CHAPTER 9

The von-Neumann Computer

In this chapter we will present how to electronically represent data and how to process it. In other words, we will get to know the electronic parts and building blocks of a machine that is able to store data in form of electrons or magnetism. We will learn the functionality of parts and components and how they manipulate the data according to an algorithm. The instructions for the algorithms are equally stored as a sequence of data items that are interpreted as machine instructions. In fact, the interpretation of data items defines whether the data represents numbers, characters, a picture (see chapter 10) or machine instructions.

A machine with this capability is commonly called **computer**.

Historically (see section 1.9) the first attempts to build a computer used mechanical concepts with toothed wheels for number representation and gears to add or multiply numbers. Even when the mechanical concept worked in principle and the challenge of mechanical precision could be mastered today, moving masses limits the processing speed and wear reduces the lifetime of such a machine.

Using electrons or magnetism instead of mechanical parts is many orders of magnitudes faster. The energy necessary for moving electrons or generating magnetism is supplied by electrical current. The switching "wear" generated can easily be compensated by electrical energy.

9.1 A Computer Model

In this section we present a simplified view of the computer architecture and in the following section we start on the electrical level to describe the parts and functional units of this architecture.

The principal architectural computer model is similar to the workbench of a clerk. It consists of an *input* and an *output basket* where the tasks are queued up for processing resp. for retrieval of the results, a *notebook* for writing down intermediate and final results, a *calculator* for doing arithmetic operations, and a *clerk* controlling all work steps.

The clerk is fetching a new task from the input basket, typing numbers and the operators into the calculator, writing down intermediate results on the

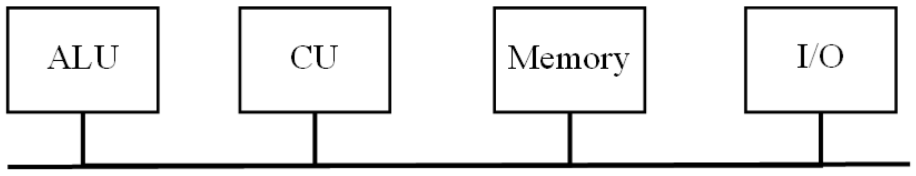


Figure 9.1 A todays von-Neumann computer architecture

notebook, and finally on completion writing the result on a paper and depose it to the output basket for retrieval.

Using the corresponding components for an electronic computer system we basically get the architecture proposed by *John von Neumann* in the year 1945 as shown in Figure 1.6. It consists of an *Input/Output (I/O)* unit, an *Arithmetic-Logic Unit (ALU)*, a *Memory* for storing data and program instructions, a *Control Unit (CU)* taking the role of the coordination of all activities, and a *Bus System* for an easy communication.

The I/O unit is responsible for the communication with the "outside world" as reading data from the input devices (keyboard, mouse, hard disc, CD, etc.) and writing data to the output devices (printer, display, disc, memory stick, etc). Without the I/O unit we could not get any results from the computer or observe what it is doing. The CU reads instructions from memory in order to know what to do with the input or data in memory. The ALU performs computations of numbers, string manipulations, and logical decisions according to the algorithm being executed.

If the communication between the components is realized via a bidirectional path linking physically all components simultaneously (*system bus*) instead of many dedicated links, we have the modern variant for the von-Neumann architecture of a typical contemporary computer as in Figure 9.1.

All data transfer between any two units is done via the system bus. The bus by nature is working in a *broadcast* mode, i.e. all devices connected receive all communication. An addressing scheme designates sender and receiver of a communication. Thus, it is always clear which device should read and process the receiving data.

As microprocessor technology advanced the ALU and the CU were put onto one integrated semiconductor circuit (*chip*), called *Central Processing Unit (CPU)*. Figure 1.2 shows the typical architecture and components of a modern PC.

Today, the CPU integrates apart from ALU and CU a special high speed memory, called *cache*, that holds prefetched and intermediate data. If a cache memory is used, the data is transferred from and to memory asynchronously via the cache by ensuring that the data in memory and in cache will always stay in synchronization. This allows full speed processing without waiting for the memory access to complete.

9.2 Binary Signals

The von-Neumann architecture draws a high level picture of the computer. So far, we have no idea of how to materialize data and how to physically process the data in the ALU. We only know that electrons would be a good media to use for its speed and low energy to operate.

The abstract representation of data as a sequence of bits and bytes was introduced in chapter 1. Later, in chapter 10 we will show that all kind of data, even audio and video data, can be coded as binary data. If we manage to assign the bits "0" and "1" a certain electrical state that may solve the materialization problem universally.[1]

Electrons generate a voltage according to its amount. We need to distinguish only two voltage levels, say 0 volt for the bit "0" and 5 volts for the bit "1". The absolute voltage values are not relevant as long as both are clearly distinguishable. We usally speak of "low value" for binary 0 and "high value" for binary 1. We will use electronic memory cells that are able to store many bits (see section 9.6.1) to represent a large amount of (complex) data.

A voltage tends to vanish by "leaking" current. We have to make the bits persistent by supplying electrical power. In fact there exist memories where each bit has to be cyclically refreshed.

The value of a bit may change over time intentionally due to data processing. As example take the number five represented as dual number 101. If we add one to the number five this will change the right most (least significant) bit to 0 and the middle bit to 1.

To change the bit values data needs to be transferred from memory to the ALU for manipulation and stored back to memory. This can be viewed as a kind of communication between the memory and the ALU. On the lowest abstraction level, bit values representing data will be transferred via system bus between any two computer components.

We call a time-varying quantity that is transferred for communication purpose a **signal**. In particular transferring binary values is called a **binary signal**.

Beside the electronic memory we can store bits on magnetic tapes (slow sequential access) or on magnetic discs (fast random access). Each bit will be stored on ferromagnetic material by magnetizing a little area depending on its value. We may assume that the bit 0 is magnetized as having a certain magnetic field direction (north pole of the magnetism) and the bit 1 will be magnetized in opposite direction.

An electromagnet is used to magnetize the ferromagnetic material. The magnetic field of a ferromagnetic material will remain until it will be re-magnetized or de-magnetized (degaussed), thus providing persistently stored data. Figure 9.2 shows a disc with longitudinal bit recording.

The materialization as voltage works fine as long as we can guarantee that the tolerance of the voltage value is less than half of the nominal voltage distance. In this case we can still reliably distinguish between bit 0 and 1.[2] This

[1]In the early days of computing the ENIAC was based on the decimal number system (see 1.9). In this case 10 different electrical states had to be recognised.

[2]The reliability of the data materialization would be even more fragile if we used a decimal system as 10 different voltage levels need to be distinguished

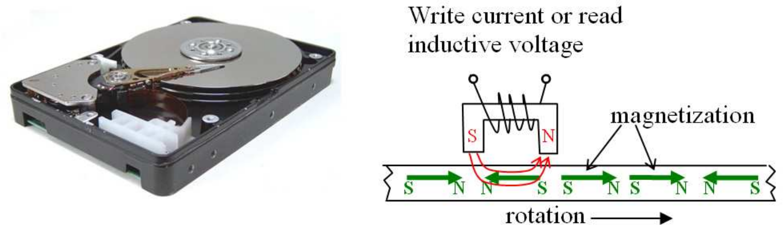


Figure 9.2 Hard disc with longitudinal recording

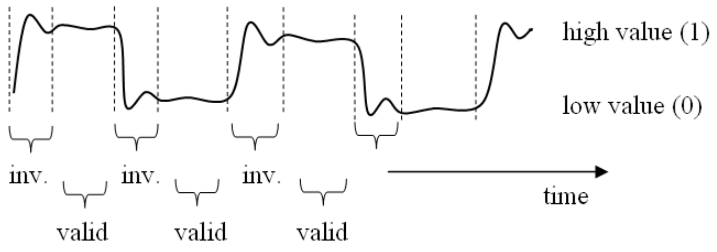


Figure 9.3 Binary signal transition

shows, that the computer is sensitive against voltage variation or drop outs. Therefore stabilized power supplies are used in all computer systems.

A similar argument holds for the magnetic store. If the fading of the magnetic field is less than half of the fully magnetised state, we can still recognize the original bit state. All ferromagnetic storage media are very sensitive to magnetic exposure as this could change the magnetic state and erase the stored bits.

Lets return to the binary signals representing data items transferred to the ALU. During the processing a value will eventually change from high to low value and vice versa.

The change from one voltage level to the other needs some time and will pass the area where there is no interpretation possible. This transient phase is also called invalid or "illegal" state. The transition time and time span where the signal or data is valid is shown in a magnified view in figure 9.3.

Any circuit processing binary (or digital) signals has to know when the signal state is valid and when to change the state. Therefore, two **clock** signals are necessary to indicate the points in time when the circuit should change state and when the data is valid and can be evaluated. All circuits in the computer hardware are *triggered* by these two clocks. The clock frequency is essential for the processing power of a computer. The higher the clock frequency the faster the processing speed.

The clock cycle of a modern personal computer reached three Gigahertz in the year 2007. This speed allows a state change within $\frac{1}{3}\,10^{-9}$ seconds ($\frac{1}{3}$ *ns*). To get an impression of this very short time interval we take the light speed as reference. Within $\frac{1}{3}$ *ns* the light travels only 10 cm. This fact explains why

AND			OR			NOT	
input		output	input		output	input	output
i_1	i_2	o	i_1	i_2	o	i	o
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Figure 9.4 Elementary binary operations a) AND b) OR c) NOT

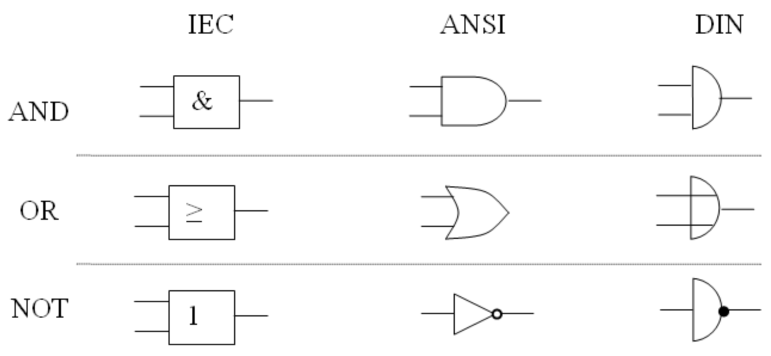


Figure 9.5 Graphical notation for binary operators

miniaturization is not only "nice to have" but is also an essential pre-condition for a high performance CPU.

9.3 Binary Operations

Binary operations are also known under the name *Boolean operations* in honor of the English mathematician George Boole (1815 - 1864) who created the *Boolean algebra (logic)*. In his theory two operations are defined on a set of two values, usually taken to be 0 and 1. This matches nicely with binary data values allowing us to think of the bits 0 and 1. The basic operations are $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT),

The AND operator takes two input values (two bits) and produces one output value (one bit). The output bit is 1 if and only if both input bits are 1 at the same time, else it will be 0. The OR operator has the same input and output parameters (i_1 , i_2, and o in figure 9.3b) but differs in function. The output bit is 0 if and only if all input bits are 0, else the output is 1. The NOT operator has on input and one output parameter. The output bit value is always the negated value of the input.

To define the operations we may list all possible results as our value domain is only {0, 1}. Figure 9.3 shows the function tables for the operations AND, OR, and NOT.

Beside the function table there exists a graphical notation for binary operations. Some examples are given in Figure 9.5.

The main benefit of using binary coding of data is that we can do any kind of manipulation with only three elementary operations: AND, OR, and NOT.

To motivate this proposition we consider first the case of functions with one input and one output. We have only four ($= 2^2$) possibilities: the NOT function which we know already, the identity function IDENT that does not change the input values, the ONE function that produces the output 1 no matter what the input is, and the ZERO function that results into 0 for all input values.

For functions with two input and one output parameters there exist already $8(= 2^3)$ ways of choosing the output. As example, we will only show how to construct an EQUIV operator that results to 1 if and only if both input values are the same. To construct this function we use one AND, two ORs, and one NOT operator. Figure **??** shows how to connect the elementary operators. We will see more examples and their implementations with the just presented elementary operators in Section **??**. In particular the most important operator for arithmetic operations the addition (+) is explained in detail.

To prove, that it is possible to construct an arbitrary binary function $f(i, j)$ based on only AND, OR, and NOT gates we select any input situation that produces a 1 as output. For example assume that $f(1, 0) = 1$, then we construct $f_{10} = i \wedge (\neg j) = AND(i, NOT(j))$. Combining the output via OR gate of all functions f_{ij} producing the output 1 for the input values i and j will yield the function:

$$f(i, j) = OR_{i,j \in \{i,j | f(i,j)=1\}}(f_{ij})$$

This function $f(i, j)$ produces a 1 if and only if any function f_{ij} yields a 1. We have OR-ed all truth table entries that have output 1 as result. All other entries produce 0.

This construction mechanism is extensible to any function of any number of input parameters. Lets demonstrate this with the MAJORITY function with three input parameters.

MAJORITY			
input			output
i	j	k	o
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The resulting function is:

$$\begin{aligned} MAJORITY(i, j, k) &= OR_{i,j,k \in \{i,j,k | i+j+k \geq 2)\}}(f_{ijk}) \\ &= OR(f_{011}, f_{110}, f_{101}, f_{111}) \\ &= f_{011} \vee f_{110} \vee f_{101} \vee f_{111} \\ &= (\neg i \wedge j \wedge k) \vee (i \wedge j \wedge \neg k) \vee \ldots \end{aligned}$$

i_1	i_2	o
0	0	1
0	1	0
1	0	0
1	1	1

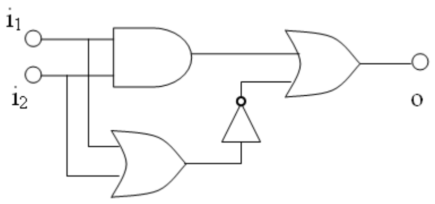


Figure 9.6 Building an equivalence operator EQUIV

To construct the MAJORITY function we only use NOT operation to invert an input value, AND gate to combine all input values that produce the output 1, and OR gates to union all outputs to form the result. This proves the proposition that any binary function can be constructed from the elementary operations.

We are free to interpret the values 0 and 1 as the logical values *false* and *true*. With this association we may read the AND operation as:

false AND false = false
false AND true = false
true AND false = false
true AND true = true

Depending on the interpretation of the two possible states (**low** and **high** value) these voltage values take the meaning of the binary data values **0** and **1** or the logical values **true** and **false**.

In the light of propositional calculus the AND operation takes two propositions each of which can be either true or false. The result of both input propositions is again a proposition that is true only if both inputs are true.

If we have two expressions like "x is even" and "$3 < x$" than the AND operation of both expressions will only be true if each expression is true which would be the case for x = 4, but not for x = 5 or x = 2.

Please note that the OR operation is not an "either or" in the common sense, as the result of two true values yields again true (see OR function in Figure 9.3).

This interpretation of the binary operations allows a computer to do propositional calculus. The computer may do logic operations that conclude new propositions from existing ones. For example, from the propositions "When it is raining, the street will be wet" and "The street is dry (not wet)" we can conclude "It is not raining". This is the starting point for reasoning with a computer. As we can use the same hardware for arithmetic and logic operations, the hardware unit for doing this processing is called Arithmetic-Logic-Unit (ALU).

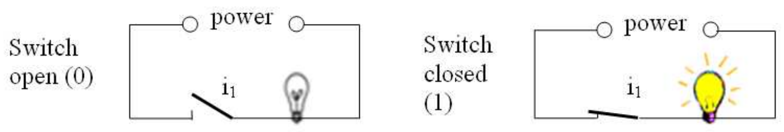


Figure 9.7 A simple switch used to operate a light bulb

9.4 Binary Circuits

In previous sections we learned that electricity can be used to materialize binary values and the kind of possible operations on it. Now it is time to tell *how* to actually build electronic circuits that hold the binary values and manipulate them.

As there are only two voltage values (low and high) to distinguish a simple switch is sufficient as basic building element to construct binary circuits. However, the switches in a computer system will not be mechanical, but electronic.

Assume that we have a power source (e.g. battery or AC line). If the switch is *closed*, the power resp. voltage (high level, "1") is output to a consumer. If the consumer is a light bulb it will be *on* or if it is a computing device like ALU it receives the binary value 1. If the switch is *open*, no voltage (low level, "0") is output to a consumer. In this situation the light bulb will be *off* or the ALU receives the binary value 0. Both situations for the light bulb are shown in the circuit diagram of Figure 9.7.

The functionality is not changed If the mechanical switch is replaced by a electronic switch. Instead of mechanically flipping the switch it will be operated by an electrical signal. This is great as we can change the state of the switch by the same means as the output generated by the switch: an electrical voltage value.

An electronic switch in its simplest form consists of only one *transistor*. A transistor is a semiconductor device with three connectors (*pins*).

The voltage u_G at the input pin (*gate*, *base*) will control the voltage at the output pin (*drain*, *collector*). The third pin, the *source* or *emitter*, is the reference point for measuring the voltage at the input and output pins.

The gate voltage creates an electric field that increases the conductivity between source and drain pin. If there is power supplied to the drain via a resistor, this will result in a current and the voltage at drain will drop close to zero. If there is no input voltage u_G than the electrical resistance between drain and source is high and so will be drain voltage u_D too.

As result a low input voltage results in a high output voltage, and a high input voltage effectuates a low drain voltage. We may say that u_G appears as amplified and inverted voltage u_D.

The transistor is used either as amplifier or as electronic switch, for both high power applications including switched-mode power supplies and low power applications such as logic circuits, called *gates*.

There exist two main technologies to manufacture a transistor. The bipolar transistor technology is based on a junction created by two differently doped (p-type, n-type) silicon crystals that control the amplification effect. In field effect transistor (*FET*) technology the junction is realized by a thin insulation layer.

a)

U_G	U_D
0	1
1	0

b)

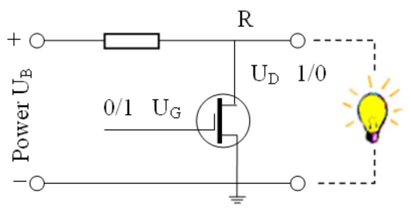


Figure 9.8 Transistor switch using a Field Effect Transistor (FET)

This enables an amplifier with very low residual control current. Colloquially speaking, the FET is controlled without any power.

The switch circuit will be used to construct the elementary binary operations AND, OR, and NOT.

9.4.1 NOT circuit

We have already seen in Figure 9.8 that a transistor can be used to produce in a natural way a negated input signal. A low resp. high voltage u_G produces the opposite voltage u_D at the drain. Both voltages are measured against the source pin.

9.4.2 AND circuit

If we use two switches in serial the result is that both switches need to be closed to make the current flow. If no switch or only one switch is closed, no current will flow. Using two transistors in serial produces a NAND circuit. That is an negated AND with the following truth table:

input		NAND	AND
i_1	i_2	o	o
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Negating the NAND produces the AND logic. Figure 9.9 shows different schematic drawings for the AND circuit.

9.4.3 OR circuit

If we use two switches in parallel the result is that any closed switch will make the current flow. Only if no switch is closed, no current will flow. Using two transistors in parallel produces a NOR circuit. That is an negated OR with the following *truth table*:

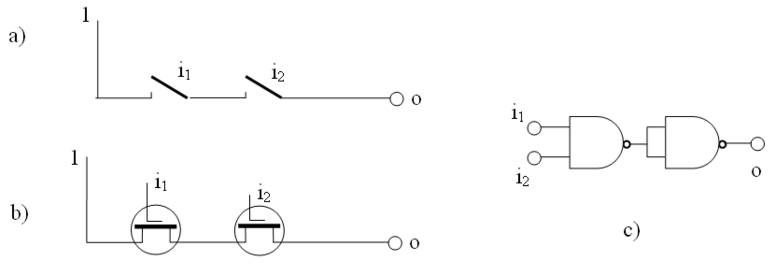


Figure 9.9 Schematic drawings for AND circuits: a) mechanical, b) electronical, c) NAND components

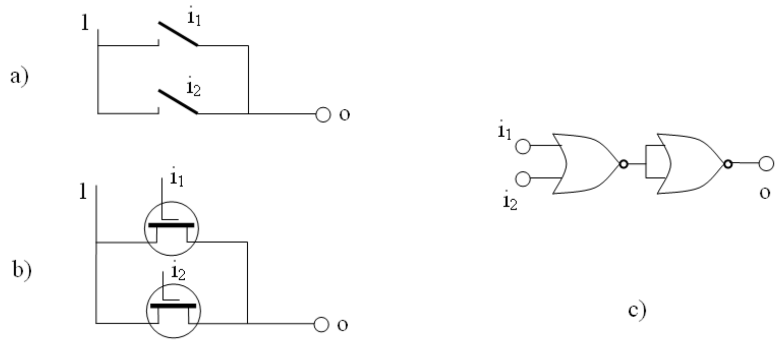


Figure 9.10 Schematic drawings for OR circuits: a) mechanical, b) electronical, c) NOR components

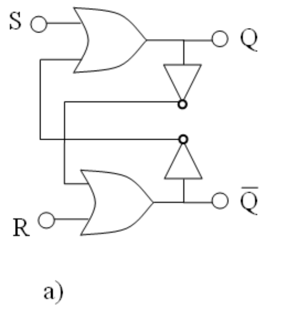
input		NOR	OR
i_1	i_2	o	o
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

Negating the NOR produces the OR logic. Figure 9.10 shows different schematic drawings for the OR circuit.

In section 9.3 we made plausible that we can construct any binary logic or arithmetic operation by using AND, OR, and NOT operators. Next we give some examples relevant for arithmetic operations.

9.5 Arithmetic Circuits

In this section we present the circuits necessary to do arithmetic calculations. Many calculations consist of more than two operands or involve many complex operations. In this case the calculations are done step by step, e.g. we compute the sum of three numbers a, b, c as $(a+b)+c$ which means that the intermediate result of $a+b$ needs to be temporarily saved and added to c. Therefore we



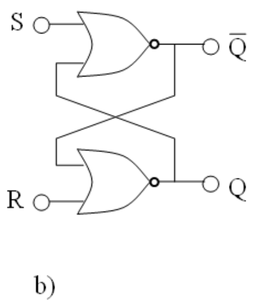


Figure 9.11 One-bit memory (flip flop) a) using OR and NOT circuits b) using NAND circuits

need to store intermediate results. This is why we show how to build a storage element first.

9.5.1 Flip Flop

The most simple storage element is a one-bit memory which we will call a *flip flop* to indicate that it has exactly two possible states, representing the binary values 0 and 1. To produce this storage element, we use two OR and two NOT circuits and wire it in the way shown in Figure 9.11. The trick is to feed back the output Q to one input pin of the first OR (OR_1) circuit. Likewise, the output $\overline{Q}$ is fed back to one input pin of the second OR (OR_2) circuit.

If both input pins R and S are receiving 0 value the circuit remains in any one of two possible output states: the output Q will show either 0 or 1.

Lets assume $Q = 1$, then OR_1 produces $o_1 = 1$ and the NOT results into $\overline{Q} = 0$. Both entries of OR_2 are 0 which yields $o_2 = 0$ and the negation results into $Q = 1$, proving that the circuits remains unchanged. Analog arguments for the situation with $Q = 0$ prove the opposite state. Both states are stable. The feedback of the output Q and its inverse $\overline{Q}$ lock the present state to persist as long as the inputs are 0.

Lets assume that $Q = 0$ and we want to set the output Q to 1. This is done when input $S = 1$ and input $R = 0$. Because of $S = 1$ we get $o_1 = 1$ and $\overline{Q} = 0$. The input lines of OR_2 are all 0 which is responsible for $o_2 = 0$ and $Q = 1$.

If we want to reset Q to 0, we use the inputs $S = 0$ and $R = 1$. Following analog arguments as for setting Q we see that in fact the result will be $Q = 0$ and $\overline{Q} = 1$.

Instead of using OR and NOT circuits it is technically simpler to use NOR gates to construct a flip flop (see Figure **??**. It can be easily proved that NAND and NOR gates can be used to produce AND, OR, and NOT gates because the following equations hold:

AND = NOT(NOT(AND)) = NOT(NAND)
OR = NOT(NOT(OR)) = NOT(NOR)
NOT = NOT(AND*) = NAND*
[* (N)AND gates with both inputs connected]

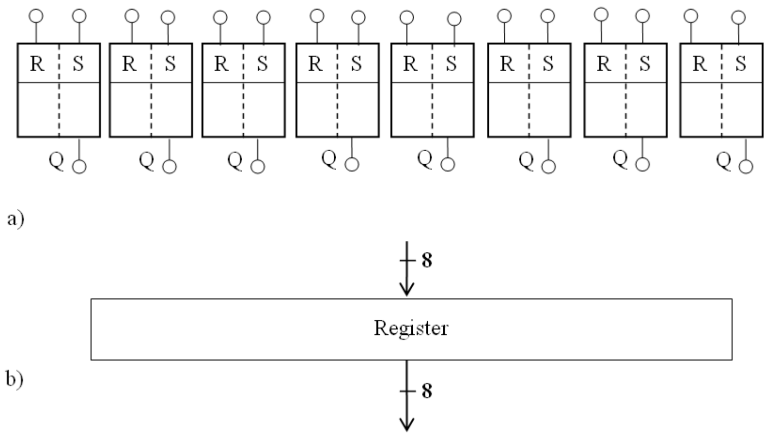


Figure 9.12 An eight bit (byte) register a) using 8 flip flops b) schematic symbol

9.5.2 Register

A *register* is a storage device for logically related bits. These bits may represent a character, a number, a command, or even a more complex data structure. The number of bits contained in a register is usually called a **machine word**. Depending on the processor the "length" of a machine word is usually 8, 16, 32, or 64 bits. Each bit is stored with a flip flop, i.e. the register is built of flip flops.

The register is loaded with a machine word from the input lines by *setting* the S input pin to 1. The register is reset to 0 (*cleared*) by setting the R input to 1.

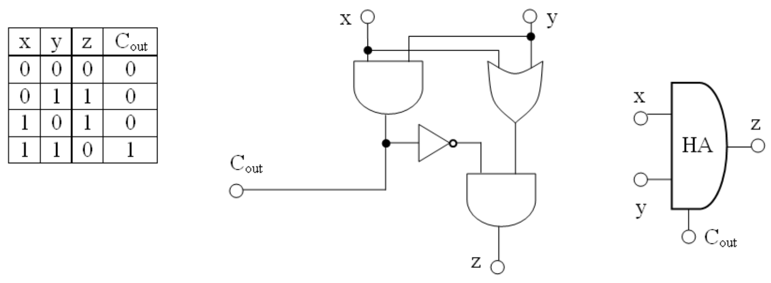
A *shift register* is a special register where the output bit o_i is linked to the input bit i_{i+1}. When the left shift input line is activated (value 1) the bits of the register are shifted one position to the left. The leftmost bit (*most significant bit* is lost and replaced by its neighbor to the right, and so on. The rightmost bit (*least significant bit* - after left shifting - is replaced by the value 0. If the content of the register is interpreted as a binary number this left shift results in multiplying the value by 2. If the lost leftmost bit was 1 this indicates that an arithmetic overflow occurred.

9.5.3 Half Adder

The most important arithmetic operation is the addition (+). Other numeric operations like subtraction or multiplication can be attributed to add a negative number or to repeated addition.

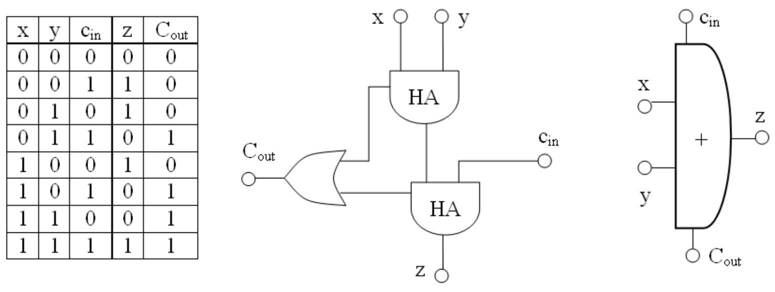
Consider the addition of two single binary digits. The truth table for the addition of $x + y$ with result z and carry c is shown in Figure 9.13a. We construct the carry bit by using an AND gate and for the result we take a negated EQUIV gate. Figure **??** already showed how to build an EQUIV gate of elementary AND, OR, and NOT logic. If we start with this circuit, negate the result and simplify the circuit we obtain the circuit of Figure 9.13b.

Lets explain the function of the adder in case of $x = 1$ and $y = 1$. As both entries are 1 the AND gate produces 1 at the carry output C. The OR gate



x	y	z	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 9.13 Half adder a) truth table b) using AND, OR, NOT c) schematic symbol



x	y	c_{in}	z	C_{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

Figure 9.14 Full adder a) truth table b) using half adder c) schematic symbol

produces 1 as output because the input is 1. The result z is the negated carry (0) and AND-ed with the OR output (1) which yields 0.

The other situations are left to the reader to check.

9.5.4 Full Adder

For a complete addition of a digit within a binary number of multiple digits we have to consider the carry bit from the previous addition. This can be done after adding x to y by adding the carry bit c to this result. We use two half adder and wire them as in Figure 9.14b. The circuit has now two carry lines: the input carry c and the output carry C.

In order to add two numbers of n bits each, n full adders are connected together by forwarding the carry output to the carry input of the next full adder, thereby forming a full adder chain. The last carry output indicates an arithmetic overflow.

9.5.5 Accumulator

To add two numbers we have to connect the output of a n-bit full adder (+) to a register (AC), called *accumulator* in the way shown in Figure 9.15. The first number is stored in the accumulator and the second is taken from memory and input to the full adder chain. The number in the accumulator is feed back to the second entry of the adder chain. When both numbers to add are present at the adder input, a clock signal (please see Section 9.2) triggers the addition. The result is stored in the accumulator with the next clock signal. This means

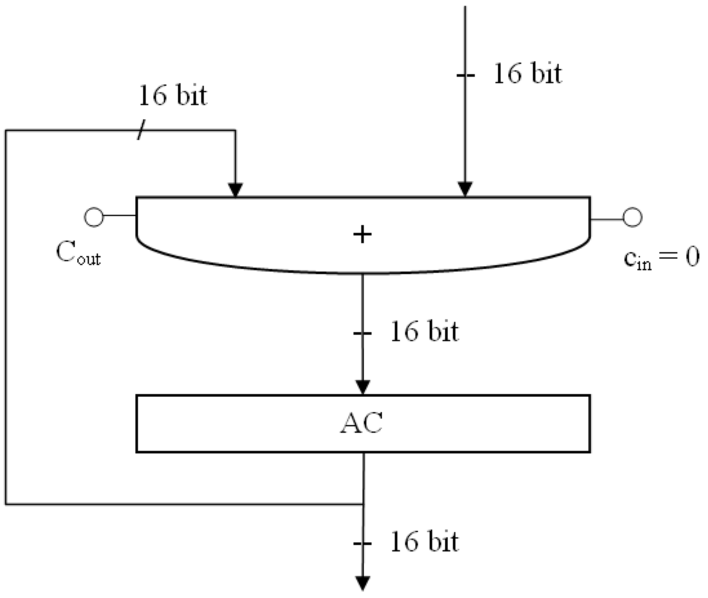


Figure 9.15 Schematic circuit of an accumulator

that an arithmetic unit using an accumulator circuit needs two clock signals to add two numbers. If the clock cycle is 5 ns (= $5\,10^{-9}$ sec) then an addition will be completed in 10 ns.

We can continue to add more numbers from memory to the value of the accumulator by repeating the operations.

To add for example $a + b + c$ the following operation steps are necessary:

Step	Operation	Comment
1	AC := 0	(clear AC)
2	AC := AC + a	(result AC = a)
3	AC := AC + b	(result AC = a + b)
4	AC := AC + c	(result AC = a + b + c)

The demonstration shows clearly why the register AC is called accumulator: it collects the (intermediate) results of the additions.

9.5.6 Encoder

An **encoder** is a device used to change input data into an output code. The code is usually digital but not necessarily. In case of digital input data this would mean that a encoder is a manipulation logic circuit that takes multiple input values that converts the input into coded output values, where the input and output values are different.

The purposes of encoding is to transform the data into a suitable form for transmission or processing. For transmission a favorable code includes compression, encryption, and fault tolerance. For processing the code should

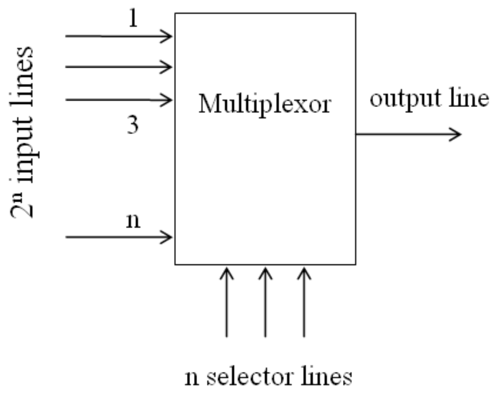


Figure 9.16 Block diagram of a multiplexor

be easy and fast to manipulate. The encoding is usually done by means of a programmed algorithm, especially if the data is digital.

A special kind of encoder is a **multiplexor**. The multiplexor combines the input from different data sources to one output line by encoding the data and serially adding the code from each input line to the output.

The multiplexor has 2^N input lines, N selector lines, and one output line. Depending on the value of the selector lines the appropriate input data is coded and passed to the output line. In the simplest form, the coding is identical to the input value. In this case the selected input line value is simply routed to the output. If the selector lines change the value, the value of a different input line is output. Selecting each input line in turn results in a serial output of all input values. This kind of operation is called *multiplexing* which explains the name of the circuit.

9.5.7 Decoder

A decoder is a device which reverses the work of an encoder. Undoing the encoding retrieves the original information.

In digital electronics this would mean that a decoder has N input lines and 2^N output lines. The input is interpreted as numerical value, the corresponding output line gets a signal (binary 1) while all other output lines are deactivated (binary 0).

A **demultiplexor** is a special kind of decoder. I takes $N + 1$ input lines and generates as signal at one of 2^N output lines. Figure 9.17 shows how an application of a multiplexor and demultiplexor can reduce the number of transmission channels resp. lines.

9.6 Computer Components

The von-Neumann architecture defines five functional units (see Figure 1.6) for a digital computer. Today, in a modular computer system these units are part of the physical components and printed circuit boards. The *main board* of a personal computer (PC) for instance contains the *CPU*, *memory*, a *bus system*

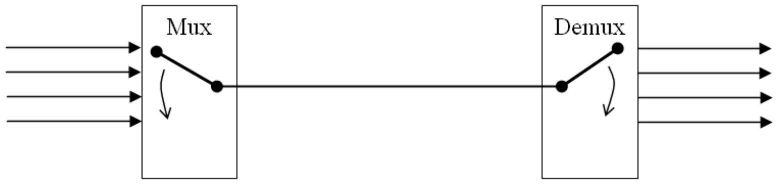


Figure 9.17 Example of a multiplexor and demultiplexor application

and some standard *I/O interfaces* like serial communication interface, network and modem adapter. Special interface cards are connected with the main board via system buses. These are e. g. *graphics adapter*, *sound card*, *disc controller*, and special I/O cards. We will present and discuss the basic units and the actual components of a computer system in the following subsections. Beside the internal components of a computer system we need peripheral devices for mass data storage (hard disc, tape drive, CD-ROM, DVD), data input (keyboard, mouse, joy stick), data output (printer, display), and communication devices (modem, wireless and local area network (LAN) devices). Some of these will be explained here under subsection 9.6.6.

9.6.1 Memory

The **memory** is a random access, high speed, solid state data storage and retrieval device. Each memory cell (*word*) is uniquely accessible by its address. The access time to write or read data does not depend on its address, it remains constant for all addresses. This is called **random access memory (RAM)** in contrast to **sequential access memory (SAM)** like tape storage devices. If the data stored can only be read, but not written, this type of memory i called a **read only memory (ROM)**.

A memory cell has a capacity of 8, 16, 32, or 64 bits, depending on the word "length" of the computer system used. The memory is physically organized as a grid where a cell is addressed by row and column number, but logically the addresses are sequentially assigned (see Figure 9.18). The maximum total capacity of a memory M_{max} can be calculated as address space s times cell size w.

$$M_{max}[bits] := s \cdot w$$

The address space depends on the number of bits used for the address. Some years ago, this have been 16, 20, or 24 bits, now it is at least 32 ($= 2^{32} = 4$ gigabytes (GB)) for a typical PC or laptop computer.

The address of a memory cell is stored in the memory address register (MAR) which is connected to the data bus. The address is divided into two parts where the lower (right) part is taken as column number and the upper (left) part of the MAR is taken as row number. Both numbers need to be decoded into the appropriate column and row address lines.

When both column and row lines and the *memory read command (MRC)* are activated (bit level 1) the content of that cell is read into the memory data register (MDR). When the *memory write command (MWC)* is activated (bit level 0), the addressed cell will be overwritten with the data from the MDR.

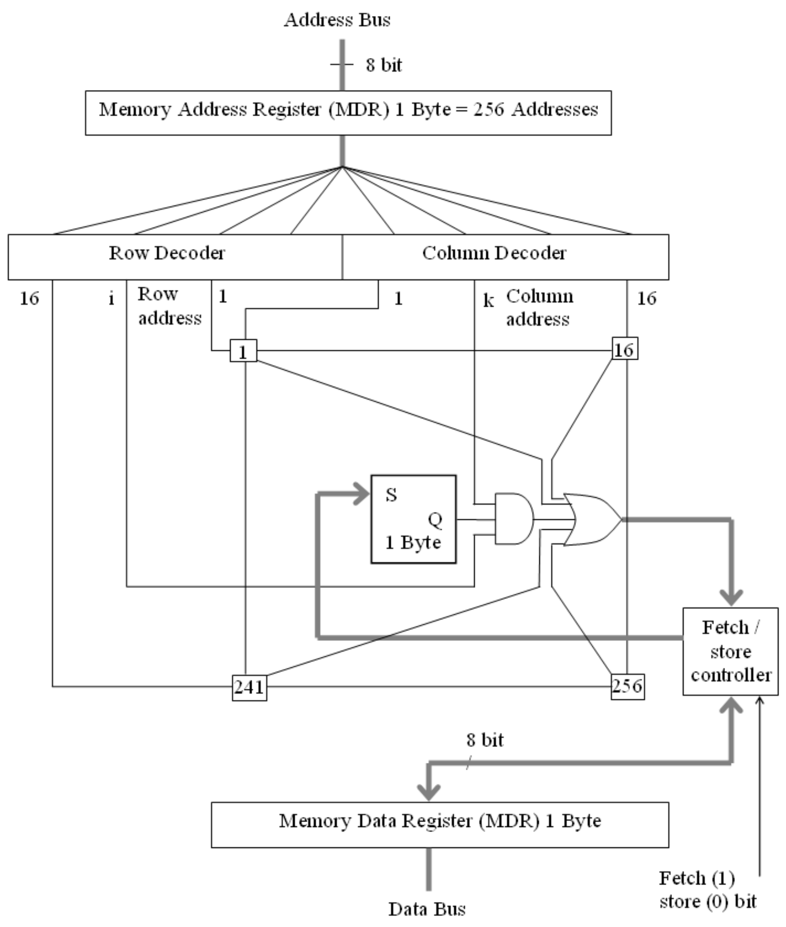


Figure 9.18 Physical memory (RAM) organisation

As the speed of a processor is developing much faster than the speed of memory (see Figure 1.8) a special high speed memory, called **cache memory** is located between both units in order to buffer data or commands and enable asynchronous fetch or store operations while processing is under way. The cache memory has substantial faster memory cells so that it can keep up the pace of the ALU.

There exists a memory hierarchy in terms of speed. The fastest data/command storage are registers. Then comes the cache memory, followed by main memory, and the slowest devices are peripheral storage devices as shown in the following table for a typical PC processor.

level	name	access time	size	comment
0	registers	0.5 ns	< 1 kB	1 - 8 Bytes/register
1	(L1-)cache	2 - 5 ns	32 kB	incl. on proc chip
2	(L2-)cache	$\sim$ 10 ns	$\sim$ 512 kB	ext. buffer
3	memory	60-80 ns	1 - 2 GB	word addressable
4	hard disc	10 - 20 ms	100 - 400 GB	single drive
5	tape drive	1 - 200 s	5 GB - (10 TB)	multiple tapes archive

The cache memory is used as a buffer memory to bridge the performance gap between main memory and the ALU. Data is first looked after in the cache memory. If it is not already there it is read from main memory and stored in cache. If the same data is read again or written, this is done only in cache. If data is updated to cache, a “[”dirty flag is set to indicate that this data needs to be stored back to main memory. But this can be done parallel and asynchronously to the data processing in the ALU.

The benefit of using a cache memory depends on the *hit ratio*, a number indicating the percentage of data that could be read from cache instead from main memory.

Let m be the access time of the main memory, let c be the access time of the cache, and h be the hit ratio of accesses that are satisfied from the cache. Hit ratios of 80 to 90% are not uncommon. Then we may calculate the average access time a of the system including memory and cache with the the following formula:

$$a = c + (1 - h)m$$

The first term signifies that we check the cache for every data access. The second term is the main memory access time for those data only that don’t get a hit on the cache.

Let’s give an example. A computer system has a cache access time of 5ns with 90% hit ratio and a main memory access time of 70ns. Substituting the values into the formula we get $a = 5 + (1 - 0.9)70 = 12$ ns which is about six times faster than without using a cache.

9.6.2 Arithmetic Logic Unit (ALU)

The **ALU** is the subsystem that performs mathematical and logical operation. We have learned in the previous Sections that the same elementary circuits may be used to build arithmetic and logical circuits. The ALU consists of at least a full adder, registers (accumulator), an operation register, data and address interface circuits.

The accumulator with full adder is dedicated for arithmetic operations. But its building blocks are suitable for logic operations like comparison ($=, <, >$) , too. Equality comparison may be tested by interpreting to value numerically and subtraction the values to compare. If the result is zero, both values are equal ($=$), if the value is negative, the subtracted value was larger ($>$), and if the value is positive, the subtracted value was smaller ($<$) than the other value.

The operation register defines what operations to do: addition, bit shifting, comparing, etc. The details of arithmetic operations have been explained in the previous section. Data and address interface circuits connect via bus system to other functional units, like memory or I/O subsystem. Via these path the addresses are selected and the data transferred between the ALU and memory or I/O subsystem.

9.6.3 Control Unit (CU)

The operations that should perform the ALU are defined by the program that is stored in memory[3], too. It is the task of the control unit (CU) to decode the operations that are coded in binary form like alpha-numerical data. The main component of the CU is the decoder that interprets the program code. As example take the following code clipping (addresses 10 to 14) showing how to add two numbers $c = a + b$ that are located in memory address 100 and 101 and store the result in address 102.

Address	code	comment
...	...	
10	CLR ACC	set accumulator AC content to 0 (clear AC)
11	LOAD 100	load value of a into AC
12	ADD 101	add the content of b to AC
13	STOR 102	store the content of AC to the address 102
14	STOP	stop processing
...	...	
100	0000 0010	the number a with value 2 at address 100
101	0000 0011	the number b with value 3 at address 101
102	0000 xxxx	the number c with value 5 after summation

In this example the content of the memory cells 10 to 14 are interpreted as commands, whereas the content at location 100 to 104 is regarded as numerical data. The commands in cells 10 to 14 are binary code but for the ease of human reading a mnemonic representation is used here. There is a register, called program counter PC that holds the current memory address of the active program. The machine command has usually two parts, the *command field* and the *address field*. The command field is decoded as command and the address field is interpreted as an address upon the command is executed. In modern computers there are commands with two or three address fields allowing complex operations on three or four data elements within one instruction (see Chapter 10).

The decoding of the actual command is performed by a decoder logic that activates the signal line that corresponds to the command.

For example, the command "LOAD 100" could be coded in an eight-bit computer as 10010100. The decoder reads the bits 0100 and sends it to the MAR to select that address in memory and activates the memory read command (MRC). Then it reads the command 1001 and activates the clock signal for the accumulator. This adds the selected memory data to the current value of AC and stores the result with the next clock cycle into AC.

[3]This is true for the active programs. Programs not active are stored on a peripheral storage media (hard disc, CD-ROM, etc) and loaded into memory upon activation.

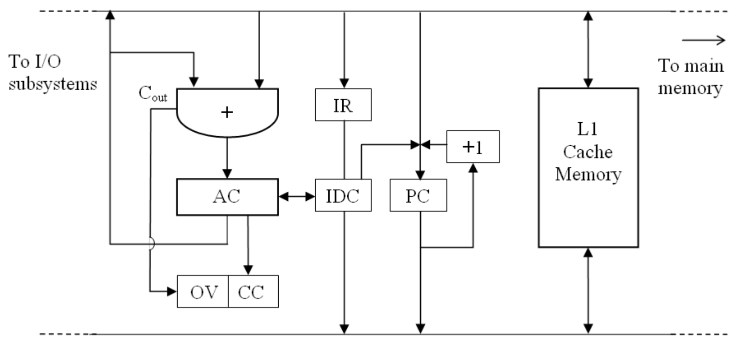


Figure 9.19 Block diagram of a central processing unit (CPU)

9.6.4 CPU

The *central processing unit (CPU)* consists at least of an arithmetic logic unit (ALU) and a control unit (CU). Registers for intermediate data, status information, operation codes provide operational storage capacity. Multiplexor link the main memory with the registers. Decoder decode commands for the CU and select memory addresses to be send via the bus system interface to main memory.

Usually a high speed memory buffer, called level 1 cache is included on the CPU chip to optimize processing speed.

To boost the performance of real number arithmetic a special floating point processor may be added to the CPU.

9.6.5 Bus Subsystem

A **bus subsystem** is a communication path that is used to exchange data between more than two participants (computer components). It consists of wires and an bus controller. The wires connect all components permanently that potentially communicate. Component A signals to the bus controller that it wants to send a message to component B. The controller assigns a time slot for A to transmit its data and indicate to all components that this message is for component B. A sends the data via the bus and all connected components can receive it. All components except B should ignore the data. This mechanism prevents simultaneous sending of two or more messages on the same wires. The controller is responsible to give any component a fair chance to communicate.

The controller is essentially a time slice multiplexor and demultiplexor as shown in Figure 9.17. The bus controller is using “one line” at different times for different communication partners.

9.6.6 Input/Output Subsystem

The computer needs to interact with the user and communicates with peripheral devices. The **input and output (I/O) subsystem** allows a computer system to communicate with the “outside world”. The I/O subsystem contains a set of controllers and buffers to interface with peripheral devices.

Compared to the CPU or memory the peripheral devices and human interaction is three to nine orders of magnitude slower. The main task for the

I/O controllers is to adjust data transfer to the slower speed of these devices without to constrict the performance of the CPU too much.

The decoupling of CPU and peripherals is realized essentially by buffering the data and leave the I/O controllers to work as autonomously as possible. In a typical scenario the data transfer is only initiated by the CPU and it takes then place controlled only by the I/O subsystem until completion. Meanwhile the CPU can continue processing. The end of data transfer from or to the peripheral device is then signalled to the CPU.

Some devices (modem, USB, LAN, etc) support only serial data communication. This required the I/O controller to send the data bit-by-bit. This is an other reason to buffer at least on word of data. The data buffer register needs to support a bit shift command to put the bits serially on to the line.

9.7 Summary

- In this chapter we have presented the von-Neumann computer architecture. Based on this computer model the pysical devices are identified. Von-Neumann's idea was to store data and programs as binary code in memory. We discussed mechanical, magnetic and electronic materialization of the binary code.
- When binary data is realized the bits 0 and 1 need to be represented as two different voltage levels. The manipulation of data leads to the changement of voltage levels (state). Similar to mass inertia the electrons need time to move. So there is a need to indicate when a state is valid. This is done by central clock signals. The faster the clock signal the faster the voltage levels change and and i consequently the processing and computations.
- The elementary binary circuits AND, OR, and NOT are explained in detail and example circuits are presented. We prove by example that any binary logic can be constructed from these elementary functions.
- Storage circuits (flip flop, register), arithmetic logic circuits (adder, comperator), and de-/encoder are constructed from these elementary binary logic circuits. Finally all these building blocks are put together to form the functional units identified by the von-Neumann architecture.

Review Terms

- von-Neumann architecture
  - Input/Output (I/O) unit
  - Arithmetic-Logic Unit (ALU)
  - Memory
  - Control Unit (CU)
  - System bus
- Binary signal
  - State
  - transition
  - Clock cycle
- Binary Operation
  - Boolean Logic
  - AND, OR, NOT
  - Truth table

  - ○ De Morgan's law
- Binary Circuit
  - ○ Low value (Bit 0)
  - ○ High value (Bit 1)
  - ○ Transistor
  - ○ Switch circuit
  - ○ FET, Source, Drain, Gate
  - ○ NAND, NOR
  - ○ Schematic diagram
- Arithmetic and logic circuits
  - ○ Flip Flop
  - ○ Register, accumulator
  - ○ Half / full adder
  - ○ Encoder, multiplexor
  - ○ Decoder, demultiplexor
- Computer components
  - ○ Main memory
  - ○ Random-access memory (RAM)
  - ○ Cache memory
  - ○ Central Processing Unit (CPU)
  - ○ Bus subsystem

Exercises

9.1 Name the functional units of the von-Neumann architecture.

9.2 How is binary data electronically materialized?

9.3 Discuss the advantage and disadvantage of mechanical, magnetic, or electronic materialization of binary data.

9.4 Why is the computer system clock needed?

9.5 What determines the clock cycle time?

9.6 Write down the truth tables for AND, OR, and NOT operations.

9.7 Develop the XOR boolean function from elementary operation with the following truth table:

input		output
i	j	o
0	0	0
0	1	1
1	0	1
1	1	1

9.8 Construct a binary circuit for XOR function using only NAND and NOR components.

9.9 Build a multiplexor for two input lines. What switching cycle is needed, if the data at any input line remains unchanged for at least one microsecond ($1\mu s$)?

9.10 What are the advantages and drawbacks of a bus system compared to dedicated connections?

Bibliographical Notes

There are many general textbooks covering computer hardware. Koegh [2001] covers the core topics of Boolean algebra, logic design, binary arithmetic, CPU, assembly language, memory systems, and network devices. The book is intended for non-technical people. A more universitary stype course in computer architecture and computer organization offers the book Clements [2006]. A more practical approach is take by Thompson [2003] providing a practical guide to buying, building, upgrading, and repairing Intel-based PCs.