters and hope there are no special cases. There usually are.

The above discussion just shows why it was important to use general purpose computers to make the Internet run. When errors were discovered only the program needed to be modified and then loaded into the computers. The modification can be done rapidly, and the new program can be loaded into computers located all over the Internet using the Internet itself. Compare the software approach to a hardware system where new circuit boards and chips need to be made and physically inserted into computers located all over the world.

Chapter summery

Programmable computers do most of the tasks that make the Internet function. The basic computer consists of a central processing unit and addressable memory. The processing unit reads a computer program stored in memory to obtain instructions which are used sequentially to process data. Each instruction is implemented by a set of logic gates which individually can only carry out simple operations such as AND, OR, and NOT, but when combined can execute an arbitrarily complex task. The program and the data that is being processed is stored in memory as a pattern of charged capacitors, and can be retrieved in less than a microsecond. Data can also be stored a thousand times more cheaply as magnetic patterns on rotating metallic disks, although, retrieving the data is about a thousand times slower.

The central processor and addressable memory are made up of a complex network of metal electrodes deposited on semiconductor crystals. The band gaps in

semiconductors allow a small current to switch a large current, which makes it possible to construct a network of logic gates. Semiconductor devices are small, require low power, and can be mass produced cheaply by photolithography.

Computer programs are written using other computer programs to translate human friendly computer languages into the actual processor instructions. Typical programs may contain millions of lines of code and it is almost impossible to prevent errors. The effective separation of large, complex programs into a smaller number of subroutines is one major goal of computer research. The invention of computer languages that result in fewer mistakes is another major goal.