

# How Microprocessors Work

by [Marshall Brain](#)

The computer you are using to read this page uses a **microprocessor** to do its work. The microprocessor is the heart of any normal computer, whether it is a [desktop machine](#), a [server](#) or a [laptop](#). The microprocessor you are using might be a Pentium, a K6, a PowerPC, a Sparc or any of the many other brands and types of microprocessors, but they all do approximately the same thing in approximately the same way.

If you have ever wondered what the microprocessor in your computer is doing, or if you have ever wondered about the differences between types of microprocessors, then read on. In this article, you will learn how fairly simple digital logic techniques allow a computer to do its job, whether its playing a game or spell checking a document!

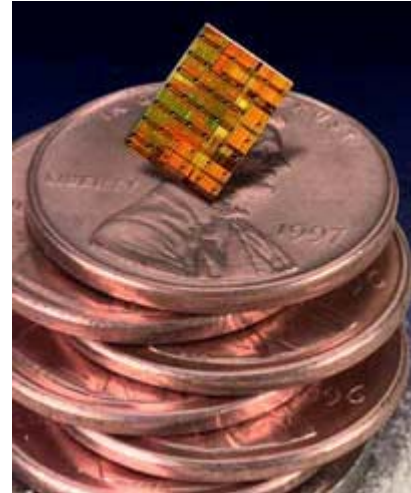


Photo courtesy [International Business Machines Corporation](#). Unauthorized use not permitted.

## CMOS 7S "Copper chip" on a stack of pennies

## Microprocessor History

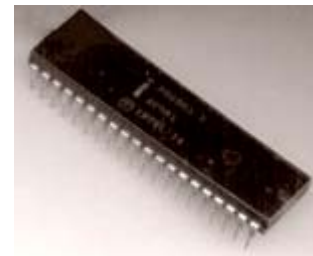
A microprocessor -- also known as a **CPU** or central processing unit -- is a complete computation engine that is first microprocessor was the Intel 4004 was not very powerful -- all it and it could only do that 4 [bits](#) at a everything was on one chip. Prior to computers either from collections of components ([transistors](#) wired one at of the first portable electronic calculators.



Intel 4004 chip

fabricated on a single chip. The 4004, introduced in 1971. The could do was add and subtract, time. But it was amazing that the 4004, engineers built chips or from discrete a time). The 4004 powered one

The first microprocessor to make it into a home computer was the Intel 8080, a complete 8-bit computer on one chip, introduced in 1974. The first microprocessor to make a real splash in the market was the Intel 8088, introduced in 1979 and incorporated into the IBM PC (which first appeared around 1982). If you are familiar with the PC market and its history, you know that the PC market moved from the 8088 to the 80286 to the 80386 to the 80486 to the Pentium to the Pentium II to the Pentium III to the Pentium 4. All of these microprocessors are made by Intel and all of them are improvements on the basic design of the 8088. The Pentium 4 can execute any piece of code that ran on the original 8088, but it does it about 5,000 times faster!



Intel 8080

The following table helps you to understand the differences between the different processors that Intel has introduced over the years.

Name	Date	Transistors	Microns	Clock speed	Data width	MIPS
8080	1974	6,000	6	2 MHz	8 bits	0.64
8088	1979	29,000	3	5 MHz	16 bits 8-bit bus	0.33
80286	1982	134,000	1.5	6 MHz	16 bits	1
80386	1985	275,000	1.5	16 MHz	32 bits	5

80486	1989	1,200,000	1	25 MHz	32 bits	20
Pentium	1993	3,100,000	0.8	60 MHz	32 bits 64-bit bus	100
Pentium II	1997	7,500,000	0.35	233 MHz	32 bits 64-bit bus	~300
Pentium III	1999	9,500,000	0.25	450 MHz	32 bits 64-bit bus	~510
Pentium 4	2000	42,000,000	0.18	1.5 GHz	32 bits 64-bit bus	~1,700

Compiled from [The Intel Microprocessor Quick Reference Guide](#) and [TSCP Benchmark Scores](#)

Information about this table:

- The **date** is the year that the processor was first introduced. Many processors are re-introduced at higher clock speeds for many years after the original release date.
- **Transistors** is the number of transistors on the chip. You can see that the number of transistors on a single chip has risen steadily over the years.
- **Microns** is the width, in microns, of the smallest wire on the chip. For comparison, a human hair is 100 microns thick. As the feature size on the chip goes down, the number of transistors rises.
- **Clock speed** is the maximum rate that the chip can be clocked at. Clock speed will make more sense in the next section.
- **Data Width** is the width of the ALU. An 8-bit ALU can add/subtract/multiply/etc. two 8-bit numbers, while a 32-bit ALU can manipulate 32-bit numbers. An 8-bit ALU would have to execute four instructions to add two 32-bit numbers, while a 32-bit ALU can do it in one instruction. In many cases, the external data bus is the same width as the ALU, but not always. The 8088 had a 16-bit ALU and an 8-bit bus, while the modern Pentiums fetch data 64 bits at a time for their 32-bit ALUs.
- **MIPS** stands for "millions of instructions per second" and is a rough measure of the performance of a CPU. Modern CPUs can do so many different things that MIPS ratings lose a lot of their meaning, but you can get a general sense of the relative power of the CPUs from this column.

### What's a Chip?

A **chip** is also called an **integrated circuit**. Generally it is a small, thin piece of [silicon](#) onto which the [transistors](#) making up the microprocessor have been etched. A chip might be as large as an inch on a side and can contain tens of millions of transistors. Simpler processors might consist of a few thousand transistors etched onto a chip just a few millimeters square.

From this table you can see that, in general, there is a relationship between clock speed and MIPS. The maximum clock speed is a function of the manufacturing process and delays within the chip. There is also a relationship between the number of transistors and MIPS. For example, the 8088 clocked at 5 MHz but only executed at 0.33 MIPS (about one instruction per 15 clock cycles). Modern processors can often execute at a rate of two instructions per clock cycle. That improvement is directly related to the number of transistors on the chip and will make more sense in the next section.

## Inside a Microprocessor

To understand how a microprocessor works, it is helpful to look inside and learn about the logic used to create one. In the process you can also learn about **assembly language** -- the native language of a microprocessor -- and many of the things that engineers can do to boost the speed of a processor.

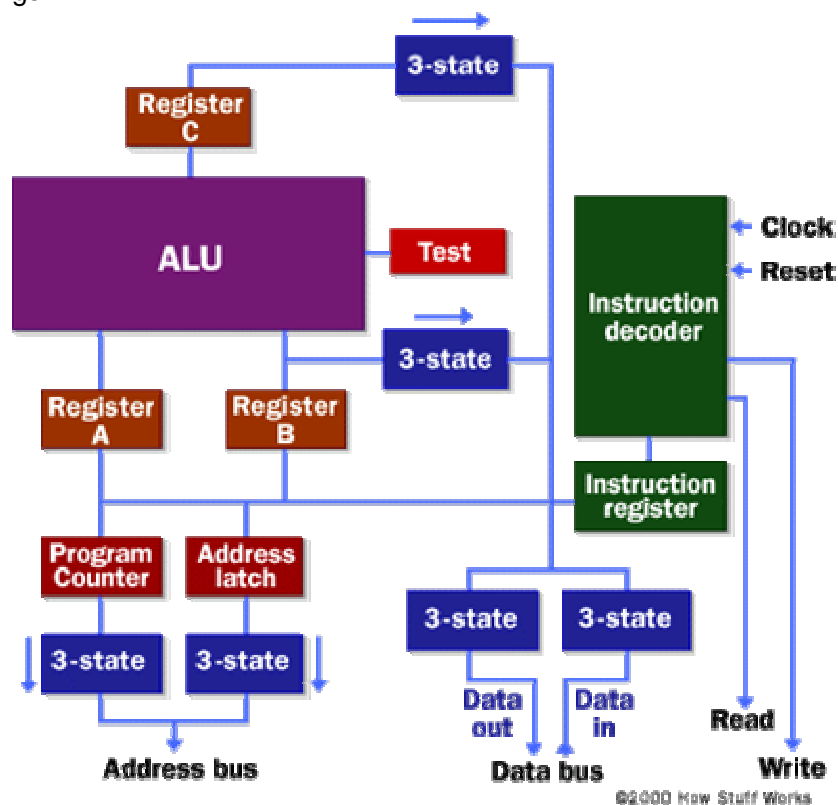


Photo courtesy [Intel Corporation](http://www.intel.com)  
**Intel Pentium 4 processor**

A microprocessor executes a collection of machine instructions that tell the processor what to do. Based on the instructions, a microprocessor does three basic things:

- Using its ALU (Arithmetic/Logic Unit), a microprocessor can perform mathematical operations like addition, subtraction, multiplication and division. Modern microprocessors contain complete floating point processors that can perform extremely sophisticated operations on large floating point numbers.
- A microprocessor can move data from one [memory](#) location to another.
- A microprocessor can make decisions and jump to a new set of instructions based on those decisions.

There may be very sophisticated things that a microprocessor does, but those are its three basic activities. The following diagram shows an extremely simple microprocessor capable of doing those three things:



This is about as simple as a microprocessor gets. This microprocessor has:

- An **address bus** (that may be 8, 16 or 32 bits wide) that sends an address to memory
- A **data bus** (that may be 8, 16 or 32 bits wide) that can send data to memory or receive data from memory
- An **RD** (read) and **WR** (write) line to tell the memory whether it wants to set or get the

- addressed location
- A **clock line** that lets a clock pulse sequence the processor
- A **reset line** that resets the program counter to zero (or whatever) and restarts execution

Let's assume that both the address and data buses are 8 bits wide in this example.

Here are the components of this simple microprocessor:

- Registers A, B and C are simply latches made out of flip-flops. (See the section on "edge-triggered latches" in [How Boolean Logic Works](#) for details.)
- The address latch is just like registers A, B and C.
- The program counter is a latch with the extra ability to increment by 1 when told to do so, and also to reset to zero when told to do so.
- The ALU could be as simple as an 8-bit adder (see the section on adders in [How Boolean Logic Works](#) for details), or it might be able to add, subtract, multiply and divide 8-bit values. Let's assume the latter here.
- The test register is a special latch that can hold values from comparisons performed in the ALU. An ALU can normally compare two numbers and determine if they are equal, if one is greater than the other, etc. The test register can also normally hold a carry bit from the last stage of the adder. It stores these values in flip-flops and then the instruction decoder can use the values to make decisions.
- There are six boxes marked "3-State" in the diagram. These are **tri-state buffers**. A tri-state buffer can pass a 1, a 0 or it can essentially disconnect its output (imagine a switch that totally disconnects the output line from the wire that the output is heading toward). A tri-state buffer allows multiple outputs to connect to a wire, but only one of them to actually drive a 1 or a 0 onto the line.
- The instruction register and instruction decoder are responsible for controlling all of the other components.

Although they are not shown in this diagram, there would be control lines from the instruction decoder that would:

- Tell the A register to latch the value currently on the data bus
- Tell the B register to latch the value currently on the data bus
- Tell the C register to latch the value currently on the data bus
- Tell the program counter register to latch the value currently on the data bus
- Tell the address register to latch the value currently on the data bus
- Tell the instruction register to latch the value currently on the data bus
- Tell the program counter to increment
- Tell the program counter to reset to zero
- Activate any of the six tri-state buffers (six separate lines)
- Tell the ALU what operation to perform
- Tell the test register to latch the ALU's test bits
- Activate the RD line
- Activate the WR line

### Helpful Articles

If you are new to digital logic, you may find the following articles helpful in understanding this section:

- [How Bytes and Bits Work](#)
- [How Boolean Logic Works](#)
- [How Electronic Gates Work](#)

Coming into the instruction decoder are the bits from the test register and the clock line, as well as the bits from the instruction register.

## 64-bit Processors

Sixty-four-bit processors have been with us since 1992, and in the 21st century they have started to become mainstream. Both Intel and AMD have introduced 64-bit chips, and the Mac G5 sports a 64-bit processor. Sixty-four-bit processors have 64-bit ALUs, 64-bit registers, 64-bit buses and

so on.



Photo courtesy [AMD](#)

One reason why the world needs 64-bit processors is because of their **enlarged address spaces**. Thirty-two-bit chips are often constrained to a maximum of 2 [GB](#) or 4 GB of [RAM access](#). That sounds like a lot, given that most home computers currently use only 256 MB to 512 MB of RAM. However, a 4-GB limit can be a severe problem for [server](#) machines and machines running large databases. And even [home machines](#) will start bumping up against the 2 GB or 4 GB limit pretty soon if current trends continue. A 64-bit chip has none of these constraints because a 64-bit RAM address space is essentially infinite for the foreseeable future --  $2^{64}$  bytes of RAM is something on the order of a quadrillion gigabytes of RAM.

With a 64-bit address bus and wide, high-speed data buses on the [motherboard](#), 64-bit machines also offer faster I/O (input/output) speeds to things like [hard disk drives](#) and [video cards](#). These features can greatly increase system performance.

Servers can definitely benefit from 64 bits, but what about normal users? Beyond the RAM solution, it is not clear that a 64-bit chip offers "normal users" any real, tangible benefits at the moment. They can process data (very complex data features lots of real numbers) faster. People doing [video editing](#) and people doing photographic editing on very large images benefit from this kind of computing power. High-end games will also benefit, once they are re-coded to take advantage of 64-bit features. But the average user who is reading [e-mail](#), browsing the Web and editing Word documents is not really using the processor in that way. In addition, [operating systems](#) like Windows XP have not yet been upgraded to handle 64-bit CPUs. Because of the lack of tangible benefits, it will be 2010 or so before we see 64-bit machines on every desktop.

Check out [ExtremeTech - 64-bit CPUs: What You Need to Know](#) and [InternetWeek - Athlon 64 Needs A Killer App](#) to learn more.

## RAM and ROM

The previous section talked about the address and data buses, as well as the RD and WR lines. These buses and lines connect either to RAM or ROM -- generally both. In our sample microprocessor, we have an address bus 8 bits wide and a data bus 8 bits wide. That means that the microprocessor can address ( $2^8$ ) 256 bytes of memory, and it can read or write 8 bits of the memory at a time. Let's assume that this simple microprocessor has 128 bytes of ROM starting at address 0 and 128 bytes of RAM starting at address 128.



ROM chip

[ROM](#) stands for read-only memory. A ROM chip is programmed with a permanent collection of pre-set bytes. The address bus tells the ROM chip which byte to get and place on the data bus. When the RD line changes state, the ROM chip presents the selected byte onto the data bus.

[RAM](#) stands for random-access memory. RAM contains bytes of information, and the microprocessor can read or write to those bytes depending on whether the RD or WR line is signaled. One problem with today's RAM chips is that they forget everything once the [power](#) goes off. That is why the computer needs ROM.

By the way, nearly all computers contain some amount of ROM (it is possible to create a simple computer that contains no RAM -- many [microcontrollers](#) do this by placing a handful of RAM bytes on the processor chip itself -- but generally impossible to create one that contains no ROM). On a [PC](#), the ROM is called the [BIOS](#) (Basic Input/Output System). When the microprocessor starts, it begins executing instructions it finds in the BIOS. The BIOS instructions do things like test the hardware in the machine, and then it goes to the hard disk to fetch the **boot sector** (see [How Hard Disks Work](#) for details). This boot sector is another small program, and the BIOS stores it in RAM after reading it off the disk. The microprocessor then begins executing the boot sector's instructions from RAM. The boot sector program will tell the microprocessor to fetch something else from the hard disk into RAM, which the microprocessor then executes, and so on. This is how the microprocessor loads and executes the entire [operating system](#).



RAM chip

## Microprocessor Instructions

Even the incredibly simple microprocessor shown in the previous example will have a fairly large set of instructions that it can perform. The collection of instructions is implemented as bit patterns, each one of which has a different meaning when loaded into the instruction register. Humans are not particularly good at remembering bit patterns, so a set of short words are defined to represent the different bit patterns. This collection of words is called the **assembly language** of the processor. An **assembler** can translate the words into their bit patterns very easily, and then the output of the assembler is placed in memory for the microprocessor to execute.

Here's the set of assembly language instructions that the designer might create for the simple microprocessor in our example:

- **LOADA mem** - Load register A from memory address
- **LOADB mem** - Load register B from memory address

- **CONB con** - Load a constant value into register B
- **SAVEB mem** - Save register B to memory address
- **SAVEC mem** - Save register C to memory address
- **ADD** - Add A and B and store the result in C
- **SUB** - Subtract A and B and store the result in C
- **MUL** - Multiply A and B and store the result in C
- **DIV** - Divide A and B and store the result in C
- **COM** - Compare A and B and store the result in test
- **JUMP addr** - Jump to an address
- **JEQ addr** - Jump, if equal, to address
- **JNEQ addr** - Jump, if not equal, to address
- **JG addr** - Jump, if greater than, to address
- **JGE addr** - Jump, if greater than or equal, to address
- **JL addr** - Jump, if less than, to address
- **JLE addr** - Jump, if less than or equal, to address
- **STOP** - Stop execution

If you have read [How C Programming Works](#), then you know that this simple piece of C code will calculate the factorial of 5 (where the factorial of 5 = 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120):

```
a=1;
f=1;
while (a <= 5)
{
    f = f * a;
    a = a + 1;
}
```

At the end of the program's execution, the variable **f** contains the factorial of 5.

A **C compiler** translates this C code into assembly language. Assuming that RAM starts at address 128 in this processor, and ROM (which contains the assembly language program) starts at address 0, then for our simple microprocessor the assembly language might look like this:

```
// Assume a is at address 128
// Assume F is at address 129
0  CONB 1          // a=1;
1  SAVEB 128
2  CONB 1          // f=1;
3  SAVEB 129
4  LOADA 128      // if a > 5 the jump to 17
5  CONB 5
6  COM
7  JG 17
8  LOADA 129      // f=f*a;
9  LOADB 128
10 MUL
11 SAVEC 129
12 LOADA 128      // a=a+1;
13 CONB 1
14 ADD
15 SAVEC 128
16 JUMP 4         // loop back to if
17 STOP
```

So now the question is, "How do all of these instructions look in ROM?" Each of these assembly language instructions must be represented by a binary number. For the sake of simplicity, let's assume each assembly language instruction is given a unique number, like this:

- LOADA - 1
- LOADB - 2
- CONB - 3
- SAVEB - 4
- SAVEC mem - 5
- ADD - 6
- SUB - 7
- MUL - 8
- DIV - 9
- COM - 10
- JUMP addr - 11
- JEQ addr - 12
- JNEQ addr - 13
- JG addr - 14
- JGE addr - 15
- JL addr - 16
- JLE addr - 17
- STOP - 18

The numbers are known as **opcodes**. In ROM, our little program would look like this:

```
// Assume a is at address 128
// Assume F is at address 129
Addr opcode/value
0 3 // CONB 1
1 1
2 4 // SAVEB 128
3 128
4 3 // CONB 1
5 1
6 4 // SAVEB 129
7 129
8 1 // LOADA 128
9 128
10 3 // CONB 5
11 5
12 10 // COM
13 14 // JG 17
14 31
15 1 // LOADA 129
16 129
17 2 // LOADB 128
18 128
19 8 // MUL
20 5 // SAVEC 129
21 129
22 1 // LOADA 128
23 128
24 3 // CONB 1
25 1
26 6 // ADD
27 5 // SAVEC 128
28 128
29 11 // JUMP 4
30 8
31 18 // STOP
```

You can see that seven lines of C code became 17 lines of assembly language, and that became 31 bytes in ROM.



The instruction decoder needs to turn each of the opcodes into a set of signals that drive the different components inside the microprocessor. Let's take the ADD instruction as an example and look at what it needs to do:

1. During the first clock cycle, we need to actually load the instruction. Therefore the instruction decoder needs to:
  - activate the tri-state buffer for the program counter
  - activate the RD line
  - activate the data-in tri-state buffer
  - latch the instruction into the instruction register
2. During the second clock cycle, the ADD instruction is decoded. It needs to do very little:
  - set the operation of the ALU to addition
  - latch the output of the ALU into the C register
3. During the third clock cycle, the program counter is incremented (in theory this could be overlapped into the second clock cycle).

Every instruction can be broken down as a set of sequenced operations like these that manipulate the components of the microprocessor in the proper order. Some instructions, like this ADD instruction, might take two or three clock cycles. Others might take five or six clock cycles.

## Microprocessor Performance

The number of **transistors** available has a huge effect on the performance of a processor. As seen earlier, a typical instruction in a processor like an 8088 took 15 clock cycles to execute. Because of the design of the multiplier, it took approximately 80 cycles just to do one 16-bit multiplication on the 8088. With more transistors, much more powerful multipliers capable of single-cycle speeds become possible.

More transistors also allow for a technology called **pipelining**. In a pipelined architecture, instruction execution overlaps. So even though it might take five clock cycles to execute each instruction, there can be five instructions in various stages of execution simultaneously. That way it looks like one instruction completes every clock cycle.

Many modern processors have multiple instruction decoders, each with its own pipeline. This allows for multiple instruction streams, which means that more than one instruction can complete during each clock cycle. This technique can be quite complex to implement, so it takes lots of transistors.

The trend in processor design has primarily been toward full 32-bit ALUs with fast floating point processors built in and pipelined execution with multiple instruction streams. The newest thing in processor design is 64-bit ALUs, and people are expected to have these processors in their home PCs in the next decade. There has also been a tendency toward special instructions (like the MMX instructions) that make certain operations particularly efficient. There has also been the addition of hardware [virtual memory](#) support and L1 [caching](#) on the processor chip. All of these trends push up the transistor count, leading to the multi-million transistor powerhouses available today. These processors can execute about one billion instructions per second!