Computer Systems (VUB 2018–2019) — notes extracted from slides —

Notes extracted by
Olivier Pirson — OPi
olivier.pirson.opi@gmail.com

January 15, 2019

Last version and LaTeX sources:
https://bitbucket.org/OPiMedia/elec-y506-computer-systems-extracted-notes

Abstract

This document contains all notes extracted from the slides of the VUB course *Computer Systems* (Jacques Tiberghien 2018–2019, VUB 4004876EER, ULB ELEC-Y506).

Contents

1 From Highlevel “Language” to Lowlevel “Code”

4. In this introduction, intended for students familiar with programming in high level languages, the steps necessary to execute the programs on actual machines will briefly be reviewed.

5. Inside a computer, all information is encoded by means of integer, fixed length, numbers. To transfer information between the outside world and the computer translating devices, such as keyboards, displays, printers, temperature sensors, accelerometers etc. are required. We call them input or output interfaces. Often information can not be processed immediately after having been introduced in the computer. For instance, if one wants to know if the temperature is climbing, the actual value obtained from the temperature sensor needs to be compared with the previous value obtained from the same sensor. Therefor the computer has a data memory, where numbers can be stored as long as needed. These numbers can eventually be fetched from that memory, combined by an arithmetic calculator and the result stored also in the data memory. The different actions performed by the parts of the computer are initiated by a control unit that sends, at appropriate moments, electrical signals to these different parts as required for the specified task. This is called “execution of an instruction”. Instructions are encoded by means of integer numbers, stored in the program memory. The control unit fetches, one by one these instructions and executes them sequentially. The list of instructions to be executed is loaded into the program memory through an input interface, called here “program interface”.

6. Keyboard, screen and mouse as traditional input/output interfaces.

7. Printers are also output interfaces.

8. The vast majority of computers don’t have traditional input/output interfaces such as keyboards and displays but are embedded in physical systems and use specific interfaces to communicate with the physical process they control.

10. As a first analogy, one can consider a computer memory as cabinet of drawers, each drawer can contain one, fixed length, integer number and is identified by a number, called “address”. Storing a number in a drawer discards automatically the number that was previously stored in that drawer.

12. Conceptually, a data handling (arithmetic) instruction is composed of 5 independent items:
- The nature of the operation that needs to be performed
- The addresses in data memory of the two operands that need to be combined by an arithmetic operation
- The address in data memory where the result of the operation needs to be stored
- The address in program memory where the next instruction will be fetched.

If only data handling instructions existed, the sequence of executed instructions would be fixed in advance and could not be influenced by the value of some data items. Therefor a second variety of instruction, the control instructions, is required. In a control instruction the value of two data items are compared and the address of the next instruction to be fetched depends on the result of the comparison.

13. As an example of how, at the lowest level we will consider, a computer performs a

given tasks we will discuss a simple electronic lock.
The input interface is a digital keyboard, the output interface an electromagnetic lock that can unlock a door.

14. Despite the fact that all data and instructions in a computer are encoded as integer numbers, we will give here names to the values of these numbers, helping us to memorize their functions.
Addresses in data memory are given a name to evokes the function of the number stored at that address. ND will contain the number of buttons that have already been pushed and SC, the value of the secret code entered through the keyboard.
The input and output interfaces are considered as special locations in data memory.
Location KFL (Keyboard Flag) contains a 0 as long as no button has been pushed. Once a button has been pushed it contains a 1, until a 0 is written in that location. When KFL contains a 1, KDA (Keyboard Data) contains an integer between 0 and 9 corresponding to the key that has been pushed. For unlocking the door, it suffices to write a 1 in DDA (Door Data)
A numerical value preceded by the # sign means that the instruction will interpret that value as an operand rather than as the address of an operand.

15. One can observe that the computer described here has two memories, one for data and one for instructions. This is the case for the vast majority of computers, designed and manufactured for a specific application. The number of digits used for data and for instructions as well as the number of numbers each memory can store can be chosen independently for each memory and optimized for a specific application.

16. When the applications the computer will have to run, are not known in advance, Optimizing separately word length and memory size for each of the two memories becomes meaningless. Then it is preferable to combine physically both memories into a single one, with a word length that is a compromise between the size of data useful for most applications and the size of instructions allowing an efficient encoding of the various instructions the control unit can execute. Such computers are called “Von Neuman computers”. Most general purpose computers like smart phones, tablets, laptops, desktops and servers have a Von Neumann architecture.

17. In most general purpose computers the memory is physically split in several parts, with different properties:
- A fast, expensive part for storing the data and the programs that are being used at that moment. This is often called “central memory”.
- A slower, less expensive memory used for data and programs that need to be kept for future use. This is often called “peripheral memories”. Examples of such memories are solid state disks, magnetic disks, flash memory sticks, CDs and DVDs.
- Archival stores, with very slow access but huge storage capacity and high reliability, based most often on magnetic tape cassettes, handled by robots.

18. One, or several central processor units are built in a single silicon chip. As such chips dissipate a lot of power, they are often mounted within cooling devices including heat pipes and fans.

19. It was already mentioned that programs and data are always coded inside the computer as integer, fixed length, numbers.
Programs in that form are said to be in machine language. Such code, loaded in the

program memory of the computer is also called “object code”.

20. For discussing the example we gave names to instructions and memory locations. This is called assembler language. Each instruction in assembler language corresponds to one instruction in machine language. Assembler language is called a “low level language” (LLL).
Of course, programs written in a LLL need to be translated in machine language before they can be loaded in the program memory.

21. Programs written in a LLL are much more understandable by humans than programs in machine language. When these programs have been written by humans, they are called “Source Code”. They are normally translated in machine language by means of a translating program, called “assembler”. The assembler produces object code (in machine language) that can be loaded in the program memory of the computer.

24. Translating a LLL program and executing it takes four steps:
First the assembler program, supposedly available in object code, needs to be loaded in the program memory of the computer.

25. Then, that translating program can read, line by line the source code written in a LLL from a peripheral memory and translate it, line by line, into object code, which is stored in memory (normally, peripheral memory).

26. The newly created object code can then be loaded in the program memory of the computer.

27. And finally, the newly loaded object code can be executed, reading user’s data and producing (hopefully!) user’s results.
Of course, it is possible that the computer used for steps 1 and 2 is different from the one used for steps 3 and 4. This is often the case when the computer running the user’s program is a very small machine, lacking peripheral memories and a friendly users interface.

29. Also addresses in program memory can receive a symbolic name: in the above example, BGN and KLP are addresses of instructions that will be fetched following certain control instructions. For instance the instruction at address KLP will be fetched after KFL has been found equal to 0 or after ND has been found different from 3. These symbolic names are simply defined by writing them in front of the instruction whose address they represent.

30. When a LLL program needs to be translated into object code, some indications need to be given to the translating program. For instance from what address in data or program memory the program will be loaded. This is commonly done by adding to the program pseudo-instructions that look like true instructions but that are not translated into machine instructions. In the above example the two ORG pseudo instructions tell the assembler that data memory starts at address 101 while program memory starts at address 11, that for variables ND and SC one memory word needs to be reserved in data memory and that variables KFL, KDA and DDA have respectively the fixed addresses 1,2 and 3, defined outside the normal data memory.
The pseudo instruction END informs the assembler that it has reached the end of the program to be translated.

31. Translating LLL is fairly simple. In fact only one problem requires special attention: when a symbolic name, such as Next in the above example, is used before it is defined.This is called “a forward reference”. When the program is translated line by line, the translator can not know what address should be used in the EQ? instruction, as Next is not yet defined.

32. Two approaches exist to solve the problem of forward references:
- The object code is kept in central memory, with empty spaces for the still unknown addresses, which are filled in when these addresses have been defined. This approach requires of course that the entire object code can be kept in central memory.
- The source code is translated twice, the first time, only to set up a symbol table in which all symbolic names and their value are stored and the second time to generate the object code.

33. Two pass assemblers read twice the source code!

34. As to compute the addresses corresponding to the symbolic names, an almost complete translation need to be made, it is a pitty that this translation is not used to produce the final object code. Some two pass assemblers generate a partial translation during the first pass and complete it during the second pass.

35. The correct working of both one and two pass assemblers can be jeopardized when the length of an instruction depends on the value of its operands. Indeed, some processors have instructions that can refer to 8, 16 or 32 bit addresses, and the assembler can not know which version of the instruction should be used before the length of the operand is known. If n is the number of forward references that can influence the length of instructions, it can be proven that by implementing a n+1 pass translator, all forward references can be correctly handled. This technique however is almost never used. Instead the programmer is requested to give a hint to the translator about which variant of the instruction should be used. If the hint is wrong, the translator can generate an error message or a warning, and the programmer can change the hint.

36. In low level programming, multiple uses of identical sequences of instructions, eventually with different parameters, occur very often as shown on the left side of the slide. Instead of writing repeatedly the sequence LDA, LDB, MUL with their operands, one can define a macroinstruction MULB with two formal parameters &1 and &2 and replace the many occurrences of the sequence LDA, LDB, MUL by references to MULB with the appropriate actual operands, as shown on the right side of the slide. An assembler accepting macros first expands, in the source code the macros (the program shown on the right is replaced into the one shown on the left) and then translates the entire program into object code.

37. Macros are not only used in low level languages, they are also used in high level languages such as c, where the #include command allows insertion of specific parts of a program defined elsewhere.

38. Macros are often handled, before translation of the source code into object code, by a preprocessor, suitable for handling macros in any programming language. Such a macro expander is in fact just a text processor that allows sophisticated merging of text operations.
For most LLL exist macro libraries containing macros to perform many short but fre-

quently used operations, such as, for instance, computing the absolute value of a signed number.

39. In fact, for the superficial observer, macros look exactly like functions, procedures or methods commonly used in high level languages. They are, however, fundamentally different, in the sense that the resulting program contains as many copies of a macro as it is "called", while only one copy of a function is present, copy which is used every time when the function is called. One could say that macros shift the overhead of function calls from execution time to compile or assembling time. They are typically faster than functions, but require more memory.

40. Large programs are seldom made by a single person, existing pieces are reused and new parts are often made by a team.
Translating the source code of such programs into object code in a single operation is unpractical. It would be preferable that the different parts could be translated separately and brought together before execution of the program. This is possible by using so called relocatable object code, that can be linked together with other pieces of relocatable object code by a specific program called linker to obtain the executable object code.

41. Relocatable code is composed of three parts:
- The object code, containing all the instructions, the addresses being generated as if each separate piece of relocatable code were to be loaded in memory, starting at address 0, together with this code a table is created containing all the locations in the code that need to be modified if the program would be loaded in memory with an other start address than 0.
- The externals table, containing all the symbolic names that are referred to in a piece of program, without being defined in that piece. The externals table contains also the locations in the code that need to be changed when the actual address corresponding to that symbolic name becomes known.
- The entry points table, that contains the symbolic names that are defined in a piece of program and that may be referenced from outside that piece. This table contains also the corresponding address.

42. The task of the linker consists in matching the externals and the entry points of the different available relocatable modules, deduce from this matching operation the list of modules that need to be included in the final object code and adapt consequently all the addresses in each module to build up a coherent executable object code.
Just as assemblers, many linkers use two passes: the first matches the entry points and the externals and by doing so establishes the list of modules to be linked and the second does the actual generation of object code.

43. In large software suites, such as Microsoft Office, not all available functions are used by all users. In fact each user requires only a small fraction of the available software. Therefor it would be a waste of time and memory space to link all the available functions in advance. Instead, some function calls are replaced with calls to the linker, with the name of the function as a parameter. This way, the function will be linked at the moment one wants to use it and those function that are not used will not be linked and not occupy memory space. This technique is called "dynamic linking". A look at the Windows and System directories in Windows will show you that Microsoft software is in fact a giant collection of .dll files, where dll stands for "dynamically linkable libraries".

44. Most programs are written in a high level language (HLL), such as Java, c, FORTRAN or Python. More than one thousand such languages are in use. This slide shows a program for the electronic lock written in a HLL called Modula 2, which is a language optimized for readability and consistency. Lines 2 to 6 are declarations. They correspond to the pseudo instructions in the assembler program, but here they are less oriented towards the technical details of the computer and emphasize the program logic. For instance, the declaration of SC reminds us that the value of SC is an integer in the range 0 to 999. The part of the program delimited by BEGIN and END lock corresponds to the executable instructions in the assembler program, but again, in a way that is much closer to way an application programmer thinks than to the way the instructions of the computer need to be arranged to obtain the desired results.

45. Of course, programs written in a HLL need to be translated into object code to be executed, but this translation is orders of magnitude more complex than the translation of assembler in machine code, because there is no longer a one to one relationship between HLL instructions and machine instructions. This, however allows to translate identical HLL source code for different processors. The same app can be translated for a laptop with an Intel processor or a smartphone with an ARM processor. A slightly oversimplified description of the translation process of a HLL into machine code shows two fundamentally different ways to proceed: the first way, called compilation, consists in translating the entire program from HLL into object code (or relocatable code). The second way consists in translating and immediately executing each statement of the source code. The program that performs such a translation and subsequent execution is called interpreter.

46. What are the main advantages of compilation and interpretation?
When the same program needs to be executed many times, it is obvious that compilation and repeated execution of the object code is more efficient. However, in case of run-time errors, such as a division by 0, error reporting might have little informative value, as the only information available at run time are the addresses of the instructions and the data involved with the error. Deducing from these addresses which instruction and which variables caused the error is generally an almost impossible task.
On the other hand, an interpreter, translating the source code statement by statement will always be able to report useful data about any incident that occurs during execution. On the other hand, translating repeatedly the same statements included in a loop that is executed many times might be an unacceptable waste of time.
Fortunately, modern compilers and interpreters borrow from each other techniques to keep references to the original source code in compiled object code and to store temporarily translations that might be needed again in the near future.
Roughly speaking one could state that interpretation is to be preferred for program development while compilation is best for processing data with stable programs. Most modern programming environments allow to evolve from interpretation towards compilation during program development.

47. A compiler (and an interpreter) have several functional parts. They are mainly the lexical analyzer, the syntax analyzer, the code generator and the code optimizer. Some compilers have also a preprocessor for handling macros. Most compilers produce relocatable code, but some produce assembler code instead. This assembler code then needs to be further translated by means of an assembler.
This two steps translation is common in compilers that were originally developed as student projects. For checking if a new compiler works properly it is indeed much simpler to read the produced assembler rather than object code.

50. Some compilers perform the translation step by step creating several intermediate representations of the program.

51. Other compilers have a more monolithic structure: the syntax analysis is the core of the compiler which calls functions that try to recognize the lexical items expected in a specific construct and, once the structure is identified, other functions that generate the corresponding machine (or assembler) code to execute the recognized statement.

2 Data representation

2. Two fundamentally different way of representing (and storing) information exist: analog and digital.
In analog representations, the representation of a quantity can vary almost continuously, by steps too small to be observed.
In digital representations, the range of values to be represented is subdivided in a pre-defined finite number of subranges, and each subrange is assigned a name. The representation of a value becomes then the name of the subrange the value belongs to. Most often the names assigned to these subranges are numerical values, proportional to the represented value.
As an example, one can consider chronometers: a hourglass represents time as the height of sand in the lower chamber while a digital watch measures time in hours, minutes and seconds.

3. In digital computers, all information is encoded by means of integer numbers. An integer number is a set of digits. These digits need to be physically stored in the memory by means of some physical property of this memory, such as voltage, current, magnetic field or optical opacity. To maximize the reliability of storage, the number of different digits to be stored is minimized, so as to ease their recognition and minimize the influence of external perturbing factors. It is easier to distinguish between light and darkness rather than to distinguish between ten different intensities of light. This implies that the integer numbers stored in digital devices are usually binary numbers, featuring only two digits, 0 and 1. Such binary digits are called “bits”.
With one bit only two different values can be represented, but adding a bit doubles the number of values. With 32 bits per word, already over 4 billion values are possible.

4. As writing down in texts binary numbers is somewhat tedious often 3 or for bits are grouped in one octal or hexadecimal digit, as shown above.
This applies to programming languages and texts about computers, but has no effect on the internal, always binary, representation of numbers.

5. As all parts of a computer are dimensioned in such a way they can be described by binary numbers, it is useful to memorize some often encountered powers of 2.
One power which is particularly useful is $2^{10} = 1024$, which is equal to 1000 with an error margin of 2.464 = 6*10 + 4, implying that 2^{64} is approximately $1000^6 * 2^4 = 16 * 1000^6 = 16 * 10^{18}$

7. Most people use arithmetic in base 10, but one can define a coherent arithmetic system in any base, not only for integers, but also for fractional numbers.
This slide shows the definition of an arithmetic system in an arbitrary base and the way

to compute the corresponding value.
Three examples, one in base 2, one in base 5 and one in base 16 are given.
It is useful to be aware, and capable of using, arithmetic in any base.

9. In the remaining part of this chapter, different examples of how information can be encoded by means of fixed length binary numbers will be given.

10. In a Vinyle record, the depth of the grove was an analog representation of the sound. In digital music, the sound is sampled a certain number of times per second (for CD's this is 44100 times per second) The number of samples per second determines, according to the Nyquist sampling theorem, the highest frequency that can be stored without aliasing (for CD's, this is approx. 20000Hz). The number of bits in each channel determines the accuracy of the representation of the amplitude of the sound (CD's use two 16 bit words, one for each channel of stereo music)

11. When music is stored as a sequence of numbers, it becomes easy to perform digital signal processing techniques to improve (or modify) the quality of the sound.
For instance, both vinyl and polycarbonate CD's can get scratched. As sound is encoded by the depth of a grove in vinyl, a scratch is almost undistinguishable from the music and will remain audible. In CD's a scratch will result in some samples that are unreadable, which is easily detectable by the software in the reader and can be hidden by the same software by interpolating between readable samples. The information is not restored, but its absence can be hidden.

12. Two techniques are commonly used to store images: bit maps and geometric patterns or other compressed image representations.

13. Storing digital images can be done in various ways. The most straightforward consists in just storing the numbers describing the properties of each pixel.
The suffix .bmp is used in MS Windows for such pictures.
The size in Mbytes of a picture stored as a bit map is obviously independent of the contents of the image, the only difference between a landscape and a uniform surface with a single circle drawn on it will be the fact that for the latter, the contents of many bytes will be identical to each other.

14. Simple images could also be described as a set of geometric patterns, which would reduce considerably the required memory size.
This is, for instance done with texts, where, instead of storing the image of each character the corresponding ASCII code is stored and in computer generated images such as Powerpoint presentations.
By analyzing pictures and extracting their features, natural images can also be represented as a set of geometric patterns, which can considerably reduce their memory size.
This slide shows some of the size reductions that can be obtained with respect to bit maps.

15. Many, more sophisticated techniques to reduce the size of stored pictures exist (e.g. Jpeg used in most cameras). Most often they reduce somewhat the quality of the images and, therefor have specific application domains. In vacation picture minute details can be erased as long as the global impression is rendered while in medical image, these minutes details are precisely what is important.

16. A very important, by volume, variety of images that need to be stored, are texts.

Rather than making images of printed pages (as fax machines do) individual characters are identified and encoded. Encoding texts and reproducing their images implies the notion of character fonts, which associate with each character code a specific graphical representation.

17. Storing separately texts and the fonts used for reproducing these texts can save a lot of memory space as illustrated by the above example. Encoding characters, rather than their graphical aspect requires 20000 times less memory!

18. While the much older Morse code (1837) used variable length encodings for characters, allowing to minimize the number of bits needed for representing a specific text, the telecommunications and computer industries adopted less efficient but more practical fixed length codes.
The TELEX network that allowed transmission of typed texts all over the world used the 5 bit per character Baudot code.
During a few years, in the 60s, 6 bit codes were quite popular because they allowed to represent the 32 letters of the Latin alphabet together with the 10 decimal digits ($2^6 = 64$)
The 7 bit American Standard Code for Information Interchange (ASCII) has been used intensively in most computers at the end of the 20th and the begin of the 21st centuries. It allowed to represent both upper and lower case letters together with digits and many commands. Many manufacturers have extended the ASCII code with an additional bit to allow also letters with accents. Unfortunately, these extensions were not standardized, still causing unexpected effects when using accents (e.g. in emails between PCs and Apple computers)
Presently , ASCII is being replaced by the 16 bit Unicode that encodes not only the Latin Anglo-Saxon alphabet, but also most of the alphabets in use anywhere in the world.

19. The 32 combinations of 5 bits used in Baudot were insufficient for the latin alphabet, the 10 decimal digits and a few commands such as return and line feed. This was solved by assigning to 30 out of the 32 combinations two different meanings, depending on the "alphabetical" or "numeric" state of the sender and receiver. The two remaining combinations were used to change the state. This trick is still used in some printers to switch between alphabetical or graphical modes.

21. The original 7 bit ASCII code had no provisions for texts using an other language than English.

22. ASCII had a rich set of commands allowing to control remote printers or to transmit texts between teletype machines.

23. Manufacturers of personal computers have extend ASCII to include accents and common symbols in the alphabet.

24. Unicode, universally used on the web, solves most of the problems associated with different alphabets by using 16 bit representations.

25. Representing numerical values by means of numbers should not be difficult. However encoding signed integers and fractional numbers requires some additional encoding.

26. As most business applications use signed decimal numbers it would be logical to use the ASCII codes for the 10 digits, the *and – signs and the decimal point to represent

numbers. This would however be a considerable waste of memory space. For instance, in a 32 bit word, 4 asci characters can be stored, meaning that numbers between 0 and 9999 could be stored, while we know that 32 bits can take over 4 billion different values!
For very specific applications, such as pocket calculators however, a 4 bit representation of decimal digits, called Binary Coded Decimal BCD) has been designed. BCD is still wasteful of memory space, but in a more reasonable way. Some business oriented programming languages like COBOL do sometimes use BCD for encoding numbers.

27. In the "absolute value plus sign" binary representation of integer numbers value and sign are handled independently from each other.
The sign is the most left bit of the representation, 0 meaning + and 1 meaning -. The remaining n-1 bits are used to represent the absolute value of the number.

28. The different encodings of integers will be described by means of examples with 4 bits. This is, of course, totally unrealistic, but allows clear graphical illustrations. The representations are shown in yellow ovals on a circle, emphasizing the modulo 2^n nature of integer number with a fixed number of bits. The represented value is shown on the outside of this circle.
For "absolute value plus sign" one observes that the number 0 has two distinct representations and that on the circular scale two discontinuities appear, between +7 and -0 and between -7 and + 0.
Absolute value plus sign representations are mainly used in devices that handle amplitude separately from sign, such as dynamic range compression for telephony.

29. To represent numbers in the "binary offset" representation, a constant positive value is added to all numbers to be represented, so that the representation is always positive.

30. On the circle of representations, the binary offset representation has only one discontinuity, between 7 and -8.
The representation 0 represents the most negative value possible and the representation of 0 is not a binary number with all zeros.

31. The "two's complement" representation is by far the most used in modern computers for representing integers. The name of the representation can not be explained without delving into the history of number representations. In short an other representation, called "one's complement" exists and is obtained by subtracting the actual value from a number with all 1s in it. For negative values "two's complement is equal to "one's complement" + 1. Not a very accurate name!!!
The main benefit of two's complement is that all arithmetic operations can be performed modulo 2^n on the representations of the numbers and will yield the correct representation of the result. This means one has no need to pay attention to signs of operands, which simplifies considerably the design of arithmetic units.

32. The circular representation presents one discontinuity, between +7 and -8. The value 0 is represented by a binary number with only bits 0.

33. As two's complement numbers are very common, it is useful to get some familiarity with them.
The first bit from the left is always the sign bit, 0 for positive 1 for negative.
Small positive numbers have leading 0s next to the sign bit. The largest positive numbers have a 1 next to their sign bit.

Small negative numbers have leading 1s next to the sign bit. The largest negative numbers have a 0 next to their sign bit.
When one wants to increase the number of bits representing a value, it suffices to reproduce to the left the sign bit as many times one want to add bits to the representation.

34. Here is an example of two's complement arithmetic: The table of addition is very simple:
0+0=0; 0+1=1; 1+0=1; 1+1=0 with carry to the left.
The table of multiplication is even simpler:
0*0=0; 0*1=0; 1*0=0; 1*1=1.

All arithmetic operations need to be done modulo 2^n , this imply means that everything to the left of the nth bit must be ignored.

35. These three slides show a formal proof of a property of the two's complement representation. Similar proofs can be established for all properties.

38. As illustrated in this example, the attractive properties of two's complement arithmetic are only valid when the range of values that can be represented is never exceeded. When this range is exceeded, the results are completely wrong!!!

39. The circular representation with 16 possible values will clarify the mechanism of overflows.
Lets first suppose that we want to add 4 to -3. We start from the representation of -3, which is 1101 and we make 4 steps clockwise. This brings us in position 0001, which is the representation of +1. Indeed -3+4=+1.
Lets now try to add 4 to +6. We start at +6 represented by 0110 and we make 4 steps clockwise. This brings us at 1010, which is the representation of -6 , so $6+4=-6$!!! This comes because while stepping clockwise, we crossed the $+7 > -8$ discontinuity, showing that we exceeded the range of values that can be represented by means of 4 bits $[-8, +7]$
What we experienced here is called "integer overflow".

40. Integer overflows occur unfortunately quite often. Some excellent Tetris players have experienced it, because Tetris stores its score in a 16 bit integer.
As an exercise: find out what the true score of this player was.
Unfortunately some other integer overflows are more dramatic. During its maiden flight, the Ariane 5 rocket had to be destroyed because an integer overflow in the output of an accelerometer had completely disoriented the rocket.

41. Detecting an integer overflow is relatively simple in an arithmetic unit. Indeed, an overflow can only occur when two numbers with the same sign are added. Adding two numbers with different signs will yield a result with an absolute value less than the absolute value of the term with the largest absolute value.
When adding two positive numbers, numbers both with a sign bit equal to 0, the result should be positive, thus have a sign bit equal to 0. If a carry occurred from the bit next to the carry bit, the result will be negative as a consequence of an overflow.
When adding two negative numbers, numbers both with a sign bit equal to 1, the result should be negative, thus have a sign bit equal to 1. Adding two numbers with a sign bit equal to 1 will result in a sum with a sign bit equal to 0, unless a carry occurred in the next bit to the right. So when two negative numbers are added an overflow occurs when there is no carry from the right.

In fact, an overflow can be detected by computing hardware-wise the exclusive or function of the carry bit on the left and right sides of the sign bit.

42. As multiplication can easily cause overflows, prevention by doubling the number of bits in the product is often useful. This can not be done after the multiplication has been done. One needs to extend the number of bits of both factors. If the factors both have n bits, the should both be brought to 2n bits by extending to the left the sign bit of each of the factors by another n bits. When that is done, the 2n bits product can be computed.

43. In addition to integer numbers, real numbers should also be encoded in computers. As between any two real values, an infinity of intermediate values exist (Bolzano-Weierstrass theorem), it is only possible to represent in a finite computer approximations of real values. Therefor an important characteristic of any encoding for real numbers is the upper limit of the relative error made when a real value is encoded. As the range of real values extends from $-\infty$ to $+\infty$, any finite encoding will also impose a range of values that can be encoded.
Before 1980 each computer manufacturer had its own proprietary encoding scheme for real numbers, and the cleverness of some of these schemes was a major asset in the competition between designers of large computers for scientific applications.
In the early eighties, the IEEE, inspired by prof. Wiliam Kahan from UC.Berkeley, proposed a standard for real encoding, named IEEE754. This standard is now in use in practically all computers. It proposes two formats, called single precision and double precision, corresponding to respectively to the types float and double in Java.

44. Real values are encoded as floating punt numbers. As shown in the this slide, a floating point number is composed of 3 variable parts and one predefined constant.
The exponent gives the order of magnitude of the value, while the mantissa gives the significant digits of the value.
A floating point number is said to be normalized, when the mantissa has no leading zero bit when the base is 2 and no leading hexadecimal 0 when the base is 16.

45. To understand the notion of normalized floating point number an every day's life example is useful. Consider an electronic scale with a three digits decimal display and four predefined scale factors, 0.01Kg, 0.1Kg, 1Kg and 10Kg.
Let's suppose that a person weighting 77.3Kg uses that scale to verify its weight.
With the scale factor 10Kg, the reading will be 008, which means a weight between 75 and 85 Kg.
With the scale factor 1Kg, the reading will be 077, which means a weight between 76.5 and 77.5 Kg.
With the scale factor 0.1Kg, the reading will be 773, which means a weight between 77.25 and 77.35 Kg.
With the scale factor 0.01Kg, an overload will occur.
It is obvious that the third scale factor gives the best accuracy, as all the digits in the display are effectively used. (there are no leading zeros).
Considering the display as the mantissa and the scale factor as the exponent, the 0.1Kg scale factor corresponds to the normalized form.

46. To evaluate the relative error between the actual real value and its representation, we consider the mantissa as an integer number, meaning that the absolute error will be less than $0.5 * base^{Exponent}$
To compute the relative error, we have to divide the absolute error by the smallest possible

real value, considering only normalized mantissas (otherwise we can artificially increase the relative error as much as we want)
If m is the number of bits in the mantissa, the smallest normalized value will be $(2^{m-1}/base)* base^E$
So, we obtain as upper limit for the relative error $base * 2^{-m}$
This result shows clearly that choosing a base larger than 2 is not a good idea.

47. If we require that all floating point numbers are normalized, we encounter a problem with very small real values while these can be important, for instance, when computing slowly converging series.
The problem is shown in the slide with a 4 bit mantissa and a base of 2. This is obviously unrealistic but is good for didactic purposes.
The smallest normalized value is $1000_2.2^{min}$ where min is the most negative value of the exponent that is allowed.
The next normalized values are each at a distance of 1.2^{min} from the previous one, which is 8 times smaller than the distance between 0 and the first one. This discontinuity in resolution close to 0 can result in convergence problems with some algorithms. The discontinuity in resolution which is a factor 8 in the slide is, of course much higher when realistic numbers of bits for the mantissa are considered.
To avoid this, some floating point formats, and especially IEEE754, allow unnormilized numbers when the exponent has the most negative value. The effect of such rule is also shown in the slide.
Again the analogy of the scale used in a previous slide can be used: supposing that the most sensitive scale factor were 0.1Kg, a child weighting 7Kg could not be weighted using the three digits of the display. Nevertheless, allowing the first digit to be 0, even if that reduces the accuracy, can be useful.

48. In the IEEE754 binary format, the base is 2, the exponent is encoded in a variety of binary offset and the mantissa is considered as a number between 1.00000_2 and 1.111111_2, the bits after the binary point are called “Significand”, their number depends on the format used. As the mantissa of a normalized number with base 2 always starts with a 1, the 1 before the binary point in the mantissa is implicit, which improves the accuracy without the total number of bits required for the floating point number.
When the exponent has the value 0, the first (implicit) bit of the mantissa is 0, the number is not normalized and gradual underflow is effective.
When the exponent has the value 255, the value is undefined or infinite (see further)

49. As an example the characteristics of the IEEE short real format will be computed here.
The smallest normalized value is reached when the mantissa is 1.0 and the exponent 1. corresponding to a value of $2^{-126} = 1.18 * 10^{-38}$.
The largest value has a mantissa as close as possible to 2 and an exponent equal to 254. The value is then $2 * 2^{254-127} = 3.410^{38}$
As the number of bits in the MANTISSA (not the significand) is 24, the relative error is $2 * 2^{-24} = 1.210^{-7}$

52. For performing calculations inside the arithmetic unit, a higher precision than the one used to encode the operand and the results is required. Such higher precision formats are defined in IEEE754. They do not need to be optimized as they are not used outside the arithmetic unit or numeric libraries.

54. For encoding 0, IEEE754 uses the not-normalized format with a zero exponent. This means that zero is encoded as a word with all zeros, which is convenient.
In scientific calculations divisions by 0 do happen sometimes. Normally this causes an abnormal termination of the program, but in some circumstances, such as inversions of very large matrixes, with some parts of no interest for the problem at study, it is preferable to continue the calculations with ∞ as result of the division by 0. An IEEE754 number with a zero significand and an exponent with all ones is used for that purpose. It should be recognized by arithmetic units and handled properly. This handling, however, requires an additional special value, called “Not A Number” (NAN) which is the result of the subtraction or division of two infinite values. A NAN is encoded with a non zero significand and an exponent containing all ones.

55. This slide gives a few examples of the encoding of simple values. It is recommended that you verify these encodings to make sure you understand all the subtleties of encoding real values.

56. Finding the minimum number of bits to necessary to encode a given dataset is a research topic with very important practical consequences. A very simple example will introduce the subject and a summary of commonly used techniques will be given in the next few slides.

57. As an example, let’s consider a dataset that represents a collection of 4 different colors. The set contains 1000 colors. It is known that the colors are not uniformly distributed, as shown, there are much more red items than yellow items.
As there are four different colors, it seems logical to encode each color by means of a 2 bit number. This results in 2000 bits to encode the entire set of 1000 colors.
If we are inspired by the Morse code, where frequent symbols were encoded by short sequences and less frequent symbols by longer sequences, we could imagine a code where red would be encoded by a single bit 0, blue by two bits 10 and the remaining two colors by the tree bit codes 110 and 111. With this code, a set of 1000 colors, with the predefined frequencies, would require 1750 bits, which is significantly less than what would be required if an uniform code is being used.

58. Can we conclude that non uniform codes are preferable?
No, this slide shows the same the same alternatives, but with similar frequencies for the 4 colors.
Here we see that non-uniform coding has no advantages, on the contrary, 250 additional bits are required.
From these examples, one can observe that non uniform coding’s become attractive when the frequencies of the different symbols is very different, but is damaging when frequencies are the same.

59. This has been studied in depth by Shannon, who proposed a way to compute the smallest number of bits needed to encode symbols with known frequencies.

60. Different coding techniques exist. Those called “entropy coding” are based upon symbol frequencies, and cause no loss of information. The first technique, used for the previous example, is called Huffman coding. An other example is the run length encoding used in the ZIP data compression programs.

61. Other compression algorithms exist. They are based upon the properties of the data to be compressed and cause a small, but acceptable loss of information, while allowing spectacular compression ratios.

3 Instructions and addresses for Sequential Computers

1. In this chapter the essential characteristics of sequential CPUs will be introduced and evaluated in relation to their applications.

2. In the first part of this course the most general format for a computer instruction was introduced. Two types of instructions were mentioned, the data processing instructions and the control instructions. In both types of instructions there are 5 separate fields, the first called "opcode" always indicates the nature of the instruction. Other fields have slightly different functions depending on the type of instruction. In data processing instructions typically two operands are combined by the arithmetic logic unit to provide a result. These instructions will contain two fields for each of the addresses of the two operands and a field for the address where the result should be stored. The last field contains the address where the next instruction can be found. In the control instructions one finds also the address of operands whose value must be compared. The result of this comparison will select one of the two choices for the address of the next instruction. In both types of instructions, it is often possible to replace the address of an operand by its value when the latter is a constant [Immediate Addressing]

3. The generic instructions in the previous screen are probably conceptually attractive but lead to weak computers. Indeed, if we assume that the addresses and words of memory have a 4-byte length and the operation codes of instructions a length of 2 bytes, performing a data processing instruction would require transfer of 30 bytes between memory and the CPU. Under these conditions the connection between memory and CPU would certainly be the bottleneck that limits the computer's performance. As the bottleneck is a direct consequence of the Von Neumann architecture that uses a single memory for data and instructions, it is often called "von Neumann bottleneck". In following parts of this course different techniques to limit the adverse effects of the von Neumann bottleneck will be presented. The Von Neumann bottleneck is also an argument for choosing a Harvard architecture for specialized computers.

4. Different techniques to reduce the effects of the Von Neumann bottleneck will be described here. The first is to use a program counter instead of the "next" field in each instruction that contains the address of following instruction to be executed.

5. Instead of providing in each instruction a field that contains the address of the next instruction to be executed, we place the instructions at consecutive addresses in memory and add to the control unit a specific register to store the address of the next instruction to be executed. This register, which is called the program counter or the P register is incremented during execution of the instruction. Next to the register P, the control unit also contains a register I intended to contain each instruction during its execution.

6. L'introduction du registre P dans l'unité de commande a pour effet de faire disparaître le champ ≪ next ≫ de toutes les instructions de traitement des données. Dans les instructions de commande, un des deux champs ≪ next ≫ disparaît, mais l'autre subsiste, puisque le rôle spécifique des instructions de commande est de modifier l'ordre d'exécution des instructions.

The introduction of the P register in the control unit has the effect of removing the field "next" from all data processing instructions. In the control instructions, one of the two fields "next" disappears, but the other remains, since the specific role of control instructions is to change the order of execution of instructions.

7. The use of a last in, first out stack to evaluate arithmetic expressions can reduce the length of the data processing instructions.

8. A stack is a structure in which data are introduced and removed solely from the top, which means that the last inputted data are those that can be fetched first. Some computers have their memory for data organized as a stack, with the arithmetic logic unit which takes its operands of the top two places in the stack and replaces the result at the top.

9. With a stack based architecture, we must no longer provide address fields in data processing instructions as operands are taken at the top of the stack and the result pushed back on the top. In return, the instruction set must be augmented with instructions designed to transfer data between any address in the memory and the top of the stack in order to bring the operands at the top of the stack and place the results in the desired memory location.

10. A stack is particularly useful to evaluate expressions because it provides automatic management of interim results in the evaluation of an expression with parentheses. First, the expression needs to be translated from the classical notation with parentheses to the Reverse Polish Notation (RPN) in which the operator is written immediately after the operands rather than between. The expression a + b is written ab+ in reverse Polish notation. The screen above shows the transcription of an expression a bit more complex with parentheses. For the evaluation of the expression, we read the expression in Reverse Polish Notation and each encountered operand is pushed on the stack. When meeting an operator, it is applied on the two operands at the top of the stack which are replaced by the result of the operation. On the screen are successively shown the status of the stack after the operands a, b, c and d have been pushed on the stack, after c and d have been multiplied, after b was added to the product of c by d, and finally after the preceding result was multiplied by a.

11. This screen simply shows the result of the evaluation begun on the previous screen.

12. We have seen in the previous two screens how a stack architecture simplifies the evaluation of complex expressions and makes the corresponding code shorter since it must not handle intermediate results. Nevertheless, one can wonder whether this kind of architecture helps to avoid the effects of the von Neumann bottleneck. Indeed, if we consider the code that corresponds to evaluating a simple expression like the one shown on the screen, we see on the left side, for a stack architecture, three transfer instructions which each contain an address and operation code and one instruction that contains only an operation code. For traditional architecture with three addresses in the data processing instructions, just three addresses and one operation code are needed, in fact, three operation codes less than with a stack architecture. One can conclude that the stack architecture becomes only interesting when the program contains many complex expressions. As software engineering encourages us to write simple and easy to understand programs, very complex expressions are not widespread in modern programs and the benefits of stack architectures is negligible or even negative.

13. Since the management of interim results using a stack does not offer great advantages and on the contrary, as will be shown in the next chapter, stacks are sometimes a source of unnecessary complexity, a better way to avoid having to write all intermediate results in memory has been invented. It consists in processor registers.

14. Inside the CPU a small number (typically 8, 16 or 32) of addressable registers with fast access are provided to temporarily keep intermediate results. As the number of registers is small, the field to contain their address in the instructions will be much shorter than those intended to contain memory addresses. To transfer data between memory and registers, specific transfer instructions have to be added to the instruction set.

15. The availability of registers in the central unit will optimize data processing instructions: instead of using the memory to store the operands and the results, registers can be used for this with a double benefit, firstly access to registers is significantly faster than a memory access and the register addresses are much shorter. Data processing instructions with three addresses that refer all three to registers exist but sometimes only two explicit addresses are needed when the result will take the place of an operand.
In some instruction sets, one of these two addresses is a memory address while the other is a register address. This technique reduces the number of transfer instructions to be used in programs.

16. Another way to reduce traffic between memory and CPU is to adopt a specific organization for control instructions.

17. Comme les opérandes à comparer sont le plus souvent les résultats de calculs antérieurs, on ajoute à l'unité arithmétique un registre spécial, appelé ≪ registre des codes de conditions ≫ dans lequel, à la fin de chaque calcul, certaines propriétés remarquables du résultat sont consignées.
Les propriétés qui sont consignées dans ce registre sont, le plus souvent, un résultat nul, un résultat négatif, un dépassement arithmétique ou un report au delà du bit le plus significatif (pour permettre des opérations arithmétiques sur des nombres plus larges que l'unité arithmétique).
Les instructions de commande choisissent alors l'instruction suivante en fonction des codes de conditions, plutôt qu'en fonction d'une comparaison effectuée entre deux opérandes. Dans l'instruction il suffit dès lors de prévoir quelques bits pour indiquer les codes de condition qui doivent être pris en considération ce qui prend évidemment beaucoup moins de place que s'il fallait inclure l'adresse de deux opérandes.

As the operands to be compared are often the results of previous calculations a special register called "Register of conditions codes" is added to the arithmetic unit to record, at the end of each calculation, some outstanding properties are the result. The properties that are recorded in the register are, most often, a null result, a negative result, an arithmetic overflow or deferral beyond the most significant bit (to allow arithmetic operations on larger numbers than unity arithmetic). Control instructions then choose the following instruction in function of the condition codes rather than in function of the comparison between two operands. In the control instruction it is therefore sufficient to provide a few bits to indicate the condition codes that must be taken into consideration which obviously takes much less space than the address of two operands.

18. One last way to reduce traffic between memory and the CPU is to increase the amount of work done by one instruction. Vector instructions illustrate this principle.

19. In many programs one finds loops like the one shown above. In fact, the same arithmetic operation on must be performed on all the elements of a vector (e.g. represented by the array type). With the kind of statements seen so far, it would be necessary for each element of the vector to fetch in memory the arithmetic instruction, the one that increments i, and finally, the control instruction that checks if the exit of the loop can be chosen. In the case of long vectors, the same instructions are to be transferred many, many times.

20. Instead of transferring a large number of times the same instructions, there could be an extra field in some instructions that would specify the number of times that the instruction must be repeated with consecutive vector elements. Such instructions are called "vector instructions" and contributed to the outstanding performance of the computers that are usually called "supercomputers", which are mainly used for numerical calculations. Such vector instructions can also come in handy in less prestigious applications. They could, for example, be very valuable when it comes to fill in the pixels of a graphical display or to search for a word in a long text.

21. Les différentes techniques visant à améliorer les performances d'ordinateurs en réduisant le trafic entre mémoire et unité centrale décrites dans ce chapitre s'appliquent à tous les styles de programmation, qu'il s'agisse de programmation en langage élémentaire ou en langage de haut niveau. Pour optimiser l'exécution de programmes initialement écrits en un langage de haut niveau, des améliorations additionnelles peuvent être apportées au répertoire d'instructions.
Il s'agit plus spécifiquement, d'assurer un mécanisme efficace pour appeler des fonctions, de prévoir des techniques d'adressage qui permettent d'accéder à des structures de données complexes et de permettre la gestion efficace des règles d'accessibilité propres aux langages à structure en blocs.
L'appel de fonctions est implémenté au niveau de la machine par des instructions de commande qui permettent de sauter vers un morceau spécifique du programme (une sous-routine), d'exécuter cette sous-routine et de retourner dans le programme original à l'instruction qui suit celle qui avait causé le saut vers la sous-routine. La manière dont l'unité de commande "retient" l'adresse de l'instruction qu'il faudra exécuter après la fin de la sous-routine (≪ l'adresse de retour ≫) peut avoir une grande influence sur l'efficacité d'exécution de programmes écrits en langage de haut niveau.

The different techniques to improve computers performance by reducing traffic between memory and CPU mentioned in this chapter apply to all programming styles, be it programming in elementary or in a high level language. To optimize the performance of programs written initially in a high level language, additional improvements can be made to the instruction set. This is more specifically to provide an efficient mechanism to call functions, to provide mapping techniques that allow access to complex data structures and to allow an efficient management of the accessibility rules dictated by block structured languages. Calling functions is implemented at the machine level by control instructions that can jump to a specific piece of the program (subroutine) to run this subroutine and to return to the original program at the instruction following the one that caused the jump to the subroutine. The manner in which the control unit "remembers" the address of the instruction that will run after the end of the subroutine (the “return address”) can have a great influence on the efficiency of execution of programs written in high level language.

22. The technique used in virtually all modern computers is to use part of the data memory to retain the return addresses. This part of the memory is organized in a “first in, last out” structure called “stack”. When executing the jump instruction to the subroutine, the address of the next instruction is pushed on the stack and a jump to the first instruction of the subroutine is performed. At the end of the subroutine, there is a specific instruction to return to the calling program. This statement is actually a jump instruction whose destination is taken from the top of the stack.

23. Thanks to the use of a stack for storing the return addresses, recursive subroutine calls become possible. For each separate call, a return address is stored on the stack.

24. A major advantage of high-level languages is the ability to organize data into complex structures such as tables, structures, linear lists, binary trees and even quite arbitrary graphs. For quick and comfortable access to such data structures, it is essential that the computer has various addressing modes, well adapted to the data structures.

25. The simplest addressing mode is direct addressing. This is the mode that has been implicitly used in all previous explanations in this course. It consists simply in directly mentioning the desired address in the appropriate fields of the instructions. This mode is frequently used for program memory addresses that are found in the jump instructions, as well as in data processing instructions for referencing simple, unstructured variables.

26. Indirect addressing consists in mentioning in the instructions, instead of the address of the operands, the address where the address of the operands can be found. This addressing mode corresponds in many high-level languages with the use of pointers. A pointer is a variable that contains the address of another variable. Instead of directly referencing the latter, one reference in the instruction the pointer through which one can access the variables themselves. This method allows to build structured variables and to pass to functions the address of an actual parameter rather than a copy of the parameter (passing parameters by reference instead of by value). Such passing by reference allows the function to change the value of its actual parameters, while passing by value limits the influence of the function to the formal parameters that are only copies of the actual parameters.

27. Sometimes indirect addressing is done through a register rather than through the memory. The address contained in the instruction is that of a register which contains the address in memory of the operand desired. This technique is, for example, used to manage a stack such as the one mentioned in the study of subroutine call instructions: A special register, called “stack pointer” contains the address of the first free place on the stack. The subroutine call instruction writes the return address to the address contained in the stack pointer and increments the contents thereof. The subroutine return instruction first decrements the content of the stack pointer and then performs a jump to the address in memory at the location indicated by the contents of the stack pointer.

28. There is talk of relative addressing when the address contained in the instruction is added to the contents of a register before being used to reference a word in memory. Relative addressing is primarily used to access elements of an array (array) or structure (record). For tables, the address of the first array element (corresponding to the index

0) is placed in the address field of the instruction while the value of the index is placed in a register. For structures, on the contrary, it is often preferred to place the address of the beginning of the structure in a register while the address of the specific field is placed in the instruction. This addressing mode is also used in the jump instruction using the program counter as reference register. This technique then allows to jump forward or backward relative to the location of the control instruction. As in most structured programs jumping from one end of the program to another is avoided, one can use mostly short addresses in the control instructions.

29. Most high-level languages are block structured and each block may possess local variables. This organization helps limiting the scope of each variable declaration in the area in which the variable is actually used and thus avoid errors that come from the indiscriminate use of homonyms variables. As each block can itself contain other blocks, a nontrivial management of the scope of variable declarations is needed. Part of this management can be done by the compiler in translating the high-level language into machine language, but there is all the dynamic allocation of memory for the different variables that must be made during program execution. Instructions that optimize this management contribute to an efficient execution of programs written in high level languages.

30. In most high-level languages, a program is a set of nested blocks. C, C++ and Java blocks are delimited by braces.

31. Whenever a block is activated by a function call, space for its local variables is reserved on the stack in data memory which is also used for keeping track of the return addresses. When a block is no longer in use (because of a return statement) the memory space reserved for the local variables is made free. The dynamic management of the stack part of the data memory is done by means of pointers called ≪ dynamic links ≫
In addition to local variables, functions also need access to global variables. For this, we will provide above the dynamic links space for an other pointer, the static link that contains the start address of the block in which global variables are located. Since all functions are defined at the same level in C and derived languages, the management of static links is trivial. In languages that allow the definition of functions within other functions such management is substantially more complex.

32. Parameters are a major asset of functions in high-level languages. It is therefore essential to provide mechanisms that ensure the efficiency of parameter passing.

33. Parameters allow you to use the same function with different variables. A little analogy will clarify this concept, imagine a very organized person writing instruction sheets for all activities of daily living. Imagine that this person writes an instruction sheet to invite a friend to the restaurant. At the time of writing the sheet the identity of the friend is not yet known and will appear under the heading "my friend." This substitute for the actual name of the friend is called "formal parameter". When the sheet is used, the identity of the friend is known, and we replace the formal parameter by the actual name of the friend. This real name will be called "actual parameter" In the example shown on the screen, the function DoCCC has a formal parameter xyz which is substituted in the first use of DoCCC by the actual parameter abc and in the second, by the actual parameter uvw.

34. There are several different methods to associate current and formal parameters. With the first method, called "passing by value", the value of the actual parameter is copied on the stack before the call. In this way, the function can possibly change the value of the formal parameter without this affecting the actual parameter value, since the formal parameter is a copy of the actual one. If the parameter is a structured type, possibly large, one must be aware that before any call, the parameter values must be copied, which can require a lot of space and time. The fact that the formal parameter is a copy of the actual parameter is a protection of the calling program against the actions of the function. One could, for example, imagine a function that calculates the sine of an angle beginning by bringing the angle in the first quadrant, but it is unlikely that the author of a program appreciates when that angle is amended by the sine function. Since only the value of the actual parameter is used, it can be any expression whose value belongs to the type of the formal parameter.

35. The other popular method for passing parameters is called "by reference". In this case, the calling program does not provide the value of the actual parameter but its address. In this way, using indirect addressing, the called function can directly access the actual parameter and modify its value. This is the mode normally used with structured types of parameters because it avoids the waste of time and space associated with the copying of the contents of the actual parameter. Whatever the size of the actual parameter, it will only require the size of an address on the stack.

36. Most functions return a value to the program that called them. This is also done through the stack. The calling program before placing parameters on the stack, reserves a place for the value that the function will return. Putting the value returned below the parameters allows, after the execution of the function, to remove from the stack all that concerned the function call, except the returned value.

4 SP0: A Stack Processor

1. In this chapter, intended as an illustration of some concepts introduced in the preceding chapter, a machine with a data memory organized as a LIFO stack is described.
This machine doesn't exist as a commercial product, but it has all the essential properties of the many stack machines that had a significant commercial success between 1970 and 1990. Such machines are no longer build, but modern machines emulate their operation. This is the reason why this chapter is an important part of this course.

2. The computer described in this chapter has a memory for programs (code), a memory for data, structured as a LIFO stack (data) , a program counter (P) , an instruction register (I), a base register (B), a stack pointer (T) and an arithmetic unit that fetches its two operands from the top of the stack (TOS) and pushes back its result on the stack (an arithmetic operation results in a reduction of one of the height of the stack)

3. The code memory is randomly addressable and is read only during program execution. The program counter P mostly holds the next execution address. Once the instruction fetched into the instruction register I, execution is conducted by electronic signals that control different computer parts.
This operation is illustrated using a C++ program simulating the operation of the computer. This basically helps to accurately illustrate the operation of a computer without using the electronic concepts. Additionally it acts as an application to simulate the execution of object code written for this computer.

4. The data memory is randomly accessible. The base register B and the stack pointer T allow to manage the stack memory. The arithmetic unit uses the two top values and replaces them with the result of the arithmetic operation.
The data memory is used for variables of active blocks, temporary variables, return addresses of subroutines, static and dynamic data links and finally for procedure parameters.

5. Among the instructions for processing data, the CHS instruction changes the sign of the element that is at the top of the stack (TOS) and the EXC instruction exchanges the values of the two upper elements of the stack.
Besides these two instructions there are four traditional arithmetic instructions and six comparison instructions that compare the values of the two upper elements of the stack and replace them with a Boolean value.
All these instructions assume a single format for the operands, as, for example, 32-bit integers (int). If you want to allow the use of other data formats, such as floating point numbers (float or double) one should provide additional instructions and even instructions to perform conversions between different formats.

6. Five instructions for transferring data are provided.
The first, LIT, allows to place a constant value at the top of the stack (TOS).
The LOD and LDI instructions copy the value of a variable at the TOS. The first uses direct addressing to access the variable while the second uses indirect addressing.
The STO and STI instructions remove the top element of the TOS to incorporate its value to a specified address directly for the first indirectly for the second.

Transfer instructions using indirect addressing are primarily intended to access functions passed by reference parameters.
The INT instruction is used to change the value of the stack pointer by adding a constant which can be positive or negative. This instruction is mainly used to reserve space on the stack for active blocks.
The LAD instruction allows to calculate a local or global address in the data memory and to place this address at the TOS. This instruction is used to define a actual parameter passed by reference.

7. Six control instructions are provided:
The JMP instruction is a simple unconditional jump.
The JPT and JPF instructions are conditional jumps. These jumps are performed only if the specified Boolean value is at the TOS. The TOS value is always popped out of the stack no matter if the jump is performed or not.
The JSR instruction calls a subroutine and the RET instruction allows to return from the subroutine.
The HLT instruction stops the program execution. Normally such a statement is not part of a processor instruction set, since it never stops. Every program ends by returning to the operating system. However, as this is a simulated processor, a statement has been provided to stop the simulation.

8. The simulator uses an enumeration to define the operation codes of instructions.
The instruction itself is a three-field structure, the first of which contains the operation code and the two following can contain the operands.

The struct type is used to declare and give a name to aggregations of variables of differential types. The variables that are part of a structure are named by giving the

name of the structure, followed by a period and the name of the structure field.
So, if I is a variable of the type Instr declared in the slide above, you can reference the entire statement with the variable name I or the different fields of I by the names I.opc , I.I and I.a

9. Memories and processor registers are represented in the simulator by variables.

10. The heart of the simulator is a do ... while loop in which the instructions are read one by one from the code memory (Code) and loaded into the Instruction register (I). The operation code of the instruction contained in the instructions register is then used to a switch statement that allows to treat separately the simulation of different instructions. The execution simulated in the do ... while loop continues until the Boolean variable Running equals false, which corresponds to the simulated running the HLT instruction

11. In the following slides, the role of register T and the stack as the location for temporary variables will be illustrated.

12. The CHS instruction changes the sign of the variable at the TOS.
The EXC instruction swaps the positions of the two elements that are at the TOS. Note that the variable just above the top of the stack is used as a temporary variable during the exchange.
The ADD instruction adds the values of the two elements that are at the top of the stack and replaces them with the value of their sum. At the end of this operation, the total stack height has been reduced by one.

13. On this slide you can see the implementation of three comparison instructions.
The first one checks whether the values of the two elements situated at the TOS are equal. If equal the Boolean value true is put in the TOS, if not false is put in the TOS.
The other comparison instructions are similarly implemented.
Since the two values at the top of the stack are replaced by a single Boolean value, the total height of the stack is reduced by one by the execution of a compare instruction.

Note:
The C code shown in the above slide and some of the following slides is not syntactically correct because a Boolean value can not be assigned to a variable of type int. Ignoring this makes the code much more readable for programming novices, the syntactic error will be ignored in favour of the clarity of semantics.

14. The unconditional (JMP) and conditional (JPT and JPF) jump instructions use the operand to reference the code memory address to where the program (eventually) will be jumping to.
In the case of unconditional jump (JMP) , execution replaces the contents of the program counter P by the address to which the programs jumps.
In the case of conditional jump instructions (JPT and JPF) , the program counter P is only replaced if the stack top equals the specified Boolean value.
In every of the above cases, the Boolean value that was at the top of the stack is removed, (popped).

15. With instructions explained so far it is already possible to show how a compiler might translate some common C language instructions to make them executable by the processor described in this chapter.

In this slide we see how an if statement is translated.
First we must evaluate the Boolean expression B. This assessment will conclude with true or false at the stack top.
If true, the JPF instruction will not cause jump and the code for the S1 block will be executed. After S1 there is an unconditional jump to the label L2, which is the result of the program beyond the if statement.
If instead the term B was false, the JPF statement will cause a jump to the L1 label above the code for the block S2. This code is immediately followed by the L2 label and rest of the program.

16. The translation of a while loop is not much more complicated than the if selection:
It also begins by evaluating the Boolean expression B which leaves a true or false at the stack top.
If the value at the stack top is false, and therefore the body of the loop should be executed, execution jumps to the L2 label that marks the beginning of the rest of the program with a conditional jump JPF .
If instead this value is true, which means that the body of the loop must be executed at least once more, the JPF instruction performs no bond and the code for S runs. At the end of S there is an unconditional jump to the start of the evaluation of B, possibly passing again through the loop.

17. Compiling a do ... while loop is even simpler:
The performance begins with the code corresponding to block S. Then the Boolean expression B is evaluated leaving the true or false at the top of the stack. JPT a statement to jump if B is true, to the label L1 which is the beginning of the code block S.
If JPT instruction causes no jump, execution continues beyond this statement.

18. In the following slides, the use local variables will be illustrated.

19. Pendant l'exécution d'une fonction, les variables locales de cette fonction se trouvent sur la pile, dans le bloc d'activation de la fonction. Le registre de base contient alors l'adresse du début de ce bloc. L'adresse d'une variable en mémoire est définie par les deux opérandes que l'on trouve dans toutes les instructions qui référencent des variables. Le premier de ces opérandes, l, à la valeur 0 pour les variables locales et la valeur 1 pour les variables globales. Le second, a, est l'adresse de la variable au sein du bloc d'activation de sa fonction.
Pour effectuer des calculs avec des variables ou pour les passer comme paramètres à une autre fonction, il faut pouvoir copier leur valeur au sommet de la pile. Pour assigner le résultat d'un calcul à une variable, il faut aussi disposer d'instructions pour déplacer des résultats du sommet de la pile vers l'adresse d'une variable spécifique.

During the execution of a instruction, local variables of the instruction are on the stack, in the stack frame of the instruction. The base register (B) now contains the address of the beginning of this frame. The address of a memory variable is defined by the two operands found in all instructions that reference to variables. The first operand, l, has the value 0 for local variables and the value 1 for global variables. The second, a, is the address of the variable in the activation of the function block.
To perform calculations with variables or pass them as parameters to another instruction, you need to copy the value at the stack top. To assign the result of a calculation to a variable, one need instructions to move the results of the stack top to the address of a specific variable.

20. The LOD 0,a instruction places value of a local variable at the TOS.
The address of this local variable is obtained by adding the contents of the base register (B) with the instruction's operand.
The STO 0,a instruction removes the value stored at the TOS and assigns it to the variable whose address is obtained by adding the contents of the base register (B) with the instruction's operand.

21. The LIT a instruction simply sets the value of the constant a at the TOS.
The INT a instruction allows to reserve a places on the stack by adding value a to the content of the stack pointer. The constant a may also be negative, it then frees up the space occupied by the stack.

22. To illustrate the use of local variables and the stack, the expression's evolution will be described in this slide.
The compiler changes the arithmetic expressions notation. The traditional notation (algebraic) is replaced by the notation called "reverse Polish" (RPN) in which the operator, instead of being between the operands is written after them. Thus the expression (a + 3) * b transforms into a 3 + b *. This notation, besides being easily translated into computer programs, has the advantage of never requiring parentheses (&).
To translate a phrase written in reverse Polish notation into a program, one needs to read from left to right. The operands are placed at the TOS and the operators are applied to the two elements which are at the TOS.
On the slide you can see the stack's status in four successive evolution stages performed by the program shown at the right.
When the final value of the expression is at the top of the stack, the value remains to be assigned to the variable x. It is the role of the instruction STO 0, x where x is the address of the variable x in the stack frame of the function.

23. Another example showing how the local variables and the stack are used in conventional programming constructions is the assembler translation of a for loop. The slide shows the arrangement of the various operations during the execution of a for loop on the left, while on the right, the corresponding program in assembler is shown.
As an exercise, the reader is invited to review the operation of this program by drawing the successive stages of the stack during the execution of the loop with values for M and N respectively 1 and 3.
The reader is also invited to note that the automation of the management of the stack are impractical in this example because it must regularly exchange the top two elements and modify the content of the stack pointer.

24. The stack is also used to keep subroutine return addresses and for dynamic and static links.

25. Before calling a procedure the base register B points at the start of the activation block of the calling procedure while the stack pointer T points at the last place occupied by this block.
If the called instruction has no parameters, the first three available places above the active block of the calling procedure will be used for the return address, the dynamic and static link. The following places will be used for the local variables of the called procedure's (it's active block).

26. The operation of call instructions and return from subroutines will be explained in three stages. This slide shows the first of these steps, the one where the jump to subroutine and return from are treated.
When calling the subroutine the content of the register P, which contains the address of the instruction following the call is placed on the stack and the starting address of the subroutine is copied to the P register in a way that the first statement of the subroutine is executed immediately after the call.
Upon return, the return address is read from the stack and copied to the register P.

27. After managing the return address, the dynamic link is managed. The content of the base register should be copied just above the return address and place in the base register the address of the freshly created dynamic link.

28. The slide shows the execution of a subroutine call instruction. The B content is written on the stack just above the return address and the address where the writing was performed is placed in the register B.
When executing the instruction to return from a subroutine begins by updating the contents of the register T, to release all the space on the stack that was used by the subroutine which one returns. Since, during the call, before the contents of the register T has been changed, the address equal to the contents of T increases 2 was placed in B, it now simply reload T in the B content decreased 2.
Then it replaces the content of B by the address contained in the dynamic link.

39. The management of static links is, as already stated in previous chapter is quiet simple when limited to programs initially written in C, C++ or Java because in these languages all functions are set at the same level, which means that variable that is not local to a function belongs to the global block, which from the point of view of the structure of the program immediately surrounding the local blocks. All static links must return to the beginning of global block.
We have already seen that in the variable address the part I allows to distinguish local variables ($l = 0$) global variables ($l = 1$). In the call instructions to subroutines, a similar convention is used in the sense that besides of the operand a indicating the address in the memory for the start of the subroutine programs, it also provides the second operand which in the original programs written in the above mentioned languages, is always 1 to indicate the names of the called functions are still reported overall relative to the location from which they are called.
It may seem absurd to allow room for an operand that always has the same value. But a computer can be made to run programs that were originally written in a language where function declarations are not all done at the same level and for which the operand I is useful.

40. The function base described above is used to track the chain of static links. Simply, in this particular case with languages derived from C, to observe that this function returns the address of the beginning of the global block if the current setting is 1 and the start address of the local block if the parameter is 0.

41. The function base can be used to manage static links. It is sufficient to observe that before the base register B has been updated for calling a subroutine, the base function, called with a current setting equal to 1 provides the address the beginning of the global block.

42. The slide shows a synthetic manner the implementation of the instructions to call a subroutine and to return.

43. Note that the call instruction to a subroutine leaves the stack pointer T directed to the address above the return address. One may wonder why T is not incremented by this instruction. The reason is simple: since in any case the programmer of the subroutine must reserve space on the stack for the activation block of that subroutine, it will insert at the beginning of the subroutine an INT instruction . He needs to give the operand of this instruction a value equal to the size of the activation block plus three places for the return address, the dynamic and static link.

44. We can now return to the data transfer instructions and no longer be limited to global variables as had been done previously. In the code that describes the execution of these instructions, simply replace B by base (LI).

45. After reviewing the instructions to call a subroutine and back, it remains to consider the instructions that are specifically designed to facilitate the transition of parameters.

46. In previous chapter, it was explained that the transition of the parameters is made through the stack on which the calling function places the value or the address of the current parameters before calling the function.
When it comes to pass parameters by value, the necessary instructions have already been described: to place the value of a variable at the TOS, it can use the LOD I.a instruction, or any instruction that evaluates the value of an expression and leaves the result at the TOS.
The called function can use the values assigned formally assigned parameters only if it uses the instruction LOD 0,a, with negative values for a, to set the parameter value on the top of the stack.
Using parameters passed by reference requires the use of indirect addressing (see below).

47. The LDI and STI instructions allow to transfer data between the TOS and any data memory location where its the address itself is stored in the memory where the instruction fethes its information.

48. The LDI and STI instructions are implemented as LOD and STO instructions but indirectly determine the operand's address, not directly. In the LOD and STO instructions the operand's address is directly in the instruction, while for LDI and STI instructions holds the address where the operand's address can be found. The address contained in the instruction is normally composed of two parts, the part I that indicates whether it is a local or global variable and the part a that determines the location of the variable within a specific activation block. On the other hand the addresses in the memory are absolute addresses that can reference to any location in data memory. The conversion of an address to two components into an absolute address is realised by adding the address of the start of the activation block to the address in this block.
The instruction LAD converts the address in two components it contains absolute address space and the address at the top of the stack. It is used to place the address of the current passes through reference parameters on the stack before calling a function.

49. The above slide shows the assembler translation of a simple C function that has three parameters, both x and y parameters are passed by value and parameter z is passed by reference. The function has no return value, but assigns to z the value equal to the sum

of x and y values.
The INT 3 instruction by which the function begins reserves space on the stack for the return address, for the dynamic link and the static link but does not reserve space for the activation block since there are no local variables.
Both instructions LOD places the values of x and y at the top of the stack, the ADD instruction removes these values from the stack but instead leaves the value of the sum and finally the STI instruction removes this result from the TOS and assigns it to variable z whose address is the third actual parameter of the procedure.

50. The slide shows the assembler translation of a call to the function described in the previous screen.
The actual parameters for the call are constant 2 and the local variables b and c respectively. The parameter c is passed by reference, that is to say that the current parameter is a pointer to c.
In the call sequence we first see the establishment of the constant 2 using the instruction LIT. Next the value b is placed on top of the stack by the instruction LOD 0, b and finally the address of c is load with the instruction LAD 0, c. The call sequence is followed by the actual call instruction JSR 1, P. It will be noticed, after the return of the function INT -3 instruction serves to remove the parameters from the stack.

51. In the example shown in slide, a function that calculates the sum of the values of two parameters is shown as well, but instead of returning the value of the sum by a parameter passed by reference, it is the function itself returning the value of the sum. The function also uses a local variable t to temporarily retain the value of the sum and perform other operations that are not shown on the screen.
Since there is a local variable, the function begins with INT 4 instruction.
Then, as in the previous example, the sum of the values of the parameters is calculated and the value of this amount is assigned to the local variable t that is stored right after the static link.
As explained in the previous section, a space is provided on the stack for the return value, just below the call parameters. Before the return of the function, the value of t is replaced temporarily on top of the stack to be copied in the space provided.

52. This slide shows an example of the use of the function described in the previous slide.
We want to assign to the variable z the value obtained by adding the value of the variable x and the value returned by the function P when called with the current settings 2 and b.
The program begins by placing the value of the local variable x to the top of the stack through the instruction LOD 0, x.
Booking a room for the return value of the function is done by placing the value 0 to the top of the stack using the LIT 0 instruction.
Then the two parameters calling P are placed on the stack and the P function is called.

53. After the return of the function the parameters of this stack is removed by the INT -2 instruction. The value calculated by the function is found at the TOS, just above the value of x.
It therefore remains only to add these two values and assign the result to variable z.

54. A recursive function is a function that can call itself.
Although for performance reasons it is not recommended to calculate the factorial function recursively, we can do it to have a very simple example of the use of recursion.
The factorial function written in C is simple since it literally transposes the mathematical

definition of the factorial function in the C language

55. The slide shows the translation in assembler, with comments, of the recursive factorial function.
It is strongly recommended to the reader to follow this function step by step when it has to calculate 3! or 4! It's particularly important to draw the successive states of the stack during the execution to profoundly understand the performing mechanisms of a recursive function.

5 RP0: A Register Processor

1. In the previous chapter we could see that a stack architecture is excellent to manage activation blocks and evaluate expressions but we also saw that at certain times, such as during the execution a for loop, the stack organization can be very hard. In this chapter a register architecture will be described. This architecture can emulate a stack architecture where this is useful, but also avoids heaviness caused by the rigidity a stack.
It would be perfectly possible to describe this processor as was done in previous chapter (this is even be an excellent exercise to check understanding of these two chapters) but another method closer to the physical reality of the processor will be used here. This method of describing all computer devices through which data can flow and analyse these transits in time. The descriptions used in the previous chapter and in this one are complementary, the first being more abstract and essentially intended to understand the logic of the instruction set, regardless of the speed of execution, while the second, more concrete, consider not only the logic of the instructions but also the temporal sequence of their execution.

2. The processor RP1 does not exist commercially, but its architecture, reduced to the minimum necessary to illustrate architectural concepts, inspired by that of the Motorola 68000 family.
On the slide, from left to right, we can see
- A set of eight registers, called D0 to D7, intended to contain data
- An arithmetic unit (ALU) with, at its inputs, two temporary registers T1 and T2.
- A CC register for holding the properties of arithmetic results
- A control unit (CU) with its instructions for register (IR)
- An arithmetic unit to increase or decrease addresses
- A set of 8 registers to keep addresses, called A0 to A7; In the last three registers A have specific functions. Thus, A7, also called PC is the program counter, which keeps the address of the next instruction to execute.
- The central computer memory.
All these components are interconnected by means of two 'bus'. These are the wire sets to transport the content of a register to another.
Both bus present in RP1 are the data bus, shown in red at the top of the slide and the address bus, shown in green at the bottom of the slide.
Besides the busses a number of wires that go from the control unit to the different computer parts. These wires allow the control unit to transmit its commands.

3. To describe a transfer between memory and CPU, we must first consider the clock that determines the timing of all computer operations. This clock is part of the control unit and defines fixed time intervals called "clock cycle". In contemporary computers, the duration of a clock time is of the order of nanoseconds. A write or read operation in the

main memory takes a few clock cycles. The totality of the required clock cycles is called a “memory cycle”.
A typical read cycle takes place as follows:
- At time t1, the address of the data to be read is sent from one of the A registers to the memory via the address bus and a read command is sent by the control unit to the memory.
- At time t2, the control unit asks the memory the status of the read operation. If ready, proceed to t3. Otherwise, the control unit inserts a waiting time tw before the control unit asks again the memory for a status.
- At time t3 the read operation is completed by the transfer of the read value via the data bus to a central processing register.

4. A write cycle begins with a time t1 during which the address is placed on the bus addresses, the data on the data bus and a write command is given by the control unit. The rest of the memory cycle is used partly by the memory to execute the write operation and partly by the control unit checking the memory status at each clock cycle. A write cycle also ends with a clock cycle t3, during which nothing here described happens.
If we consider normal memories that have an access time of about 50ns we understand that for each read operation many tw waiting times will be inserted. This finding helps explain why the clock speed is a poor indicator of the performance of a computer, since an increase in the clock speed may have the primary effect of increasing the number of times tw to insert into each memory cycle.

5. The RP1 data processing instructions operate on data in the processor registers. For most instructions this is exclusively for Dx data registers, but for some instructions, registers Ax addresses are also allowed. This latitude is indicated by the use of Rx notation meaning Dx or Ax. The RP1 data processing instructions have two fields to reference operands, the result of the operation being systematically relocated to the place where was the first operand.
Besides the four arithmetic operations, RP1 has a CMP compare instruction which is in fact identical to the subtraction operation but does not replace the first operand with the result. The properties of the result of the arithmetic operation is noted in the CC register. This is so for all other information processing instructions.
Finally, there is an instruction, ADI that can add to the contents of any data or address register the value of a constant included in place of the reference of the second operand in the instruction.

6. In a register machine, one needs transfer instructions to:
- load constants into registers (LDI, Rx, a)
- copy the contents of a register to another (LOD Rx, Ry)
- copy the contents of a word from memory into a register (LOD Dx, ...)
- copy the contents of a register in a word of memory (STO Dx, ...)
The architecture of RP1 offers for the last two transfer instruction categories two alternative transfer instructions to reference a memory address, the direct addressing and relative addressing to the contents of a address register.

7. Besides the unconditional jump (JMP) one can find among the control instructions a series of instructions for conditional jumps. As explained in Chapter 3.2, it’s the status of specific bits of the CC register that determines if the jump is to be performed or not. Only a few RP1 instructions for conditional jumps are shown here. They are those that depend on the bit Z (zero), N (negative) and V (overflow during the calculation).

Among the instructions for managing the sub routines there is obviously the call subroutine (JSR) and the return from subroutine (RET) but there are also instructions for managing dynamic links (LNK and ULK).
Finally, there is, similar for the stack processor described in the previous chapter, a HLT instruction useful if one wishes to write a simulator for this processor.

8. The first instruction that will be analysed here is the instruction LOD Rx, Ry which copies the contents of the register Ry to the register Rx.
It requires only a single memory cycle. The first time the contents of the program counter PC is on the bus addresses and a read command is given.
The second time, the contents of the program counter that was still on the address bus is increased by one with the little arithmetic unit associated with the address registers, and rewritten in PC. Thus it is already preparing the start of the execution of the next instruction.
At the time t3, possibly after some time of tw unrepresented waiting, the instruction is transferred from the memory in the IR register to be decoded and executed. It is not until that the controller 'knows' what the instruction is that the instruction is executed.
To copy the contents of Ry in Rx not additional memory cycle is required, just one clock cycle is needed to copy the contents of the register Ry in the temporary register T1 and a second clock cycle to copy the contents of this temporary register into Rx. Both transfers between T1 and another register are obviously via the data bus.
The copy of the content of Ry into Rx can not be done without intermediate step as the 8 D registers and 8 A registers are organized as addressable memories in which, each time, only one register is accessiible.

9. Another instruction that requires only one memory cycle and which introduces an important architectural concept is the ADD Dx, Dy instruction that serves to add to the contents of Dy to Dx.
The first three clock cycles are obviously identical to those described for the LOD instruction Rx, Ry, since the nature of the instruction to be executed is known only from the end of t3.
At t4 the content of Dx is transferred to the temporary register T1, then, at t5, the Dy content is transferred to the T2 register and the command to add is given to the arithmetic unit.
Normally there should be a t6 to transfer the result of the addition in Dx. Instead, it immediately starts executing the next instruction and takes advantage of the t2 of the instruction to transfer the result of the addition. Indeed, during the cycles t1 and t2 of the first memory cycle instruction execution, the data bus is not used and can be used to terminate execution of the previous instruction.
This temporal superposition of the end of the execution of one instruction and the beginning of the next is called “pipelining” and is a technique that is used extensively in all modern processors to increase performance.

10. The instruction CMP Dx, Dy compares the values in the Dx and Dy registers. It requires 5 clock cycles and is therefore completely identical to what was described for the ADD Dx, Dy instruction, except that the command provided to the arithmetic unit is a subtraction instead of an addition.
The main difference is seen during the execution of the next instruction, since at t2 the results available at the output of the arithmetic unit is not copied into a register.
The only result of the CMP Dx, Dy instruction remains in the CC register to which, at each arithmetic operation the properties of the result are reported.

11. The LDI Dx c instruction allows to load the constant c into the Dx register. It is exists of two words, the first is the actual instruction, the second is the constant c. It therefore requires two memory cycles.
The first is limited to three clock cycles and ends when the first word of the instruction loaded in the IR register.
During the second cycle, the incremented content of PC once again put on the address bus and, once again incremented. At t3 of the second cycle, the constant c is placed on the data bus by the memory and copied into the Dx register.

12. The LDI Dx, a instruction allows to load the value stored at memory address a in the Dx register. It is a long, two words instruction, the first is for the actual instruction, the second is the address a.
It requires three memory cycles, two to read the instruction and to read the value to be transferred.
The first is limited to three clock cycles and ends when the first word of the instruction in the IR register.
In the second cycle, the content of PC is incremented, put on the address bus and, once again incremented. At the t3 of the second cycle, the content of memory address a is put on the data bus by the memory and copied to the A6 register also called TP.
In the third cycle, the TP content is placed on the address bus and a read command is given to thememory. At the t3 of the third cycle, the value in the memory at the address a has become available on the data bus and is copied to the Dx register.

13. Les explications données à propos des instructions arithmétiques et de transfert devraient suffire pour permettre de comprendre comment une expression peut être évaluée dans RP1.
Les registres D vont être utilisés pour former une pile sur laquelle on peut évaluer une expression comme cela a été fait sur la pile de l'ordinateur décrit au chapitre précédant.
L'expression est d'abord réécrite en notation polonaise inversée. Le compilateur va ensuite lire cette expression et traduire les opérandes rencontres par des instructions LOD ou LDI et les opérateurs par les instructions arithmétiques appropriées.
Ainsi, pour l'expression polonaise inversée a 3 + b * le programme commencera par placer la valeur de a dans le registre D0 et la constante 3 dans le registre D1. Ensuite le contenu de D1 sera additionné au contenu de D0. Après cela, la valeur de b sera placée dans D1 et le contenu de D0 multiplié par le contenu de D1. Il ne restera ensuite plus qu'à transférer le contenu de D0 dans la variable x en mémoire.

The explanations about arithmetic instructions and transfer should be sufficient for understanding how an expression can be evaluated in RP1.
The D registers will be used to form a stack on which to evaluate an expression as has been done on the stack of the computer described in the previous chapter.
The term was first rewritten in reverse Polish notation. The compiler will then read and translate this expression operands meetings with LOD or LDI instructions and operators through appropriate arithmetic instructions.
Thus for Reverse Polish expression 3 + b * the program will start by placing the value in a register D0 and the constant 3 in the register D1. Then the contents of D1 will be added to the contents of D0. After that, the value of b will be placed in the content D1 and D0 multiplied by the contents of D1. It then only remains to transfer the contents of D0 in the variable x in memory.

14. The JMP a instruction allows an unconditional jump to the address a. This instruction has a length of two words, the first being the actual instruction, the second the address to which it is necessary to jump to. During the second memory cycle of the execution, this address is transferred to the TP register.
Then begins execution of the next instruction, but instead of placing the PC content on the address bus during the first clock cycle the content of TP that is placed on the bus. At t2, as for any other execution, the small additional arithmetic unit adds 1 to the contents of the bus addresses and places the result in the PC registry. In this way, program execution continues well from the address that was contained in the statement of unconditional jump.

15. The execution of a conditional branch instruction is quite similar to that of an unconditional jump instruction, one major difference, which is at the beginning of the execution of the next instruction.
At the end of the second memory cycle, the TP register contains the address of the instruction that will execute if the condition of the jump is satisfied while the PC register contains the address of the instruction following the conditional jump instruction, that it will run if it is not satisfied with the condition of the jump. The control unit therefore decides at that moment, according to the CC register status if it's required to place the contents of TP or PC on the address bus.

16. Using the conditional and unconditional jump instructions one can implement conventional control instructions of high-level languages. The slide above shows how an if statement is translated into assembler RP1.
First the Boolean expression B is evaluated so that the evaluation ends with a calculation that results in a 0 result if the expression is false and a 1 result if the expression is true. After evaluating B the instruction JPZ L1 skips to the label L1 if the value of B was false. If, on the other hand it were true, the code for S1 is executed. After this code unconditional jump to the label L2 is done which is the continues the rest of the program. The code for S2 is placed between the labels L1 and L2.

17. The translation in assembler of the while ... do loop is easy. first the expression B is evaluated as in the above example. A JPZ L2 instruction skips to the rest of the program if the value of B was false. If, on the other hand, it was true the code for S is executed and at the end of it, an unconditional jump JMP L1 causes a reassessment B.

18. The code resulting from the translation of a do..while statement shown above.
On can see that it starts with the code that corresponds to S, followed by the code corresponding to the evaluation of the Boolean expression B. Following this assessment a JNZ instruction skips to the beginning of the S execution if the value B was TRUE, otherwise continue with the rest of the program.

19. The translation in assembler of the for loop allows to highlight the advantages of an register architecture with respect to a stack architecture.
First expressions M and N are evaluated and their values are placed in the D0 and D1 registers. The CMP D1, D0 instruction allows to verify if it's necessary to continue the execution of the for loop or if we can move on rest of the program. An instruction JPN L1 skips to the rest of the program if the contents of D1 was larger than D0. After JPN instruction, the code for S is executed. It should be noted that this code can not use the D0 and D1 registers, since these are necessary for the management of the for loop. After the S code, an ADI D1,#1 instruction is used to increment the variable a stored in the D1

register. After incrementing the JMP L2 instruction causes the return to the beginning of the loop.
The big difference between this code and code for the stack processor is the result of the fact that the D0 and D1 remain accessible while with the stack processor only top is accessible and that any reading of this top makes it inaccessible.

20. The most interesting aspect of a subroutine call instruction is the safeguard mechanism of the return address. In the RP1 a part of the memory is used to build a stack in which the return address, the dynamic and static links, activation blocks and the function parameters are stored. The address of the top of the stack is retained in the register A5, also called SP.
During the first 6 clock cycles (plus any waiting time tw) the execution of a subroutine call instruction is quite the same as a regular jump instruction. At the end of t3 of the second memory cycle one can find the TP address of the subroutine and in PC the address of the instruction following the call. The second memory cycle is extended by two clock cycles during which it increments the contents of the stack pointer SP, so that it points to the first available space on the stack. Then starts the third memory cycle in which we write the contents of the PC register in the memory at where the SP points.
At the end of the third cycle the executing of the next instruction starts whose address will be found in TP register instead of the PC register.

21. The execution of a subroutine return comes down to moving the return address stored on the stack to the program counter PC.
When at the end of the first memory cycle the RET instruction is identified, the second cycle begins immediately. At t1, the content of the stack pointer SP is placed on the address bus with a reading command. At the t2 the contents of the stack pointer is decremented by one and at t3 the return address is loaded into the TP register.
After the end of the second cycle the execution of the next instruction begins by putting the contents of TP on the address bus.

22. In the stack processor described in the previous chapter, call and return instructions subroutines managed not only the return addresses but also the dynamic and static links. We saw that this was not the case for RP1. This computer has two specific instructions (LNK and ULK) to manage dynamic links. It does not have specific instructions for static links because their management is so, usually, very simple.
As in the case of the stack machine, there must be a base register. Here, the A4 register will be used for this function.

23. The LNK instruction is used to update the registry after a call subroutine. It is normally the first instruction of the subroutine.
The first memory cycle of the execution of LNK has 5 clock cycles the latter two are used to increment the stack pointer.
At t1 of the second cycle, the content of the stack pointer is placed on the address bus, the content of the registry is placed on the data bus and a write command is given.
At t2 of the same cycle the base register is updated by copying the present value on the address bus into the base register.

24. The ULK instruction is used to restore the old values of the stack pointer and the base register just before leaving the subroutine. During the last two clock cycles of the first memory cycle, the value in the base register is copied to the stack pointer to release all the space occupied by the subroutine. In the second cycle, the value of dynamic link is removed from the stack and placed in the base register.

25. The above slide shows the use of the instructions JSR, LNK, ULK and RET.
Before calling the function P, the stack pointer SP points to the last used word on the stack and the base register A4 points at the start of the activation block of the function in which the call to P is found.

26. After calling the function P, the address of the following instruction in the calling function, the call (the return address RA) is on the stack.

27. The first instruction in the function P is the LNK A4 instruction to update the base register A4. It puts on the stack, above the return address, the old value in A4 and places in A4 the address on the stack in this dynamic link.
The following statement (ADI SP ...) increments the contents of the stack pointer to reserve space for the activation block of the function P.

28. The last but one instruction of the function P is ULK A4 instruction that restores the value in A4 before executing the LNK instruction at the start of the function P. It also frees all the space on the stack for the function P by placing in SP the address return address.

29. After the return of the function P execution continues from the return address RA, which is also removed from the stack. (Actually, it remains on the stack, but the stack pointer was decremented, it became practically inaccessible and will be replaced by other data at the next write on the stack)

30. To complete the study of the call subroutines in RP1, we must also consider the parameter passing mechanism.

31. The parameter passing is done in exactly the same way in RP1 as in the stack processor studied in the previous chapter. The parameters are placed on the stack, on top of activation black of the calling function and below the return address. For parameters passed by value it is the current value of the parameter that is placed on the stack while for parameters passed by reference, this is their address.
The only difference with the stack processor lies in the values returned by functions. While in the stack processors a place was reserved for this value below the stack parameters, it's usually in a data register that the value of a function is returned in a register processor.

32. A recursive function is a function that can call itself.
Although for performance reasons it is not recommended to calculate the factorial function recursively, we can do it to have a very simple example of using recursion.
The factorial function written in C is simple since it literally transpose the mathematical definition of the factorial function in the C language

33. The slide shows the translation in assembler, with comments, of the recursive factorial function.
It is strongly recommended the reader to follow step by step the execution of this function when it has to calculate 3! or 4! It is particularly important to draw the successive states of the stack for this execution to understand the mechanisms of implementation of a recursive function.

6 Input / output interfaces

1. In previous chapters, the operation of a CPU has been described quite well with details. One aspect, however, was completely ignored. It is that of communication between the computer and the outside world. This will be discussed in this chapter.
The computer is coupled to peripheral equipment or communications networks through electronic devices called interfaces.

2. In some computers, such as PCs, there are conceptually simple interfaces to display text or images on the screen. Part of the main memory has double access: first the processor can read and write to this memory as with the rest of the memory, but on the other hand, the controller screen, often called "GPU" can also access this memory to extract the information to display on the screen.
The display of text or drawings on a screen does not require synchronization mechanism between the computer and the screen. The two components can work at their own pace without worrying about the pace of the other.

3. When the rhythms of the computer and peripheral equipment must be synchronized, an interface is used that essentially contains a register for data ("data") and a synchronization bit ("sy").
To illustrate the ideas let's consider a primitive printer as peripheral equipment.
When a program needs to send a character to the printer, the character is written into the interface's DATA register and the Sy bit is set to true. The printer, when it is ready to print, checks the status of Sy bit. When the Sy bit is true, it reads the contents of the DATA register and sets the Sy bit to false.
The program regularly checks the status of Sy bit. When the Sy bit is found false, the next character is loaded in the DATA register and de Sy bit is set to true again.

4. In the above slide you can see a sample C function that uses the interface described in the previous slide.
It should be noted that the while ... do loop used to wait for the Sy bit is reset before the character to be printed is placed in the Data register.
During the execution of the loop, the program simply waits until the printer is ready, continuously verifying the status of the Sy variable.
This technique has already been used in the example of the electronic lock described in the first chapter of this course is called "polling"

5. As has already been explained in the first part of the course, the interrupt mechanism can be used to avoid non-productive waiting activity that characterizes the polling.
The control unit of a computer with an interrupt mechanism has one or more input terminals. A state change at an interrupt terminal causes the interruption of the current program: during an interrupt, the controller completes the execution of the current instruction, but instead of fetching the next instruction to the address determined by this statement, it will seek it on another address as determined by the signal interrupt.
Normally, as when calling a subroutine, the address of the next instruction of the interrupted program is preserved, so that it can resume execution of the program after the treatment of the interrupt.
The program that executes because of an interrupt is usually called "interrupt handler".

6. The architect of a control unit must make two important choices that relate to interrupt mechanism: it is the location of the interrupt handler and how to save the return

address.
For the location of the interrupt handler, three strategies are used in practice: the first consists in associating with each interrupt terminal a physical address where the first instruction of the corresponding interrupt handler must be; the second is to associate each terminal a physical address where the first manager of the instruction address must be located and the third is to load the equipment causing the interupt to supply itself the first instruction. These three mechanisms are explained in more detail in the following slides.
Saving the state of the interrupted program (often called "context") can also be done in various ways, but one should realize that it is not enough, as in the case of a subroutine call to save the return address and possibly the dynamic link, but also the contents of all registers subject to change by the interrupt handler should be saved, since we can not know when the executing of an interruption will occur and it may well be that it occurs just when all registers contain intermediate results that will be used during the execution of the rest of the program .

7. In some processors, particularly those fitted to PCs, a fixed address is associated with each interrupt terminal and, after an interruption, the first instruction to be executed will be the one found at this address.
In the processors used in PCs, address 0 is the RESET interruption, address 4 to the first normal interrupt, address 8 to the next and so on.
During a reset signal (RESET) there is no need to save anything in the current program. Normally an unconditional jump instruction will be found at address 0 jumping to the to the start address of the operating system.
The addresses corresponding to the other interrupts, normally contains a subroutine call to the concerned interrupt handler. In this way, during a break, the return address is automatically saved on the stack and ending the interrupt handler by a return instruction, it automatically returns to the interrupted program. For each interrupt handler, the programmer remains to take care of the backup of all context data likely to be modified by the interrupt handler. For instance on the stack above the return address. Before returning to the main program, thus before the return instruction, the context data must be restored to the situation before the interrupt.

8. In other processors, including those that equip computers in the Apple brand, each interrupt terminal is assigned a fixed address in memory, where the control unit can fetch the address of the beginning of the interrupt handler corresponding. In such processors, the control unit will load itself backing up a stack contents of all registers. These processors therefore have a specific instruction to return from the interrupt handler while restoring the contents of all saved records. This way is undoubtedly much more friendly to the programmer, since he does not have to worry about saving the context of the interrupted program, but if that context has a considerable size, the response time to an interrupt can be prohibitive for some critical applications. For this reason, in some processors, this is only part of the context is saved automatically. In this case it is to the programmer to safeguard the rest of context if it may be modified during the execution of the interrupt.

9. Finally, some processors designed to process a large number interrupts from different origins have no mechanism to immutably associate an addresses to specific interrupts. During a break, they just start the read cycle of the next instruction, but without addressing the memory. Instead, they give the equipment that caused the interruption, the authorization to place itself an executable statement on the data bus to be loaded into the instruction register. The equipment that caused the interruption normally sends a

routine call instruction with the address of the desired interrupt handler.
With such a mechanism, called "vectored interrupt" it is clear that it is the programmer who has the sole responsibility of the context backup.

10. The use of interrupts to manage I/O operations uses exactly the same interfaces as described in the study of the polling method.
In the above slide, we see that the Sy bit is directly connected to one interrupt terminal of the processor. In this way the program should not continuously query the status of Sy bit but may engage in other activities, or even be suspended to allow the execution of another program, since at the time of the state change of Sy bit an interrupt occurs. This interrupt results that the program awaiting for the change of state of Sy bit can send the following data in the DATA register.

11. The function shown in the above slide shows how printing operations can be programmed when the printer interface can generate interrupts. Italics in the program corresponds to parts of code where details would confuse the example.
We assume the existence of a buffer, called buffer, managed as a single queue, "first come, first served".
In normal use, the print function simply places the character to print in the buffer.
The function also contains an active waiting loop
while (bufferfull) do {}
but it is used only in case of saturation buffer ..
If, when print was called, the buffer was empty, the printer was inactive and will not cause disruption. This is why the initial state of the buffer is checked by the instruction
if (buffer was empty) start pdriver
This causes, if necessary, the activation of the driver (pdriver) of the printer (which is actually the interrupt handler of the printer) as when an interrupt occurs.

12. The interruptions caused by the printer are managed by a specific program, called "printer driver". The driver consists of an infinite loop
while (true) do
in this loop two actions are performed repetitively:
- First, the first character is removed from the buffer and sent to the interface's DATA register,
- then the driver is placed blocked state until an interrupt is caused by the printer.
If at the time of activation of the driver the buffer was empty, the driver is immediately blocked, without hope of release by an interrupt.

13. To estimate the I/O rates achievable using interrupt mechanisms one can considered the following orders of magnitude:
- a processor which executes 50 million instructions per second
- a driver that contains fifty instructions in its main loop
This results in approximately one microsecond to transfer one byte and a maximum achievable flow rate in the order of a megabyte per second (1 MB/s).
As the other throughputs for local networks are of the order of 1 to 20 megabytes per second and throughputs of the same order are found in hard drives and CD drives, one realizes that the operations I/O managed by interrupts are much too slow for fast devices.
The slow I/O operations directly managed by interrupts have a direct consequence to the universality of the CPU. Because the CPU can perform a variety of tasks, fifty instructions are required to accurately describe the task of simply transfer a byte between memory and I/O interface.

14. To speed I/O a specialized processor is added to the computer. This processor is only capable one single task, which involves transferring bytes between memory and I/O interfaces. Just one simple control signal suffices to conduct a transfer, without reading any instruction from memory. This specialized processor is called "Direct Memory Access Controller".
Before the transfer, a program executed by the CPU prepares the transfer by loading the source address, destination address and the number of bytes to transfer in DMAC. The Sy bit of the interface is connected to the transfer control of the DMAC. Whenever Sy status indicates the availability of the interface, DMAC transfers the next byte. When all bytes of the specified block have been transferred, the DMAC informs the requesting program causing an interrupt. So we can say that the DMAC divides the number of interrupts required by the block size of the transferred data.
As the DMAC may transfer a byte, it must have authority over the address and data bus. To acquire this authority before each transfer, DMAC request authority over the bus to the CPU. The latter then ends the current memory cycle, transfers the bus to the DMAC and refrains from using the bus until the DMAC has not completed the transfer of the byte. They say the DMAC steals memory cycles to the CPU and therefore slows slightly program implementation.

15. A Direct Memory Access Controller must at least have four registers:
1. The first contains the address where the byte transfer must be read ("Origin Address").
2. The second contains the address where this byte must be written ("Destination Address").
3. The third contains the number of bytes to be transferred ("Byte Count").
4.The fourth is used to configure the DMAC for a specific transfer.
This configuration determines how the contents of various registers must be adapted in each byte transfer. For example, when it comes to copy a block of data from main memory to a disk controller the Destination address remains constant, while with every byte transfer the Origin address increments and the Byte count decrements by one.

7 Microprogramming

1. In this chapter, the underlying principles of the design of the control unit will be summarized.

2. The control unit is the central element of the computer that controls all activities. It reads instructions from program memory and translates these instructions into electrical signals instructing each component of the computer to perform specific tasks at the right moment.

3. The control unit is a sequential Boolean circuit, with, as inputs:
- Instructions fetched from the program memory and stored, during their execution, in the Instruction register.
- If existing, the condition code bits associated with the ALU,
- The interrupt lines
- A memory ready line from the memory system
- A bus request line from a Direct Memory Access controller
Based upon these inputs and timed by means of the internal clock, the control unit generates digital sequences intended to control all parts of the computer.

8 Central Memory & Caches

1. Until now, the main memory was considered a homogeneous set of words, identifiable by their address and able to maintain as long as the power supply is present, the value of a binary number with a predetermined number of bits. In this simplistic view of main memory, all memory words have the same access times and addresses is a cardinal number whose value can vary between 0 and memory capacity, expressed in words.
In this chapter, these notions are questioned.
First it will be shown how it is possible, by the use of memories whose access times are different, to achieve a large memory whose average access time is significantly lower than that of most of it's components.
Then various techniques for managing the allocation of portions of the memory tasks will be introduced.

2. In the first part of the course a distinction was made between fast main memory and much slower peripheral memory.
Here are the different access times of the main memory taken into consideration.
On the one hand, the range of the access times of semiconductor memory spans almost two orders of magnitude, between one and a hundred nanoseconds, but the fastest memories are more than a hundred times more expensive than the slower.
On the other hand, the access time to consecutive addresses is often ten times faster than the access time to randomly selected addresses. This phenomenon comes from the fact that within a chip different memory cells are arranged in rows and columns. The access time is divided into selection time for the column, selection time for the row in the column and a read time itself. When reading consecutive addresses in the same column, we must not repeat the selection of the column, resulting in considerable time savings.

3. A particular organization allows to best exploit the differences in cost and speed between different memory speeds. This is the use of auxiliary memory, called "caches". This principle can be used for different levels in an information storage hierarchy. Here the explanation will be based on the cache that is in almost all computers between main memory and processor.
The cache is a fast memory connected in parallel with the main memory on the bus that unites memory and processor. When the processor reads a word of main memory, neighbours words are copies in the cache, taking advantage of shortened access time for reading consecutive addresses. This way, when the processor will launch a new reading command, it is likely that the requested word is already found in the cache and therefore should not be read from the main memory. Statistically, the access time to memory thus lie somewhere between the access time to the cache and the access time to the main memory.

4. To achieve practically a cache the storage space is conceptually divided into smaller pages, four words in the figure shown here, from 8 to 64 words in reality. If the addresses are 32 bits, we can say, in the case of pages of 4 words, the two least significant bits designate a word in a page and the 30 most significant bits designate a specific page. Physically, the cache contains a number of pages, associated with each page, a place to record the page number.
During an read operation the number of the requested address page is compared to the number of the pages in the cache. If the requested page is present the read operation is kept within the cache, otherwise the full page is read from memory and transferred to the

the cache. If the page is this the read operation is done only in the cache, if the entire page is read into main memory and transferred to the cache. While loading a new page in the cache an old page must be dropped. Ideally, it would be a page which we will no longer be used, but it is difficult to predict the future, in general we choose the page that has not been accessed for the longest time, or, simply put, the oldest.
When writing into memory, the presence of a cache raises difficult problems, in that it must ensure consistency between the contents of the cache and the main memory. Fortunately, the frequency of write operations being much lower than readings, inefficient but reliable techniques can be used. Thus, one can always perform write operations simultaneously in the cache and the main memory. Other systems remove cache as soon as one word has been changed.

9. The concept of cache can be used at different levels in the architecture of an information processor.
Between CPU and main memory we'll find virtually always a cache, and sometimes even two different caches. This is due is the growing disparity between processor speed and access time to main memory. Thus in modern PC's we'll find on the same processor chip a first very fast of few kilobytes whose access time is well suited to processor speed and then a second cache, a few hundred kilobytes, significantly slower. The adequacy of the characteristics of the various caches, the processor and the main memory is critical to the performance of a computer and much less depended of the CPU's clock speed.
Between disk and main memory cash technology is also used. Since moving heads of a disk from one track to another is an important part of the total access time, it is common practice to transfer the whole track to the main memory since it's likely that the following areas, which often belong to the same file will soon be required as well.
At a totally different level, the Internet service providers keep copies of frequently requested web pages close to users.

10. Already in chapter 2 of this part of the course we have seen that to compile and run programs originally written in a high level language with block structure, it was interesting to break down the address of a variable in two parts, first identifying a block in which the variable is declared and the second indicating the address of this variable within the block. In chapters 3 and 4, implementations of this idea using a stack, a registry and static and dynamic links have been shown. As modern operating systems tend more and more to allow the memory coexistence of several independent programs, but who still can use almost simultaneously shared objects, it becomes increasingly important to equip the processor of a more flexible address management mechanism that allows to separate as much as possible the logical addresses, used in different individual programs, physical addresses, used globally by hardware circuits.

11. The segmentation mechanism consists in breaking a logical address into two parts, the segment name and address within the segment. The various segments correspond to consistent logical entities within a same program, such as, for example, an activation block of a function. The individual segments can therefore be different sizes.
Each segment can be materialized by any contiguous block of physical memory as long as it is sufficiently large.
To use segmentation we must have a mechanism that reflects the segment name and address within the segment in an address in the physical memory of the computer. This same mechanism must ensure that the specified address within a segment is not greater than the size of the segment in question because otherwise there would be a physical address that might belong to a different segment.

Segmentation allows, among others, in various programs that are simultaneously in the memory of a computer operating in multiprogramming have each their own memory space with independent logical addresses of where these programs are loaded into physical memory.

12. Dans les adresses logiques utilisées dans un système avec segmentation de la mémoire, les adresses logiques sont constituées de deux champs, le premier est le numéro du segment et le second est l'adresse au sein du segment.
La traduction de l'adresse logique en une adresse physique se fait à l'aide d'une table des segments. Pour chaque segment, cette table contient l'adresse en mémoire physique où commence le segment et la taille du segment. Certaines tables contiennent également des indications qui permettent par exemple d'empêcher des opérations d'écriture dans ce segment spécifique.
Lors d'un accès à la mémoire, le numéro de segment est utilisé comme index dans la table des segments et l'adresse du début du segment est ajoutée à l'adresse au sein du segment pour trouver l'adresse physique désirée. Si l'adresse au sein du segment excède la taille de celui-ci, l'accès effectif à la mémoire physique est inhibé et une interruption est générée pour arrêter le programme qui essaye d'accéder des parties de la mémoire qui pourraient ne pas lui avoir été accordées. (Dans le système d'exploitation Windows cette erreur se traduit par un écran bleu et un message ≪ general protection faillure ≫). Si la possibilité de restreindre les operations autorisées dans certains segments existe, toute violation de ces restrictions se traduira également par une interruption du programme qui a causé la violation.

In the logical addresses used in a system with segmented memory, logical addresses consist of two fields, the first is the segment number and the second is the address within the segment.
The translation of the logical address into a physical address is done using a table of segments. For each segment, this table contains the physical memory address begins the segment and the segment size. Some tables also contain information which allow for example to prevent write operations in this specific segment.
During a memory access, the segment number is used as an index in the table of segments and address of the beginning of the segment is added to the address within the segment to find the desired physical address. If the address in the segment exceeds the size of it, effective access to physical memory is inhibited and an interrupt is generated to stop the program tries to access portions of memory that might not him have been granted. (In Windows operating system this error results in a blue screen and a message "general faillure protection"). If the possibility of restricting the operations allowed in some segments exist, any violation of these restrictions will also mean a program interruption that caused the violation.

13. As the place in physical memory can only be allocated to a new segment if it's available contiguous free space is equal or larger then the segment size, the creation and destruction of frequent segments inevitably leads to the appearance of small unused memory blocks.
To avoid this type of memory fragmentation, it would be better to allocate memory as fixed-size blocks. Such blocks are called "pages" and the technique that implements them is called "paging".
Both the logical memory space that physical space is divided into fixed-size pages (typically 4096 bytes). The memory management system then allows to match any logical page any physical page. This approach allows to use all the unused blocks of physical

memory to create a single logical address block adjacent.

14. In the case of a paged memory addresses consist of a page number and an address within the page. A page table establishes correspondence between the logical page numbers and physical page numbers. In the case of paging the length of page should no be included, since the latter is a predefined constant. A field that is used to impose restrictions on the use of content pages is obviously possible.
As in the case of segmentation, logical page number serves as an index into the page table where we find the corresponding physical page number. The physical address is obtained by concatenating the number of the physical page and the address within the page.

15. The number of bits of the physical page number must not necessarily be the same as the number of bits of the logical page number. A physical page number greater than that logical pages predicts more physical memory than initially planned by the designers of the architecture of the computer. Such a possibility is highly desirable in the case of architecture dating from a time when memory was very expensive and which remain in service because of the existence of many difficult to adapt programs.
The technique known as the "expanded memory" supported by DOS and Windows systems is a well known example of using paging to extend physical memory without affecting the size of the logical memory.
The original PC had a logical and physical addressing space limited to 1 Mega byte. In this space, four pages of 16 kilobytes each had been set to use the expanded memory. In the space reserved for I/O interfaces, it was possible to insert a memory card with a maximum of 2048 pages of 16 kilobytes. Using write instructions in the control register of the memory board, considered as I/O registers, it was possible to associate any physical page with each of four logical pages included in the original address space.

16. Another application of the pagination is found in the Windows system when it comes to run an old DOS program written for and display the results in a window managed by Windows.
In chapter 3.5, it has been seen that in a PC display on the screen is done by placing appropriate information at predetermined memory addresses, the video processor takes care of displaying the contents of this reserved memory area. All the old programs planned for the DOS system thus wrote what should appear on the screen to specific memory addresses and the results were displayed on the entire screen. For the Windows paging mechanism matches the logical addresses associated with the display of regular physical addresses, so that what the program thinks DOS writes to the screen is not displayed. Another program that is part of Windows, then reads the physical pages with DOS program "thought" that they corresponded to the screen, reformats the content and displays them in a traditional Windows window.

17. It is also the pagination which is used to implement virtual memory, which, as already described in chapter 2 of part one, allows matching to certain pages of the logical memory pages physically stored on disk. To do this, simply include in the page table an additional column that indicates whether the page is in main memory or device. When a read or write operation is directed to a page that is in the device memory, an interrupt of "missing page" is generated and the program who requested read or write operation is suspended until that the operating system has brought the required page in main memory.

18. The missing page interrupt is a very particular operation because if usually the controller that completes the execution of the current instruction before responding to an

interruption, this is not possible with the missing page interrupt since it is the current instruction that caused the interrupt and needs the missing page to be completed.
When a missing page interrupt of the control unit should return the status of the program in that it was in before the start of the instruction that caused the interrupt. After the missing page was brought into main memory, the instruction needed will be replayed in its entirety.
Such an interrupt mechanism does not exist in all processors, which explains why some modern versions of operating systems that support virtual memory can not run on older computers, that seem entirely consistent with their successors.

19. It goes without saying that segmentation and paging can be combined. Using segments consist of a whole number of pages, combining the functional organization linked to the segmentation with flexibility and efficient use of the paging-related memory.
In the case of a combination of segmentation and paging, a logical address is made up of three fields, the first indicating the segment number, the second indicates the number of the page within the segment and finally the third one indicates the address within the page.
In such a configuration, the segment table, instead of containing the address of the start of physical memory segment will contain the address where the table specific to this segment pages.
Modern PCs use segmentation and pagination: their architecture allows the definition of 214 (=16384) segments each of which can contain 220 ($\simeq$ 106) pages of 212 (= 4096) bytes. It goes without saying that such a large logical memory only makes sense if a virtual memory mechanism allows placing most logical memory on supported devices.

20. One might conclude that a computer with segmentation and paging is desperately slow because each memory access requires two access this memory to consult the tables of segments and pages.
Fortunately, it is very likely that for a given time interval, the vast majority of access will be in a small set of pages. Indeed in any program strictly written instructions to be executed successively are one behind the other and the data used are also in the vicinity of each other. So it makes sense to keep the result of address translations for later reuse.
In the processor a kind of special cache is found, called “Translate Look aside Buffer” in which the results of several tens of the most recent address translations are kept. Each line of such a TLB contains the one hand, the numbers of a segment or of a logical page and the other, the number of the corresponding physical page.
During each memory access, the logical address desired is compared to the content of addresses in the TLB. If this address is already present, the physical page number is immediately provided by the TLB, otherwise a missing page interrupt is generated and it is the operating system which will consult the tables of segments and pages to find the desired physical page and possibly even bring in main memory if it were resident on disk.

9 File management

1. In this chapter, the organization of memory devices and files will be discussed.
It would be perfectly possible, and this is done in some systems, to provide a logical address space very large (order of magnitude 264 byte) and consider all devices remembered as a virtual extension of main memory. In this case, no special management system is required for peripheral memories, since the management of all the memories would be provided by the central memory management system.

This approach however is not widespread and the vast majority of operating systems separately manage a (small) part of the peripheral memory as a virtual extension of the main memory and the rest as a set of files namely accessible to users and application programs. This chapter will describe in broad file management.

2. In this chapter will be discussed in turn the following topics:
1. Physical disk organization
2. The link between the logical organization and the physical organization of files
2.1. In Microsoft's DOS operating system
2.2. In UNIX and Linux systems
2.3. In systems derived from Microsoft Windows NT
3. The directories that associate names to files
4. The functions of the operating system that allow access to files.

3. As already mentioned in Chapter 1 of the first part of the course, the magnetic disk for storing information consist of several magnetic surfaces that rotate at high speed around a common axis. The information is written and read using magnetic heads that need to be in the immediate vicinity of the magnetic surface (in fact, under the effect of aerodynamic forces, they float above the surface to an altitude of some microns). Each magnetic surface is "flown" by a head. All the heads are mounted on a movable comb species for positioning heads over a number of predefined circular magnetic paths.
The set of tracks on different magnetic surfaces equidistant from the axis is called a "cylinder". All tracks of the same cylinder are accessed simultaneously, without the need for moving heads.
Information is written to disk as independent records of a few thousand bits. Such a record is called "sector". A specific record is locatable in a disk if you know the area, the cylinder and the area where it is located. Generally this three-dimensional addressing is replaced by a one-dimensional address obtained by numbering all records so that access in numerical order will be given the physical characteristics of the disc, as fast as possible.

4. One may wonder what is the optimal sector size of a disk.
The main advantage of small areas is to reduce wasted space (often called "slack") by the need to assign an integer number of sectors in each file. In the case of large files we lose statistically half a sector per file. In the case of small files a considerable place can be lost. In the case of low-capacity drives therefore preferably small sector sizes are selected. The main advantage of a large sector size is the increased data throughput due to an increase in the size of blocks transferred. An additional advantage lies in the numbering of the sectors: the larger they are, the less there are for a given total capacity, of reducing the number of bits required for a sector number. For larger disks large sector sizes are preferred.
For reasons of complexity of the disk controllers and their drivers however, it is highly desirable to adopt a uniform size to the sectors, whatever the discs are. That is why in some systems, including Microsoft, the term "cluster" was introduced, which is a group of consecutive physical sectors that are considered by the software as one sector in the sense that the software transfers that whole clusters, and manage clusters numbers rather than physical sectors numbers. Although the term "cluster" is in fact unique to Microsoft systems it will be used in the explanations of all the following slides as the name of the smallest identifiable data block and transferrable by the file management software.

5. An amusing illustration of competing considerations that may determine the cluster size is provided by the PC floppy disks. There are two formats for 3.5 diskettes: the

format "DD" and the format "high density".
The physical characteristics and the size adopted for clusters are shown in the slide above. We see that the clusters of 2 sectors have been adopted for double density floppy disks while the high-density disks are clusters of one sector.
The reason for these strange choice is historical: the double density floppy disks were used at a time when few personal computers had a hard drive. All the software and all data were stored on floppy disks. The transfer speed of disks was therefore critical to system performance and it was logical to adopt fairly large clusters. When the high-density diskettes were introduced, all computers where equipped with hard drives. Floppy disks were only used for backup or data exchange.
So it was logic that for these new disks a cluster size was adopted to minimizes wasted space in "slack".

6. The above slide shows technical data of a hard drive of a few hundred megabytes.

7. The first topic to be addressed in the area of file management is the relationship that should exist between the logical organization and the physical organization of a file. Since physically the complete file is stored as a set of fixed sized records, it is natural to also consider the logical files, known to the user, as sets of blocks whose size is the same as that physical record size. Such blocks are called "clusters" in the rest of this chapter. Each cluster has a physical cluster number (PCN), determined by its physical position on the disk, and a logical cluster number (LCN), determined by its logical position in the file to which it belongs.
The subject is introduced in this slide and the following is the establishment of the correspondence between the logical and physical numbers of the clusters of a file.
Depending on the operating system in question, the correspondence is realised in various ways.

8. In Microsoft's DOS, each file, also called "stream" is a sequence of clusters. Each cluster contains a multiple of 512 bytes of data and each cluster is associated with a pointer that gives the physical number of the next cluster of the file. This pointer may be of particular value that indicates the end of a file.
We see that DOS files are basically sequential files, since the exploration of any file must necessarily start with the first cluster (or at least by the pointer associated with the first cluster).
A special file ("free") exists throughout a disc which contains all unused clusters.
When a file is deleted it is actually simply inserted ahead of the free file and when a new file is created, its clusters are taken from free file. From this observation we can deduce that during the destruction of a file the data is generally not immediately lost, they are only so when a cluster belonging to the destroyed file has been recycled into one other file. If unwanted deletion of a file occurred, recovery starts with avoiding to create new files and to use a computer tool that allows to change the pointers associated with different clusters.

9. In the DOS system, pointers associated with different clusters of a disk are not part of the cluster itself. They are grouped in a particular table, including two copies are stored on each disk. This table is the "File Access Table" which is often abbreviated as "FAT". The FAT is retained by two independent copies because after erasing the FAT all data stored on the disc becomes practically unusable. Unfortunately, the most common cause of loss of FAT is malicious intervention of a virus, and virus writers are aware that there are always two copies of the FAT on a disc.

10 and 11. Pointers contained in the FAT are 16 bits long, which limits the number of clusters on a, disc 2^{16} (65536). This limit is, as shown in the slide above, a serious n case of the use of modern disks whose capacity can reach several tens of gigabytes.
We see that for a 8 GB disk cluster size is at least 128 kilobytes. This also means that any file, even if it contains only a few bytes, must have a minimum size of 128 kilobytes. This behaviour explains why, when a disk whose capacity exceeds one GB, the DOS users tended to partition the disk into two, three or four partitions, managed by DOS as if they were separate disks .
This problem was fixed in the Windows system by abandoning de facto the FAT, although the new disk management mode is called FAT32. In contrast to the 16-bit pointers used in the FAT described herein the original FAT is now called FAT16.

12. In UNIX and LINUX correspondence between physical and logical clusters is not established for the entire drive as in the case of DOS but is established through a separate table for each file. This table has the name “inode” and contains 13 physical cluster numbers.
The top 10 are the physical numbers of the first 10 logical file clusters. The 11th is the physical cluster number that contains the physical numbers of the following 256 logical clusters. The 12th is the physical number of a cluster that contains the physical numbers of 256 clusters which in turn contain the physical numbers of 256 * 256 following logical clusters. Finally, the thirteenth, according to a similar mechanism, allows access to the 256 * 256 * 256 last clusters of the file.
It’s clear that only the parts of this tree structure required to access effectively filled file clusters take up space on the disk. This allows, unlike what happens in DOS, to create very large files very stingy, without unnecessarily occupy space on disk.
The inodes structure promotes in a very pronounced fashion the access to short files and the beginning of long files. This option is statistically very well adapted to applications of office automation and software engineering.

13. To eliminate the limitations to the FAT16, Microsoft designed a completely different file system (New Technology File System). In this system cluster numbers are passes from 16 to 32 bits. To reduce the size of the equivalent of the FAT, that fact that often consecutive physical addresses are assigned to logically consecutive clusters was considered.
In NTFS, there is for each file a table containing the logical cluster numbers, the numbers of the corresponding physical clusters and the number of consecutive clusters for which this association is applicable.
One can conclude from this organization that fundamentally sequential nature of MS / DOS files was conserved, but more pragmatic and more compact tables have replaced the old FAT16.
Later in this chapter the location of these new tables will be clarified.

14. Besides the correspondence between the numbers of physical and logical clusters, the file manager also manages the file naming to be manipulated by the user. For this, the File Manager uses directories that contain at least the name of each file and a pointer to find the table that shows the correspondence between the logical and physical clusters of these files.
The vast majority of current management systems use tree-directories where the directories are similar files to other files. On the slide above a directory tree (unique to MS / Windows) is shown: at the top there is the initial directory, called “root” and noted \, which contains references to 8 files. Five of these files, shown in green and designed with square corners, contain data or programs while the other three, represented in orange and

drawn with rounded corners, are themselves directories.
In such a structure, the proper name of a file is preceded by a list of all directories that must be crossed from the initial directory to get to the proper name. Such a tree structure allows to manage a very large number of files by grouping them by subjects in a hierarchical structure.

15. For example, the structure of a DOS directory is shown in the slide above.
For each file, there is an 8-byte name, and an extension of the name in three bytes. The extension of the name is traditionally used in DOS to indicate the nature of the file contents. Thus an .exe extension is used for a file that contains the object code of an executable program while the .ppt extension features a presentation created with PowerPoint program.
The date and time the file was created (or more precisely, of the last modification of the content) is included in the directory to help identify successive versions of the same file.
Evidently a pointer to the first physical cluster is found and an indication of the total length of the file expressed in bytes. File size on disk is the first multiple of the cluster size that exceeds this length.
Finally, there is one byte, each bit of which has a special meaning, called “attributes” of the file. Bit number 4 of the attributes byte distinguishes a normal file from a directory.
Other attributes allows to protect certain files against accidental or malicious changes or prevent displaying their name in the directories
Finally, the archive bit, used in conjunction with some backup programs, allows, during a backup, copy only files that have changed since the last backup.

16. UNIX and LINUX systems also use tree directories (in fact, it is UNIX that such directories became popular), but the semantics of UNIX trees is a little more complex in the sense that the same file can be referenced in several directories and therefore have several different names.
This possibility can be interesting when, for example, a file is used for several different projects and we do not want to work with copies to avoid consistency problems when the content of the shared file will evolve.
The management of such shared files raises a series of interesting and complex issues among which one could cite some particularly complex examples:
- How can we avoid the inadvertent creation of reference loops that would cause the directory reading program (eg. ls) in an infinite loop?
- What should we do when a user of a file decides to destroy it while others still need it?
- If one wants to wait before recycling the space occupied by a file that all users have destroyed, how can we verify that this is so?

17. The NTFS file manager from Microsoft has adopted an original format for its directories: each disc contains a large table named “Master File Table”.
In the MFT each file occupies a space of 1. 2 or 4 Kbytes. In this space there are several fields of variable size and number, used to describe the file, possibly to ensure their safety and allow to find its contents.
Thus, each file can have several names: a name in the traditional DOS format being 8 + 3 characters, unlimited length of name by the standards of Windows and possibly even other names specific to other management systems which can be emulated in Windows NT environment.
In other fields we can find a description of the techniques of encryption and / or compression that have been applied to the contents of the file or a very detailed list of its attributes.

Finally, in the last field, there is information relating to the actual content of the file. The nature and organization of this information may differ from one file to another, and will be described in the following slides.

18. If the file concerned is a small data file, they are directly stored in the block of the MFT corresponding to that file. In this way we avoid extra disk access when viewing a small file as there are thousands in any Windows system.
If, on the other hand it's a large file, for which there is not enough space in the MFT, the logical and physical cluster correspondence table is placed in the MFT as discussed previously, allowing to find effectively from the data contained in the MFT, the contents of the entire file.

19. A somewhat similar strategy is used for directories. When it comes to small directories, indexes in the MFT to all listed files are included in the block of the MFT that is the directory.
If, on the other hand it's a large directory, it is organized as a tree structure called "BTREE" which only the root and the numbers of the physical clusters that contain the rest of the structure are contained in the MFT. Using a BTREE to represent large directories can significantly shorten the time it takes to find a file in the directory (In a traditional file this time is proportional to the number of files in the directory while in a large NTFS directory this time is proportional to the logarithm of this number)

20. File management is one of the tasks of the operating system of a computer. So the operating system must contain a series of functions that can be called by application programs to perform file operations.
The features generally offered by operating systems are:
- The creation and destruction of files with names provided by the user.
- The opening and closing of an existing file. Opening a file consists of searching for the name in the directory and create memory buffers that will be used by the driver to transfer data. Closing a file is basically to flush any buffers and write them in the file on disk. When inadvertent shutdown of a program, it may be that files are left open and correspondence tables between physical and logical clusters on the disk have not been updated and the file is therefore in an incorrect status. That is why, after every unexpected shutdown of a computer, the operating system should examine the entire disk to recover potential inaccessible clusters.
- The transfer of data between the application program and associated disk buffer, transfers between disk buffer and disk being independently via the direct access controller memory.
- The positioning a sequential file to a location specified in the file
- The reading or changing attributes of a file.

10 Concurrent Processes

1. In this final chapter some aspects related to multiprogramming that is to say the quasi simultaneous execution and interlacing of multiple programs or program parts will be introduced.
These topics are important because they can appear in many aspects of the life of an engineer and because non-IT engineers are not always aware of the pitfalls it contains.

2. The topics covered in this chapter are:

- The concepts of process and lightweight processes
- Critical sections and their various implementations
- The various attempts to achieve a pure software implementation
- Implementations with support of hardware
- Semaphores, which put among others, the critical sections in the reach of the application programmer
- The deadlocks and how they can be prevented or get out.

3. A so called “process” is a sequential program or a sequential program part during its execution. The emphasis here is on the sequential nature of the process, which implies that all instructions of a particular process are executed in a perfectly definite order by the order of the instructions and by the data values.
In any multiprogramming system, several processes are executed alternately (see Part I, Chapter 2).
The number of simultaneously running processes is determined by the number of processors available to the computer, each processor can ensure the activity of one process.
Many programs consist of several processes that must be executed concurrently. For example, in a graphical interface, each button is an independent process that waits until the user selects it. When this happens, it sends a message to the process responsible for executing the selected command.

4. What should happen when the process manager decides that the active process P0 must yield its status to process P1?
Firstly, with the help of an interrupt, the execution of P0 is stopped and its state (the value of all variables used by this process) is saved in a memory part PCB0 reserved for this purpose.
Then the state of P1 is reloaded from PCB1 where this state was saved when P1 had ceased to be the active process.
Finally, by means of a jump instruction, the process manager activates processes P1.
If later, P1 must again cede the processor to P0, the inverse operations must be done.
While the process status is saved and that of another reloaded, the computer does not provide a “productive” work. This lost time is actually the main cost of multiprogramming.

5. The state of a process is comprised by the value of all variables that belong to this process, as well as the contents of all the tables used by the management system to manage the process, and in particular, tables of segments and pages unique to this process. As traditionally we mean by “state” the value of all variables, we often use the term “context” for the set of all data to be stored during a change of an active process.
The size of the segments tables and the pages can be considerable, the cost in time for a change of active processes may be significant, which would be a serious argument against the trend in software engineering of increasingly decomposing each program in a large number of processes.
However, one can observe that when multiple processes are part of the same program, it is not essential to protect their own memory space of each process against the activities of his colleagues. Therefore for each program a specific memory space is defined, with tables of segments and pages, on which all the processes that make up the program have access to. This creates 2 types of processes, natural processes, well protected, with a large context and light processes, whose state to save is small as these are “squatting” memory space of a large process.
Such lightweight processes are called “threads”.

6. The cooperation of several processes at the same spot can be the source of sensitive issues that will be introduced in this and following slides by means of an example.
The example in question is that of a rain gauge allowing to measure the rainfall from a distance.
The rain is collected through a funnel into a small calibrated bucket. When the bucket is filled, it reverses, activates an electrical contact and resumed very quickly to its initial position. The process P1 executes an infinite loop (while (true) do) during which it waits for the tipping bucket (do {} while (full);) and increments the value of a variable v at each reversal.
The rain gauge is available by phone. A process P2 is also running an infinite loop (while (true) do) during which it waits for the phone to ring (do {} while (call);), answers the call with the value of the variable v (write (v)) and resets the value v to 0.
Calling the rain gauge one can be informed of the amount of rainfall since the last call.

7. Unfortunately, the system described in the preceding screen does not always work correctly. To understand what can happen, consider the code is actually executed by the processor in the gauge. Since there is a correspondence between the instructions of the object code and assembler, the above screen shows pieces of assembly code for the processor RP1 described in chapter 3.4.
On the left side of the slide we see the translation in assembly language of the instruction v = v + 1 from process P1 and the right side one that matches the instruction v = 0 of P2.
Since a phone call can occur at any time, it is possible that the execution of the instruction v = v + 1 is interrupted just after the contents of D1 was incremented but before the contents is copied to v.
The reset v to 0 performed following the phone call will have no effect since the new P2 value (reset) will be overwritten by the old value calculated by P1.
With a little imagination one can find other situations where the rain gauge, instead of giving too much reading would give too little reading. As such, this gauge will be unreliable.

8. The problem is that the two processes P1 and P2 share a same variable v they can each change the value while the other keeps the old value in one of its registers.
To avoid such mistakes we should be able to ensure that while one process is using the variable v, that the other can not, in any way, have access.
To realise this one could define a critical section in each process to which the management system ensures that no other process can execute its critical section.
In the example the instruction v = v + 1 from P1 and the instructions write(v); and v = 0; from P2 should be protected by critical sections.

9. To understand the critical sections, it is useful to use a railway analogy.
Considering a channel cross. It is clear that no 2 trains can not engage simultaneously in the cross which is the analogue of a critical section. Signals to 2 entries of the crossing to prevent access in case of danger.
We must find an algorithm that ensures that no two 2 trains simultaneously will access the critical section, without causing unnecessary delays.

10. A first idea that comes in mind is based on the perfect symmetry of the system and uses coupled lights that display the same colour green if the crossing is free, red if it's not.
The arrival of a train to the crossing of the entry point causes the red signal, the end of

the passing of the process ensures delivery to the green.
This system is dangerous (and has also been the basis for true rail disasters) because, although it is unlikely, one can never rule out that two trains will occur simultaneously at the entrance of the critical section. They will cause both signals to turn red, but also both trains entered the critical section.

11. The implementation of this (bad) idea in computer process is simple. Just use common Boolean variable to all processes that need to be protected. This is the free variable shown in the slide above.
The risk of this algorithm lies between the instructions do {} while (free!); and free = false;. At this point, the process knows it can enter its critical section but has not blocked access to other processes to their critical sections. The risk is real that several processes are simultaneously in their critical sections, which is the negation of the very principle of the critical section.

12. To prevent 2 trains to simultaneously enter the crossing simply arrange that the 2 signals are green at the same time. Simply wire crosswise.
When the train p arrives and p has green light, he enters the crossing and at its end of passing causes the change of signals. The Q train can therefore pass the crossing in his turn.
The crossing safety is assured, but 2 successive trains can not pass through the same channel. The 2nd will have to wait until a train has passed by the other path.

13. The translation in the preceding algorithm program is simple. Just create a shared variable turn of enumeration type which can take two values pturn and qturn.
The code to insert in the process P is just to wait until the turn has pturn value in order to enter the critical section and do not forget to assign turn to qturn value when the critical section is complete.
The code to insert in the Q process is the same, mutatis mutandis.
Again safety is assured, but strict alternation between P and Q process is imposed, which is generally not acceptable.

14. To avoid the strict alternation imposed by the preceding algorithm, one can use a somewhat more complex algorithm:
Upon arrival at the crossing each train announces it's arrival by assigning true to Boolean variables pneed and qneed respectively. Upon departure, each train resets it's associated variable to false.
Both signals are normally red but each can be green if the values of pneed and qneed show both the desirability and safety of the crossing manoeuvre.
There is however still a major problem: if 2 trains arrive simultaneously, the pneed and qneed variables take both true, the lights will stay red and trains remain forever stuck at the crossroads

15. The double booking is programmed as easily as the previous algorithms, just use two Boolean variables pneed and qneed. The code to add to the process P to control access to its critical section is shown in the slide above.
Like the trains, processes P and Q may end up stuck trying to run indefinitely their instructions do while (! qneed) and do while (! pneed).

16. In fact it is only by combining the technique of each his turn and that of the double reservation that we can find a satisfactory algorithm to implement, by the software, a

reliable critical section without blocking risk.
The turn is used only when in the situation where the double booking leads to a blockage. We come out of this blocking by favouring P or Q alternately.
This combination is named after its inventor, the Dutch computer scientist Deckers. The slide above shows the code to be included in the process P. The code for the Q process is the same, mutatis mutandis.
The Deckers algorithm is a software implementation of critical sections. It is even possible to formally prove that this implementation is correct. Unfortunately, the Deckers algorithm has a major drawback, the waiting process continues to consume CPU time. For proof just look in the above slide instruction
do {} while (turn = pturn!);
For this reason it is rarely used in practice.

17. Since the software implementation of critical sections leading to a waste of resources, hardware mechanisms are provided to achieve critical sections.
In systems with a single processor, it is sufficient that the active process inhibits interrupts to create a critical section, because another process can only be activated through an interrupt. If such a technique is used, one can not lose sight of that direct memory access controller is also a kind of processor activity that could change the value of shared variables. Fortunately, the operation of such a controller is simple enough that we can easily prevent his interference.
For multiprocessor systems, inhibition of interruptions is not sufficient to create a critical section. In such a system there are indeed many simultaneously active processes as there are processors. This will require specific hardware devices implementing critical sections if we are to avoid the Deckers algorithm.
In the next slide such a device, called “Test & Set bit” will be presented.

18. A Test & Set bit is a Boolean variable whose value can be changed by a read!
If the variable is false, a read operation does not change this value. By cons, if it is true, the read operation returns this value but, at the same time, changes the value of the variable from true to false.
If we reconsider the incorrect algorithm that uses a simple reservation Boolean to try to achieve a critical section, but by implementing the Boolean free with a test & set bit, we see that getting the authorization to enter the critical section and blocking the entry to any other process can now be done in one indivisible transaction. No 2 processes can thus observe simultaneously the availability of the critical section and make a reservation they believe to be exclusive.

19. Mechanisms such as those described for making critical sections are delicate. It is therefore essential not to allow the application programmer to take care to implement them but to include them in the specific functions management system for this purpose. Such functions can then be called by application programs to create critical sections without their users having to know the fine details of their implementation.
The most common tool in this area is the “semaphore”.
The slide above shows the semantics of the 2 functions that manipulate a shared variable a called semaphore.
The function wait(& s) waits for the semaphore to become a positive value, and then reduced its value by one.
The function signal (& s) increments the semaphore value.
One can see that the wait function contains a waiting loop do {} while ($s <= 0$); from which one could conclude that this function, like Deckers algorithm must remain active

during waiting periods. As will be shown later in this chapter it is not at all the case, the wait implementation shown here is merely intended to explain this function but not how it does it.

20. The slide above shows how a critical section can be implemented using semaphores. A semaphore, mutex initialized to the value 1 is shared between different processes that will use a critical section. Before entering its critical section, each process must call the wait(& mutex) function. This function will block the process in question as mutex has a negative or zero value. Leaving the critical section, the process must call the signal (& mutex) function to possibly unlock another process blocked in the wait function.

21. Semaphores even allow to generalize the concept of critical section: if, for example, a computer has 10 CD burners at any time, more than 10 CD burning processes can not be active. This condition can be assessed using exactly the same technique as that used for critical sections, but by setting the semaphore to the value 10 rather than 1.

22. Semaphores can also be used to synchronize processes: for example, if a process P1 contains a S1 instruction to be executed before the S2 instruction of a process P2 can be performed, just use a semaphore synch initialized to 0, and place a call to wait(synch &) before the S2 instruction and a call to signal(& synch) right after S1.

23. To implement the wait and signal functions to manipulate semaphores without involving active wait resource waste, functions are included in the process management of the operating system.
If in the function wait(& s), after subtracting 1 from the value of the semaphore s, it appears that it becomes negative, the process that called wait is suspended and added to the list of pending processes associated with s.

24. Thus we see that in the state diagram introduced in chapter 2 of the first part, the transition between the active state and the off state is the result of a call to the function wait(&s) when the semaphore s has a zero or negative value. In the initial discussion that transition was presented as the result of the initiation of an input-output operation, but in fact a lot of other causes can block a process and input-output operations use like all other potential causes of blockage, the semaphore mechanism to ensure the desired synchronization.

25. The implementation of the function signal(&s) is also part of the process manager. Furthermore incrementing the value of s, it checks in the list of blocked processes if there are blocked processes that could be unblocked after incrementing the value of s.

26. The execution of the signal function by a process P1 has no visible effect in the state diagram of the process P1. Indeed this execution can only occur when P1 is active and signal has no effect on the status of the process that runs it. By cons, the execution of signal(& s) by P1 can have an effect on the state of a process P2 if it was blocked by the execution of wait(&s). In this case the execution of signal(&s) may have the effect of changing the state of P2 from “blocked” to “pending”. If several processes were pending, the execution of the signal function obviously can not liberate only one. The choice of the released process is not specified in the definition of semaphores and is part specific options of each operating system. In general, this longest blocked process will be released.

27. A final important issue that can arise in the use of concurrent processes is the in-

evitable mutual blocking.
Situations of fatal blocking are not uncommon in everyday life, even there are often caution statement that, if followed to the letter, would inevitably lead to such blocking.
Thus, in the late nineteenth century, the Ohio state legislators wanted to prevent vigorously any risk of collision at the point where 2 railway lines belonging to competing companies crossed. To do this they enacted a law that stated that any train would stop at a mile before the crossing, and could not leave until another train would have gone ahead.

28. To be able to understand the mutual blocking mechanism, it is useful to use a chart which plots the evolution of the processes likely to hang each other.
Such a diagram has as many dimensions as there are process taken into consideration. For practical reasons easy to understand the above slide is limited to represent the state of 2 processes. Each axis is scaled from 0In a system with a processor that works alternatively to the different processes, the diagram representing the evolution of the 2 considered processes will have a staircase shape, because every time there can be only one process progressing. In the case of a multiprocessor system, the state will be represented by a continuous line linking the origin (start of the 2 processes) at the point representing the end of the 2 processes.
In the following slides, only solid lines will be used, because the study of mutual blockings it is not necessary to distinguish between mono- and multi-processors.
Mutual blockages can have several causes. For simplicity, only blockages caused by the sharing of resources will be considered here. Other blockage causes can easily been reduced to this one. We will show by parallel coloured lines to the axes which resources are needed for each process in which phases of their execution.

29. The processes P1 and P2 both use for part of their execution 2 resources, R1 and R2. Using R1 and R2 are respectively indicated by the red and blue lines parallel to the axes. Two different routes are represented, the first green does not lead to a blockage, while the second, purple, led to a deadlock.
In the green course, P1 is progressing rapidly at the start and gets the resources R1 and R2 before P2 requests them. When P2 need resources, it can not get them and remains blocked until P1 release these resources. Then P1 and P2 can reach their final state without any problems.
In the purple route, P2 is the first to ask R1, which receives it. Then P1 is requesting R2, it also receives it. Later, P1 needs R1, which has already been assigned to P2. P1 is blocked until P2 releases R1 but before releasing R1, P2 asks for R2. P2 will be also blocked until P1 releases R2. Both processes P1 and P2 are therefore blocked and will remain indefinitely since they depend on each other to be released.
In the diagram you can see that from the moment the purple trajectory entered the red rectangle that is to say, when P2 and P1 have received R1 R2 the situation was hopelessly compromised. Both processes could still continue inside the red rectangle but could not get out. The red rectangle is called "deadlock zone".

30. What can be done to avoid a deadlock?
Two strategies are possible: the prevention and correction.
To understand these different strategies one can compare the deadlock area to a hole in which one might fall. Prevent blockage could be to plug the hole or to install a fence around. The correction simply provides a ladder out of the hole after one fell there.
The following slides will shown how the blocking area can be reduced to zero by imposing a particular order for the request of shared resources and how to avoid getting into a

blocking area by performing some due diligence to all allocated resources.
Finally, a technique allowing to get out of a deadlock situation will also be described.

31. In the case of a two-process system, the blocking area is the area that corresponds to the state where both of the process already holds a resource that will be necessary for the pursuit of the other process.
By forcing all processes to request at once all the resources they need, the surface of the blocking area is reduced to zero and there can no longer be any blocking. The only drawback to this technique is often premature reservation of resources to other processes.

32. An method somewhat less radical than the one before to eliminate the blocking area is to number all resources and force all processes to acquire resources in increasing order of their number.
Thus in the example used in the previous slide, P1 should ask R1 and R2 in a single request since it needs R2 before R1, while P2 can start by asking R1 and does only ask for R2 it's really needed.
This technique reduces, like the previous one, the zero blocking area and may be slightly less expensive in terms of unnecessarily reserved resources.

33. The technique known as "banker" invented by Nico Haberman, does not try to remove the blocking area but prevents from entering by blocking any process requiring a resource allocation that would enter the system in the blocking area .
To implement this algorithm each process must announce in advance all the resources it needs (if there are any more than one resources required as a process using only one resource will never cause blocking).
The blockages prevention system maintains a table that lists all the resources that each process is likely to ask as well as a graph whose nodes are all known system processes.
Whenever a process requests a resource the prevention system completes the graph by adding an arrow oriented from the applicant process to all other processes that may ask the same resource and does not grant the resource if the new graph contains no loop.
The principle of the algorithm will be explained in the next slide, with an example that uses more than two processes and more than two resources since an example with 2 processes and 2 resources fail to show the essence of the algorithm.

34. In the example shown in the slide above, P1 requested R1 and obtained satisfaction since the graph contained only one arrow from P1 to P2. P2 then asked R3, arrows from P2 to P3 and P4 have been added, but as there was still no loops R3 was granted. Finally P3 asks R2, a P3 to P1 arrow has been added to the graph, resulting in the appearance of a loop P1, P2, P3. This means that if R2 was given to P3, the latter process holds a resource that can be requested by P1, which already owns a resource that can be requested by P2, which in turn holds a resource that can be requested by P3, in other words a situation that may cause a fatal blockage.
The banker's algorithm was used in most multi-tasking operating systems 1970 and 80 but that the time required for a search loop in a graph grows roughly exponentially with the number of nodes in this graph results in the prevention of blockages becomes prohibitively expensive in a modern system where hundreds of processes can share system resources.

35. To get out of a deadlock, you must first detect the blockage, which can be done by establishing the same graph as that used in the banker's algorithm, but limited to the processes which were observed for a certain time. Using the same graph, it can determine which resource allocation in which process permits the entry into the blocking area. Then to get out of the deadlock, there is nothing more then to restore the blocking process in the state it had before getting the fatal resource. To restore a process in a previous state, you must have kept a copy of the state of the process at that time. This implies that the status of each process is fully backed up before each resource allocation, so the cost is not negligible, but often more acceptable than the cost of prevention algorithm because, anyway, the frequent backup the system state is required to prevent data loss during technical incidents. This would mean that the deadlock is in contemporary systems, treated like any other hardware failure or system logic.

11 Computer Performance Evaluation

1. In this chapter, some aspects of computer system's performance evaluation will be introduced.

2. Performance evaluations can have several, quite different objectives:
The most common motivation is the selection of a product for a given application. It is quite obvious that such evaluations will be subject to criticism from the vendors who were not selected. A very careful objective approach is required to answer such criticisms.
A second, even more delicate, motivation for performance evaluations occurs when a purchased system does not fulfill the expectations of its buyer. Such situation results often in a neutral expert comparing the actual performance with the specified one.
Existing systems are often expanded during their lifetime by increasing the memory size, by adding disc storage, etc. It is advisable to evaluate the impact of such extensions before the investment is made.
Finally, vendors that plan to launch new systems preferably compare there planned system with those of the competition before making huge investments in the design of the new system.

3. It is not possible to define the performance of a computer system independently from its workload. Therefor, the correct definition of the workload is absolutely necessary before performance comparisons can be undertaken for selecting a new system. However, defining precisely and quantitatively a workload that represents the actual workload the future system will have to handle is often even more difficult than the performance evaluation itself.

4. When new systems are to be purchased for replacing existing ones, the workload of the existing system is often used to compare the performances of the candidates. This is OK when the new systems have architectures that are quite similar to the ancient ones. Otherwise, the candidates with architectures close to the ancient will have a significant advantage, because, over time, the existing applications have most often been optimized for the existing system and these optimizations might have adverse effects on systems with a radically different architecture.
As an example, one could consider matrix inversions, if the arithmetic unit is powerful while memory is scarce other algorithms will be used than those used when there is plenty of memory, but the arithmetic unit doesn't have facilities for handling efficiently indices.

5. A final consideration that needs to be made prior to any performance evaluation or specification is the exact contour of the system considered. It is obvious that the final user is interested by the global performance of the entire system, but it is much simpler to specify and evaluate the performance of well defined parts of a system. Unfortunately,

the global performance of a system is a highly non linear function of the performances of its components. In fact, in most systems, the global performance is determined by the one single component that constitutes a bottleneck in a specific application.
This means that often performance studies will have to try to identify bottlenecks and dimension the different components of a system in such a way that they do not result in large expensive parts of the system being underutilized.

6. An extreme, real-life, example of the importance of paying attention to the entire system rather than to its components comes from a supermarket chain that suffered considerable losses due to a hardware failure of the computer system that controlled all their cash registers, and subsequently contracted with a hardware vendor that guaranteed, at considerable cost, a 100
7. Any objective comparison of the performance of systems should be based upon well defined, measurable indices.
There might be other important criteria for qualifying the performance of systems, such as their "user friendliness", but despite many efforts, this highly subjective index has never been clearly defined nor made measurable.

10. The availability index is most often expressed as a percentage of time the system is operational.
One should realize that truly interactive services, such as automatic teller systems, cash registers in shops and on line services such as Google or Facebook need to have impressively high availabilities, in the order of at least 99.99
11. Availability figures are studied in general by means of failure probabilities and average recovery times.

12. Many different indices are being used to evaluate the amount of work a computer can perform per unit of time. This multitude of indices is a clear indication that that there is no really good index for that purpose!
Some indices are based upon hardware characteristics of the processor, such as the clock speed, or the number of instructions per second that can be executed.
Others take a more global view by measuring how long it takes to execute a specific test program.

14. Many different indices are being used to evaluate the amount of work a computer can perform per unit of time. This multitude of indices is a clear indication that that there is no really good index for that purpose!
Some indices are based upon hardware characteristics of the processor, such as the clock speed, or the number of instructions per second that can be executed.
Others take a more global view by measuring how long it takes to execute a specific test program.

15. The influence of the clock speed of a processor on its productivity is very hard to determine. Depending on the circuitry used, the execution of similar instructions might take a quite different number of clock cycles, as illustrated by this (very old) example. Such clear differences are no longer visible in modern processors due to various performance enhancing techniques such as pipelining, but they nevertheless subsist, even between processors having the same instruction set (such as the processors for PCs manufactured by Intel and AMD)

16. An other reason why processor clock speeds give a very poor indication of the productivity of a computer is the slowness of central memory compared to the speed of the processor itself. When accessing memory, almost all processors need to wait till the memory has accomplished the requested read or write. The average access time of memories is reduced by caching techniques, so that productivity will much more be influenced by the size of the cache than by the speed of the processor clock.

17. In order to avoid the uncertainties linked with the clock speed, productivity is often expressed in terms of instructions executed per second. To compare the productivity of computers with exactly the same architecture and the same instruction set, this can be a useful index, but when it is used to compare the productivity of very different architectures, such as RISC and CISC, the MIPS figures are almost meaningless.

19. The productivity of large scientific computers is often estimated by counting the number of floating point operations it can execute per second. As all scientific computers use various forms of parallel processing and vector instructions to increase their productivity, the overall performance will depend heavily on how well the problem solved can use the available resources. As an example, with a single instruction, the Cray supercomputer could add two 64 dimensional vectors, but if your problem is in a traditional 3 dimensional space, you will not be using all the available resources.

20. By multiplying the number of processors the FLOPS figures are spectacularly growing, exceeding now the 1018 FLOPS, but this increase in raw calculating power is poorly reflected in productivity figures, as accessing the data to feed the arithmetical units has become the bottleneck in many applications.

21. For supercomputers a new hardware index is gaining importance over the FLOPS figure, it is the TEPS figure, that gives the number of pointers across memory pages that can be followed per second.
This new index is a clear consequence of the observed inability of application programs to keep the available floating point arithmetic units busy when huge data sets need to be analyzed.

22. As hardware indices do not permit to predict accurately productivity, test programs, called benchmarks, are most often used to evaluate that productivity in more or less standardized conditions.

23. The main advantage of benchmarks is that they include the entire system in their measurements.
The main disadvantage is that benchmark programs do not always represent accurately the true workload, so that their results are not always useful.

24. Two varieties of benchmarks exist: synthetic and natural benchmarks.
Synthetic benchmarks are small programs, easily portable, that can often be parametrized to tune the generated workload in function of the real workload.
Most of the synthetic benchmarks were designed at a time cache memories and optimizing compilers were not yet often used and therefor they do not allow an accurate estimation op the productivity of modern workloads on modern computer systems. For instance many synthetic benchmarks are small enough to reside permanently in the cache of modern computers.

25. The most popular synthetic benchmark is the Whetstone benchmark, designed in

the seventies to represent a workload typical for scientific calculations. It is available in almost all programming languages and is still widely used and quoted in performance comparisons despite the fact it is no longer representative of contemporary workloads.
Some computer manufacturers even designed workstations specially optimized for executing the whetstone benchmark.

26. This small example of the whetstone code shows how productivity of specific examples of code (matrix operations) is evaluated.

27. To avoid all methodological errors associated with synthetic benchmarks, more and more often natural benchmarks, based upon real application programs are being used for productivity tests. They require however considerable efforts to be installed on a large variety of machines.

28. A benchmark, which is somewhat halfway between synthetic and natural benchmarks is the Linpack Benchmark, which repeatedly inverses a matrix. The size of the matrix can be chosen and is only limited by the memory size of the tested computer. As matrix inversion play a major role in scientific calculations, this benchmark gives a good measure of the productivity of computers for scientific applications.
The author of this benchmark (Jack Dongara) used to publish, every three months, the results obtained when he ran his benchmark on all computers he had access to.

29. This slide shows the productivity of various computers (expressed in MFLOPS) computed by the linpack benchmark for a 100 by 100 and a 1000 by 1000 matrix. These results are compared with the theoretical productivity of these computers.
The remarkable inefficiency of the INTEL iPSC Delta 512, one of the first multiprocessor supercomputers, illustrates quite well the difference between theoretical FLOP figures and the real productivity.

30. In order to reduce and share the costs of benchmarking major computer manufacturers have created a non profit organization, financed by these manufactures, to design and run natural benchmarks and make the results of these comparisons widely available.

31. For PCs several magazines (now almost all on-line) publish comparisons based upon real benchmarks.
Some are lousy: Byte Magazine used the sieve of Erathostenes to compare the productivity of PCs, but people who buy a PC to compute the first 1000 prime numbers are not really representative of the PC market. Ziv-Davies (PC Magazine) tests PCs with a natural benchmark composed of the 8 most sold PC packages the previous year.

32. As these benchmarks are available for reasonable prices, one can use them if PC performances need to be tested.
Considering the efforts needed for doing just that, running PC benchmarks makes only sense for large purchases of thousands of PCs.

33. Sometimes, the distinction between productivity and responsiveness is not clear. In case of transactions that require a lot of processing, a good productivity is a necessary condition for good responsiveness, but it is not a sufficient condition, the process handling the transaction must also have a sufficient priority to ensure a good responsiveness, regardless of the productivity. Moreover, Responsiveness is often experienced through an access network that can add significant delays to the total duration of a transaction.

34. Defining clearly what an interactive response time is requires a careful analysis of the structure of a command and the corresponding answer. If, for instance, one wants to measure the response time of a web server, a distinction is needed between the response measured at the server or the response measured at the client, because, in the second case, one is not only testing the server, but also the entire communication link between server and client.

36. This can be useful for detecting bottlenecks or estimating user satisfaction.

37. Various techniques can be used to evaluate performance indices.
First of all one can make measurements on real systems with a real workload. This technique is quite good for measuring batch productivity.
In interactive environments, it is fairly difficult to create a real, reproducible workload. Therefor, interactive performance is often evaluated by means of a simulated workload, where a computer is used to simulate the users.
As in many circumstances, the actual computer whose performance needs to be evaluated is not available, it can be replaced by a simulator, that runs the test programs and provides indication about time and resources needed for their execution.
Finally, for some more theoretical studies, Markov chain models can be used to establish analytical models of the behavior of a computer system.

38. The measurement of the responsiveness and/or the productivity of an interactive multi-users system will be briefly described here.

39. First of all, a precise script, describing all interactions between user and computer should be established.
When writing such a script the author should not forget that the computer can give unexpected answers, such as "division by 0"or "stack overflow" to some commands. The script should recognize unexpected answers and react to them in such a way that the continuation of the test is not jeopardized. One should also avoid to build too complex scripts, with many branches. For instance, one could specify that, for any unexpected answer, the user should log out and start again his/her session.

40. The example of a script shows how a UNIX login procedure could be described. The Begintransaction and Endtransaction declarations allow labeling transactions and indicate where timestamps are required.
The Timeout declaration specifies that after 12 second without any expected answer the script shoul be discontinued and replaced by the script called crashproc.

41. Such a script could be executed by human operators in front of workstations, but this would generate a lot of timing problems and script errors.

42. To prevent difficulties with human operators, they are usually replaced by a computer program interpreting the script. This program could run on the computer under test, but using a separate computer for it is preferable to minimize interference between the real-time user emulator and the system under test.

43. It is important that the computer executing the scripts is powerful enough, as its response time will be added both to the measured response time of the system under test and to the think time specified in the script. A response time of the User emulator which is not negligible with respect to the response time of the system under test and the specified think time can totally ruin the measurement results.

44. As an example of an analytic model used for computer performance predictions, the simple time sharing system presented in this slide is useful.
We consider n interactive users who submit at random moments requests to the processor. The processor processes these requests and sends back the answer to the user. On average, the processor needs $1/\mu$ seconds to process a request and after receiving an answer, the user thinks on average $1/\lambda$ seconds before sending a new request.

46. We introduce a state variable j that counts the number of users waiting and we want to compute the probability Pj that the system is in state j at a given moment.

47. Supposing that the system is in state j, we can compute the probability that the system will evolve towards
- state j+1 because one of the n-j thinkers stops thinking and generates a new request
- state j-1 because the processor completes the tasks it was doing.

This means that we can establish a set of n+1 homogeneous linear equations giving all the Pj as a function of λ and μ,

48. Solving that system allows us to express Pj in function of n, λ, μ and P0 and, by remembering that, as the system always has one of the n states, the sum of all Pj must be 1, we can find an expression for P0 as a function of n, λ and μ.

49. To introduce the notion of response time R in this model, we should compute the flow of request entering and leaving the processor.
Entering the processor, the fraction of users who are not waiting for an answer will generate λ requests per second. That fraction is the average think time divided by the sum of the think time and the response time R.
The processor will generate μ answers per second, when it is not in the idle state and we know that the probability of being in the idle state is P0.
By stating that the system is in a steady state, the flow of requests entering the processor and the flow of answers leaving the processor must be equal, which results in an expression that allows us to compute R as a function of n, λ and μ.

50. As, for a large number of users the probability of the processor being idle is very small, P0 will be close to 0 and one can observe that the response time will grow linearly with n, with a slope of $1/\mu$.
If there is only one or a few users, the response time will be close to $1/\mu$, as that is the time the processor needs to serve a single user.
The asymptote for large values of n intersects with the n axis at μ/λ. This is called the saturation point, where response times start to grow rapidly with the number of users.
This model shows that, even overloaded, time sharing systems will not crash, they will only become slower and slower.