

BERUFSAKADEMIE MANNHEIM  
Information Technology International

Lecture Notes

Programming I  
Part 1 Introduction

Dipl.-Betriebswirt (BA) VOLKAN YAVUZ  
{xmachina GmbH

November 2001–February 2002  
Classes: TIM01AGR && TIT01EGR  
Term: 1

**Contents**

<b>1 Motivation</b>	<b>1</b>
<b>2 Problem and Solution</b>	<b>2</b>
2.1 Problem Description . . . . .	2
2.2 Problem Specification . . . . .	3
2.3 Algorithm . . . . .	4
2.4 Summary . . . . .	5
<b>3 Programming Languages</b>	<b>5</b>
3.1 Generations of Programming Languages . . . . .	6
3.2 Programming Paradigms . . . . .	8
3.3 Compiler, Interpreter and Virtual Machine . . . . .	9
<b>4 Computer Architecture</b>	<b>9</b>
<b>References</b>	<b>13</b>

**1 Motivation**

*Computer Science* is the science that deals with the systematic processing and storing of data, and especially the automatic processing with means of a computer.

A computer is a machine that is able to perform simple operations at a very high speed. In order for a computer to perform meaningful work, it has to be programmed, i. e. it has to be instructed on how to solve a (real-world) problem by applying a great number of simple operations step-by-step. This set of instructions as a whole is called a *computer program*. A *computer program* is a formulation of an *algorithm* in a *programming language*.

**Note:** The seemingly greek word *algorithm* actually derives from the name of the mathematician MOHAMMED IBN MUSA ABU DJAFAR AL KHOWARIZMI, also known as AL CHORESMI (approx. 783–850). Together with other scientists, he translated greek writings on the subjects of mathematics and medicine into the arabic language in the “House of Wisdom” at the royal court in Baghdad. A widely known book with the arabic title “Kitab al muhtasar fi hisab al gebr we al muqabala” (“Kurzgefaßtes Lehrbuch für die Berechnung durch Vergleich und Reduktion”) had been translated into latin, and each section started with the words “Dixit algorismi”. Therefore the term *algorithm*. [see Klaeren]

Remarkable about an algorithm is that it decomposes a possibly complicated mathematical procedure in such a way into many single steps, so that by persistently and consecutively performing those simple steps, one finally reaches the correct solution, even though the procedure as a whole remained incomprehended, let alone the mathematical background. This greatly recudes the possibility of human failure and also lays the foundation for mechanical i. e. unintelligent processing. [see Klaeren]

### Example 1 (Recipe for Brownies)

**Ingredients** 1/2 cup melted butter, 1 cup white sugar, 2 eggs, 1/2 cup self-rising flour, 1/3 cup unsweetened cocoa powder, 1/4 teaspoon salt, 1 teaspoon vanilla extract, 1/2 cup chopped walnuts (optional)

#### Directions

1. Preheat oven to 175°C. Grease and flour a baking pan.
2. In a medium bowl, beat together the butter and sugar. Add eggs, and mix well. Combine the flour, cocoa and salt, stir into the sugar mixture. Mix in the vanilla and stir in walnuts if desired. Spread evenly into the prepared pan.
3. Bake for 25 to 30 minutes in the preheated oven, or until the edges are firm. Cool before cutting into squares. □

## 2 Problem and Solution

Given a real-world problem<sup>1</sup>, several steps have to be performed until the solution can be solved by running a computer program.

Starting with a description of the real-word problem to be solved and omitting all irrelevant details (*abstraction*), we get an (informal) problem description<sup>2</sup>. It is important to note that the problem description describes *what* has to be done, and not *how*.

### 2.1 Problem Description

**Example 2 (Informal Problem Description)** There are cars and motorcycles on a parking lot. In total, there are  $n$  vehicles with a total number of  $m$  cycles. Determine the number  $C$  of cars. □

The above written description of the problem is the first step but not entirely sufficient to derive an algorithm and implement a program. Natural language statements are ambiguous<sup>3</sup>, incomplete and sometimes even contradictory<sup>4</sup>. By formalization and using a more precise form of notation, the problem description is transformed into a *problem specification*.

1 *dt.* Realweltproblem

2 *dt.* (informelle) Problembeschreibung

3 *dt.* mehrdeutig

4 *dt.* widersprüchlich

## 2.2 Problem Specification

**Definition 1 (Problem Specification)** The problem specification is a formalised, precise, and unambiguous description of a real-world problem and the desired solution.  $\square$

**Example 3 (Problem Specification)** Obviously, the number of cars  $C$  plus the number of motorcycles  $M$  equals the number of vehicles  $n$ . Additionally, we know that cars have 4 cycles, and motorcycles have 2. This leads to the following equations:

$$\begin{aligned} C + M &= n \\ 4C + 2M &= m \end{aligned}$$

Transforming the first equation gives  $M = n - C$ , and inserting the result into the second equation gives:

$$\begin{aligned} 4C + 2(n - C) &= m \\ 2C &= m - 2n \\ C &= \frac{m - 2n}{2} \end{aligned}$$

Evaluating the formula with  $n = 3$  (number of vehicles) and  $m = 9$  (number of cycles) leads to the following computation:

$$P = \frac{9 - 2 \cdot 3}{2} = \frac{2}{2} = 1.5$$

This means that there are 1.5 cars on the parking lot. Obviously, the problem can only be solved if the number of cycles ( $m = 9$ ) is even.

Evaluating the formula with  $n = 5$  (number of vehicles) and  $m = 2$  (number of cycles) results in:

$$P = \frac{2 - 2 \cdot 5}{2} = \frac{2(1 - 5)}{1} = 1 - 5 = -4$$

meaning that “four cars are missing”. Obviously, the number of cycles has to be at least twice the number of vehicles. On the other hand, the number of cycles may at most be four times the number of vehicles. This means that to get meaningful results (meaningful in terms of the real-world problem),  $m$  has to be even, and  $2n \leq m \leq 4n$ .

This leads to the following problem specification, considering pre- and post-conditions<sup>5</sup>, which are inherent<sup>6</sup> to the original real-world problem, but which haven’t been mentioned in the problem description as they were regarded common sense<sup>7</sup> or self-evident<sup>8</sup>.

<b>Input</b>	Two natural numbers $m, n$
<b>Precondition</b>	$m$ even <sup>9</sup> , $2n \leq m \leq 4n$
<b>Output</b>	A natural number $C$ in case postcondition is satisfied, else “no solution”.
<b>Postcondition</b>	$\exists$ a natural number $M$ , with $C + M = n$ and $4C + 2M = m$ . $\square$

Using the problem specification, we can provide an algorithm, i. e. a set of instructions on *how* to solve the problem.

5 *dt.* Vor- und Nachbedingungen

6 *dt.* inhärent, innewohnend

7 *dt.* gesunder Menschenverstand, der

8 *dt.* selbstverständlich

9 *dt.* gerade

### 2.3 Algorithm

**Example 4 (Algorithm)** This is an example algorithm, solving our problem:

<b>Input</b>	Two natural numbers $m, n$
<b>Precondition</b>	$m$ even, $2n \leq m \leq 4n$
<b>Procedure</b>	Evaluate $C = \frac{m-2n}{2}$
<b>Output</b>	A natural number $C$
<b>Postcondition</b>	$\exists$ a natural number $M$ , with $C + M = n$ and $4C + 2M = m$ . <span style="float: right;">□</span>

As we can see, the aforementioned<sup>10</sup> algorithm doesn't solve just one problem, but a whole *class of problems*.  $m$  and  $n$  are said to be *parameters* to the problem, which can be called a *parameterised problem*.

**Example 5 (Algorithm)** Further refinement of example 4 yields the following result, which shows much similarity to a computer program:

**Precondition:**  $m$  even  $\wedge 2n \leq m \leq 4n$   
**Postcondition:**  $\exists$  a natural number  $M$ , with  $C + M = n$  and  $4C + 2M = m$   
 $n \leftarrow$  number of vehicles  
 $m \leftarrow$  number of cycles  
**if**  $m < 0 \vee m$  uneven  $\vee m < 2n \vee m > 4n$  **then**  
     write 'Wrong Input'  
**else**  
      $c = (m - 2 * n) / 2$   
     write 'Number of cars:'  $c$  □  
**end if**

**Definition 2 (Algorithm)** An *algorithm* is a well-ordered, finite collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.

The following properties have to be given:

**Finiteness** The whole procedure resp. algorithm has to be formulated in a finite number of operations, called steps (static finiteness), and during execution the algorithm claims only a finite number of resources (dynamic finiteness).

It might seem obvious to demand that the whole algorithm be formulated in a finite number of operations but sometimes an infinite number of operations is hidden as in:

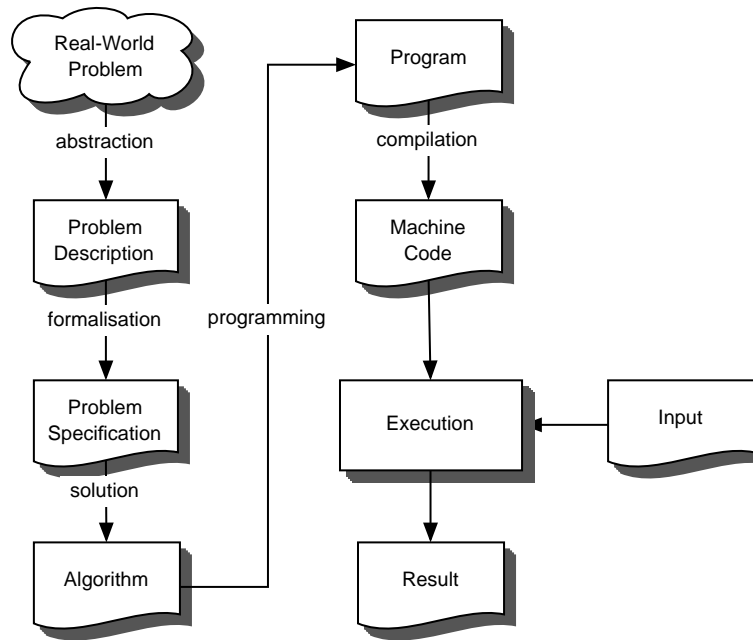
$$a = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

**Effectiveness** Each single step has to be computable effectively. Effectiveness (theoretical computability) must not be mistaken for efficiency (economically sensible computability). Some operations are not effectively computable, such as

- Count all digits of  $\pi$  that are 3
- Count from 1 to  $\infty$

Some operations are effectively computable but not efficiently computable, such as:

<sup>10</sup> dt. obengenannt



**Figure 1:** From Problem to Solution

- Compute all possible moves from a given game of chess until the game is finished, and thus choose the best consecutive move.

**Termination** The procedure halts in a finite amount of time.

**Determination** The sequence of operations is unambiguously determined for each step. This property is not always asked for as in algorithms when the next step is determined at random. □

## 2.4 Summary

Figure 1 gives an overview of the process described before. Starting with the real-world problem, that has to be solved an informal problem description is created by abstraction. By further formalisation and more precise formulation, a problem specification is written. Using this problem specification, an algorithm is developed, which is then coded using a programming language. The resulting program is then compiled and run. During runtime, the necessary data is input, yielding the desired results, thus the solution to the original real-world problem.

## 3 Programming Languages

The art of programming is  
 the art of organising complexity.  
 – Edsger Dijkstra

For algorithms to be performed by a computer, they have to be formulated in a way digestable by the computer, i.e. a program has to be written in a certain *programming language*.

The programming language used to implement a given algorithm enforces two classes of rules, syntactic and semantic. The *syntax* of a programming language specifies the formal rules, like constraints for identifiers, the set of reserved words (*keywords*), the beginning and the ending of the code, etc.

Additionally, the programming language also specifies the *semantics*, i.e. the *meanings* and *effects* of all statements.

The most simple programming language would consist of commands to perform arithmetic operations, to store values for further use, and to compare values to decide which steps to process next.

### 3.1 Generations of Programming Languages

The human's mind capacity to process information is rather limited. Miller concludes that humans can only cope with seven things at the same time. The solution to this is

- Abstraction, i.e. to concentrate on relevant facts and to disregard irrelevant details.
- Modularisation, i.e. to provide functionality as a "black-box" that can be used without the need to understand its inner workings, and building further functionality on already available modules.

Thus, demands made on programming languages include

- easy to read
- easy to understand
- easy to modify

Programming Languages have evolved through the time, and are usually said to belong to a *generation*. Earlier generations of programming languages focused very much onto the technical peculiarities of the computing hardware used, while newer generation programming languages focused more onto the problems to be solved. This evolution can be characterized as a shift from *machine orientation* to *problem orientation* (see figure 2).

The shift from *machine orientation* to *problem orientation* is motivated by the fact that the computer is seen as a tool to solve a problem in a specific solution domain. Thus, the challenge lies not in programming the computer using its machine code, but to efficiently use the computer to solve a real-world problem. The higher-level the abstractions a given programming language supports, the more a programmer is relieved from the burden of concentrating on the machine with all its peculiarities, resulting in more intellectual capacity available to concentrate on the problem to be solved.

Following a common approach to describe generations of programming languages:

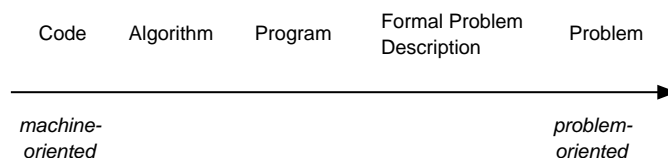
**1GL** or first-generation language is the *machine language* the processor actually works with, expressed on binary level with '0's and '1's.

**2GL** or second-generation language is *assembly language*. Assembly language still translates one-to-one to *machine language*, but an *assembler* accepts mnemonic abbreviations for machine level operations and translates them to '0's and '1's. A typical assembler code fragment looks like this

```

1      cld                ; reset direction flag
2      lodsb              ; load byte from string
3      test    AL,80h     ; test for end of string
4      pushf              ; save flags
5      and     AL,7fh     ; insure valid character
    
```

**Figure 2:** Evolution of programming languages



```
6      mov    AH,14      ; write tty
7      int    10h       ; bios video service
8      popf                   ; restore flags
9      jz     Send      ; do until end of string
10     ret                    ; return to caller
```

**3GL** or third-generation programming language is a *high-level* programming language, such as already mentioned languages like FORTRAN, PL/I, C oder Java. A *compiler* usually translates the *source code* first into *assembler language* which is then further translated into machine language. Java, and some other languages like Lisp, are compiled into *byte code* in contrast to *native code*. While native code can directly be executed by the machine or CPU (Central Processing Unit), byte code runs on a *virtual machine* which emulates a CPU in software. Other languages like APL, Forth, and Smalltalk are not compiled, but *interpreted* instead. Programs in 3GL translate one-to-many to machine language, i. e. that one single statement in a high-level programming language may need many machine language instructions.

**4GL** or fourth-generation programming languages are designed to be close to natural language. This also includes a *paradigm shift* from *imperative* (telling the computer *how* to solve a problem) to *declarative* (telling the computer *what* the solution or the desired result is) programming. SQL (Structured Query Language) is a typical 4GL (Fourth Generation Programming Language):

```
1  select *
2  from students s
3  where exists (
4      select *
5      from professors p
6      where p.dateofbirth > s.dateofbirth);
```

**5GL** or fifth-generation programming uses a complete visual or graphical development environment to construct and debug programs. This may also include visualization of key concepts of the underlying 3GL programming language (like class hierarchies in object-orientation) or tools for the creation of user interfaces, which in turn might be described in 4GL.

Booch gives another taxonomy of programming language generations [Booch, 1994, pp. 46].

### First Generation (1954–1958)

- Mathematical Expressions (FORTRAN I<sup>11</sup>, ALGOL 58)

Those languages were mainly developed for applications in technical and scientific areas. They relieved the programmer from the burden of writing assembler or even machine coded thus enabled him to use familiar mathematical expressions and formulae. This generation was the first step “away from the machine” to the actual solution domain.

### Second Generation (1959–1961)

- sub routines, separate compilation units (FORTRAN II)
- block structure, data-types (ALGOL 60)
- data description, file handling (COBOL)
- list processing, pointers, garbage collection (LISP)

The focus during this generation lay on the algorithmic abstraction. As computers became more and more powerful, more problems became viable for computer-based handling.

<sup>11</sup> derived from the words *formula translator*

### Third Generation (1962–1970)

- consolidation and summarization of several concepts from FORTRAN, ALGOL, and COBOL (PL/I)
- several ALGOL successors (ALGOL 68, Pascal)
- classes, data abstraction (Simula)

At the end of the 1960's due to the development of transistors and integrated circuits, prices for computing hardware fell dramatically, while the processing power grew exponentially. As the problems to be solved grew, programming languages had to support more abstraction.

### “The Generation Gap” (1970–1980)

- Smalltalk as successor of Simula
- Ada derived from ALGOL 68 and Pascal and influenced by Simula, Alghard, and CLU
- CLOS derived from LISP, LLOPS, and Flavours
- C++ derived from C and Simula
- Eiffel derived from Simula and Ada

Literally thousands of new programming languages were developed during the 1970's, all trying to compensate the deficiencies of previous programming languages in the development of large software systems. Only very few of those programming languages survived, but the concepts were incorporated into successors of older programming languages. Most interestingly, the concepts of *object oriented* programming languages were developed during this phase.

## 3.2 Programming Paradigms

Independent of the generation the programming language belongs to, it is also based upon a certain concept or *paradigm*. In this lecture, we will concentrate on *imperative* programming languages mainly. It has been shown that any problem that can be solved by any one of the paradigms can also be solved by all the others. However, certain types of problems lend themselves more naturally to specific paradigms.

**Procedural/Imperative** Derived from the latin word *imperare*, this programming paradigm clearly “commands” the computer, thus telling *how* the results are to be achieved. Typical commands are “store the value of 3 into variable *a*” or “jump to program position *p*”.

Imperative programming languages are the most widely used ones. As the paradigm maps well to the underlying machinery, those languages were developed very early, and were sufficiently efficient on the available computer hardware.

Examples are Ada, ALGOL, Basic, C, Cobol, FORTRAN, Modula-2, Pascal, PL/1, Simula. Also, the *von-Neumann* architecture is clearly reflected in imperative programming languages.

**Functional** The *Functional* Programming paradigm views all subprograms as functions in the mathematical sense. Programs are functions, taken in arguments as input, and returning values as output. Functions can also be treated as first-class objects, involving the possibility to pass functions as arguments to other functions. Functional programming languages provide a high level of abstraction, suppressing many of the details of programming and thus removing the possibility of many classes of errors.

```
1 function fac (n: N) N :  
2     if n <= 1 then 1 else n * fac (n-1).
```

Examples are LISP, Scheme, Haskell, and LOGO.

**Logical** The *Logical* Programming paradigm takes a declarative approach to problem-solving. Consisting of three sections, first a series of definitions that define the problem domain are made. Then, logical assertions about the problem in question are made, establishing all known facts. Finally, queries are made, and deducible solutions are returned. The role of the computer becomes maintaining data and logical deduction. The programmer's role is kept to a minimum, in declaring all known facts. In contrast to imperative programming, where it is explicitly formulated *how* to use the declarative knowledge to solve the query, in logical programming, this is done by the computer.

```
1 fac (0, 1). "The factorial of 0 is 1"
2 fac (1, 1). "The factorial of 1 is 1"
3
4 fac (N - 1, F) => fac (N, N * F).
5
6     "If the factorial of N - 1 equals F, then
7     the factorial of N equals N * F"
```

To evaluate the factorial of 6, the computer returns that value of  $F$  for which the predicate 'fac (6, F)' holds true.

Examples are PROLOG.

**Object Oriented** The *Object Oriented* Programming paradigm views real-world objects as separate entities, involving state and behaviour. As objects do operate independently, they are encapsulated, isolating implementation details from how they are accessed (their interface) and preventing access to the inner workings of a class (encapsulation). Cooperation between objects is done by message passing. Objects are organized into classes, from which they inherit structure and behaviour. Classes are organized into hierarchies, where derived classes (subclass) inherit structure and behaviour from its superclass, optionally supporting additional structure or behaviour. It is also possible to override behaviour inherited from a superclass.

Especially inheritance is the single most distinguishing feature of the object oriented programming paradigm, and the chief benefit over other programming paradigms. This feature gives relatively easy code reuse and extension, without the need to access or change existing source code, and maybe breaking existing code.

### 3.3 Compiler, Interpreter and Virtual Machine

A compiler's task is to process a program written in a high-level programming language and generate machine code that can actually be executed by the computer. Such resulting *binaries* are platform specific, meaning that they can only run on machines that have the same type of CPU and run the same operating system.

Although interpreters seem to be omitting the compilation phase as they execute a given program almost instantly, the essential functionality (syntactical and semantical parsing) is still performed.

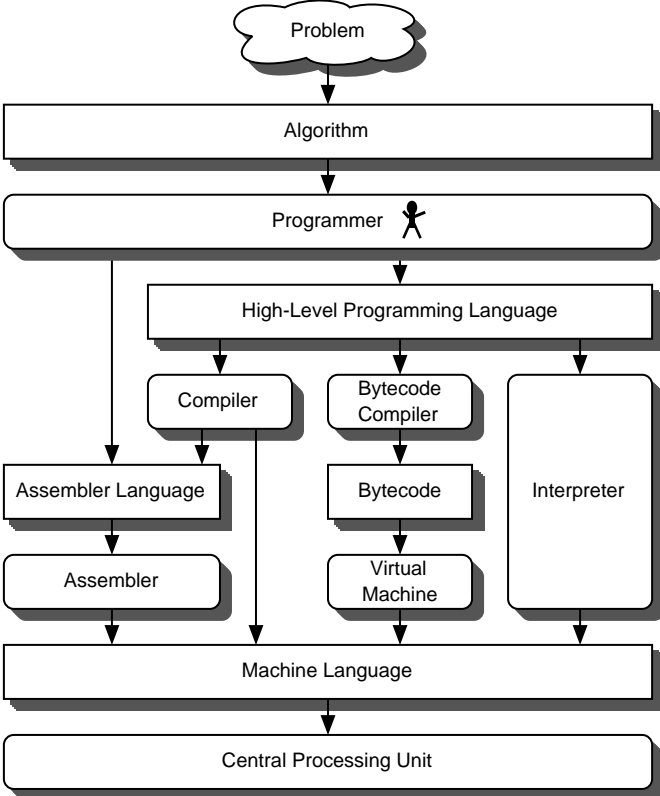
Programming languages targeted at virtual machines overcome the disadvantage of platform specific binaries. As long as the virtual machine is available on the target platform, those programs can be run there without re-compilation.

Figure 3 on the next page summarizes the differences between compiled and interpreted programming languages. Also included is the concept of a virtual machine, executing byte code.

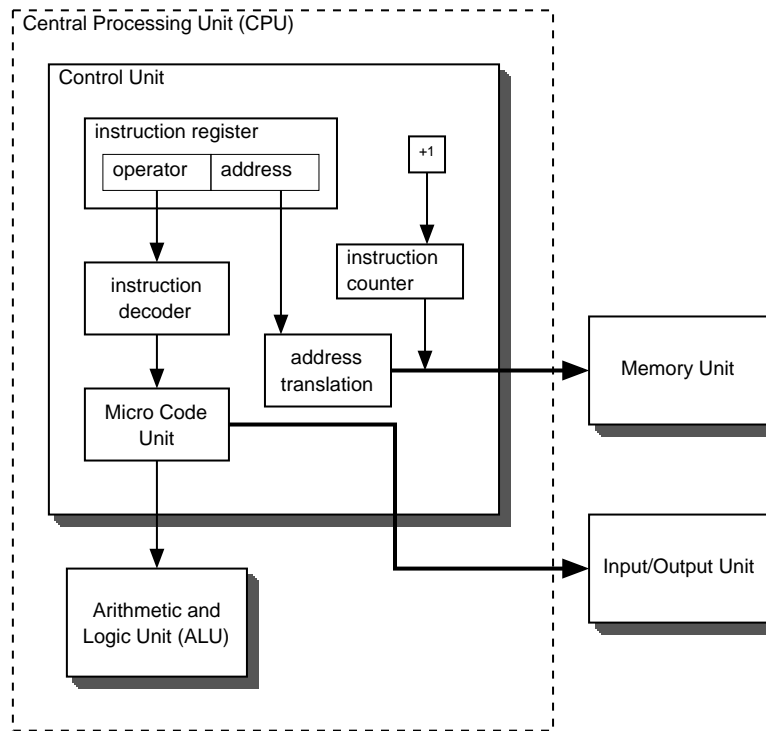
## 4 Computer Architecture

Von Neumann published his ideas about the requirements for a general purpose electronic computer in 1945 [Goldstine and von Neumann, 1963, Burks et al., 1963]. Almost every computer built since then is rooted in the *von Neumann* architecture. The first computer of this type to be actually constructed

Figure 3: Differences between compiler, interpreter and virtual machine



**Figure 4:** The von Neumann Architecture



and operated was the Mark I, designed and built at Manchester University in England. It ran its first program in 1948, executing it out of its 96 word memory.

The key point in his ideas was the concept of a *general* purpose computer, that not only was able to store its data and intermediate results of operations, but also the instructions or operations. In contrast, a special purpose computer could have its instructions hard-wired and thus be part of the hardware. Von Neumanns principal contribution was to encode the instruction in numerical form (just like the data the instructions operate on) and save them, along with the data, in the computer's same memory.

The architecture consists of mainly four parts as depicted in figure 4. Although not covered by von Neumann, those parts usually interface with each other using several buses, als depicted in figure 5 on the next page.

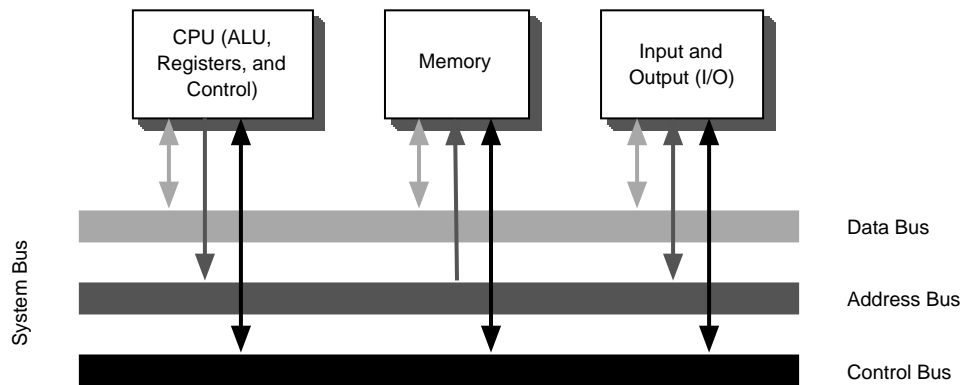
**Control Unit** The Control Unit is the heart of the computer. It's responsibilities mainly include loading the operations in the correct order from memory, decoding the operations that are encoded in numerical form, executing them and supplying all other units with the necessary control signals.

**Arithmetic and Logic Unit** All arithmetic (e.g. addition, subtraction) and logical (e.g. and, or, not) operations are performed by the ALU (Arithmetic and Logic Unit). The necessary operands (typically two) are made available to the ALU by the Control Unit. Control Unit and ALU together build the CPU.<sup>12</sup>

**Memory Unit** All data and operations are stored in the Memory Unit. Digital memories can store exactly two states, '0' and '1', sometimes also described as 'low' and 'high' or 'no' and 'yes'. This corresponds to 1 *bit*. 8 bits give 1 *byte*. Sometimes, several bytes (usually 4 or 8) are grouped

<sup>12</sup> All arithmetic operations can be derived from the basic operations of shifting, bitwise negation (complement), and addition. Subtraction is performed by addition of the complement, multiplication by repeated addition, and division by repeated subtraction.

**Figure 5:** The System Bus Model



together, giving one *word*. To quantify memory capacity, “kilo”, “mega” and “giga” are used. As memory addressing is also digitally based, the factor is not  $10^3$  (1.000) but  $2^{10}$  (1.024). 1 MByte is thus 1.048.576 bytes and 1 GByte 1.073.741.824 bytes.

**Input/Output Unit** The I/O-Unit (Input/Output Unit) provides all necessary functionality to interface with peripheral hardware. This may include machinery to read in program that are saved on punch cards, or to save results of program runs on taper tape.

The main principles of the classical von Neuman architecture are as follows:

- The computer consists of four units (control, arithmetic and logic, memory, input/output).
- The computer’s structure is independent of the problem to be solved. To actually solve a problem, an appropriate program has to be entered into memory. Without such a program, the machine is not able to operate.
- Program, data, intermediate and final results are all stored in the same memory.
- The memory consists of memory cells with identical size, that are numbered consecutively and that are uniquely identified and accessed by their address.
- The program’s operations are executed in exactly the same order as they are stored in memory. The instruction counter determines the next operation to be executed. Using special operations, deviations from that fixed order can be achieved.

Computers based on the von Neumann architecture belong into the SISD (Single Instruction, Single Data Stream) class of architectures. These architectures are characterized by the following properties:

- one CPU, consisting of Control Unit and Arithmetic and Logic Unit
- a strictly sequential execution of operations

**List of Tables**

**List of Figures**

1	From Problem to Solution . . . . .	5
2	Evolution of programming languages . . . . .	6

3	Differences between compiler, interpreter and virtual machine . . . . .	10
4	The von Neumann Architecture . . . . .	11
5	The System Bus Model . . . . .	12

### List of Acronyms

1GL.....	First Generation Programming Language
2GL.....	Second Generation Programming Language
3GL.....	Third Generation Programming Language
4GL.....	Fourth Generation Programming Language
5GL.....	Fifth Generation Programming Language
ALU .....	Arithmetic and Logic Unit
CPU .....	Central Processing Unit
CU.....	Control Unit
I/O-UNIT.....	Input/Output Unit
MU .....	Memory Unit
SISD .....	Single Instruction, Single Data Stream
SQL.....	Structured Query Language

### References

Grady Booch. *Objektorientierte Analyse und Design*. Addison-Wesley Deutschland GmbH, Bonn; Paris; Reading, Massachusetts, 1994. ISBN 0-201-42277-8.

A. W. Burks, H. H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. In A. H. Taub, editor, *John von Neumann Collected Works*, volume V, pages 34–79. The Macmillan Co., New York, 1963.

H. H. Goldstine and John von Neumann. On the principles of large scale computing machines. In A. H. Taub, editor, *John von Neumann Collected Works*, volume V, pages 1–32. The Macmillan Co., New York, 1963.

Herbert Klaeren. Calculemus! Die artifiziiellen Paradiese der Informatik. URL <http://www-pu.informatik.uni-tuebingen.de/users/klaeren/herakles/herak1\%es.html>.

George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956. URL <http://psychclassics.yorku.ca/Miller/>.