CHAPTER 4 — The Processor

From previous chapters you know that the major hardware components of a computer system are:

- Processor
- Main memory
- Secondary memory devices
- Input/output devices

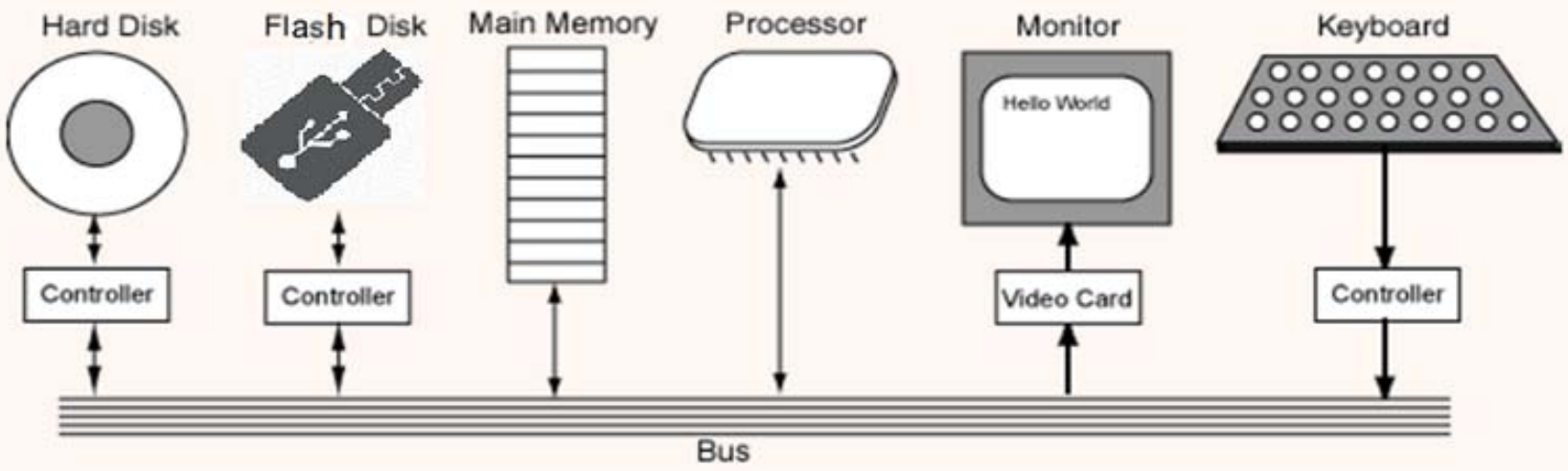
This chapter examines the "brain" of the computer system — the processor. Then it shows how the programs you write are translated into instructions for the processor.

Chapter Topics:

- Machine operations and machine language
- Example of machine language
- Different types of processor chips
- High level programming languages
- Language translators (compilers)
- Language interpreters

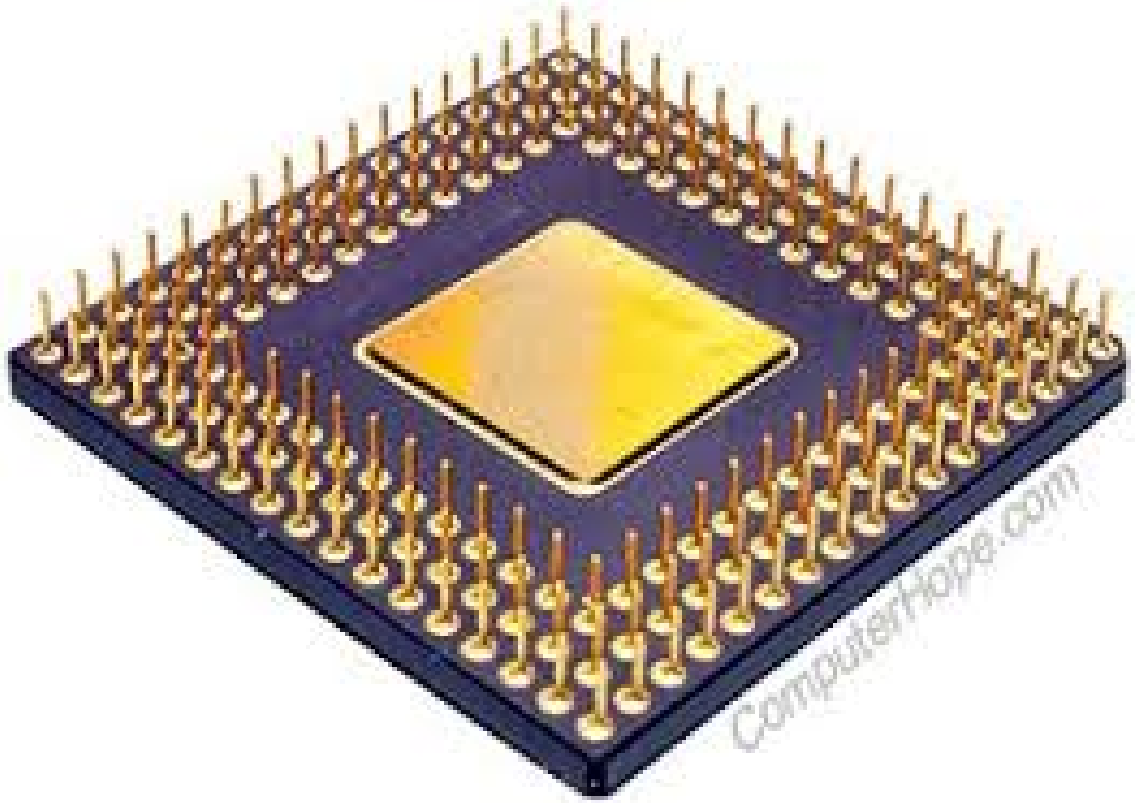
Characteristics of Computer Memory

Main memory is as vital as the processor chip to a computer system. Fast computer systems have both a fast processor and a large, fast memory. Here is a list of some characteristics of computer memory. Some characteristics are true for both kinds of memory; others are true for just one.

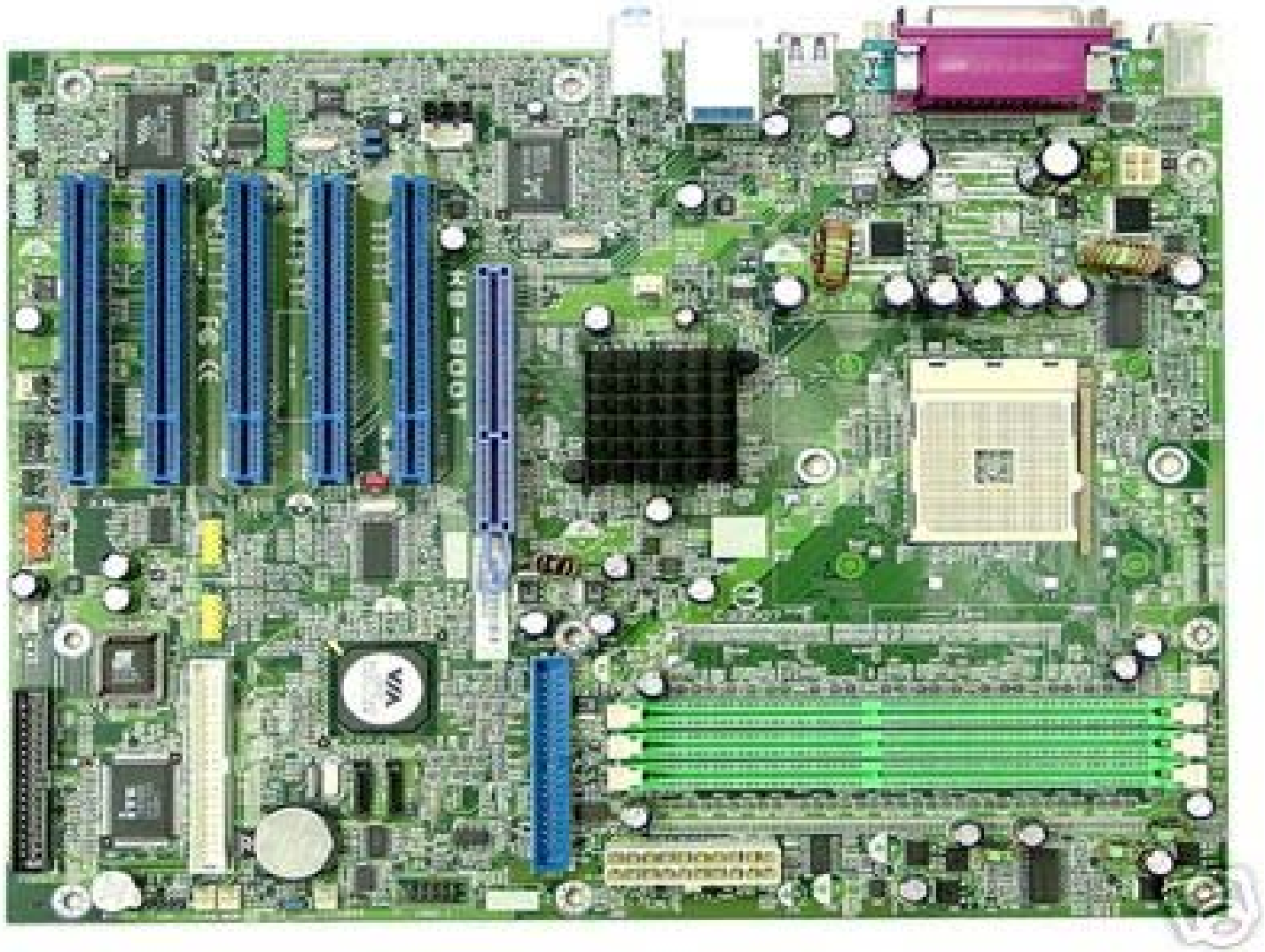


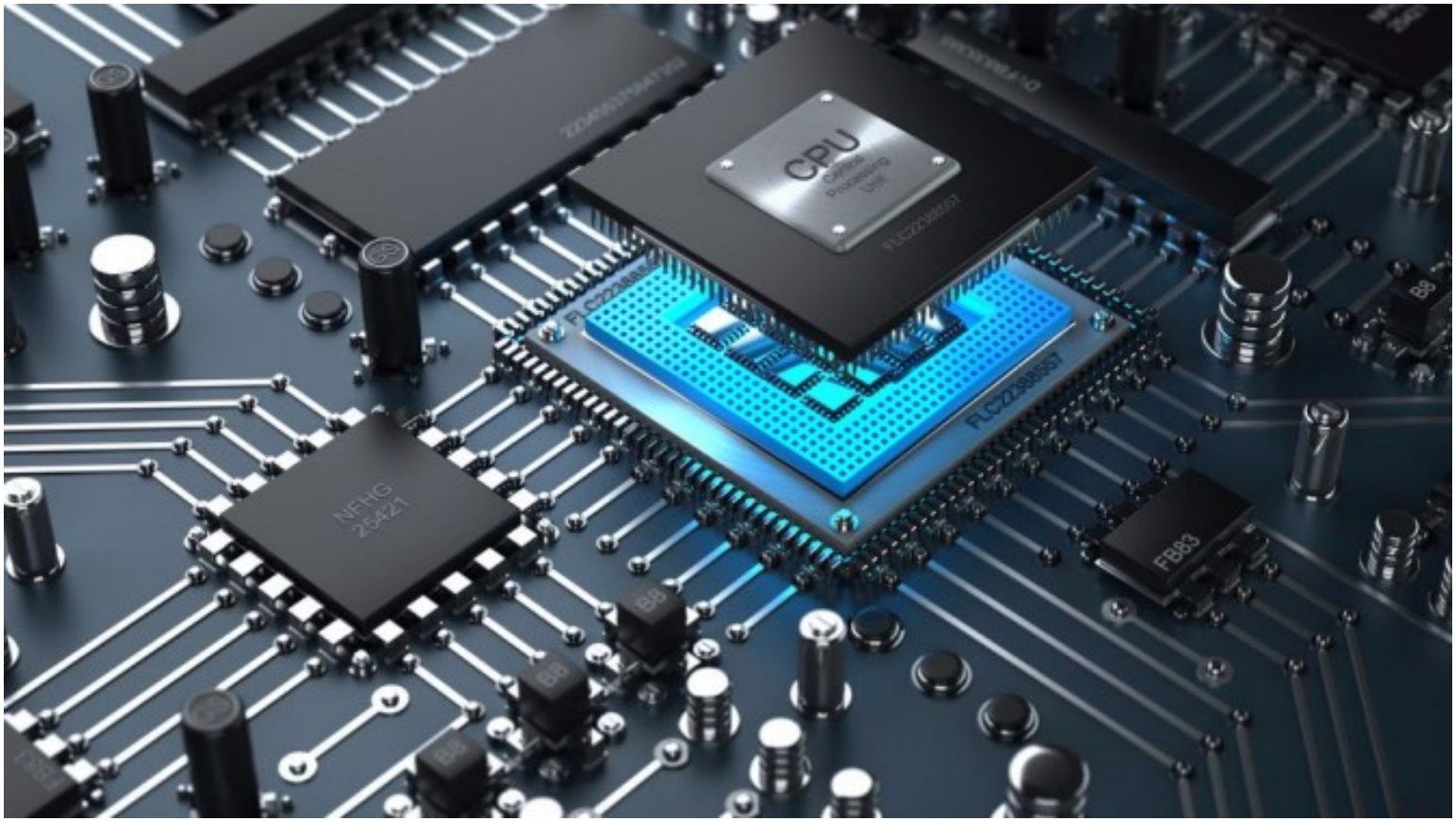
Main Components of a Computer System

A processor


ComputerHope.com

A processor



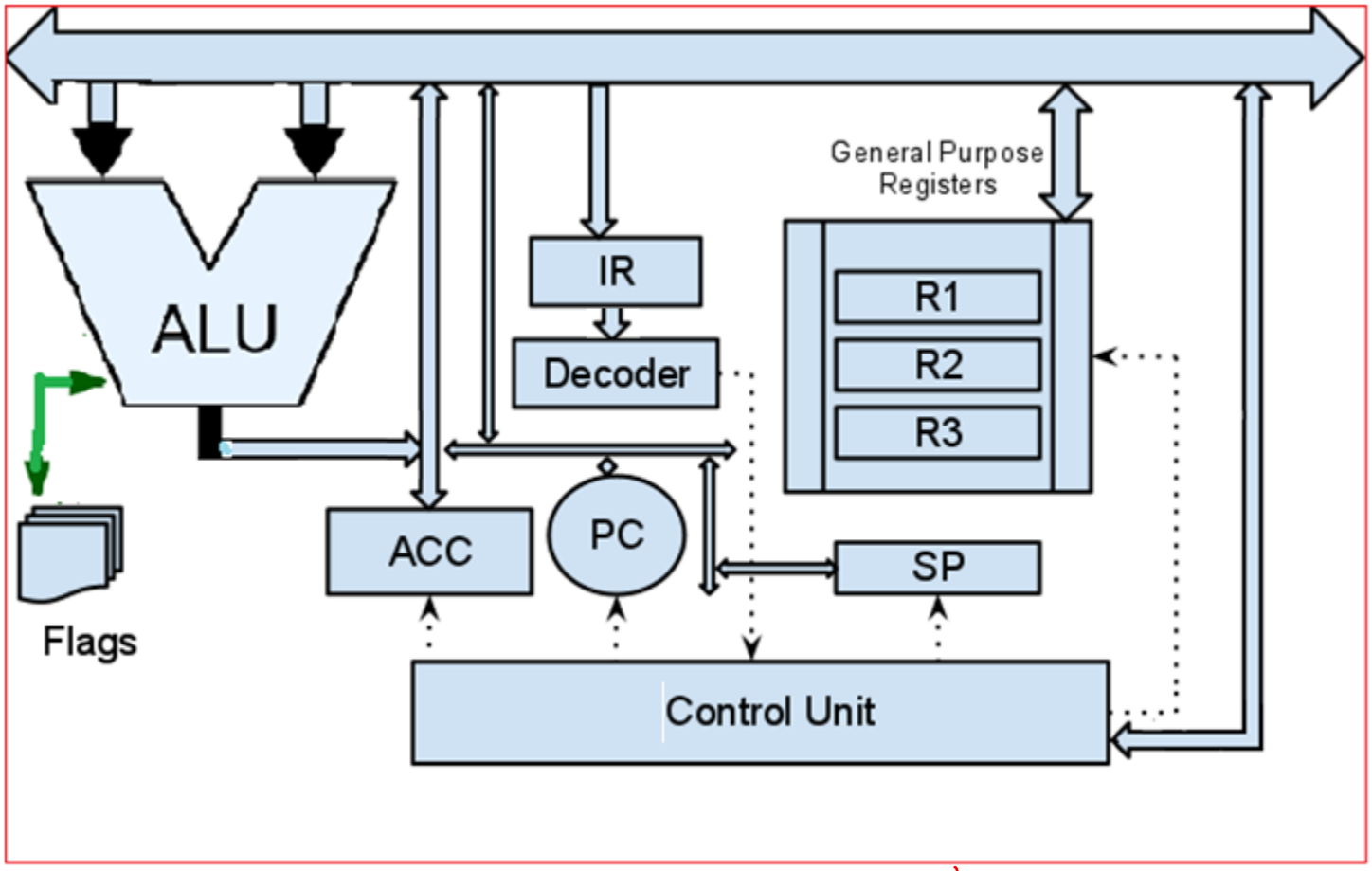

CPU
Central Processing Unit
FB83

Electronic Operations of a Processor

When a program is running on a computer the processor is constantly performing very many, very tiny electronic operations. For example, one such operation reads one byte of data from main memory into the processor. Another operation tests if one of the bits in the byte is a "1" bit. Most processors are able to perform several hundred types of small operations like these.

Those are the only things that a processor can do. It has a small set of tiny electronic operations that it can to perform, and that is all. These little electronic operations are performed one at a time. But millions of them are performed per second. Millions of small operations can add up to a large and useful action.

Everything that a processor does is built out of these tiny operations! Luckily, you don't need to know these details to write programs in Java. The purpose of a "high-level language" like Java is to organize the tiny electronic operations into large, useful units represented by program statements.



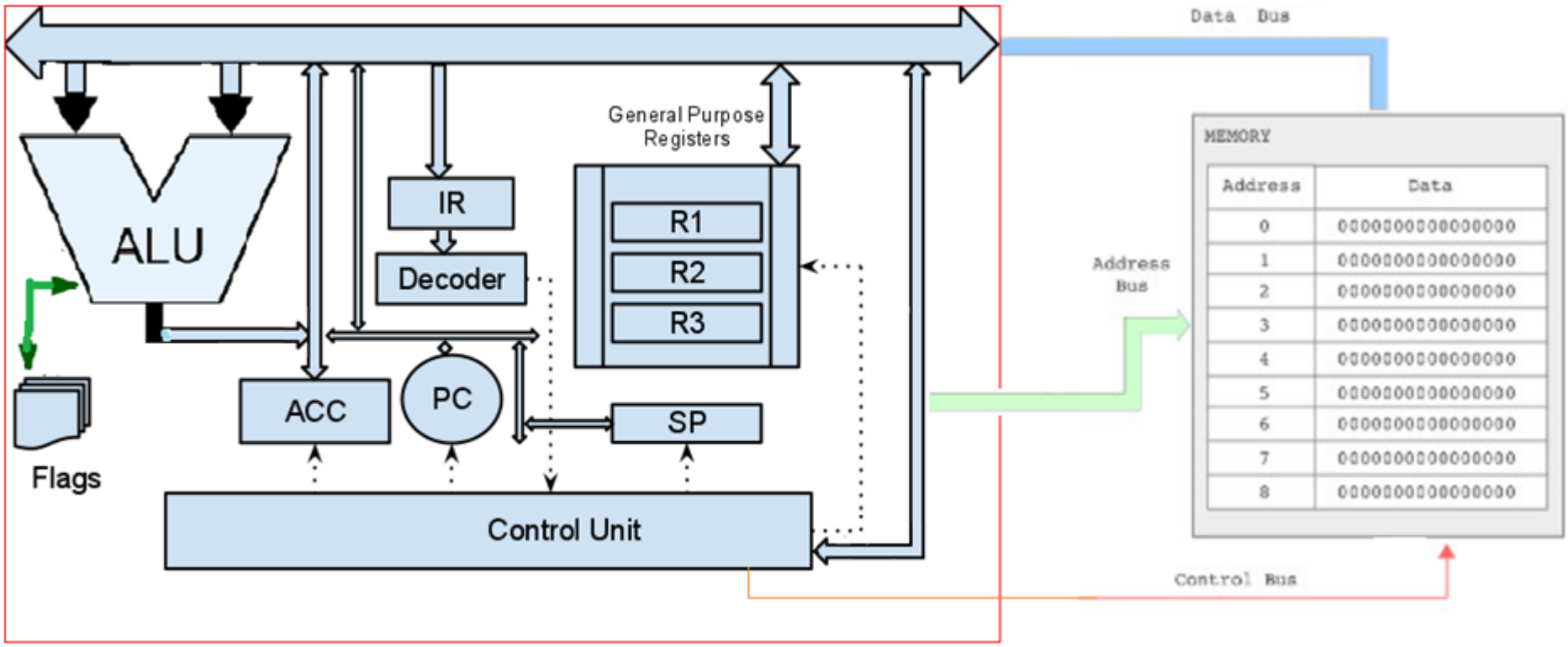
Accumulator

Instruction

Program Counter

Stack pointer

Arithmatic logic Unit

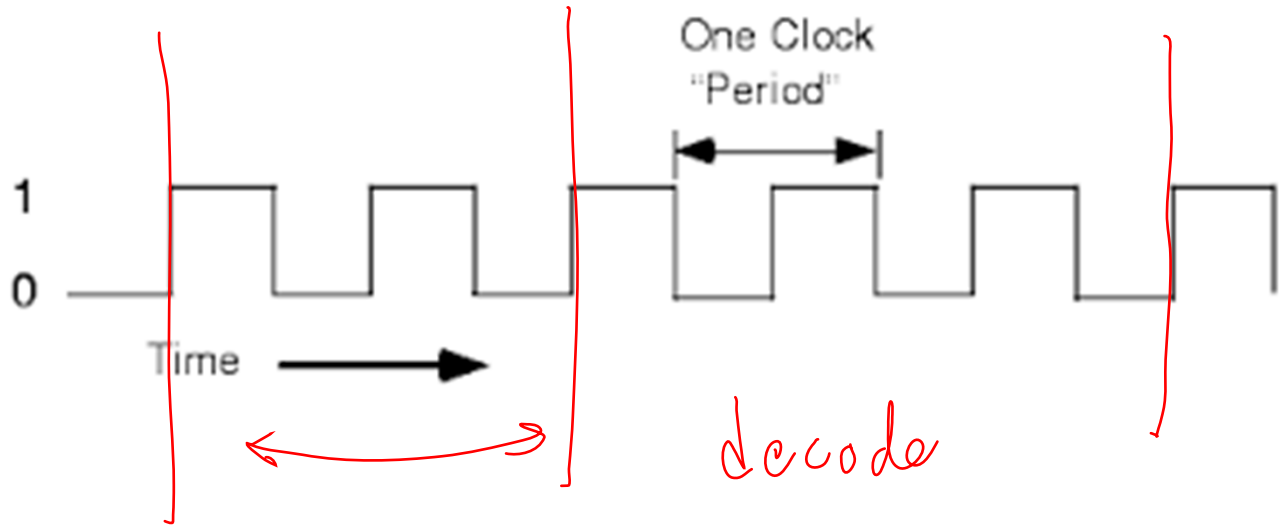

Data Bus
General Purpose Registers
IR
R1
R2
R3
ALU
Decoder
Address Bus
MEMORY
Address
Data
0 0000000000000000
1 0000000000000000
2 0000000000000000
3 0000000000000000
4 0000000000000000
5 0000000000000000
6 0000000000000000
7 0000000000000000
8 0000000000000000
ACC
PC
SP
Flags
Control Unit
Control Bus

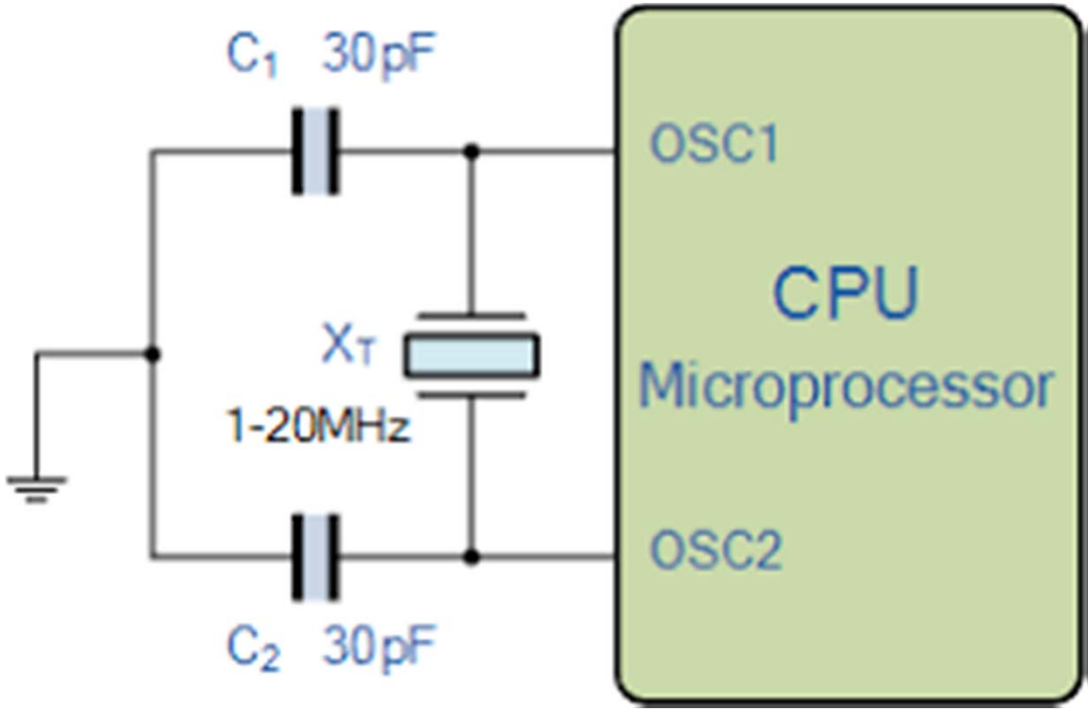
Clocks

Digital systems are built so that the "on" "off" (binary) value is only tested at certain times, giving the wire (or transistor, or...) a chance to change its state between these times. This is why computer systems have a "clock" — to keep all these times synchronized. So faster clocks mean wires can be tested more times per second, and the whole system runs faster.

Processor chips (and the computers that contain them) are often described in terms of their clock speed. Clock speed is measured in **Hertz**, where one Hertz is one clock tick per second. The symbol **MHz** means **mega Hertz**, a million clock ticks per second.

A 700 MHz Pentium processor checks binary values 700 million times in each second. In between these times values are allowed to change and settle down. The faster a processor chip is, the more times per second values can be tested, and the more decisions per second can be made.


One Clock
"Period"
1
0
Time
decode


C1 30pF
OSC1
CPU
Microprocessor
XT
1-20MHz
OSC2
C2 30pF

Electronic Operations of a Processor

- logical
- arithmetic
- comparator
- move (in memory)
- I/O (out of memory)
- jump

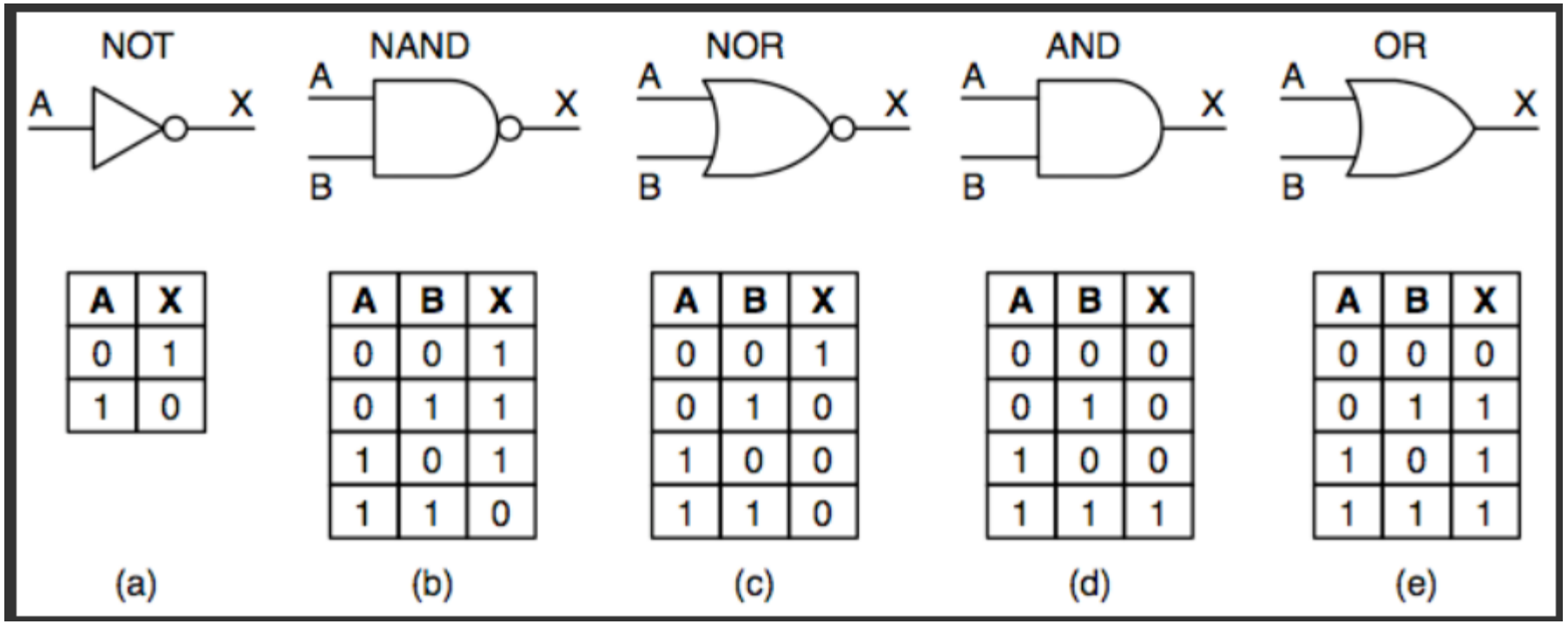
logical

- AND
- OR
- NOT
- XOR

arithmetic

* ADD
* SUB
* DIV
* MUL

Electronic Operations of a Processor



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

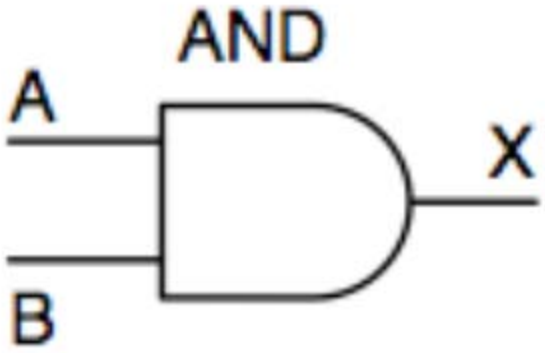
(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

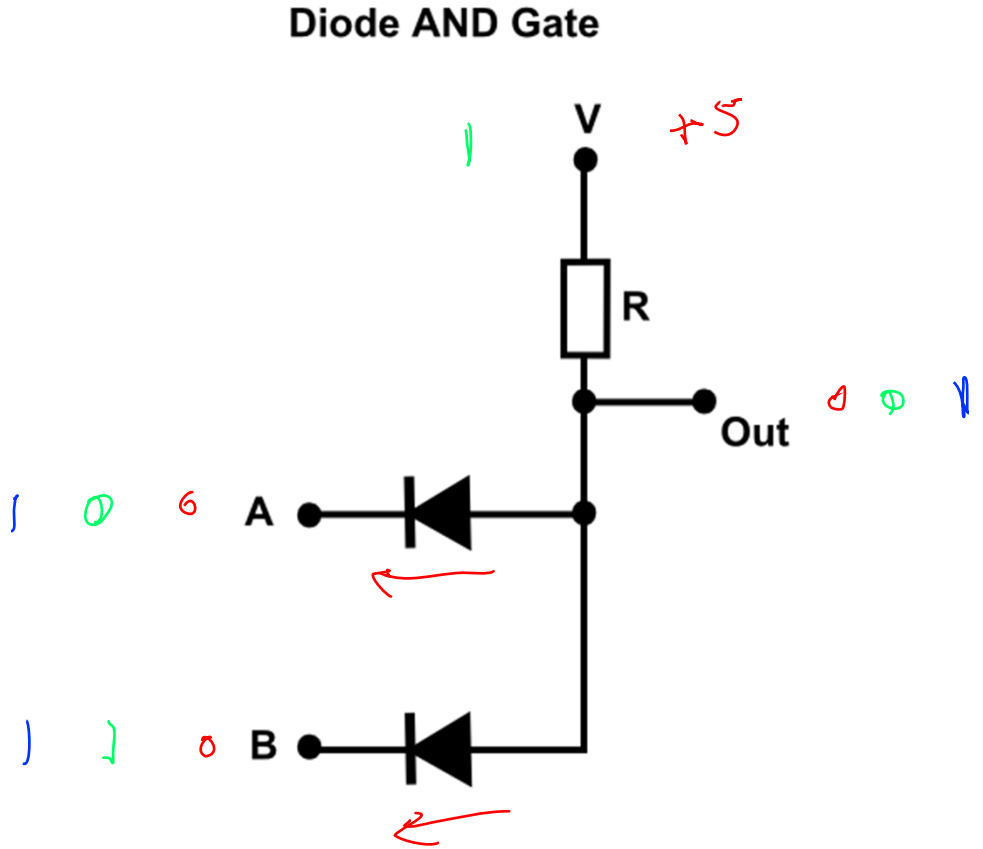
(d)

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)



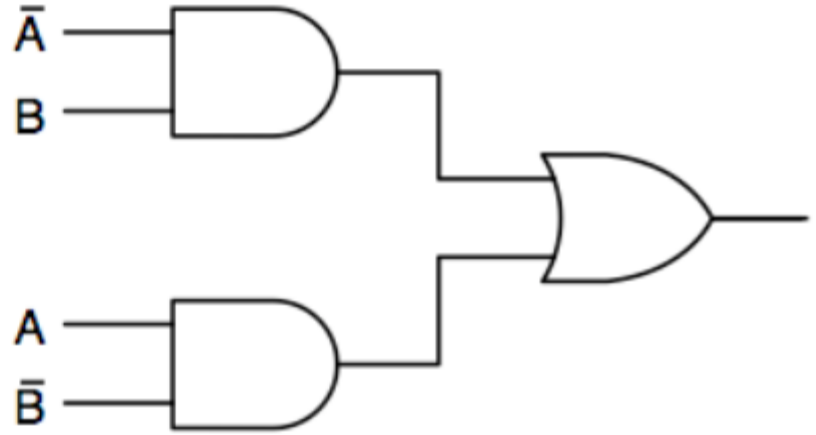
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1



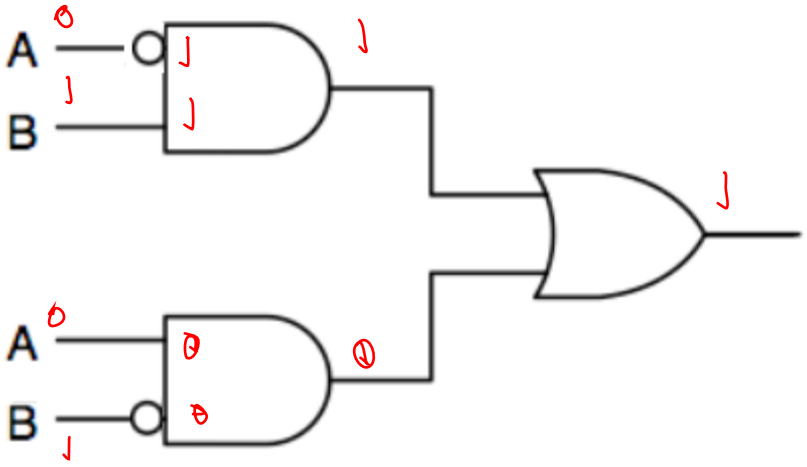
Electronic Operations of a Processor

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

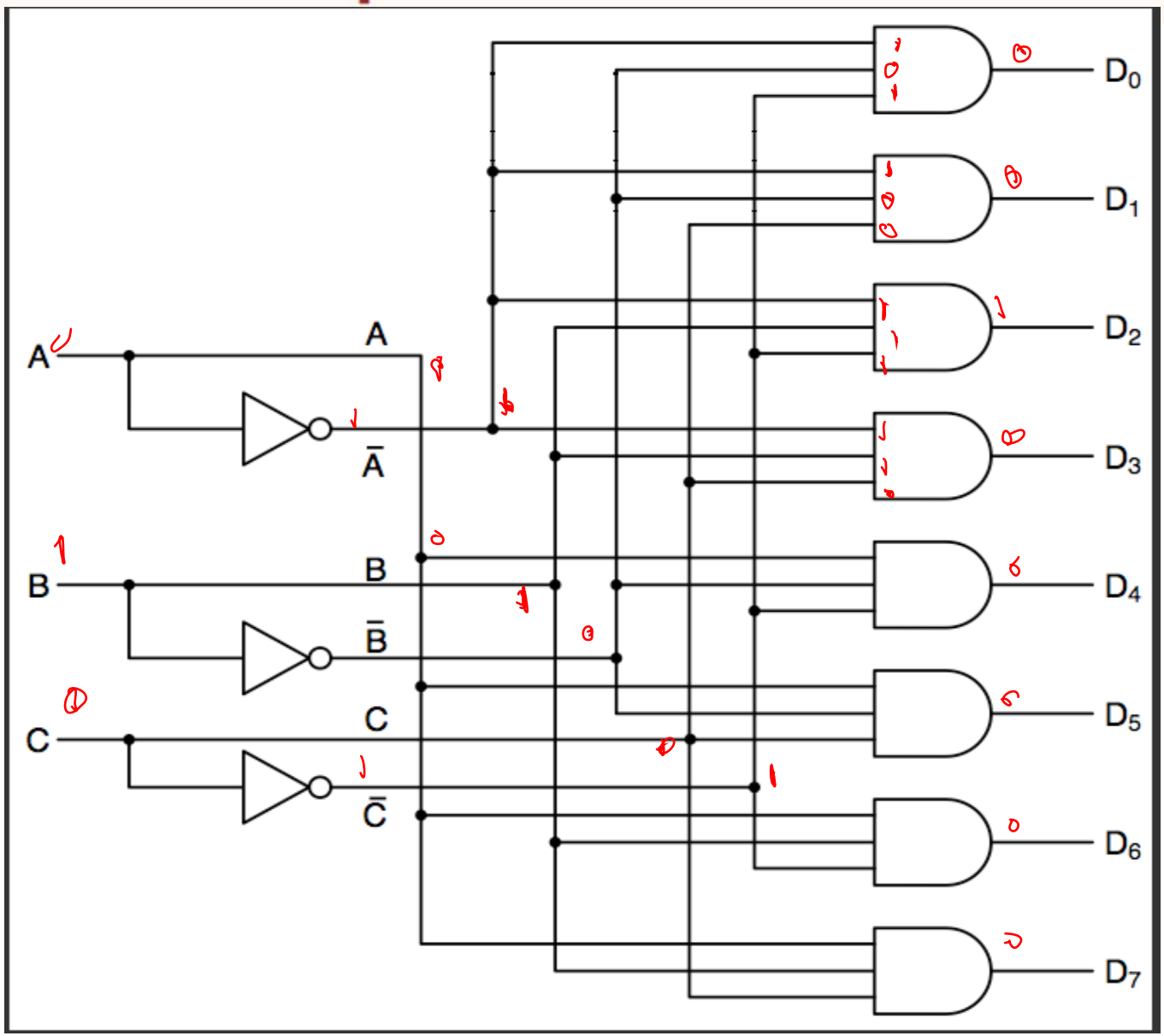
(a)



(b)



Electronic Operations of a Processor



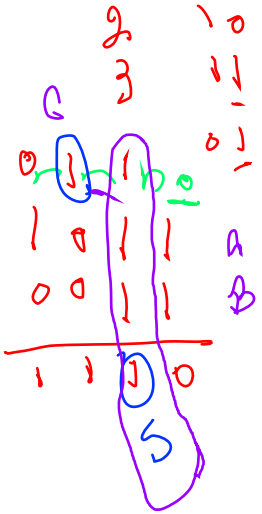
Decoder

Electronic Operations of a Processor

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

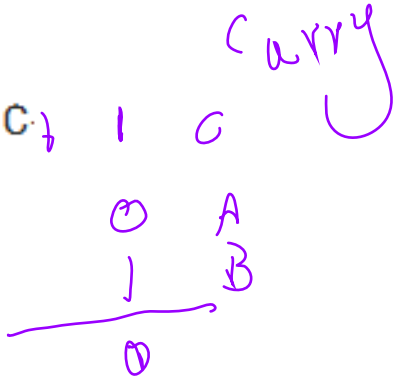
Logical Expression for SUM:

= A' B' C + A' B C' + A B' C' + A B C



Logical Expression for C-OUT:

= A' B C + A B' C + A B C' + A B C



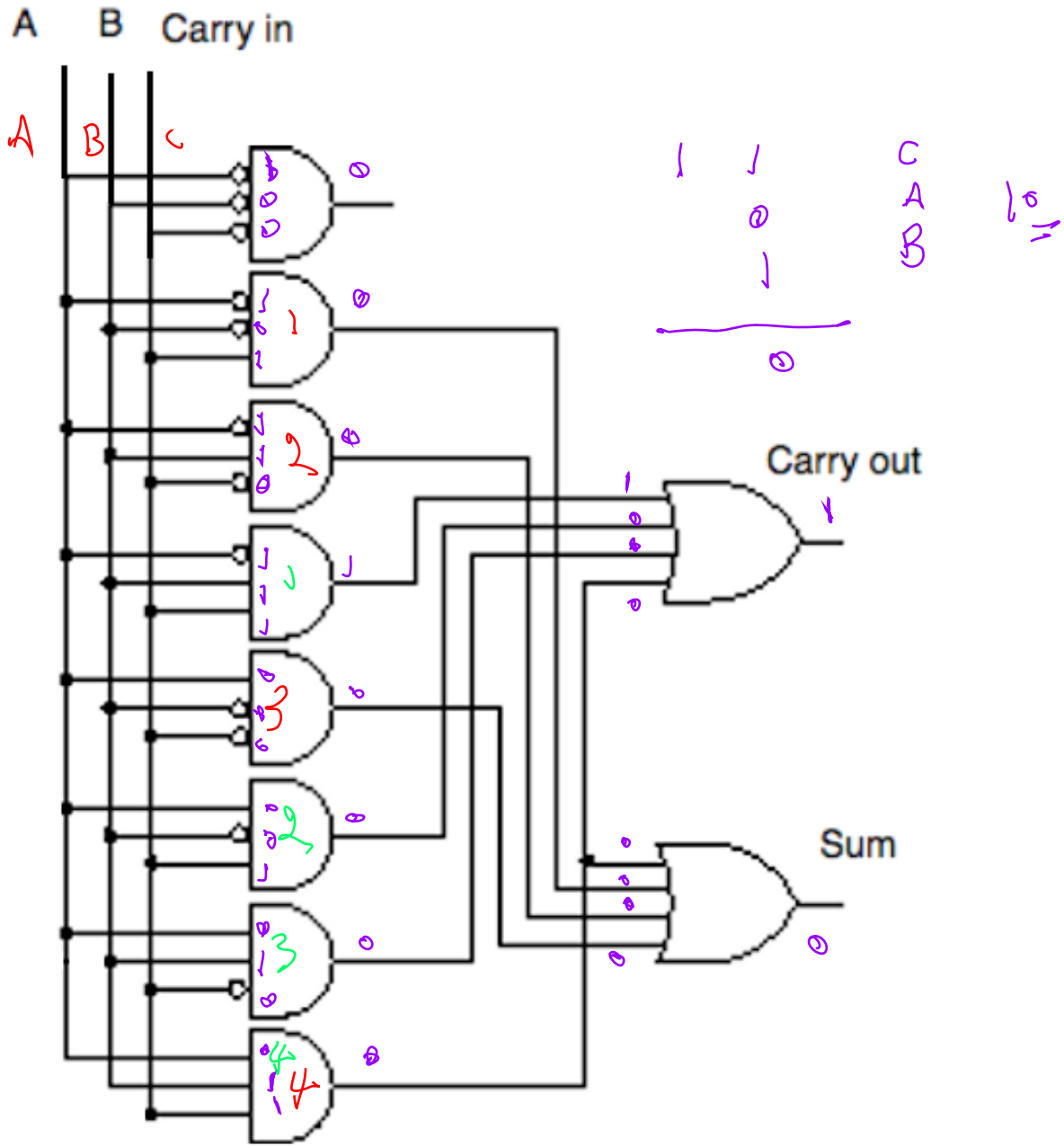
A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Logical Expression for SUM:

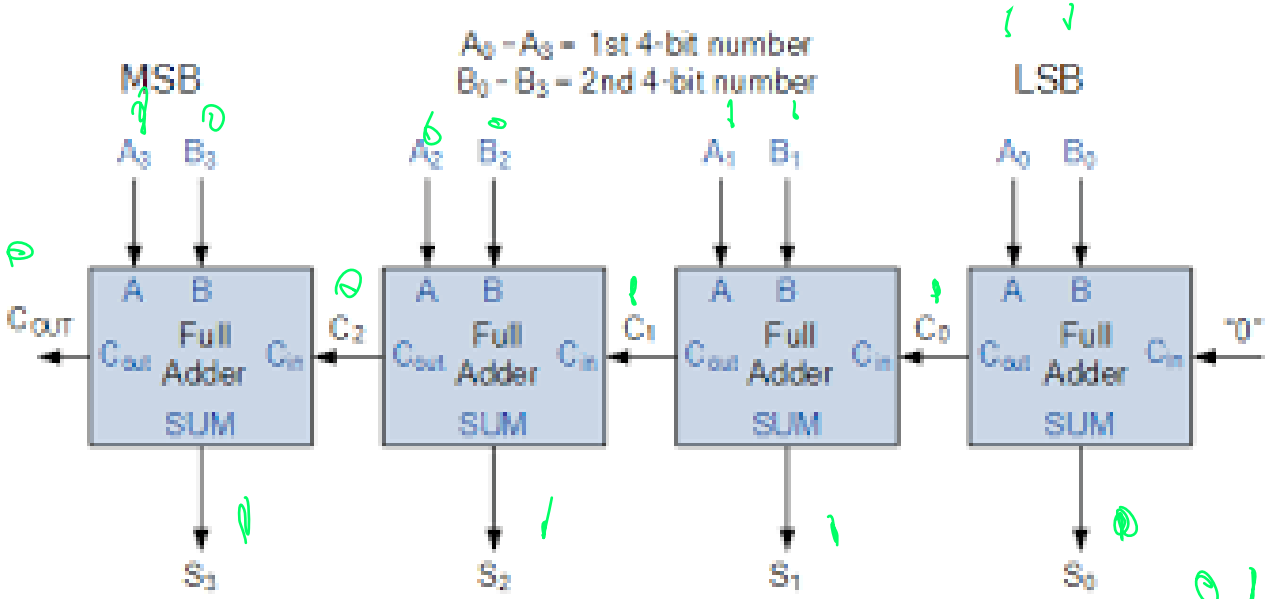
= A' B' C + A' B C' + A B' C' + A B C

Logical Expression for C-OUT:

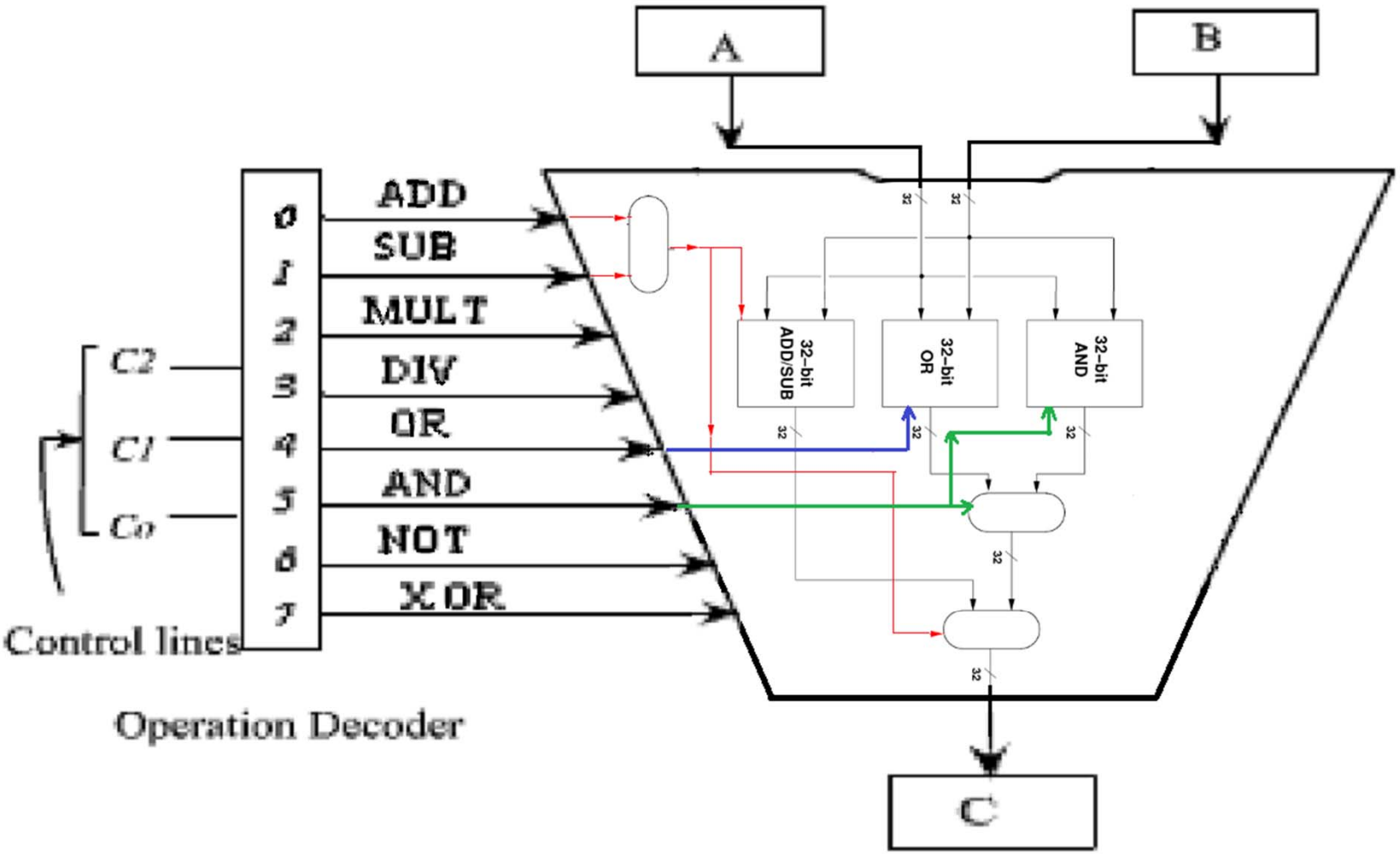
= A' B C + A B' C + A B C' + A B C

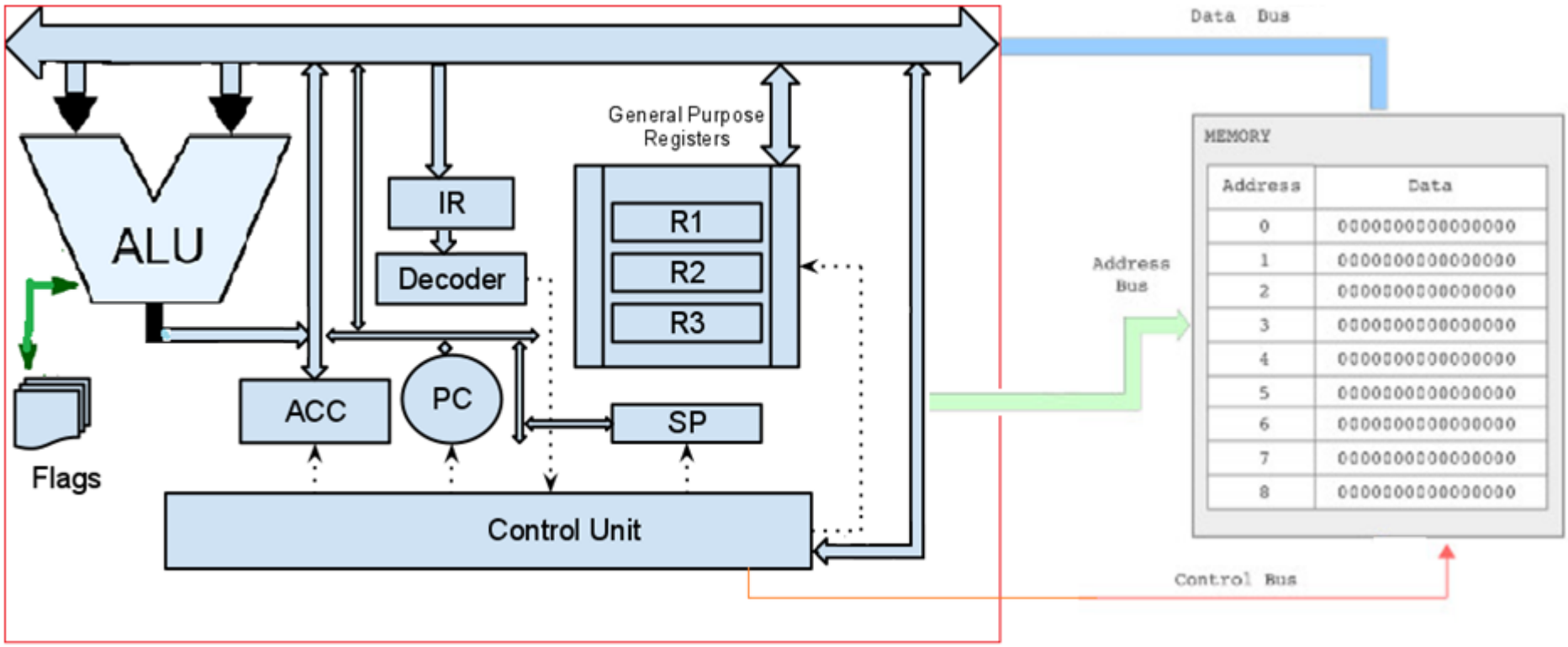


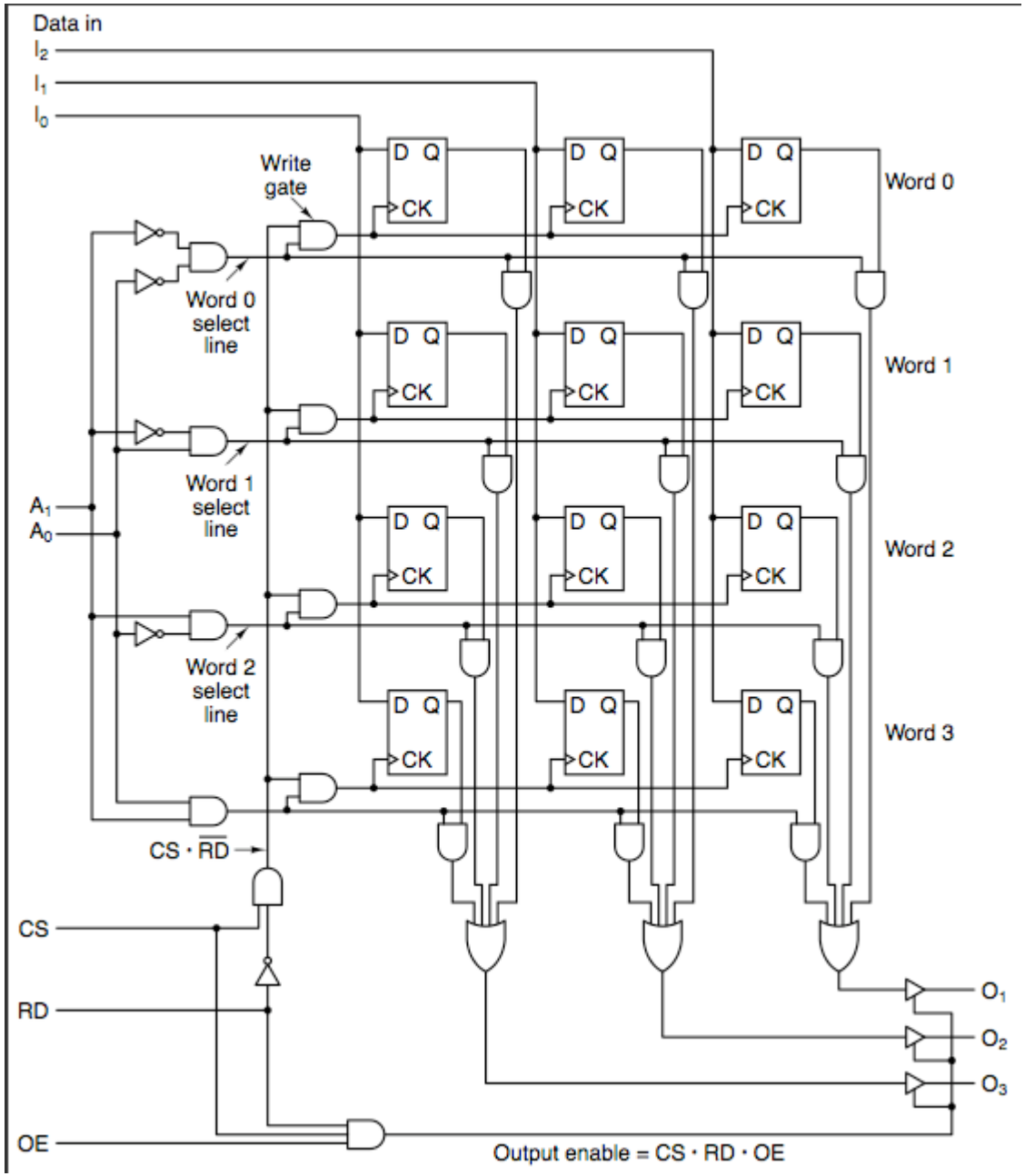
Electronic Operations of a Processor



Electronic Operations of a Processor




Data Bus
General Purpose Registers
IR
R1
R2
R3
ALU
Decoder
Address Bus
MEMORY
Address
Data
0 0000000000000000
1 0000000000000000
2 0000000000000000
3 0000000000000000
4 0000000000000000
5 0000000000000000
6 0000000000000000
7 0000000000000000
8 0000000000000000
ACC
PC
SP
Flags
Control Unit
Control Bus


Data in
I_2
I_1
I_0
Write gate
D Q
CK
Word 0
Word 1
Word 2
Word 3
Word 0 select line
Word 1 select line
Word 2 select line
A_1
A_0
CS · $\overline{RD}$
CS
RD
OE
O_1
O_2
O_3
Output enable = CS · RD · OE

Machine Instructions

Users and programmers of computers usually don't think about the millions of tiny electronic operations that go on each second. The situation is (very roughly) similar to when you are driving your car. You think about the "big operations" it can perform, such as "accelerate", "turn left", "brake", and so on. You don't think about the valves in your engine opening and closing 24,000 times per minute.

Each tiny electronic operation that a processor can perform is called a **machine operation**. A processor (a "machine") performs these one at a time, but millions of them in a second.

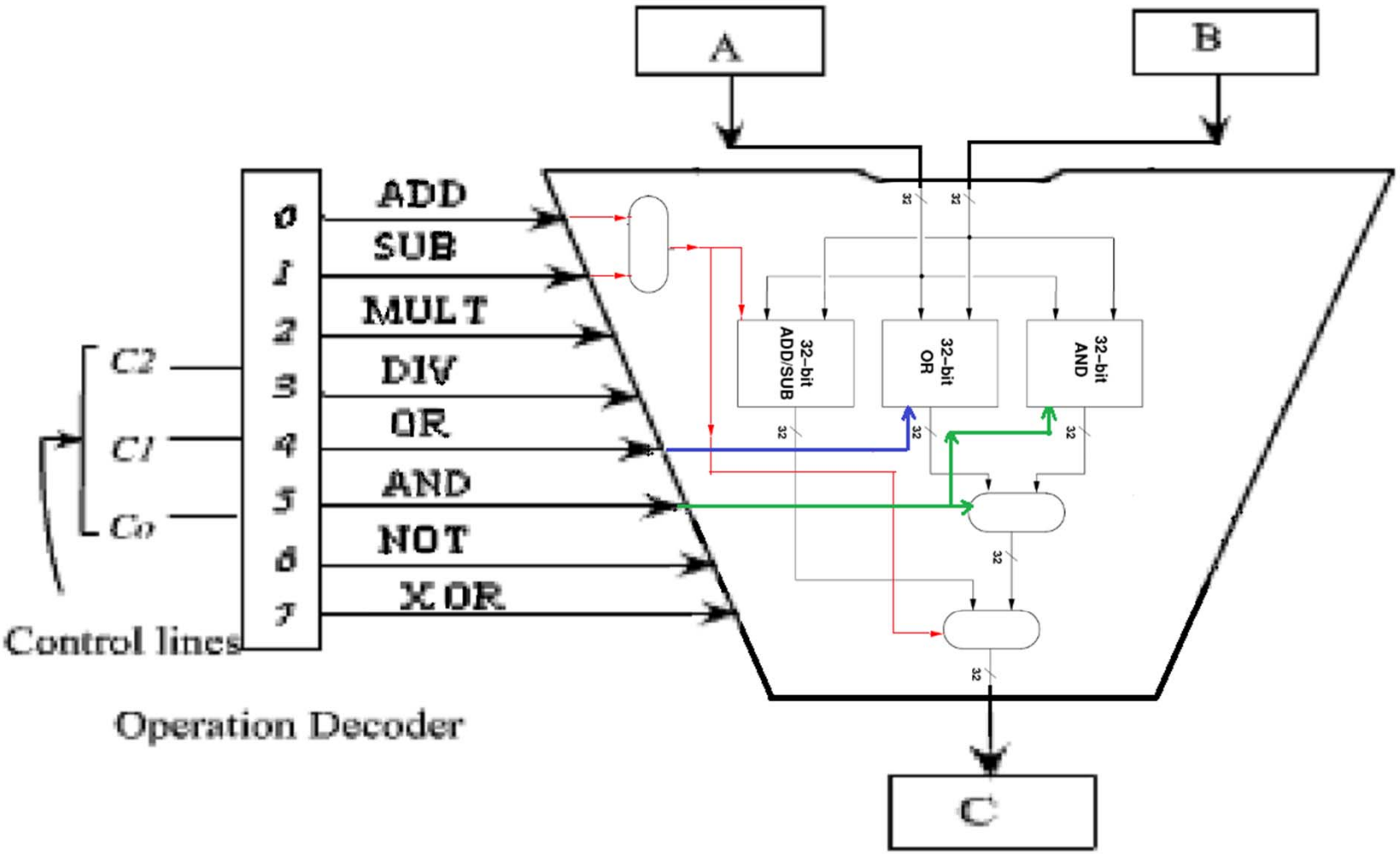
A **machine instruction** consists of several bytes in memory that tells the processor to perform one machine operation. The processor looks at machine instructions in main memory one after another, and performs one machine operation for each machine instruction. The collection of machine instructions in main memory is called a **machine language program** or (more commonly) an **executable program**.

Don't panic if the above seems incomprehensible. It takes some getting used to. (And to *really* understand it all takes several courses.)

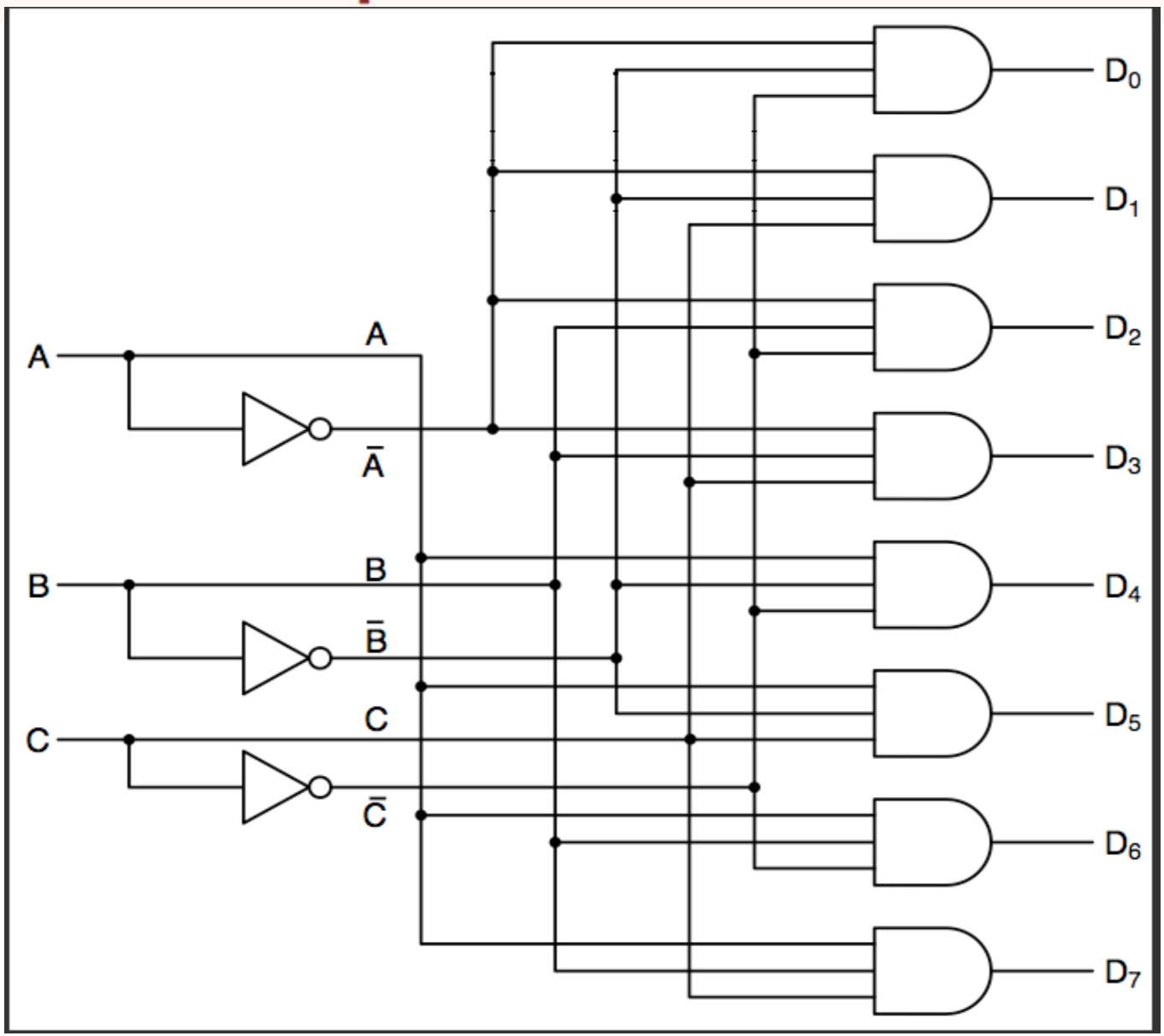
Machine Instructions

Sub r1,r2	000
Add r1,r2	001
Mov r1,2	010
Mov r1,3	011
Mov r2,0	100
And r1,r2	101
Or r1,r2	110
Mult r1,r2	111

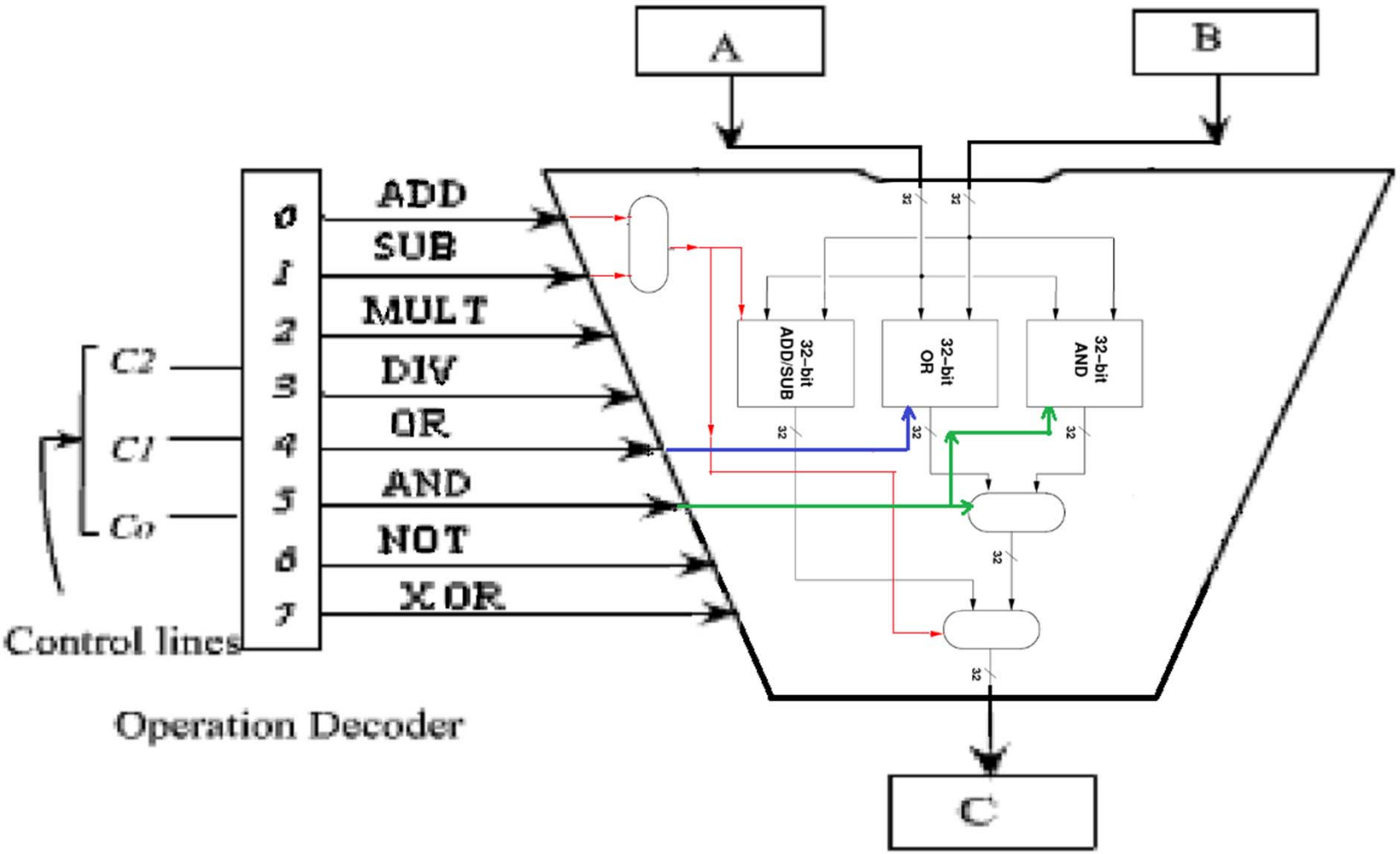
Electronic Operations of a Processor



Electronic Operations of a Processor



Electronic Operations of a Processor



Executing Instructions

The word "execute" is often used to mean "perform the machine operation that an instruction asks for." So you can say that "executing the instruction 0000 0000 stops the toothbrush," or "millions of instructions execute per second." "Execute" is also used for an entire program or part of a program: "to execute the program, turn the switch to on."

Most machine language programs are made up of instructions that are executed again and again. In the toothbrush program there was an instruction that caused the processor to start the program again from the beginning. In a real computer, millions of instructions execute per second, so something like this is necessary if the 512 megabytes of a typical computer are to hold a program that runs for more than a few seconds.

A group of machine instructions that can be repeatedly executed is called a **loop**.

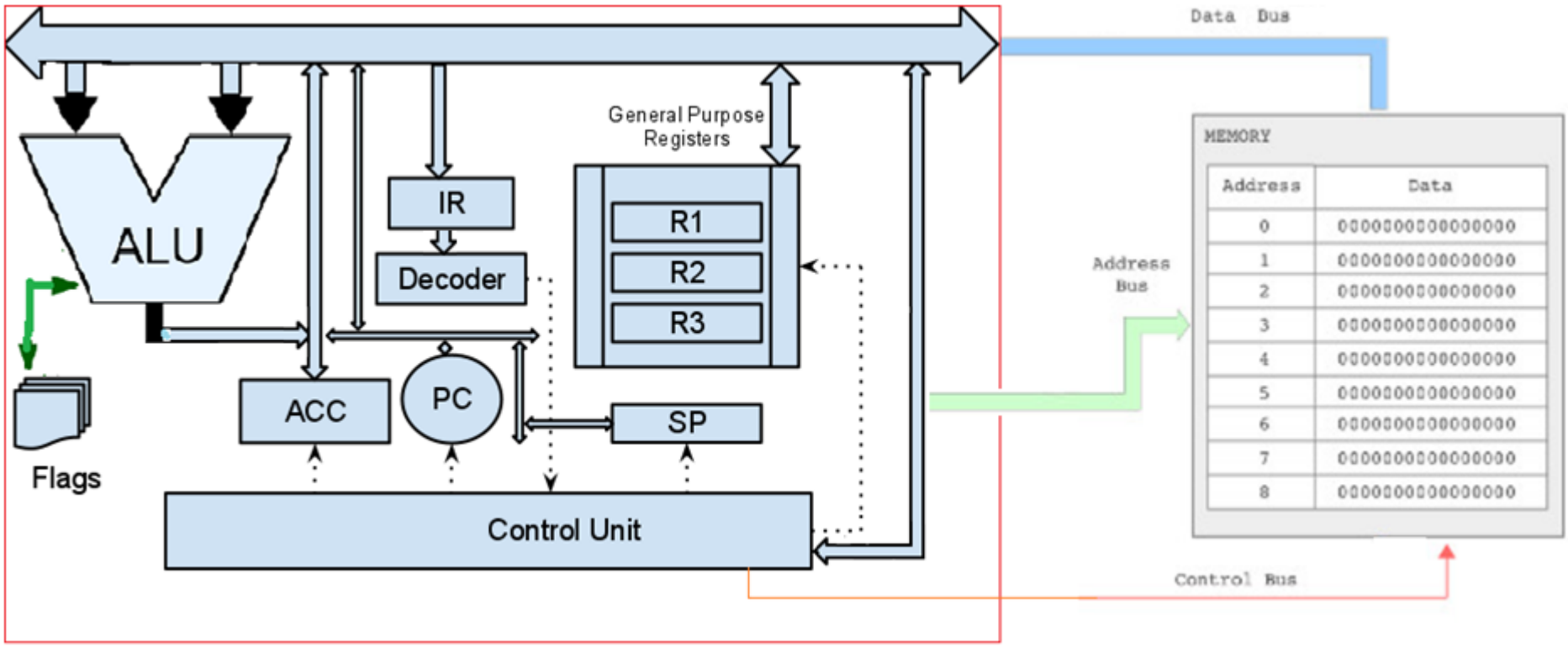
A typical processor is made up of several million transistors, all on one small wafer of silicon called an **integrated circuit** (also called a **chip**.) The toothbrush processor could probably be built with just a few hundered transistors. Integrated circuits are used for other electronic parts of a computer: for example main memory is implemented with memory chips.

Machine Instructions

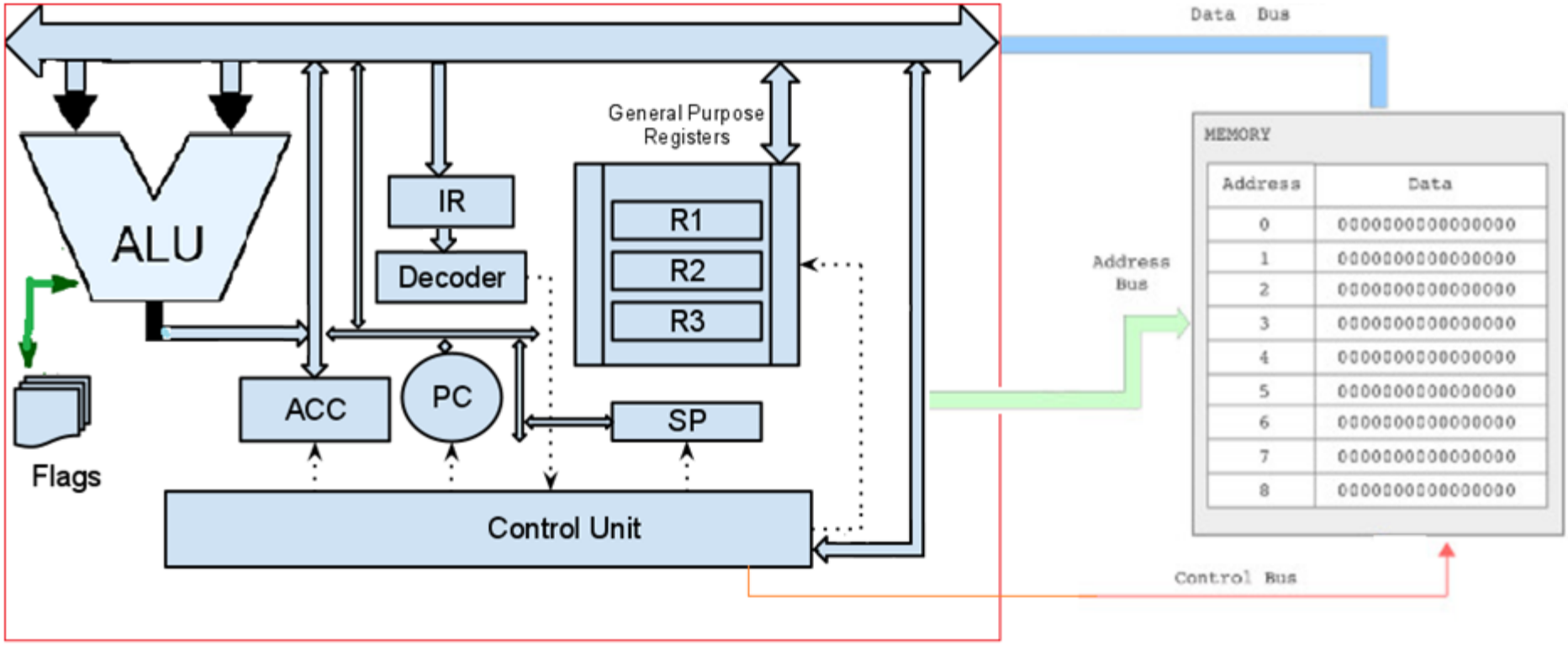
2+3

mov r1, 2 → 010
Mov r2,3 → 100
Add r1,r2 → 001

Sub r1,r2	000
Add r1,r2	001
Mov r1,2	010
Mov r1,3	011
Mov r2,3	100
And r1,r2	101
Or r1,r2	110
Mult r1,r2	111


Data Bus
General Purpose Registers
IR
R1
R2
R3
ALU
Decoder
Address Bus
MEMORY
Address
Data
0 0000000000000000
1 0000000000000000
2 0000000000000000
3 0000000000000000
4 0000000000000000
5 0000000000000000
6 0000000000000000
7 0000000000000000
8 0000000000000000
ACC
PC
SP
Flags
Control Unit
Control Bus

mov r1, 2 → 010
Mov r2,3 → 100
Add r1,r2 → 001



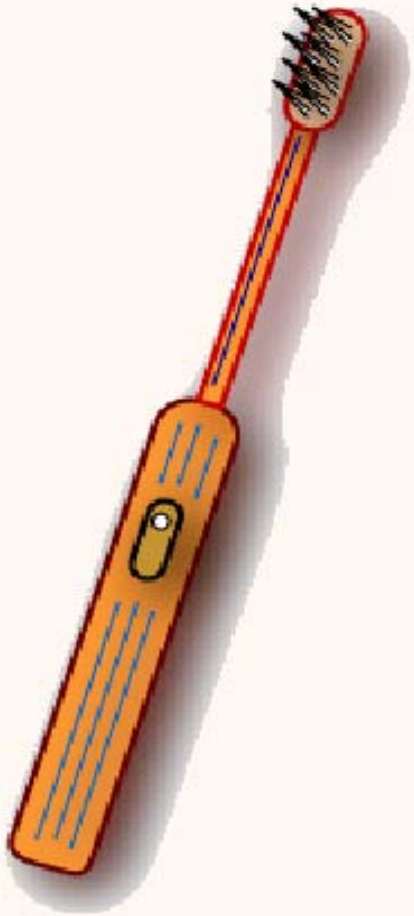
Example of embedding system

Example of Machine Language

Let us say that an electric toothbrush has a processor and main memory. The processor can rotate the bristles left and right, and can check the on/off switch. The machine instructions are one byte long, and correspond to the following machine operations:

Machine Instruction	Machine Operation
0000 0000	Stop
0000 0001	Rotate bristles left
0000 0010	Rotate bristles right
0000 0100	Go back to start of program
0000 1000	Skip next instruction if switch is off

The toothbrush is wired so that when the switch is turned on, the instructions are performed one at a time, in order, starting at address zero. The program can be re-started at address zero with the instruction 0000 0100. The toothbrush is supposed to rotate the bristles left and right while the switch is on, and stop when the switch is off.



Machine Language Program

There are at least two sensible programs for the toothbrush. Here is one program:

Address	Machine Instruction	Meaning
0	0000 0001	Rotate bristles left
1	0000 0010	Rotate bristles right
2	0000 1000	Skip next instruction if switch is off
3	0000 0100	Go back to start of program
4	0000 0000	Stop running

In the electronic memory of the toothbrush the program is stored as a sequence of bits:

0000 0001 0000 0010 0000 1000 0000 0100 0000 0000

The processor starts at the beginning and performs the action described by each code. This is, of course, a silly example. Electric toothbrushes are not controlled by computer processors. And, the machine instructions of actual processors are much more detailed. But the essential ideas of the example are these:

- A machine language program is a sequence of machine language instructions in main memory.
- A machine instruction consists of one or more bytes (in the example, only one).
- The processor runs a program one machine instruction at a time.
- All the little machine operations add up to something useful.

If toothbrush user leaves the switch "on" for a while, the program repeats its operations many times. This is how most programs in real computers run — many little operations add up to a useful function, which is then repeated many times.

Different Processors

There are many types of processors used in computer systems. You probably know something about the processors used in most desktop computers, Intel Corporation's Pentium processors. But there are other types of processors, such as the processors used in cell phones and game machines. A computer system is designed around its processor. The electronics of a computer system are designed for a particular type of processor.

Different types of processors have different machine operations and different machine languages. A machine language program for a typical desktop system (with a Pentium processor) would make no sense to a computer built around a different processor type.

However, the machine operations available with all types of processors can be used to build the same things. All processor types have enough power in their little machine operations to create the same large applications. Anything one processor can do with its machine language program, another processor can do with a program written in its machine language. For example, cell phones are built around a variety of processor types, but all cell phones can do the same things.

The **architecture** of a processor is the choices that have been made for its machine operations, how they have been organized and implemented, and how it interacts with main memory and other components. Architecture is concerned with the general plan and functions of a processor; it is not much concerned with electronic details. A course in computer architecture is required in most computer science departments.

High Level Programming Languages

It is very rare for programmers to write programs in machine language like we did for the electric toothbruch. The executable files (the directly runnable machine language programs) for most applications contain hundereds of thousands of (if not millions) of machine language instructions. It would be very hard to create something like that. As an experiment, look through your hard disk with the file listing utility (the explorer on Microsoft systems.) Look at the size of the something.EXE files. (Remember that there are usually several bytes per machine instruction.)

Most programs are created using a **high level programming language** such as Java, C, C++, or BASIC. With a high level language, a programmer creates a program using powerful, "big" operations which will later be converted into many little machine operations.

For example, here is a line from a program in the language "C":

```
int sum = 0;
```

The machine operations that correspond to this line will set up a small part of main memory to hold a number, store the number zero there, and arrange things so other parts of the program can use it. It might take a hundred machine operations to do all this. Obviously, it is easier for a human programer to ask for all these operations using "C".

Source Programs

Programers create programs by writing commands in a high level language. A high level language program consists of lines of text that have been created with a text editor and are kept in a file on the hard disk. For example, here is a complete program in "C" (Java will be discussed later):

```
#include <stdio.h>
main()
{
  int sum = 0;
  sum = 2 + 2;
  printf( "%d\n", sum );
}
```

This program could be saved on the hard disk in a file called addup.c. Like all files, it consists of a sequence of bytes. Since it is a text file, these bytes contain character data. You can edit the file with a text editor and print the file on a printer. It *does not* contain machine instructions. If the bytes are copied into main memory, they cannot run as a program without some extra work being done.

> A **source program** (or source file) is a text file that contains instructions written in a high level language. It can not be executed (made to run) by a processor without some intermediate steps.

```
#include <stdio.h>
main()
{
  int sum = 0;
  sum = 2 + 2;
  printf( "%d\n", sum );
}
```

This program could be saved on the hard disk in a file called addup.c. Like all files, it consists of a sequence of bytes. Since it is a text file, these bytes contain character data. You can edit the file with a text editor and print the file on a printer. It *does not* contain machine instructions. If the bytes are copied into main memory, they cannot run as a program without some extra work being done.

> A **source program** (or source file) is a text file that contains instructions written in a high level language. It can not be executed (made to run) by a processor without some intermediate steps.

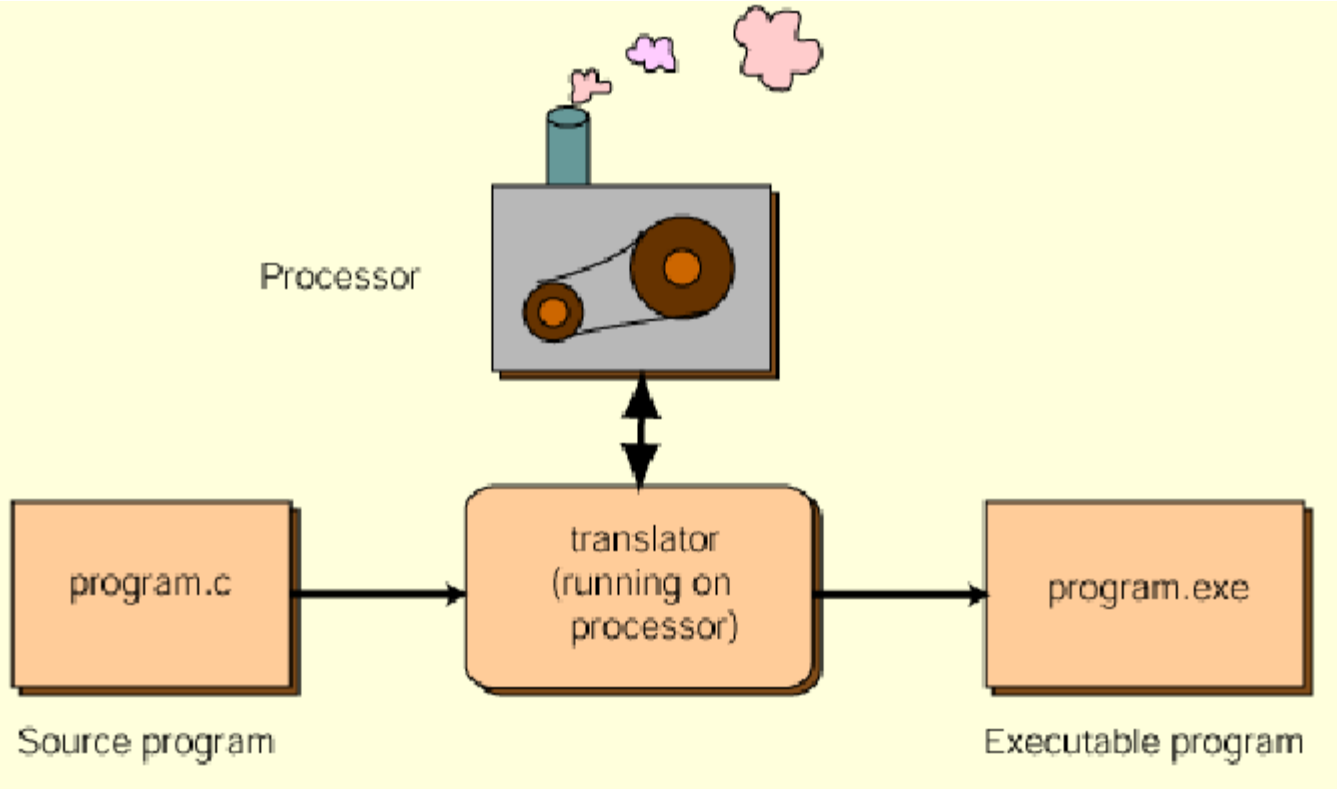
Usually a source program is **translated** into a machine language program. An application program called a **translater** takes a source file as input and produces an executable program (machine language program) as output. For example, the "C" program addup.c could be translated into an executable program. The executable program might be called addup.exe and can be saved on the hard disk. Now the executable version of the program can be copied into main memory and executed.

The word **compile** means the same thing as **translate**. So one can say that a source program is compiled into an executable program.

Program Translation

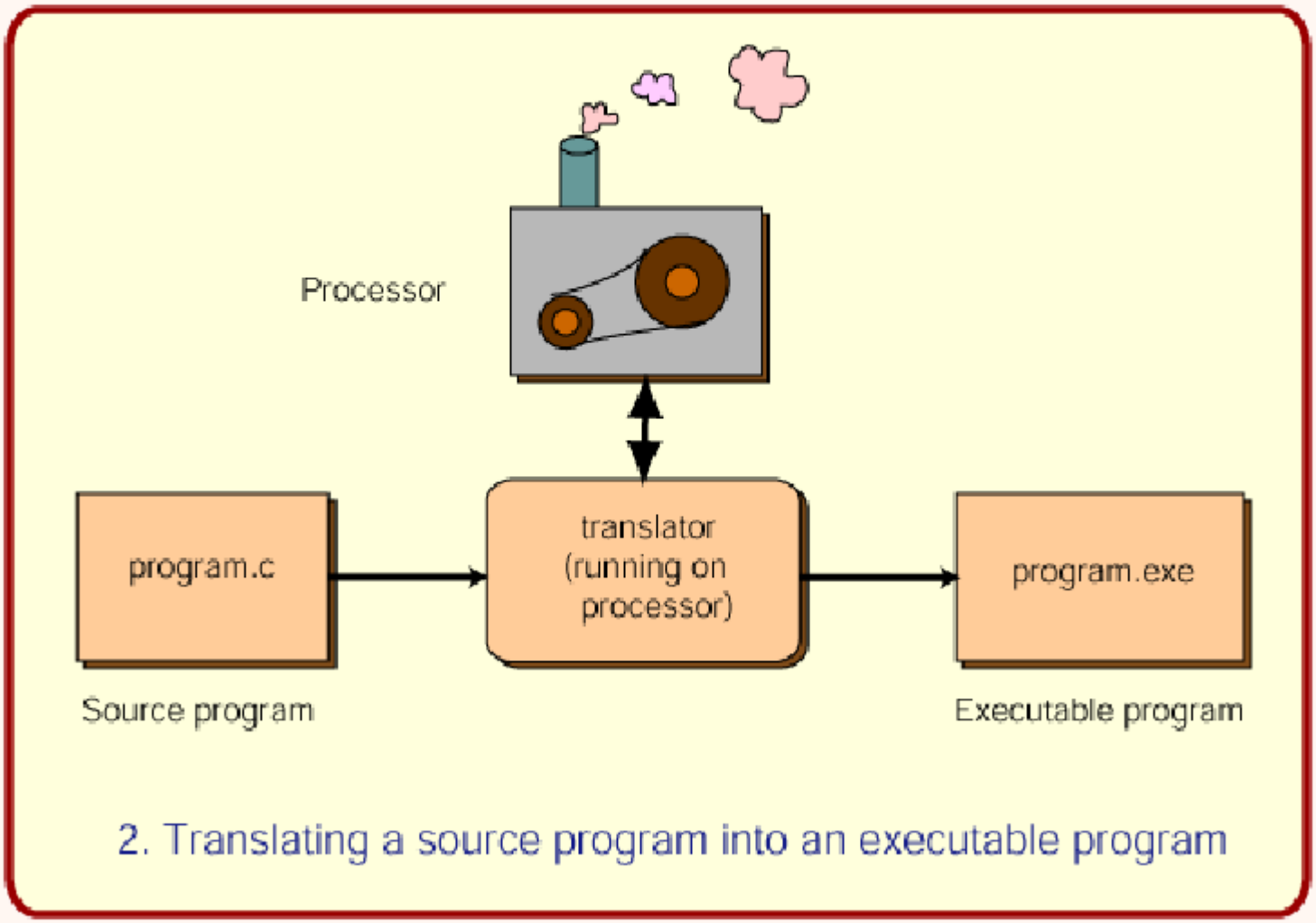
Here is an overview of what goes on:

1. The source file is created using a text editor.
   - It contains instructions in a high level language.
   - It contains bytes that represent characters.
   - The source file is kept on the hard disk.
   - The source file can not be run by the processor.
2. A translator (compiler) program translates the source file into an executable file.
   - The source file remains unchanged; a new executable file is created.
   - The executable file contains machine instructions.
   - A translator translates from a specific high level language (like C) into machine instructions for a specific processor type (like Pentium).
   - The executable file is also kept on hard disk.



Program Translation

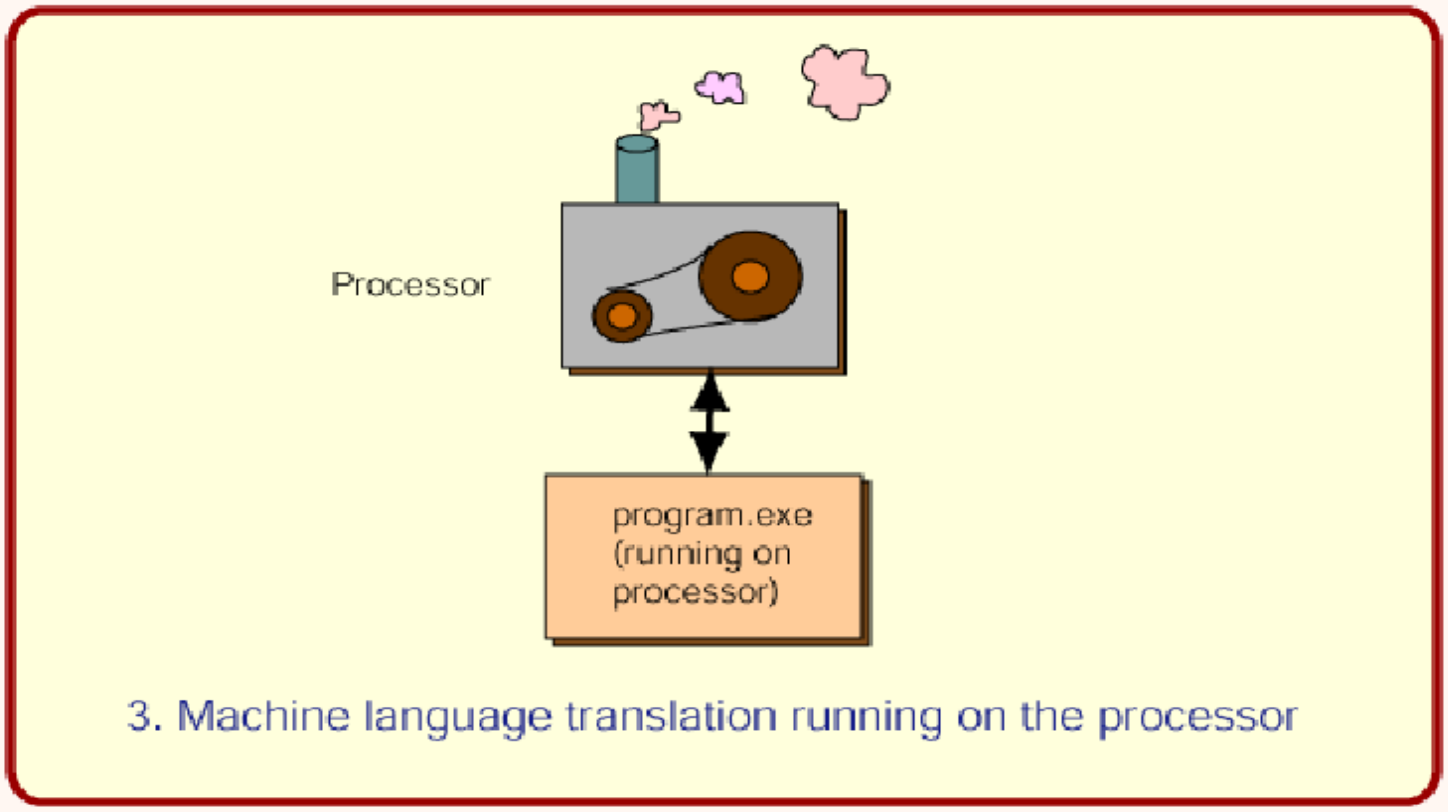
Here is a picture that shows what usually happens with programs written in "C" (Java is different; it will be discussed in the next chapter.)



2. Translating a source program into an executable program

Here is an overview of what goes on:

1. The source file is created using a text editor.
   - It contains instructions in a high level language.
   - It contains bytes that represent characters.
   - The source file is kept on the hard disk.
   - The source file can not be run by the processor.
2. A translator (compiler) program translates the source file into an executable file.
   - The source file remains unchanged; a new executable file is created.
   - The executable file contains machine instructions.
   - A translator translates from a specific high level language (like C) into machine instructions for a specific processor type (like Pentium).
   - The executable file is also kept on hard disk.
3. The program is run by copying machine language from the executable file into main memory. The processor directly executes these machine language instructions.



3. Machine language translation running on the processor

Portability

Ideally, only one program needs to be written in the high level language. That source file can then be translated into several executable files, each containing the correct machine instructions for its intended processor. This is how the same game can be made for desktop computers and game machines. (Ideally) one source file is created, which is then translated into executable files appropriate for each platform.

The idea of using one source file for executables that run on different processors is part of **software portability**. You would like to write a program just once (in a high level language) and then be able to run it on any computer system by translating it into that systems machine language.

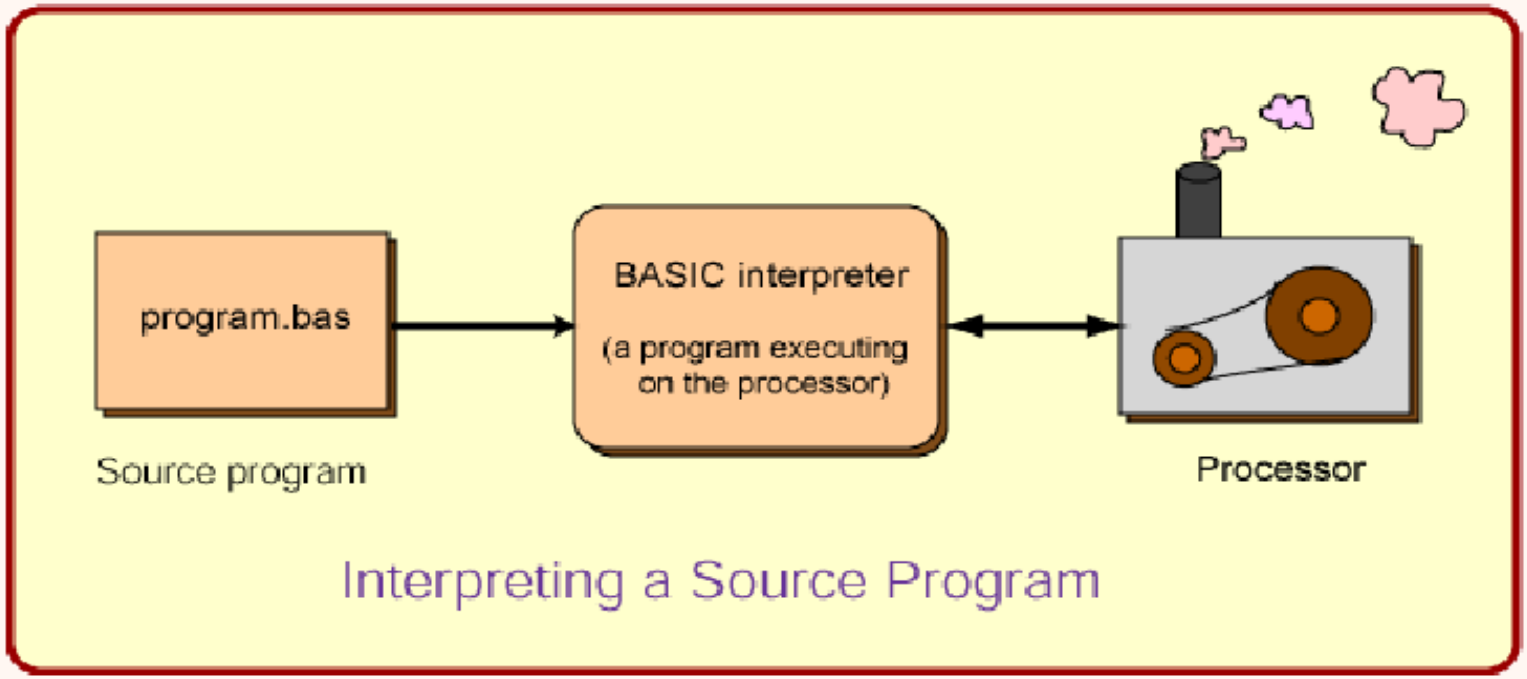
Usually, unfortunately, things do not work out that nicely. There are enough little problems so that it takes a substantial amount of human effort to get a program running on a new system. One of the big advantages of Java is that it is automatically portable between computer systems that have Java support. No human effort is involved at all.

Interpreter

Programs written in a high level language are never directly executed by the processor. You have already seen one way to execute such a program: use a translator to create a machine language program that can be executed directly.

Another way to is to use an **interpreter** for the language. An **interpreter** is a program that acts like a processor that can directly execute a high level language.

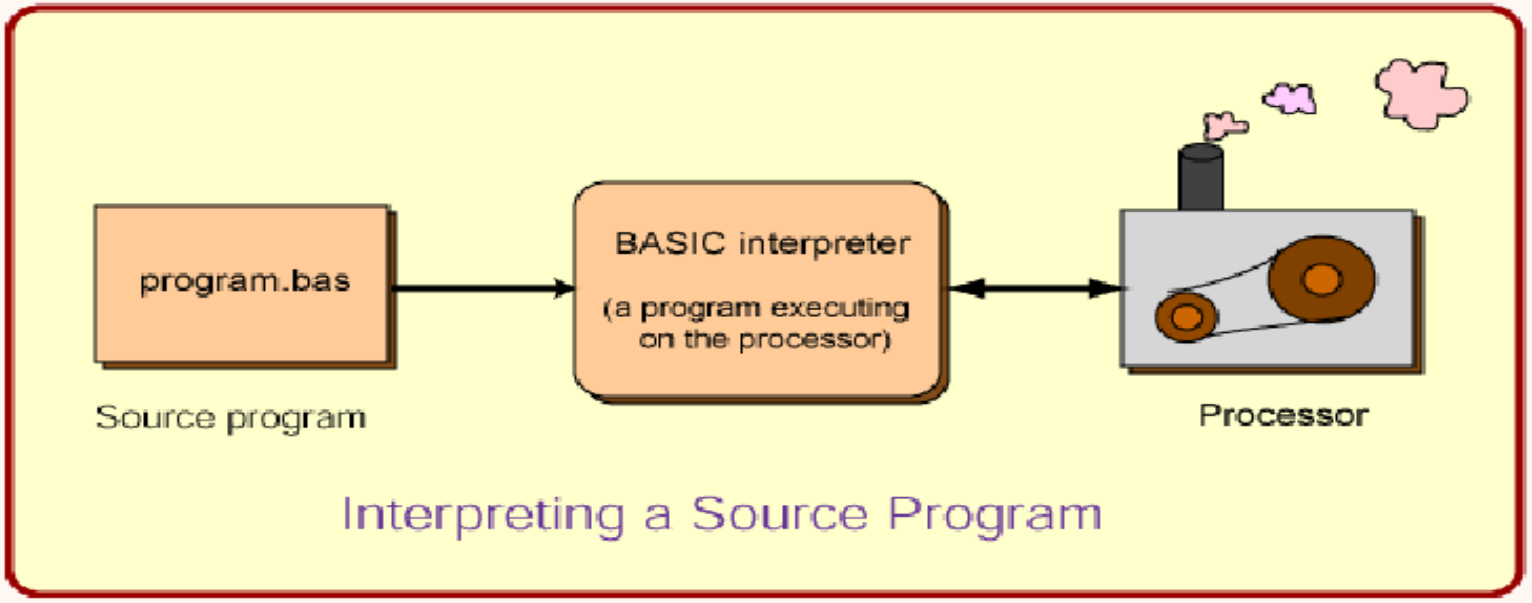
This is a fairly complicated thought. The figure might help:



Interpreting a Source Program

In this figure, the source program "program.bas" has been written in BASIC (a programming language) by a programmer with a text editor. It is being interpreted by the BASIC interpreter, which is running on the processor. The BASIC interpreter reads each command in the source program and does what it says.

This is a fairly complicated thought. The figure might help:



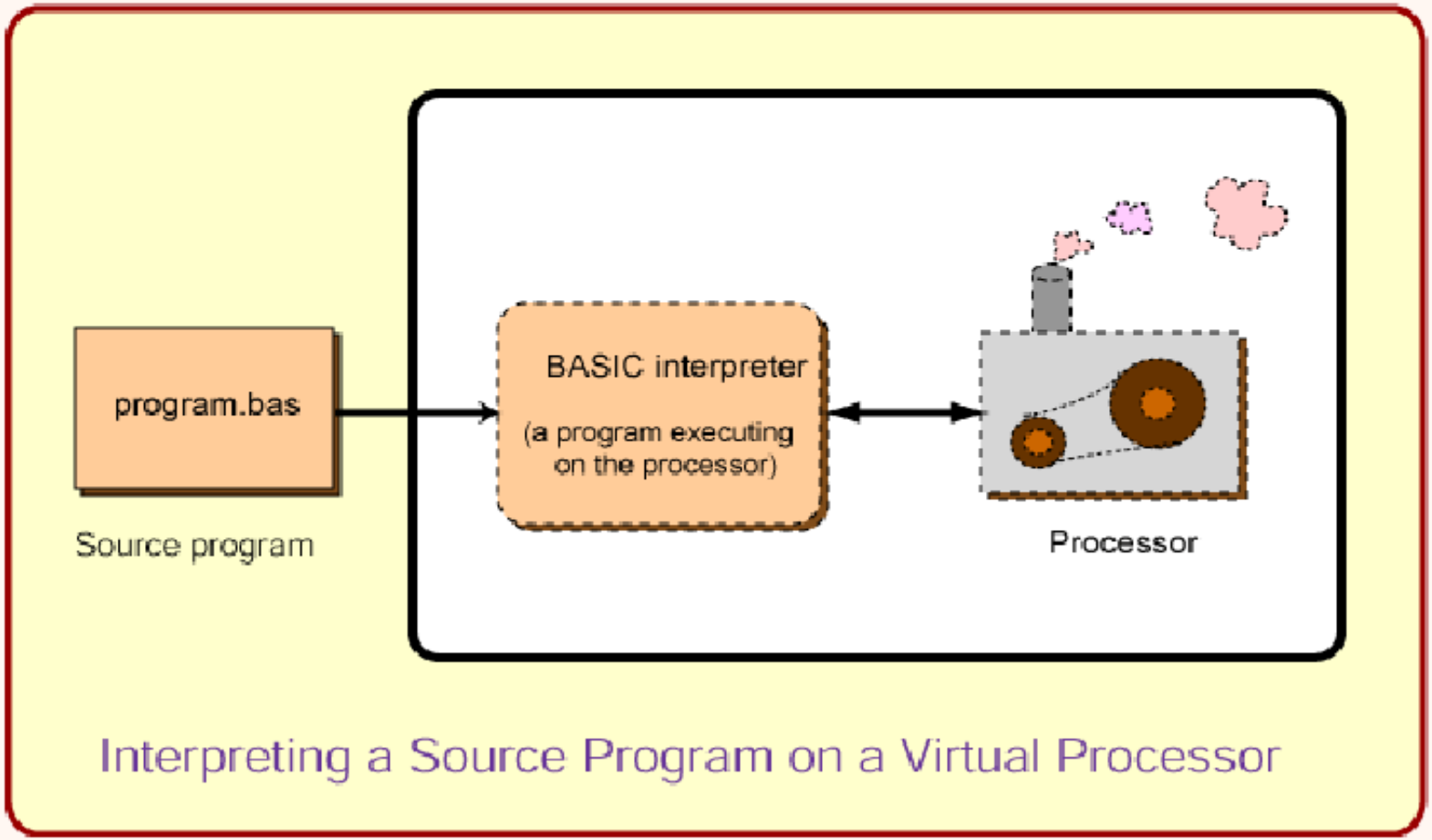
In this figure, the source program "program.bas" has been written in BASIC (a programming language) by a programmer with a text editor. It is being interpreted by the BASIC interpreter, which is running on the processor. The BASIC interpreter reads each command in the source program and does what it says.

You have probably done this many, many times without realizing it. It is exactly accurate (although an unusual use of the words) to say that the computer game DOOM (or any other game) is an interpreter for the commands that the user enters using the mouse and keyboard. The commands for the game DOOM are not the usual type of commands used in a general purpose programing language, but none the less, they are commands. The user enters commands, and the interpreter (the game) executes them one by one. The set of commands and the rules for how to use them correctly form a programming language for the DOOM interpreter.

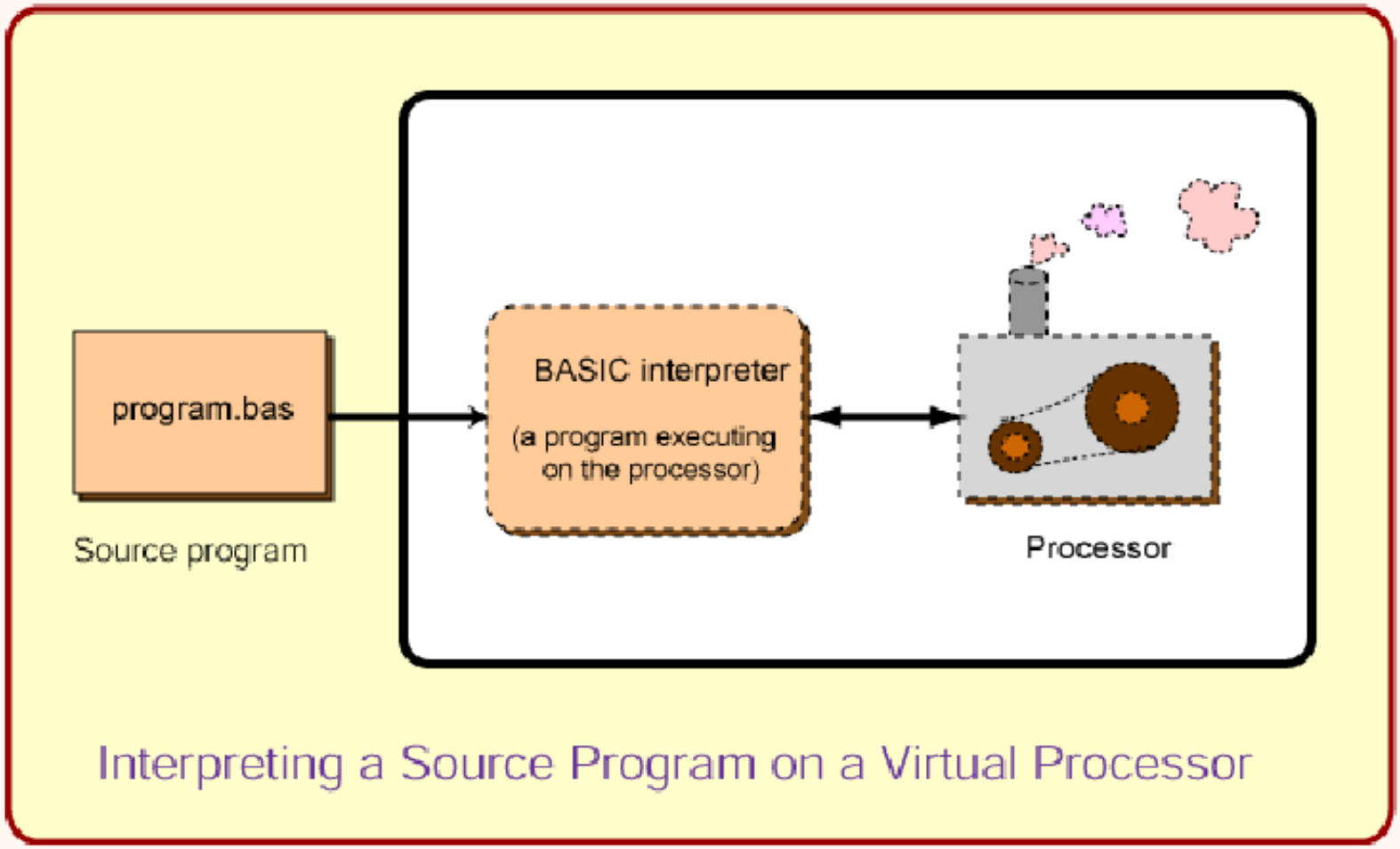
When DOOM is running in "demo mode" (a sample game played without user input) it is in fact running a program written in the DOOM language. This program is not being directly processed by the Pentium chip in your computer. The DOOM game is interpreting it command by command.

Virtual Machine

When an interpreter is running a BASIC source program, both the interpreter and the source program are in main memory. The interpreter consists of machine instructions that the hardware can execute directly. The BASIC source program consists of commands in the language BASIC that the interpreter knows. From the perspective of the BASIC program, *it looks like the commands in BASIC are being directly executed by some sort of machine.* Here is the figure, modified to show this:



Interpreting a Source Program on a Virtual Processor



Interpreting a Source Program on a Virtual Processor

This is really the same as the previous figure, but now a box has been drawn around the actual (hardware) processor and the interpreter that it is executing. The combination "looks like" a machine that can directly act upon BASIC source programs. The word **virtual** is used in situations where software has been used to make something look like the real thing. In this case it looks like we have a machine that can directly execute BASIC, so we can say that we have a BASIC virtual machine.

Speed

The situation with computer languages is somewhat like that with human languages:

- **Translator:** takes a complete document in one language and translates it into a complete document in another language, which can then be used at any time.
- **Interpreter:** acts as an intermediate between a speaker of one language and a speaker of another language. Usually works a sentence at a time. You and your French translator (say) could in combination be regarded as a "virtual French speaker".

Using a human interpreter as an intermediate is slower than conversing directly in a particular language. The same is true with computer language interpreters. The interpreter has to do quite a bit of work to deal with the language it is interpreting. The extra work makes it look like the virtual processor is much slower than the real one.

End of the Chapter!

You may wish to review the following. Click on a subject that interests you to go to where it was discussed.

- Machine operations and machine instructions.
- Executing a program.
- Differences in types of processor chips.
- High level programing language.
- Source program.
- Program translation.
- Program interpretation.
- Portability.

The next chapter will discuss how the language Java fits into the concepts this chapter has discussed.