Adam Furmanek

From Bulb to C#

Event Sponsors

Strategic Sponsors

Demant

avanade

Gold Sponsors

Relativity

ProDataConsult

medius

KMD
An NEC Company

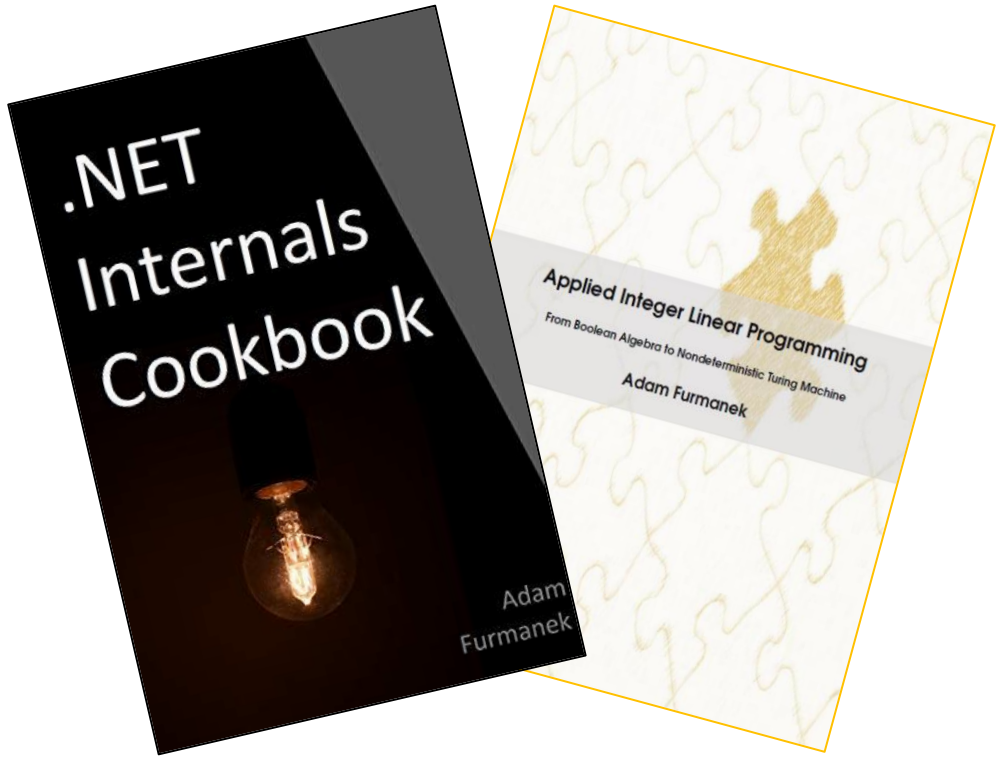
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Random IT Utensils

IT, operating systems, maths, and more.

Agenda

Bulb — it's all we need.

From bulbs to semiconductors.

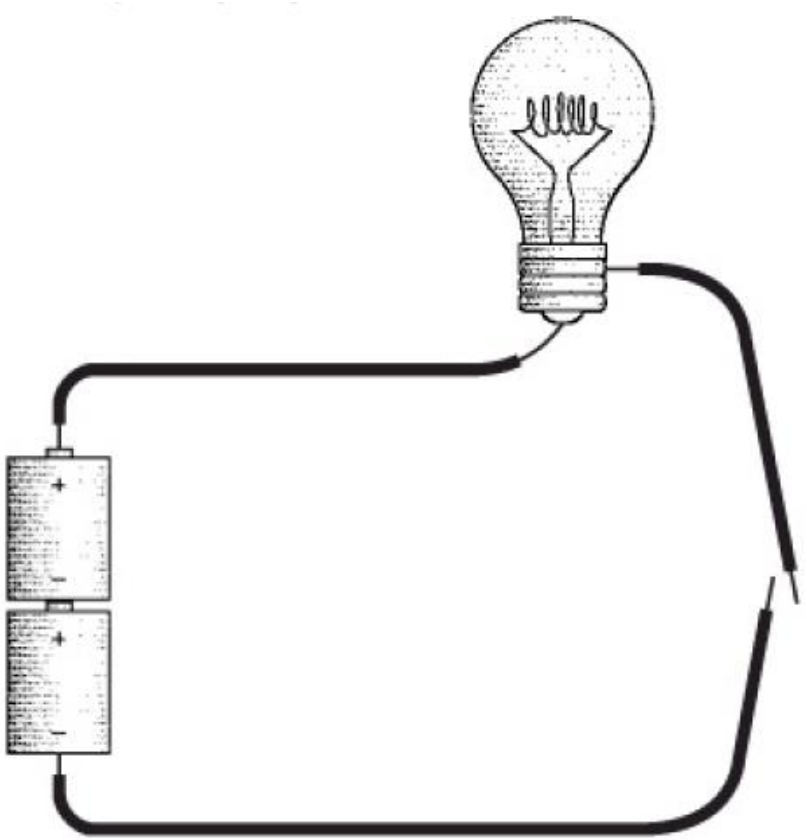
Computer architecture.
- It's all a bunch of bytes
- Von Neumann, Harvard

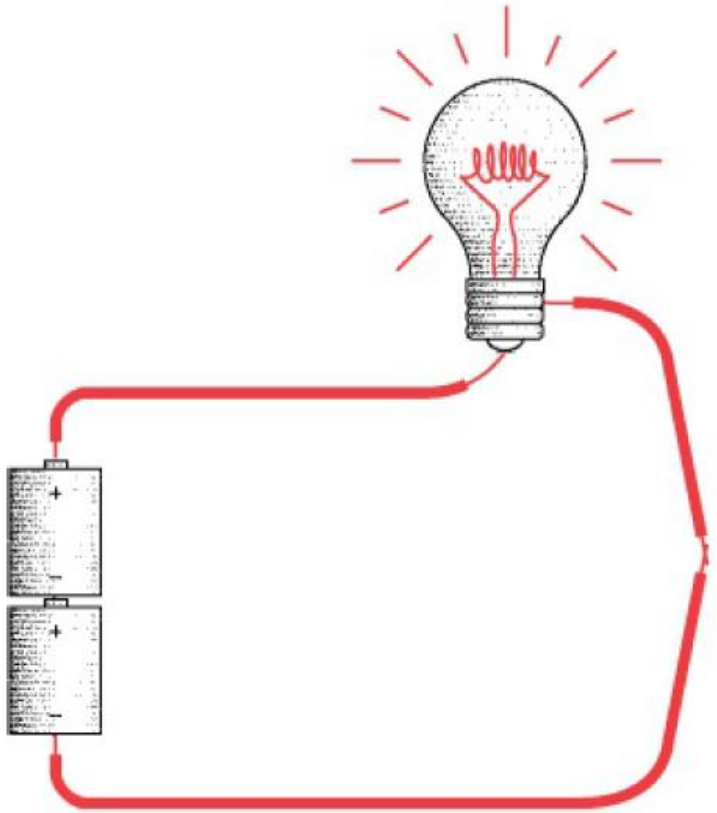
CPU architecture.
- CISC, RISC, EPIC and others
- x86 and a bit of history

Codes:
- Microcode, machine code, assembly.
- Operating System level code.
- User mode code.
- Managed code.
- Aaand C#.

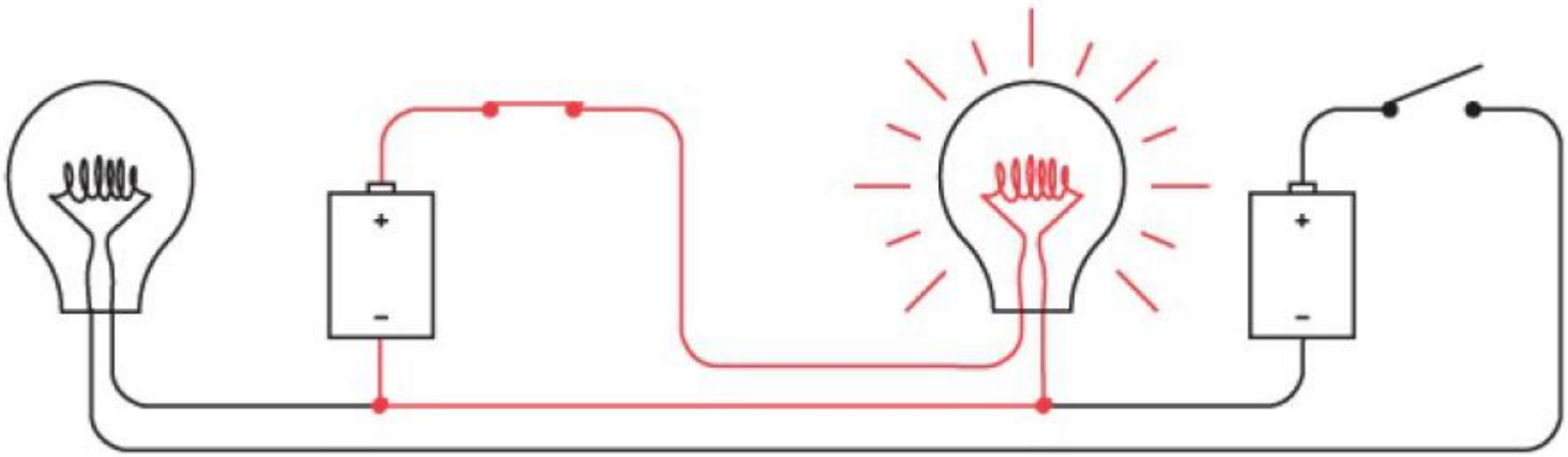
Bulb — it's all we need

Bulb

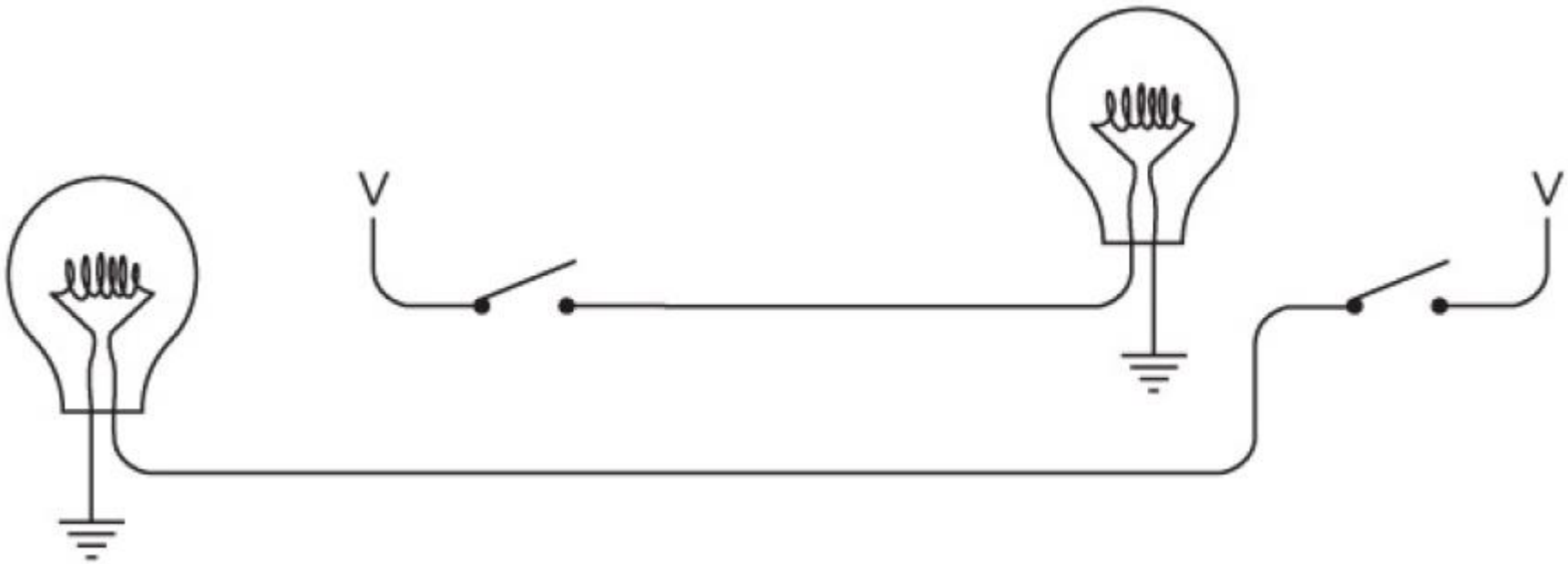




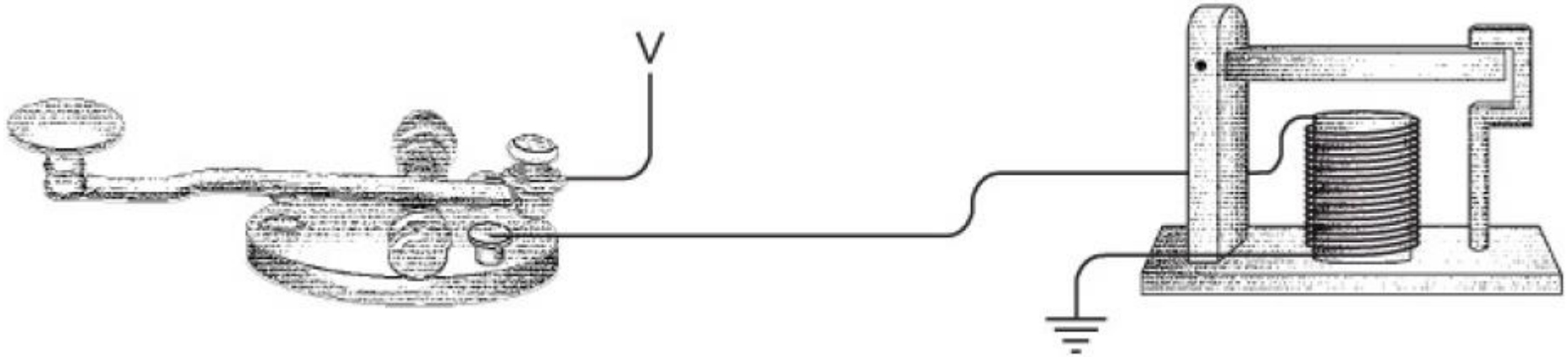
How to communicate?



How to communicate?



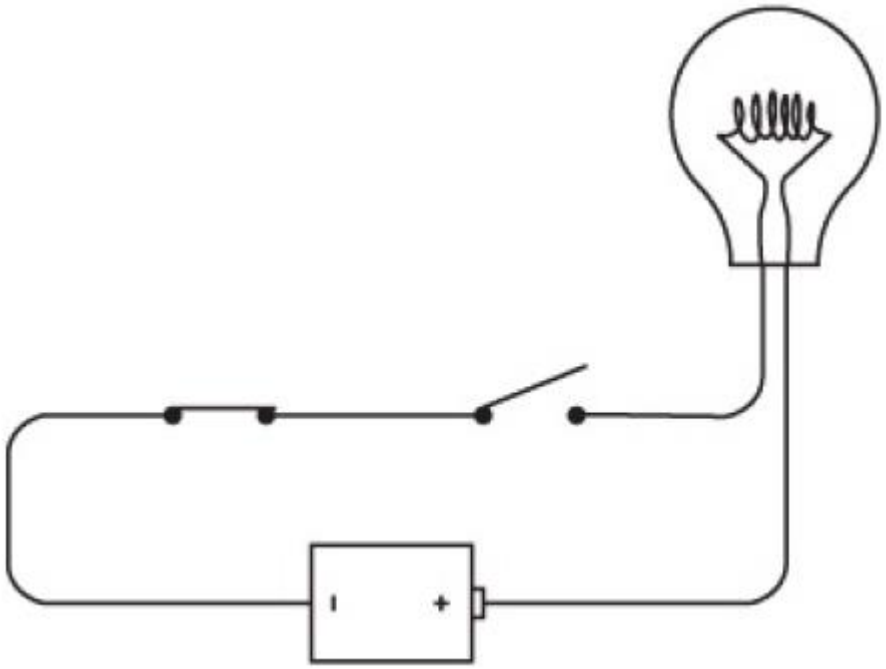
Telegraph

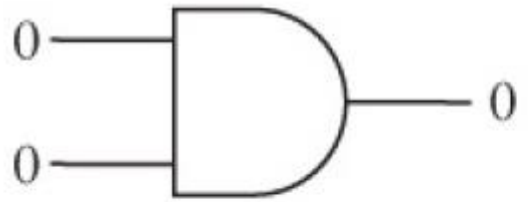


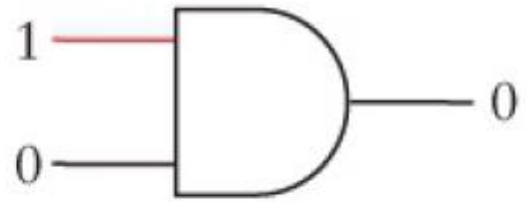
Boolean Logic

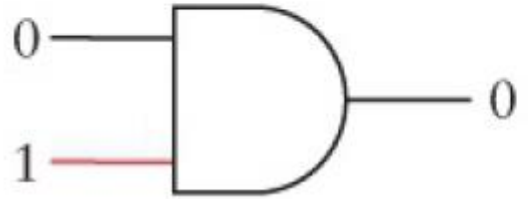
Operation	Function	Symbols
Conjunction	AND	&&
Disjunction	OR	||
Negation	NOT	~
Exclusive Or	XOR	^
Not AND	NAND	
Not OR	NOR	

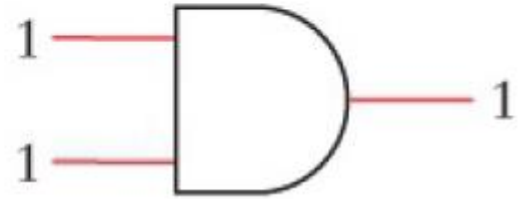
AND



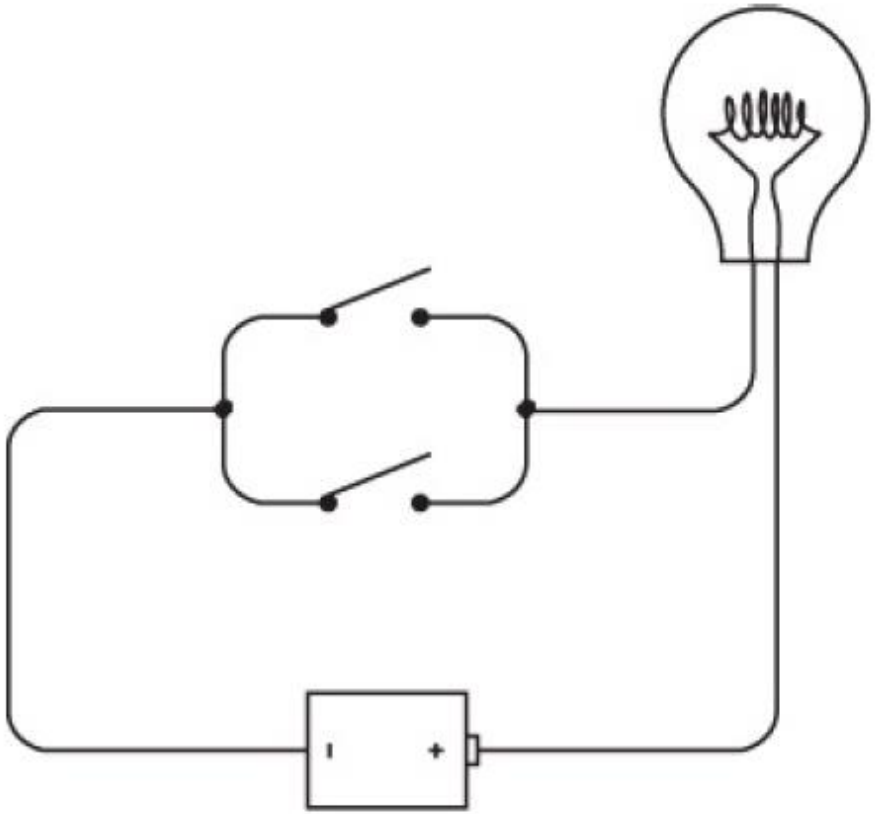


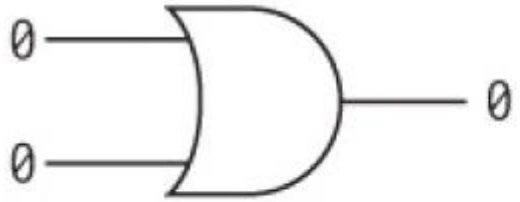


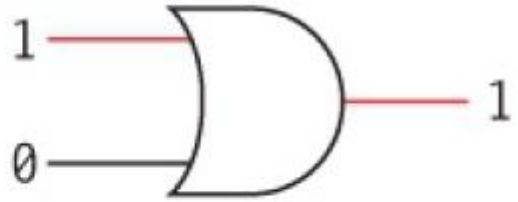


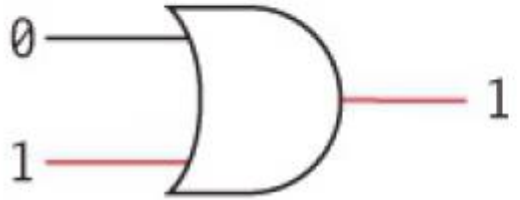


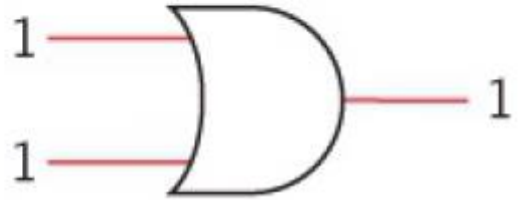
OR



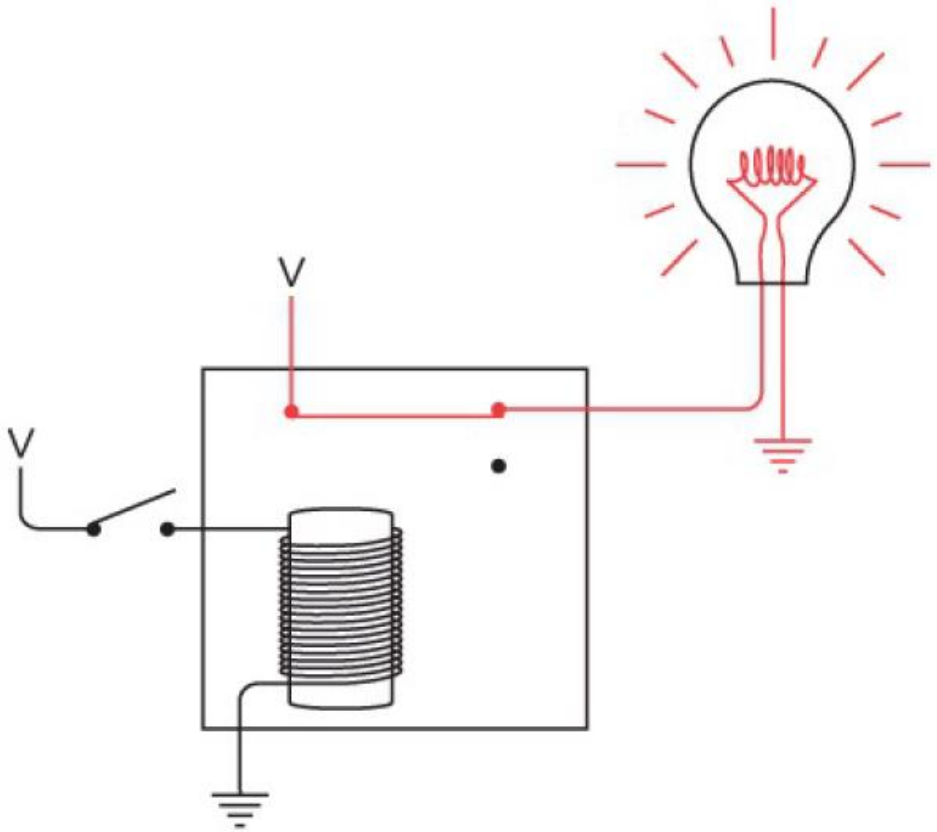


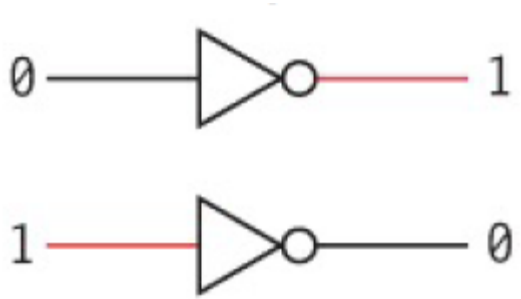






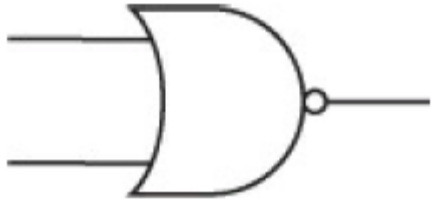
NOT (inverter)



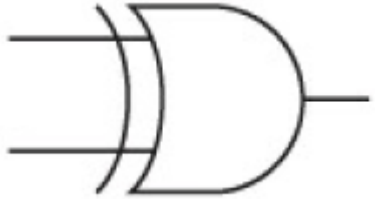


Others

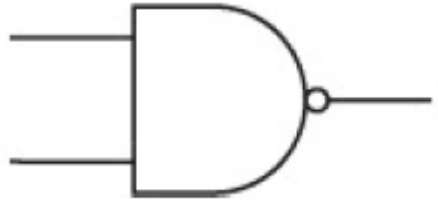
NOR



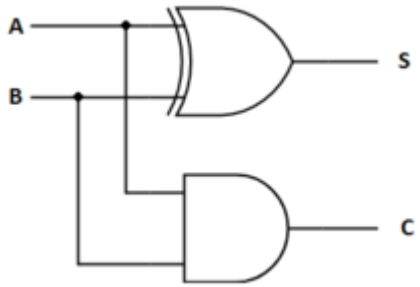
XOR

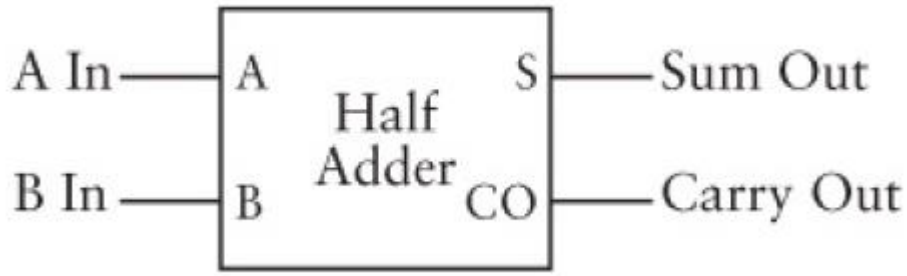


NAND



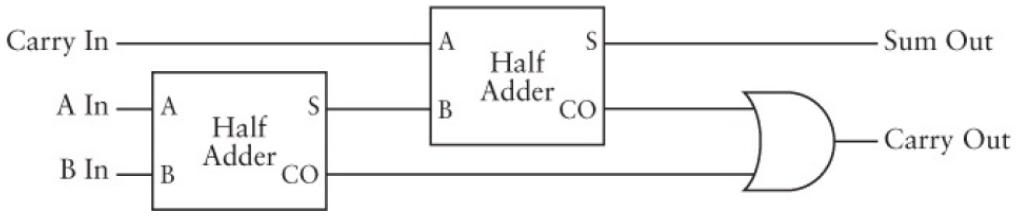
Half Adder





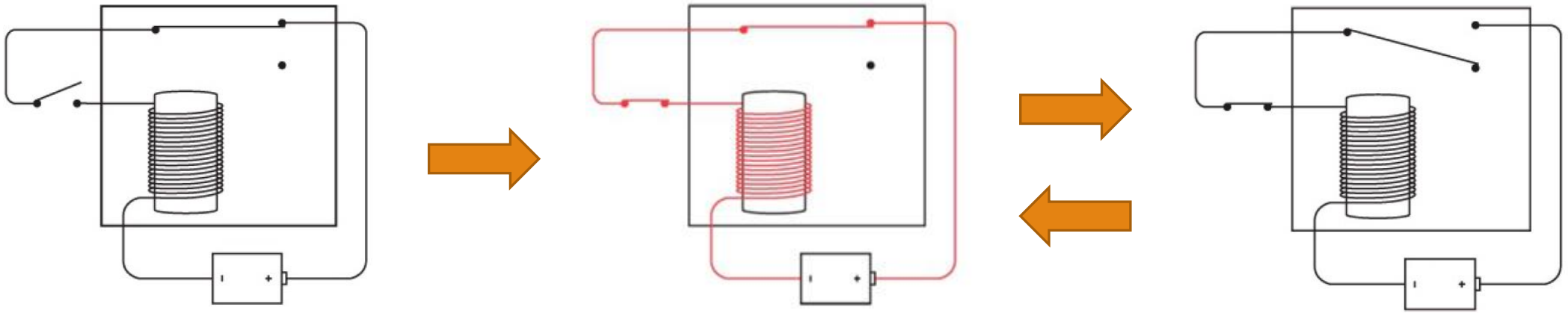
a_i	b_i	s_i	c_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

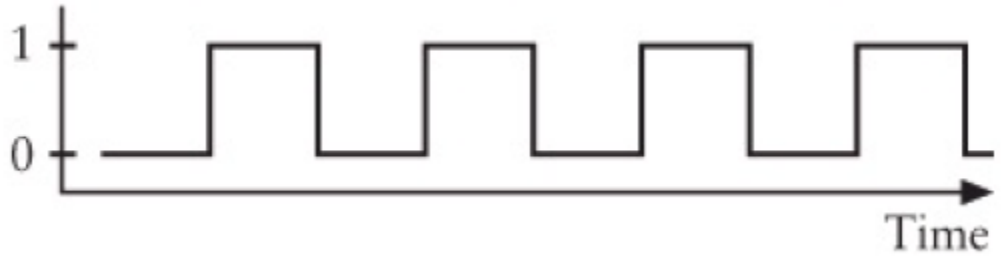


a_i	b_i	c_{i-1}	s_i	c_i
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

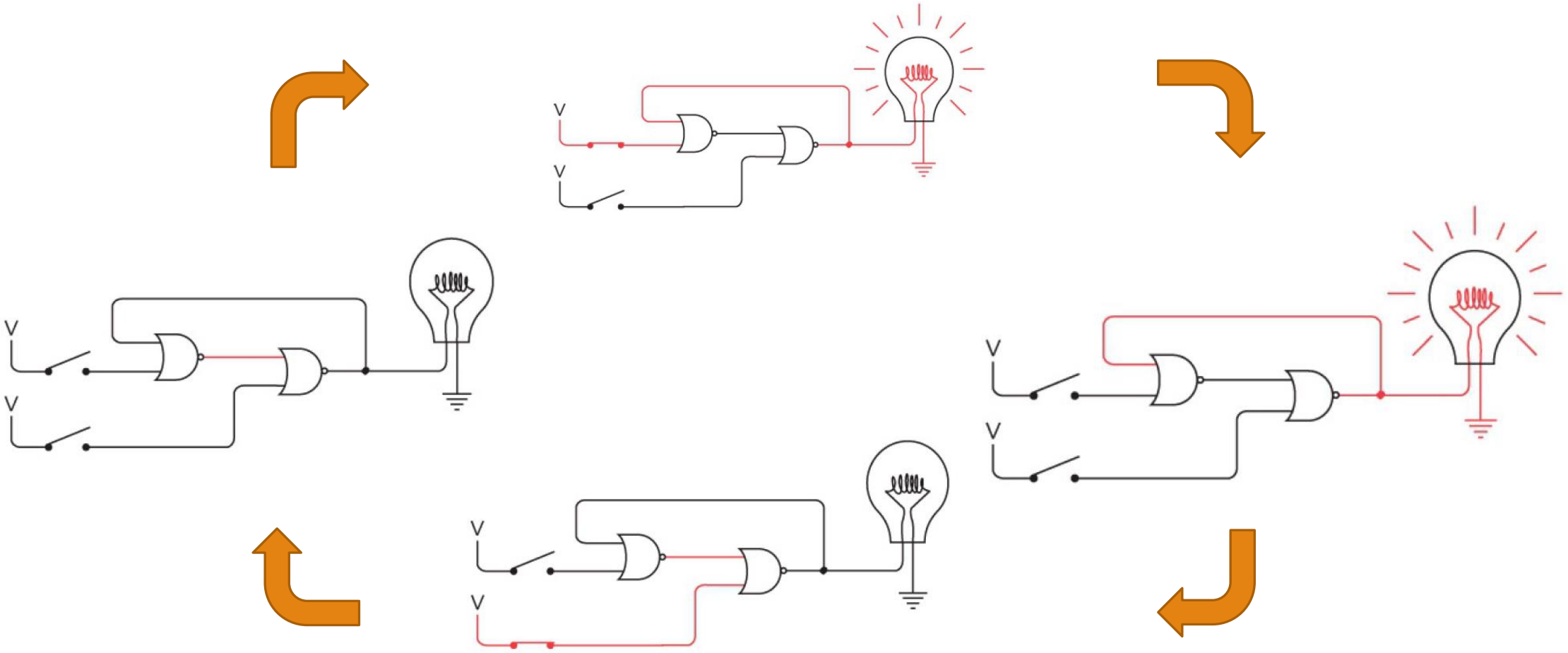
Oscillator



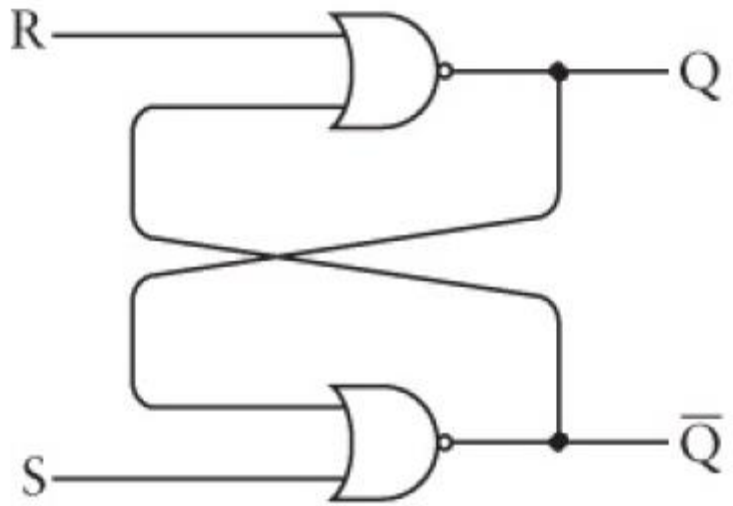
Oscillator



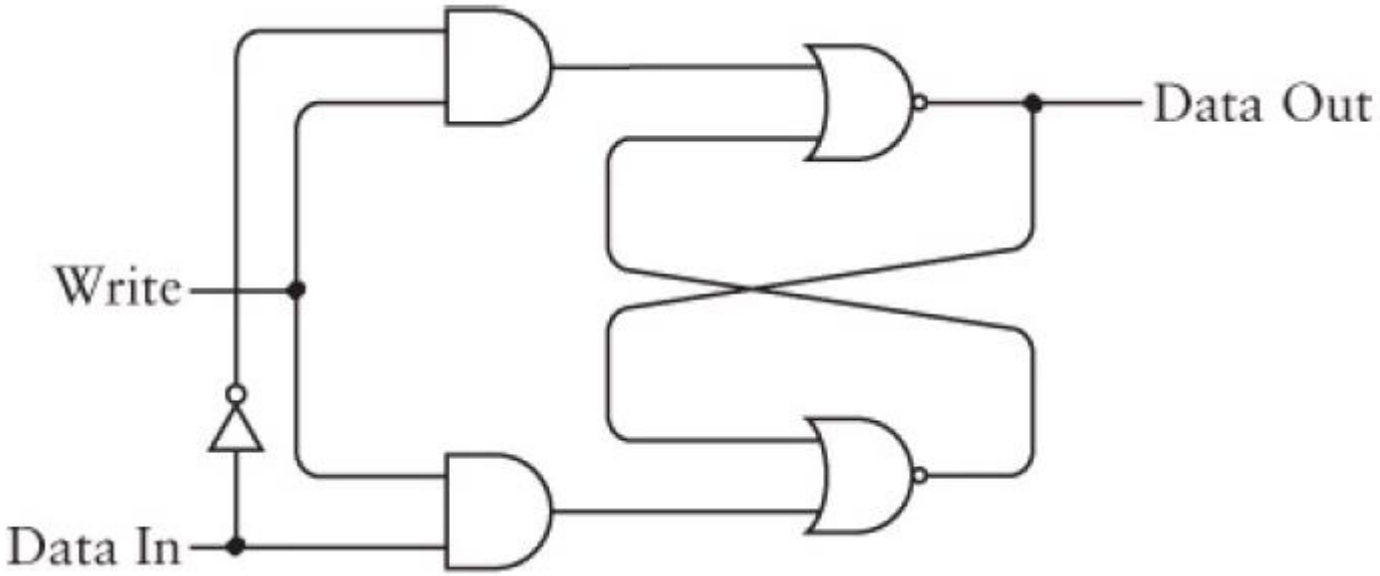
Reset-Set (RS) Flip-Flop



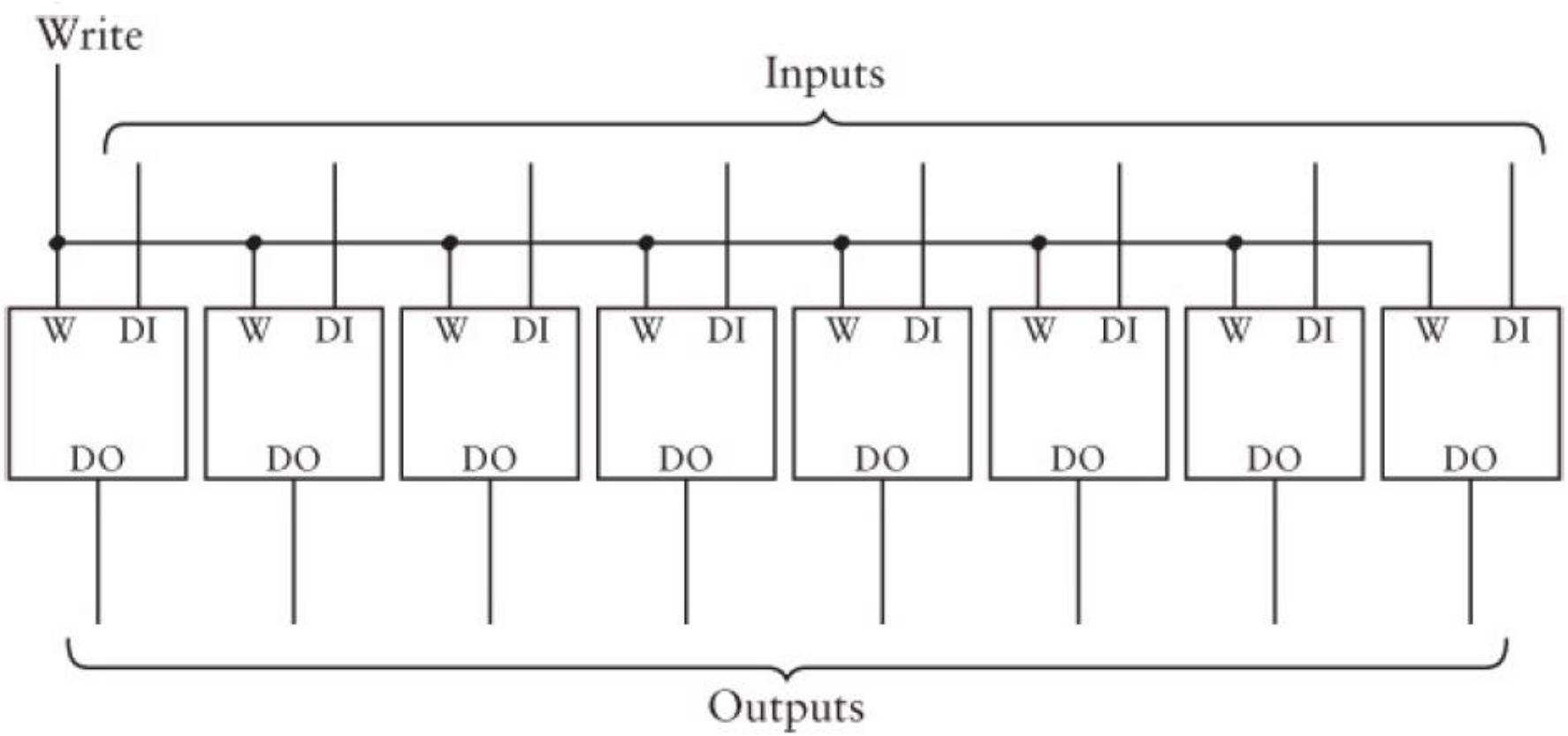
Reset-Set (RS) Flip-Flop



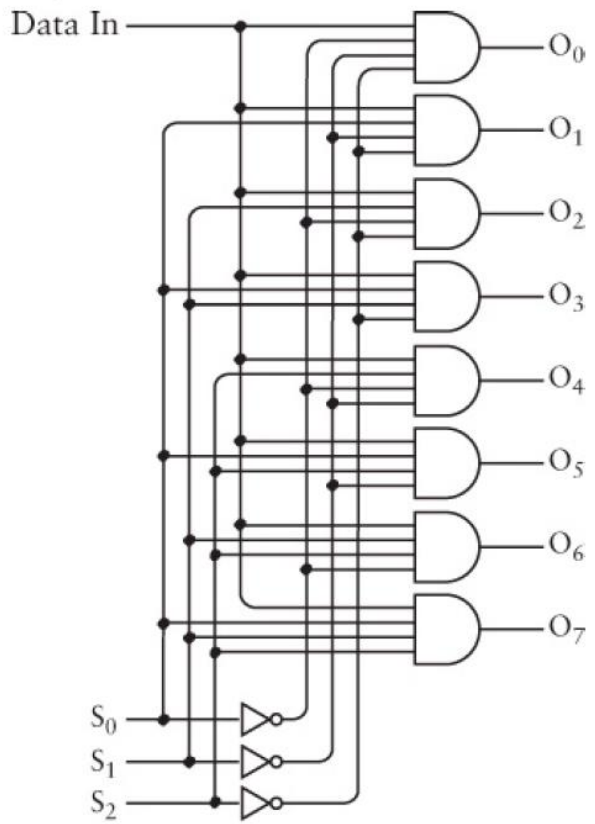
Level-triggered Data-type Flip-Flop



Multibit latch



3-to-8 Decoder




```
bool S0;
bool S1;
bool S2;

bool O0;
bool O1;
bool O2;
bool O3;
bool O4;
bool O5;
bool O6;
bool O7;

if(!S0 && !S1 && !S2) O0 = true;
else if(S0 && !S1 && !S2) O1 = true;
else if(!S0 && S1 && !S2) O2 = true;
else if(S0 && S1 && !S2) O3 = true;
else if(!S0 && !S1 && S2) O4 = true;
else if(S0 && !S1 && S2) O5 = true;
else if(!S0 && S1 && S2) O6 = true;
else if(S0 && S1 && S2) O7 = true;
```

How to draw an owl

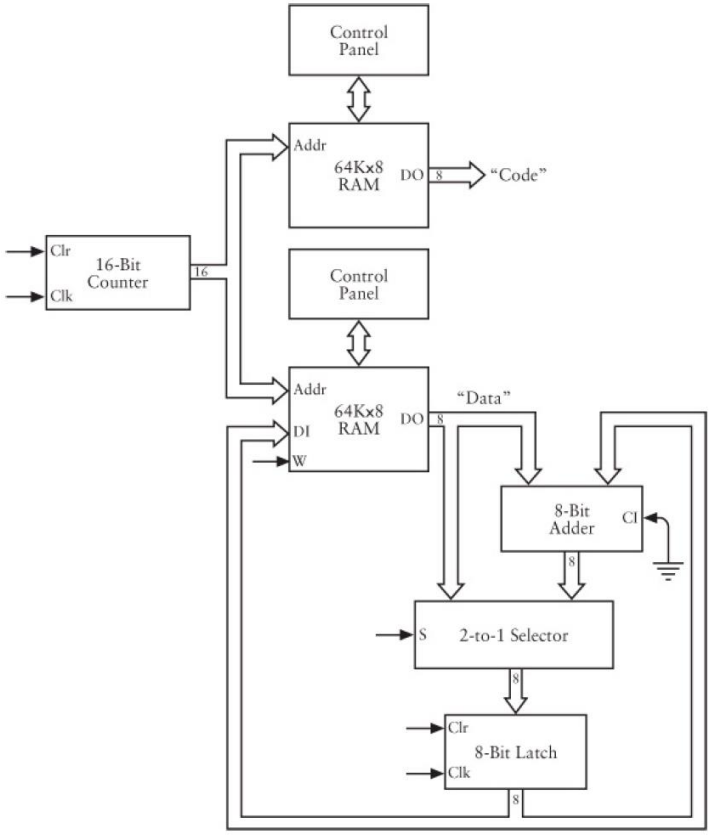
1.

2.

1. Draw some circles

2. Draw the rest of the owl

Computer


Control Panel
Addr
64Kx8 RAM
DO
8
"Code"
Clr
16-Bit Counter
Clk
16
Control Panel
Addr
64Kx8 RAM
DO
8
"Data"
DI
W
8-Bit Adder
CI
8
S
2-to-1 Selector
8
Clr
8-Bit Latch
Clk
8

From bulbs to semiconductors

It’s all about physics

Conductors

- Very conducive to the passage of electricity
- Copper, silver, gold
- Technically we can „kick” the lone electron out so it’s free to move

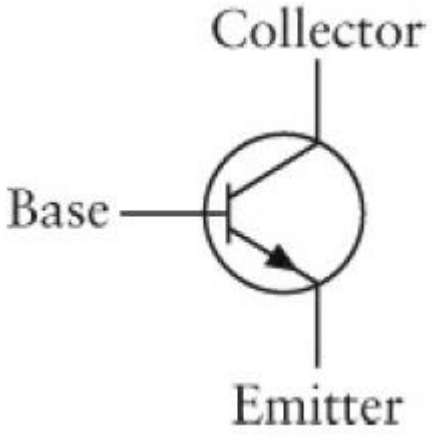
Insulators

- Barely conduct electricity
- Rubber, plastic

Semiconductors

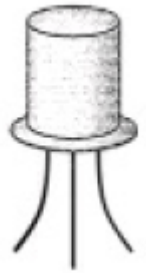
- Not because they conduct half as well as conductors but because their conductance can be manipulated
- Can be doped – combined with certain impurities
- Pure semiconductors aren’t good conductors

NPN transistor

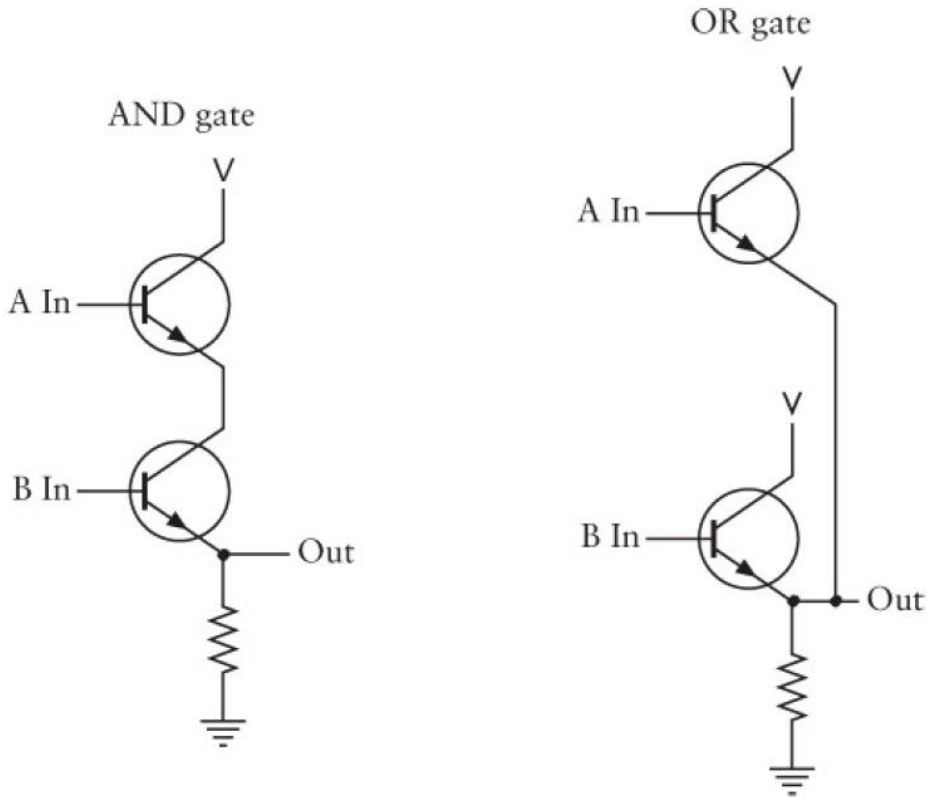


Small voltage on the base can control a much larger voltage passing from the collector to the emitter.

Invented by William Shockley, John Bardeen and Walter Brattain in 1947.



Gates with transistors



Is it easier to make computer with transistors?

Pros

- We can fit many more transistors in smaller space
- They are much stabler than other solutions (like vacuum tubes)
- We can build blocks of transistors (chips) doing well-known things (like half adder)

Cons

- We still have to worry about interconnections
- The smaller the connections the more heat we get

Integrated Circuit

Commonly called the chip.

Manufactured through a complex proces of layering thin wafers of silicon that are precisely doped.

It’s expensive to develop a new integrated circuit but it’s cheap when they are mass produced.

Different technologies to build ICs — Transistor-Transistor Logic (TTL) and Complementary Metal-Oxide Semiconductor (CMOS).

By building more and more sophisticated blocks we end up with System On Chip (SOC).

Computer architecture

Computer

CPU.

RAM.

Some way of getting instructions into RAM (input device).

Some way of showing results (output device).

Non-volatile memory (storage).

All these elements must communicate! How do we put them together?

Bus

All integrated circuits are mounted on circuit boards.

These boards must communicate and they do that with a bus.

Bus is a collection of digital signals:

- Address signals – to address the memory
- Data Output signals from the CPU
- Data Input signals to the CPU
- Control signals – to coordinate actions (e.g. indicate that CPU wants to write)

Bus

Just like CPU has manual so devices know how to talk to it, the same way bus can be standardized.

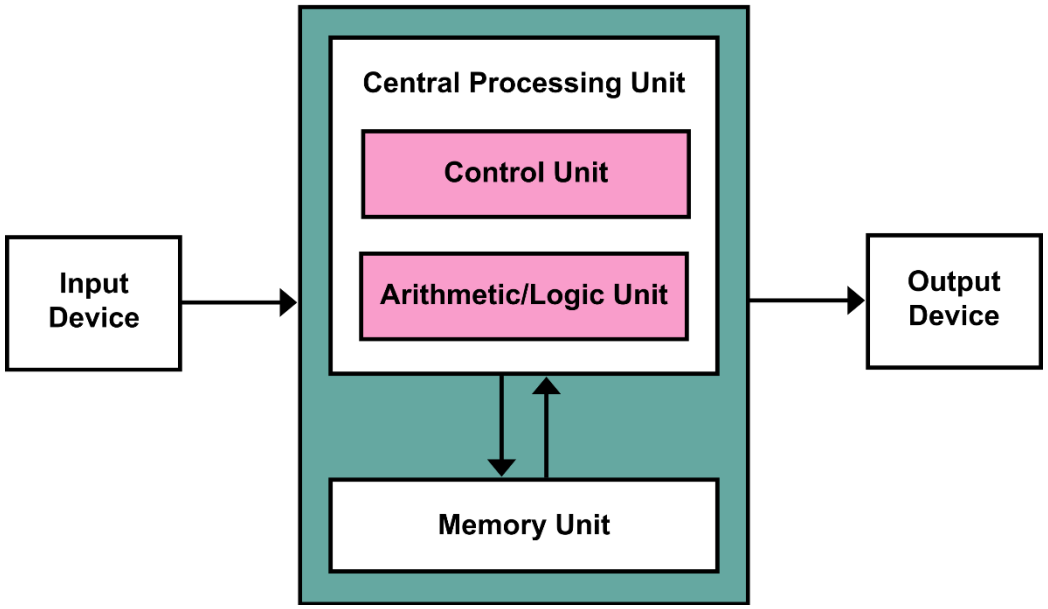
Industry Standard Architecture – designed by IBM for the original PC.

S-100 bus for the 8080 chip.

Micro Channel Architecture (MCA) bus.

IIC designed by Philips.

Von Neumann architecture (Princeton)



Data and instructions are both stored in the primary storage.

Instructions are fetched from the memory one at a time.

Processor decodes the instruction and executes it.

Other architectures

Harvard architecture

- One dedicated set of addresses and data buses for memory access, another one for instructions (so data and instructions are separate)
- Can be faster since it can access both of them at the same time
- Distinct code and data address spaces

Modified Harvard architecture

- Caches for instruction and data, sharing the same address space
- Allows treating instructions as a read-only data

The computer you have is conceptually a von Neumann architecture but technically a modified Harvard architecture.

CPU architecture

CPU architecture

Term reused in many contexts.

Instruction set architecture (ISA) – a design of physical instructions the CPU is capable of executing.

Microarchitecture (computer organization) – the way a given ISA is implemented.

Specifies how a CPU works – what is the cycle, what is the pipeline, how are instructions ordered etc.

Many other things – endianess, register length, addressing, security, programming model etc.

Instruction Set Architecture

Defines

- Supported data types
- Registers
- Hardware for managing memory
- Memory consistency, addressing, virtual memory
- Memory model

Typically classified by architectural complexity.

ISA

Complex Instruction Set Computer (CISC)

- Many specialized instructions
- Allows to „do more” with „less”
- Instructions can vary in length
- x86
- Technically often translated to the RISC by the CPU

Reduced Instruction Set Computer (RISC)

- Only frequently used instructions
- Other operations implemented as subroutines
- Much easier to execute as instructions typically have the same structure

Other used architectures

- Very Long Instruction Word (VLIW)
- Explicitly Parallel Instruction Computing (EPIC)

Conceptual architectures (not widely used)

- Minimal Instruction Set Computer (MISC)
- One Instruction Set Computer (OISC)

ISA specifies instruction encoding, length, parameters, etc

IBM PC

In 1974, Intel produced the 8080 – 8-bit microprocessor. Later used in Altair 8000 which was the first home computer.

In 1976, Intel produced the 8085 – 8-bit microprocessor, fully compatible with 8080. Smaller than the predecessor.

In 1978, Intel produced the 8086 – 16-bit microprocessor able to access 1MB of memory. It wasn’t compatible with 8080.

In 1979, Intel produced the 8088 – identical to 8086 but externally accessed memory in bytes so could use chips designed for 8080.

8088 was used in 5150 Personal Computer – the IBM PC.

x86 and AMD64

x86

- 16-bit CISC architecture.
- Backwards compatible with 8-bit one

x86-32 (IA32)

- 32-bit extension of x86 architecture
- Introduced in 80386 CPU
- Has a compatibility mode with x86 (so you can run old applications)
- Introduced MMX and SSE

x86-64 (AMD64 or Intel 64 or EM64T)

- Developed by AMD after Intel failed with their IA64 architecture
- Compatible with x86 architecture, capable of running 32-bit applications.

SSE and others

Typical CPU instructions are „simple”

- Calculations: MOV, ADD, SUB
- Control: JMP, CALL, RET
- Comparisons: CMP

CISC allows to run much more with „one” instruction

- Add 512 bits at once
- Compare strings (arrays of bytes)
- Encrypt using AES

They are introduced using extensions: MMX, SSE, SSE2, AVX

If a CPU doesn’t support them, they will be emulated with reduced performance.

Codes

Microcode

Sits one level below the machine code.

Can be used to emulate operations which are not done in the hardware.

x86 translates CISC instructions into a series of micro-operations. This is hardwired for most instructions but for some rarely used it’s done in a microcode.

Not portable, very coupled with the CPU it’s running on.

Updated via UEFI/BIOS updates or regular system updates.

Runs in the CPU directly, not accessible to the „regular” programmer.

Machine code

The one which we „implemented” with bulbs.

Hard to read, rarely written by hand.

Instructions of different lengths.

Instructions encoded as numbers with endianess in mind.

Executed by the CPU directly. Accessible to the „regular” programmer in either kernel or user mode.

Assembly language

Readable from of machine code.

One assembly instruction may represent multiple different machine instructions

- MOV in assembly is translated to different moves depending on the arguments

Exposes memory and computer architecture to the programmer.

Assembled by the assembler and then linked into executable.

Accessible to the „regular” programmer in either user or kernel mode.

Operating System language

The API exposed by the operating system

- WinAPI
- System V
- POSIX

Provides functions for the device management and OS configuration.

Accessible to the „regular” programmer in either kernel mode (drivers) or user mode (regular applications).

User vs Kernel mode

CPU must control who can access peripherals.

Typically introduces a notion of Rings

- Ring 0 (kernel mode) is a lowest level where all CPU instructions can be executed and all devices are accessible
- Ring 3 (user mode) is a highest level where memory access is limited, devices are not accessible

Operating System runs in Ring 0 and switches to Ring 3 to execute user applications.

Rings 1 and 2 not used. Sometimes virtualization hypervisors use Ring 1 to increase performance.

Stack is provided by the CPU and is one per Ring. Heap is provided by the operating system (or implemented in user mode enitrely).

User mode native applications

„Native” applications can be implemented in any „native language”

- Typically some high level languages are used like C++

Can be also written using assembler or generated directly as a machine code.

They don’t have access to the peripherals, they need to ask Operating System to do the job.

OS then enters the kernel mode by calling special CPU instructions.

Managed applications

To increase the portability of the code, we introduce another platform on top of the native user mode

- JVM
- CLR
- Web browser
- Python or other runtimes

They typically don’t use the OS functions directly. Instead they call manager wrappers exposed by the platform.

Direct access is possible and is called „interop”.

Managed languages

To increase portability, manager platforms use their own languages. Byte code in JVM, Intermediate Language in CLR.

That language is typically compiled to the machine code using Just In Time compiler.

Platform makes sure the application is „correct” – all memory accesses are verified, pointers are avoided, exceptions in place of physical segfaults.

We can write in managed languages directly or generate them from higher level languages.

High level managed language

C#, Java, other languages translated into some intermediate form.

Typically portable.

Most of the times are not aware of the quirks of the platform they run on.

Compiled to the lower level managed languages by the compiler as a part of application development.

So how does it work?

1. We write application in high level managed language.
2. Code is compiled by the compiler to the low level managed language.
3. Platform runs the code by compiling it with a Just In Time compiler. We now have a machine code running in the user space.
4. Machine code calls Operating System functions. They are written in high level native code mostly or assembly.
5. High level native code (or assembly) is compiled to the machine code running in kernel space.
6. Machine code is then fetched by the CPU and translated to the microoperations using microcode.
7. Microoperations are physically executed by the transistors. The same things we did with bulbs are executed billion times each second.

Summary

Computer is a relatively simple concept. What is hard is how to have a great speed of execution and development.

It took us long time to build standards. Same way like now we have multiple languages, we had mutliple incompatible CPUs years back.

Machine code is not the lowest level. Sometimes it’s not even close.

Q&A

References

https://www.youtube.com/watch?v=CQtSS6g00h0 – How Transistors Work

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html - Intel 64 and IA-32 Architectures Software Developer Manuals

https://fawzi.wordpress.com/2009/05/24/virtualization-and-protection-rings-welcome-to-ring-1-part-ii/ - Ring -1 for virtualization

„Code: The Hidden Language of Computer Hardware and Software” by Charles Petzold

Event Sponsors

Strategic Sponsors

avanade

Relativity®

Gold Sponsors

N-iX

<epam>

KMD
An NEC Company

HITACHI
Inspire the Next

Hitachi Energy

<vpTech/>

SCALO

Silver Sponsors

Softesco

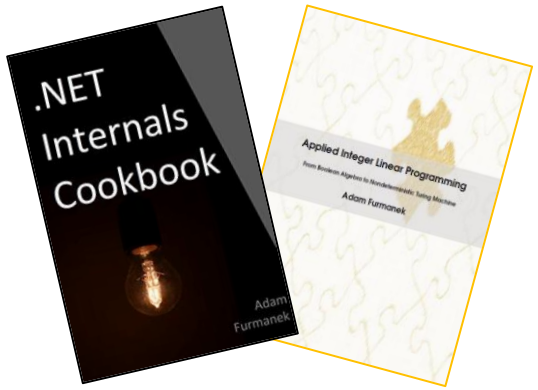
grupa pracuj

Point72

Please rate this session using

.NET DeveloperDays mobile app

(available on Google Play and AppStore)



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM