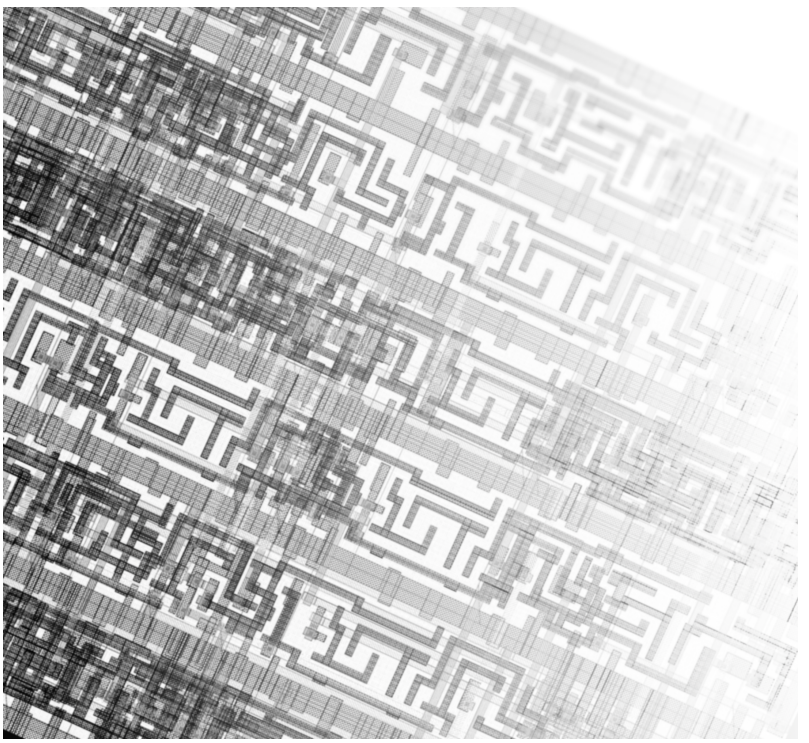
The Fundamentals 2



Partial Layout of a 65nm CMOS ASIC design. The image shows the multi-layer metal track patterns, as generated by a computer aided design tool, in preparation for generating manufacturing data for the fabrication of the silicon chip. Large lateral lines represent power grids, smaller lines are interconnections to individual transistors. The area shown covers a region of about 5x5 microns.

Photo Chris Crispin-Bailey 2019.

2.1 Some initial observations

In order to make progress with learning the principles of computer architecture, organisation, systems, and operating principles, there are many concepts that are often assumed to be as good as common knowledge. This can be confusing for the beginner, and of course this is whom this text is aimed at. So we should take some time to establish a few basic ideas before jumping in to details.

By the end of this text, we might wish to be able to understand and perhaps evaluate computer systems in terms of their components, their behaviour, and their performance. Inevitably we will need to simplify some concepts and details, and it is for the enthusiastic reader to go beyond that level of basic understanding, and fill in the greater level of expertise that is possible with further study.

Let us begin by asking what we think a computer system is: It is undoubtedly a collection of hardware and software. But that only gives us the thinnest veneer of a definition. So let us probe a little deeper. Exactly what do we mean by 'hardware'? We can attempt to define hardware in terms of a system of components and characteristics, perhaps along the following lines:

Key elements and characteristics of Hardware

- It includes **electronics**, **chips**, **circuit boards**, **cables** and **connectors**.
- But also, by extension, it includes **peripherals**: the things we plug into our computers to build larger and more complex configurations.
- Some peripherals are all but essential. Without a **keyboard** and **mouse**, or at least a **touch-screen**, most computer systems have limited usability for the average user.
- All electronic devices consume **power** and generate **heat**, and most include at least some **digital electronic** circuits.
- Digital electronics operate on the level of **binary numbers** and **binary signals**.
- Many of the performance limitations of computers relate to the **speed** at which circuits can manipulate these binary signals, and many others relate to how quickly binary information can be **moved** from one place to another.

Software is also a highly complex composition of elements, with varied purposes, capabilities, and characteristics:

Key elements and characteristics of Software

- **Applications** that you buy and **install** on your laptop/phone/-computer.
- **Utilities** that are part of the operating system, be it WINDOWS, Linux, or another option.
- The **operating system** itself, and software embedded within peripherals, typically known as **firmware**.
- Low level software that is part of the standard computer motherboard, known as **BIOS** (**basic input/output system**).
- Software that runs as part of a **web-page**.
- Software that **compiles** (converts) program **source code** into processor **machine code**.
- Software that **interprets** code written in one style into code in another style, to permit it to run on a given system.
- Software programs that you write yourself and then run on the system, referred to as **user programs**.
- Software that we **do not want**, such as **viruses**, that interfere with normal system operation.
- Software that is installed to **detect and prevent viruses** from interfering with normal operations.

Clearly, there are many aspects to both hardware and software, and we will need to bring all of these things together to give a complete picture of a modern **computer system**. We will begin to sketch out the basics in this first chapter, and then build toward a compete picture.

2.2 Important units of measurement and specification

We will encounter a number of possibly unfamiliar units of measurement when we begin to study computer systems, and we will also encounter parameters that are widely used to specify systems behaviour and requirements. It is worth reviewing a few of these right away, to get a fair background appreciation of some of these before we begin to use them in earnest.

Power: Electrical power consumed by a computer component or system, often measured in watts (W), milliwatts (mW) , microwatts (uW), and even nanowatts (nW). [1].

[1] Standard scientific units for power in computer systems are typically defined as watts (W), milliwatts (mW) ($1x10^{-3}$) , microwatts (uW) ($1x10^{-6}$), and occasionally even nanowatts (nW) ($1x10^{-9}$).

Frequency: A measure of how often a given activity happens in any one second. In a computer system, typically, this refers to a basic operating cycle called a clock cycle. The basic measure of **frequency** is **Hertz (Hz)**. Most computer systems operate at very high frequencies, and therefore MegaHertz (MHz), and GigaHertz (GHz) are normal units of measurement.

Clock Cycle: The frequency of activities is typically regulated by a **clock signal**. This is a repetitive pulse that dictates each new event cycle, and operates at a frequency as mentioned above.

Data Capacity: The number of data items a device or system can manage, or store. Typically, data capacity is quoted in large multiples of bits or bytes, and typically using binary measurement units rather than decimal (see below).

Data Rate: The quantity of data that can be transferred from one place to another in a given amount of time, usually one second. For example a connection between two components may have a data rate of 100 million bytes per second (or about 95.3 binary Megabytes/sec).

2.2.1 Kilo, Mega, Giga, and more

Defining terms is important, as is recognising their standard notations. A **bit** is a single binary value and a **byte** is a group of 8 bits. Traditionally, a **kilobyte** (KB) is 1024 bytes, a **megabyte** (MB) is 1024x1024 bytes, and a **gigabyte** (GB) is 1024 megabytes. Computer systems even have capacities as high as 1024 gigabytes: this is known as a **terabyte** (TB). Exceptionally, and in particular for super-computer systems, we may have many hundreds or thousands of terabytes of data memory, or storage. A further subtlety in notation, to be well noted, is the use of **MB** to denote **megabytes**, and **Mb** to denote **megabits**. Read and write these units with care whenever possible. Generally in this material we will use megabytes, megabits, or perhaps Mbits for brevity, to avoid opportunities for such misunderstandings whilst we learn together.

This terminology can cause initial confusion for non-computer scientist practitioners, since the standard scientific units of kilo, mega, giga, etc are more generally based upon **decimal** powers 10^3, 10^6, 10^9, etc. This has become more of a problem since the introduction of new base-10

definitions of Kilo, Mega, Giga, etc[2], in preference to the still widely used 'deprecated' binary definitions. This new standard has not yet become universally prevalent, and much of the computer systems world, including older documentation and huge volumes of legacy software and hardware, is still based upon the older definitions.

[2] In particular the definitions made under IEC 80000, where kilo=10^3 bytes, mega=10^6 bytes, giga=10^9 bytes, and so on.

However, the **binary** version of these units correspond to 2^{10}, 2^{20}, 2^{30}, giving slightly different values. A binary megabyte for example is 1,048,576 bytes, and a million bytes is 95.3 *binary* Megabytes[3]. For a more complete picture of new and old standards, see TUTORIAL **??**.

[3] To convert from millions of bits or bytes into binary megabytes, simply divide the subject figure by 1.024 twice, and for billions of bytes to gigabytes, divide three times.

IMPORTANT !

Note that in this material we default to using the well established approach of using binary megabytes, gigabytes, etc, unless stated otherwise. This is because they are more relevant to physical hardware structures such as memory devices, disk storage sectors, and so on, as we will find in later chapters. When base-10 units are appropriate, for instance where we discuss bus/network data rates, terms such as 'millions of bytes per second' will be used where possible.

Where you find other texts or online sources referring to kibi, mebi and gibi, these are the modern reference terms for binary kilo, mega, and giga quantities used in this material.

2.3 A conceptual computer system

The basic model of a computer system is traditionally represented by a block diagram showing the most important components. Of course every real-world system is different, and may have additions, novel variations to the normal practice, etc. This is especially so when computers are **application-specific** rather than **general-purpose**. General purpose computers are designed to perform adequately across a wide range of uses, but they are rarely optimal in any of those cases. Compromises are made to make the computer flexible and widely applicable, and potentially more cost effective. Where a computer system needs to be highly efficient at one well defined task, then application-specific computer designs can achieve very high performance in that one niche area.

So what does a general purpose computer consist of? Let us take a look, starting with the block diagram model given in Figure 2.1. A very simple computer system, as a first-order model, might be presented thus. It

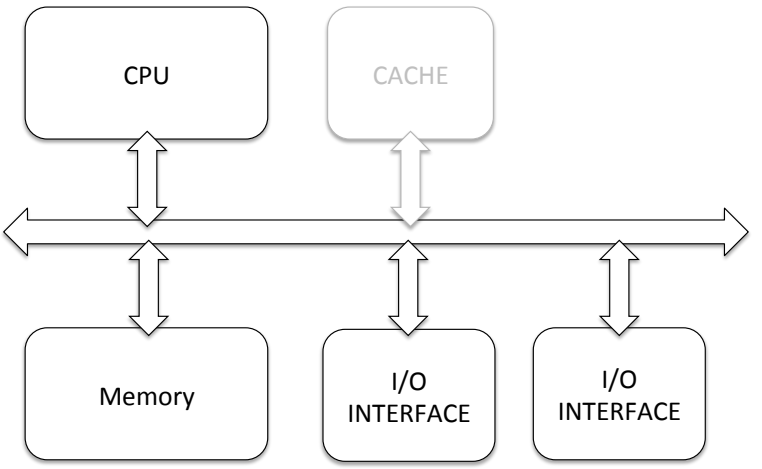


Figure 2.1: Block diagram of a general purpose computer system. Diagram shows CPU: Central Processing Unit (top left), cache memory unit (top right), Main memory (bottom left), and several IO interfaces (bottom right).

consists of a **Central Processing Unit** (**CPU**), often simply referred to these days as a **processor**, sharing a connection to a **system bus** with a number of additional modules. These include the essential **memory** and **input/output interface(s)**.

2.3.1 von Neumann model

As far back as 1945, the computer scientist **John von Neumann** defined a computer architecture as consisting of a CPU, a memory, and input/output devices. This has been the most frequently used model for computer systems ever since. It is amazing to think that this is a definition that is rapidly approaching its 80th year. Connecting the individual modules of our computer system is a **System bus**. This may well be a new concept for the reader, so let us take a moment to define what a bus is before we continue:

Definition 2.3.1 ***Bus: Simple Definition***
*A **bus** generally consists of a group of wires, working together to achieve a communication of information between several devices in a coordinated fashion.*

Busses allow many devices to share one communication route for information. However, it is important to note that only one device can control the bus at a time (the **bus master**). This is a limitation that we need to

be aware of, as it has significant impact for performance. A system bus is thus just a particular type of bus, in which some of the wires designate an **address**, some designate **data**, and some designate essential **control signals**. We will find out what these groups of wires represent later in detail.

The system bus allows the CPU to communicate with other essential modules. These include: the memory, where data and instructions are stored, and suitable electronic circuits to permit IO devices to be connected to the system. An **IO device** is an input/output device, for example, a **keyboard** is an input device, and a **video screen** is an output device. We will study these devices and their related **electronic interfaces** in more detail in a later chapter.

There is one more module in our diagram: the **cache**. Cache is a special kind of memory, which is much faster than normal memory. This was not explicitly defined in von Neumann's original model, but is worth mentioning at this point in passing. We will see later that this special fast memory allows a computer system to boost its performance by exploiting the way programs and data are accessed in the memory space the computer uses.

So, we can say at this stage that, in the simplest computer system, all devices connect to a system bus, the CPU is always the bus master, and all devices appear as if they are part of the visible memory of the computer system. This allows for very straightforward programming models to access devices and modules within the system, though not necessarily the most optimal performance.

If we analyse any modern computing system, it will have at least the same minimum features as our simple **von Neumann model**. A system without a CPU cannot do computations, a system without memory cannot store a program, and a computer that has no input/output capability cannot accept data to process or give out its results. So, all of these components are regarded as essential building blocks.

2.3.2 The von Neumann Bottleneck

Although the von Neumann model is perhaps one of the most successful concepts in computer design, it does have some drawbacks. Consider what happens when the CPU is busy **fetching** an **instruction** from memory. All of the bus wires will have signals present, and therefore they cannot be used by another device in the system.

If a CPU wishes to fetch data from memory, it cannot also fetch an instruction from memory at the same time. Likewise, if an IO device was capable of accessing memory directly[4] then it would similarly be prevented from doing so whenever the CPU was already accessing memory. This constraint, due to many devices wanting to use one shared bus, is known as the **von Neumann bottleneck**.

[4] This is indeed possible in some systems. The concept is known as direct memory access, or DMA. DMA capable IO devices are able to take over bus control and act like a master bus controller in order to coordinate data transfers between two or more bus entities.

In an attempt to alleviate the von Neumann bottleneck, designers invented an alternative architecture, known as the **Harvard Architecture**. In this system, there are two busses, one for program memory, and a second bus for data memory and IO. This permits instructions and data to be accessed **simultaneously**, permitting a speed gain.

However, in a modern computer system there are actually a wide variety of options for multiple bus systems, and covering all of them at this point may be a distraction.

There are many more fundamental questions we have yet to ask: what exactly is a CPU?, how does memory work?, how does cache give a performance boost?, how do IO devices connect to computers?, and how do they behave? We will be turning our attention to these questions in the following chapters.

2.4 The software viewpoint

As we discovered earlier, the concept of software covers a very diverse set of possibilities. Understanding some of the more fundamental concepts will help us in our journey toward a fuller understanding. In this text we will explore many, if not all aspects of the major software elements of a computer system. But let us start by familiarising ourselves with some basics.

Incredibly, some of the earliest examples of software, if we stretch the definition to its limit, include **punched cards** used in automated weaving machines, dating back to the 1700's. The **Jacquard loom**, used such cards to designate the weave pattern for each line of the cloth being woven. In effect, each set of holes punched in a card roll acted as an **instruction** to a weaving machine. The punched card rolls allowed a rather complex set of weaving instructions to be performed by an automated machine, resulting in accurate reproduction of a complex pattern. There is a lot to see here in a modern definition of software, where we would arguably accept that software is a series of instructions relating to a complex set of actions to be performed by a digital circuit (the

CPU), and rather than manipulating threads of cotton, we manipulate data values.

As we have noted, software can exist in many forms. At the simplest level, we can divide software into **source code**, which is in some way intelligible to a human, and **machine code**, which is nothing more than a series of binary numbers, and rarely understood by visual inspection by a programmer. The first systems had to be **programmed** in machine code, because nothing else existed. For most purposes this was a laborious and undesirable task, and made the creation of highly complex programs quite difficult. As a result, programming techniques had to evolve.

Programming languages began to emerge in the late 1950s and evolved rapidly over the next 30 years. Early languages such as **ALGOL** and **COBOL** are remembered fondly by ageing programmers, but the diversity of languages available now is huge by comparison[5]. Most modern computing languages, such as **Java**, **C**, **Python**, and so on, are founded upon principles of these first languages in some form or another.

[5] It is hard to know if there is a definitive answer to this. Some have suggested there are well over 500 languages in existence that have been used widely enough to be considered as candidates, but of course this number increases every year.

In modern computing situations we use two types of so-called **high level languages** (**HLLs**). The first type performs **compilation**, using a software utility called a **compiler**. A compiler automatically converts high level language statements into sequences of low-level machine code. It is the machine code that is executed on the computer, not the high level language statements typed in by the programmer. If the compiler is very good, then it will generate very efficient code.

A second approach is to perform **interpreted execution**. An **interpreter** (another kind of utility program) takes the programmer's program statements, or a condensed version of them, and reads through that sequence of high level language statements. It then executes a predefined piece of low level code for each statement it recognises.

Meanwhile, we still have a third option: to write programs in low level machine code, using a textual representation of each instruction, known as **assembly language**.

But didn't we just say that low level programming is laborious and usually undesirable? This is true, but there are situations where the very compact and highly efficient programming style of assembly language is still useful, especially where very low level activities such as directly operating IO device signals at very high speeds is required.

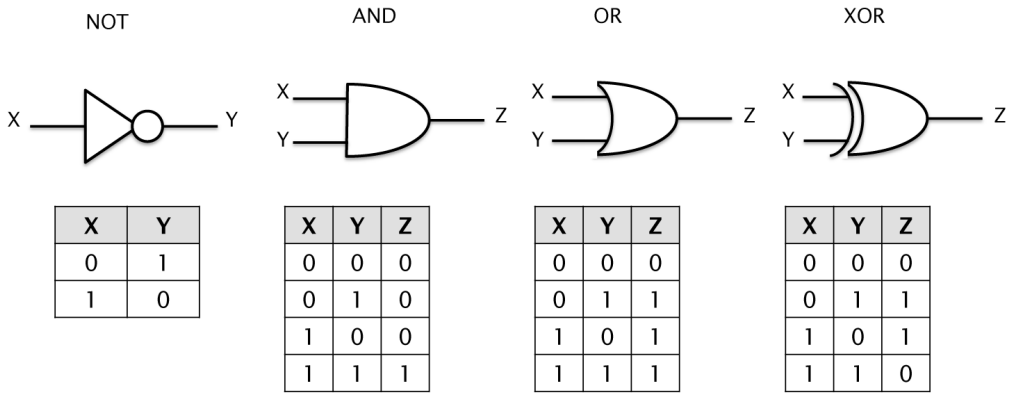
2.5 Putting electronics in the picture

In this text, we try to stay away from deep electronics knowledge. A full understanding of the physics of **transistors**, **semiconductors**, **logic gates**, and the challenges of designing very high speed circuits, would require significant engineering knowledge. However, we cannot have a complete picture of a computer system without some basic understanding of how digital circuits achieve computations. Let us therefore review a few key concepts here too, in preparation for our journey of discovery.

[6] Transistors have many properties, and are useful for amplifying and manipulating electronic signals of all kinds. When used in one particular mode, they operate in a way that approximates an on-off switching function.

- **Transistors:** Invented in the late 1950's, a transistor is a kind of switch, at least as far as digital electronics is concerned[6], and when properly used they can operate in a binary signalling regime (on and off, one and zero).
- **Transistor Operation:** A simplified description of transistor behaviour in switching mode might be as follows: A typical transistor uses a control signal (**the Gate**) to control the flow of a signal from the input (**the Source**) to the output (**the Drain**). However, multiple transistors can be combined to create logical switching functions.
- **Logical Operation:** Imagine that our Gate control signal can switch from zero to one, and by combining two transistors in the right way, we can create a switching arrangement that produces corresponding outputs one and zero. That is to say, the output is the opposite of the input. This is known as an **inverter**, or a logical NOT operation: the simplest logic gate. The output is NOT the input, if you wish to think of it in those terms.
- **AND and OR:** We can also combine transistors in such a way that they create gates with two inputs. Then the circuit can be arranged such that a signal output can only be binary one when both inputs are one (the **AND gate** function, where input-a AND input-b must both be one at the same time to generate an output of one). Likewise, a slightly different arrangement causes the circuit to generate a one if either input is one (generates one if input-a **OR** input-b are one), creating an OR gate.

What we have just described is the behaviour of three very simple **logic functions** with the definitions of AND, OR, and NOT. These circuits are referred to as **logic gates**. So we typically talk about the **AND gate**, the **OR gate**, and the **NOT gate** (or just an inverter). A further gate, the **XOR gate** performs an **exclusive-or** function, where an output is one only when any one (but not both) inputs are one. These components have diagrammatic symbols, and their behaviours can be defined by tables of



X	Y
0	1
1	0

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.2: Standard Logic Gates, symbols, and input/output 'truth tables'. Diagram shows NOT gate (Left), XOR gate (right), AND gate (centre left) and OR gate (centre right). Each gate has one or more inputs, labelled x,y, etc., and a single output, z. Output signal behaviour is defined by the combination of input(s) and the type of gate.

inputs and outputs known as **truth tables**. Figure 2.2 shows the symbols and truth tables for our four logic gates.

Any of these gates can be combined with an inverter to give an opposite function. AND gate becomes **NAND gate**, OR gate becomes **NOR gate**, XOR gate becomes **NEXOR gate**[7], and an inverter becomes a **buffer** (a gate that has no logical effect[8]). Without going into too much detail, mixtures of these eight logic gates can also be combined into more complex circuits, including circuits that add binary bits together (arithmetic), or perform other mathematical operations such as multiplying, comparing, and so on. These circuits are called **combinational circuits**.

Meanwhile, logic gates can also be used to build **counters** and other **control circuits** to permit a computer chip to control sequences of events. These circuits require a time pulse to control their repetitive behaviour: this is the **clock signal**. A clock is simply a repetitive on-off-on-off signal, like a clock tick-tock-tick-tock. Because these circuits use a clock to synchronise their behaviour, they are known as **synchronous circuits**.

Finally, by using logic gates in a particular way, such that some of their outputs feed back as input signals to the same circuit, it is possible to have a signal circulating in a loop as part of the electronic behaviour of the circuit. This allows information to be retained, and the result is sometimes referred to as a **data latch**, or a **flip-flop**, which can store a zero or one. A single bit of storage is not particularly useful, but if we combine eight latches, for example, then we have created an **8-bit**

[7] NEXOR also often referred to as XNOR

[8] This may seem pointless, but it permits certain capabilities, such as delaying a signal, among a few other purposes.

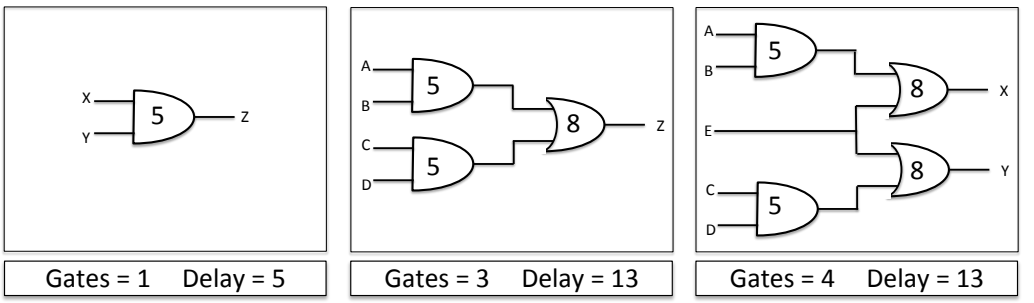


Figure 2.3: Simple Examples of Speed and Complexity. Figure (a) shows a simple fast circuit. Figure (b) shows a more complex and slower circuit. Figure (c) even more complex, but **not** slower. The numbers indicate the signal delay caused by each gate. For instance, '5' might represent 5 nanoseconds.

register, a component that can store an 8-bit binary number: one byte of data. This binary storage principle is the basic embodiment of **computer memory**. At this stage, if you are not familiar with the idea of **Binary** and **Hexadecimal** number systems, then this is something to address now. Do some research, or read the provided material in Tutorial **??**.

Hopefully from this description it should be apparent that transistors allow us to create logic gates, and logic gates allow us to create circuits that can perform arithmetic, store information, and control sequences of events. These will all appear to be very relevant in the next chapter.

[9] The idea that more gates means slower circuits is based upon the idea that logic gates are cascaded together, such that their delays accumulate from end to end. Actually, some gates can operate in tandem (side by side) so this is not an entirely concrete truth, though in general terms it is likely.

2.5.1 Speed versus complexity

Nothing is free. All of these logic gates and logic circuits we are talking about **consume power** and **generate heat**. The more gates that are combined, the more power is consumed and the slower the circuit becomes[9].

Slower circuits are revealed when we look at the **clock frequency** of a system. A clock should pulse as fast as the circuit can tolerate, to make it work as fast as it can. But if the circuit is slow then we must also have a slower clock. This is one reason why so many different CPUs have very different clock frequencies. What we have just described is the dilemma of **speed versus complexity**: more complex circuits might be better at performing a particular function. But, if they are also slower, then does this actually mean they do any more useful work?

Figure 2.3 illustrates, in a very simplistic way, the main concepts of speed versus complexity. Consider Figure 2.3(a), where a single gate is undoubtedly a simple circuit. Suppose that this logic gate has a **signal delay** of 5ns, then we can say that the complexity is low and speed is high[10] . Now consider Figure 2.3(b) which has three gates, so the complexity has increased. Meanwhile, any path through the circuit from input to output requires 13ns. In other words, it is slower. Finally, Figure 2.3(c) has four gates (even more complex) but, surprisingly, it still has a delay of only 13ns (more complex but *not* slower).

[10] When we talk about **speed**, we think about **frequency** (MHz, GHz, etc.), and **delay** is expressed as **time** (microseconds, nanoseconds, etc.). These are related, since **f=1/t** and **t=1/f**. Therefore, if t=5ns, f=200MHz, compared to f=77mHz if t=13ns.

Circuit (b) is more complex, and slower, than circuit (a). Intuitively you might also expect circuit (c) to have increased delay compared to (b). After all, circuit (c) is certainly more complex than circuit (b). Whilst it is true that increased complexity often, even typically, causes increased delay, it is sometimes possible to have more gates but not slow down the circuitry. Indeed, in circuit (c) what we have done is use parallelism, because there are actually two separate circuits operating side by side. This is a key principle underlying the idea of multicore processors, a topic we will explore in later chapters.

Speed versus complexity is one of the big arguments in computer architecture, and one that has not yet reached a conclusion. It has had a major influence on processor design, not the least of which was the **RISC versus CISC** debate. During the 1980's, processor designs were heading toward increasing complexity, following the concept of **Complex Instruction Set Computers** (**CISC**). However, it was observed that many of these instructions were rarely used effectively. This spurred a new concept: **Reduced Instruction Set Computers** (**RISC**), where a streamlined but very efficient set of instructions was implemented in the processor design, meaning less complex circuits and higher clock speeds.

The same argument has influenced the idea of **multicore processors**: the idea of putting many CPUs on a single chip, to increase performance. Instead of one processor with ten times the complexity, why not have ten processors just like the ones you already have? (ten circuits operating in parallel). As the number of transistors available on a chip at a given price[11] continues to increase, these arguments will continue to evolve too.

[11] transistor count is one measure, another term often used is 'silicon real-estate', and silicon real-estate per dollar is therefore a measure of cost versus complexity.

2.6 Summary

We have just explored a few basic ideas relating to computer systems. We have considered what the scope of hardware and software encompasses, and we have seen that those two words innocently hide a very large field of knowledge. We have also learned some basic concepts of computer systems architecture, explored some simple aspects of digital electronic circuits, and defined some key terms of reference.

With our beginner's approach, we should not attempt to cover all of these aspects in expert depth, and that indeed is not our purpose. However, in the following chapters we will expand many of these ideas to gain deeper knowledge, hopefully without information overload!

2.7 Terminology used in this chapter

Address	Application-specific
Applications	Assembler
Assembly language	BIOS
Cache	Clock signal
Combinational circuit	Compiler
Control signal	CPU
Data	Data capacity
Data latch	Data rate
Firmware	Flip-flop
Frequency	General-purpose
Hardware	High level language
IO device	Interpreter
Inverter	Logic gate
Logical AND	Logical NOT
Logical OR	Logical XOR
Machine code	Operating system
Peripherals	Power
Register	Software
Source code	Storage element
Synchronous circuit	System bus
Transistors	Truth table
Utilities	Virus

These terms are defined in the glossary, Appendix A.1.