Computers and Computing

A Personal Perspective

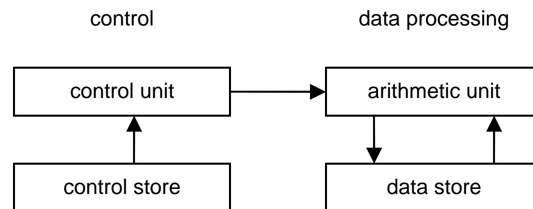
Niklaus Wirth, December 2015

The Era Before

Historians date the origin of computing far back almost in the middle age. Here we prefer to start when computers became commercially available and started to play a role in business and science. This was around 1960 or even slightly before. Only a few inventions had caused this development.

On the physical level, it had been the advent of electronics that made computers objects of interest. Electronics offered the necessary speed. Up to these times, the active elements of electronic circuits were vacuum tubes. Heated cathodes emitted electrons, sucked up by the surrounding anode. In between a grid allowed to control the flow of electrons by applying a negative field. Early computers contained thousands of such tubes which, due the cathode heating and the high voltage at the anode consumed a considerable amount of energy, a few Watts every tube. With their numerous tubes and heavy power supplies, these early computers were quite monstrous and filled entire rooms.

On the logical level, the innovation of John von Neumann (1944) made computers into what they are. Before, they typically consisted of a control unit and a separate arithmetic unit. The former fetched instructions from an instruction store one after the other, the latter interpreted them and accordingly altered data in the data store.



Von Neumann introduced two fundamental concepts. First, the two stores were united. This allowed the computer to generate data and then interpret them as instructions. Thereby, the computer mutated from a special purpose device (with always the same program) to a *universal machine,* the programmable computer. And this is what distinguishes the computer from all other technical devices up to this day.

The second fundamental idea was the conditional instruction. It has a different result depending on previously computed values, i.e. the state of the machine.

Experimental computers developed as prototypes, usually at universities, between 1945 and 1960, all adopted these fundamental ideas.

Pioneering Years

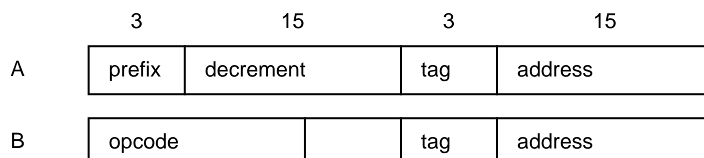
It was in 1959 in Québec, Canada, when I first dealt with a computer. I had registered in a course on Numerical Analysis, taught by Prof. Goodspeed. But the computer was not up to the professor's name. In fact, it was not only slow, but "down" whenever we had concocted a small program as part of an exercise. Hence, my first encounter with a computer was dismal and discouraging. I cannot explain how I kept up my interest in computing. But I sensed it would be important in the future.

The available computer was an Alwac III-3E. It rested on several thousands of electronic tubes, mostly double triodes, and it occupied a full room. A tube had typically a life-time of 10'000 hours. Hence, you could reckon that one - and thereby the entire system - would die every few hours. And it did! Programming was done by constructing tables of instructions in hexadecimal code. I thought there was potential for improvement.

My second go at computers was a year later in Berkeley. There was a large IBM 704 computer, one of the first built with transistors instead of tubes, and therefore much more reliable. It was accessible only via punched card decks delivered to the Computing Center. Card decks from various customers were piled up by operators, and the system automatically processed one deck after another. This was called *batch processing*. Results could be fetched several hours or even days later on endless paper as texts consisting of digits and capital letters. Programming was in the new language *Fortran* (formula translation). This was a definite improvement over hexadecimal code, but I could no longer understand how the computer operated. Again, I felt that there was room for improvement.

In the years 1960-1965 large, remote computers dominated the computing world. They became called *main frames* and filled large, air-conditioned rooms. All brands were more or less the same, differing by word-size, memory size, and, of course, execution speed. They typically featured one or two accumulator registers. Three innovations deserve being mentioned. The first is the *index register*, whose value is added to an operand address during the fetch cycle.

As a representative of these main frames, we very briefly outline the IBM 704. It featured a memory (store) of 32768 (= 2^{15}) words of 36 bits each. At the time, character sets consisted of 64 (= 2^6) characters; hence, computers featured word sizes which were multiples of 6 (6-bit bytes!). Numbers were stored in the sign-magnitude form, a single bit for the sign, 35 bits for the absolute value. The 704's core consisted of the 36-bit accumulator register (in which supposedly sums were to be accumulated), the MQ register (holding the multiplier for multiplications or the quotient for divisions), and 3 15-bit index registers. There existed two instruction formats:



Typical type-B instructions were LDA, STA, ADD for loading, storing and adding a variable (in memory) to the accumulator. The effective address was computed as the sum of the address field and the index register specified by the tag field. There were only a few type-A instructions. An example is TIX, implying a transfer to the given address, and subtraction of the decrement from the specified index register. This complex instruction was designed to implement fast loops with consecutive index values.

The second innovation was the *interrupt*, implemented in the later 7090 model. Here an external signal from an input/output device may cause the interruption of the current instruction sequence and issue a call of a subroutine, a so-called interrupt handler. The consequences for programming were considerable.

The third innovation were *floating-point* numbers. There had occurred many heated discussions on how to represent fractional (real) numbers. The options were fixed-point and floating point In the latter case, a (real) number x is represented by two integers packed into a single word, say e and m, with $x = B^e * m$. and $1.0 <= m < B$. Typically B = 2. The 704 took about 80 μs for a floating-point addition, and 8 μs for an integer addition.

The most numerable main frames were the IBM 704, 7090, 7094, the UNIVAC 1108, and (a bit later) the CDC 1604 (with 48-bit words and 6 index registers).

Apart from the 704, I encountered in 1961 a much smaller computer available at the institute led by Prof. Harry Huskey. This was a Bendix G15, designed by Huskey based on "Turing's computer" at the NPL in England. It was a stand-alone machine, and one had to sign up time on it by the hour and a few days ahead. It was an ingenious design with few tubes and a magnetic drum store. It used a serial adder, essentially with a single tube. Programming consisted of constructing tables of encoded instructions What made programming particularly intriguing, or rather cumbersome and tricky, was that their placement on the drum tracks was of crucial importance for execution speed. Any word appeared under the reading head only once during a drum revolution, i.e. about every 40 ms in the worst case. Evidently, the G15 was an interesting device for puzzle-minded guys, but no pointer to the future. However, I could now fully understand how such a machine functioned. This is what constitutes progress.

Analog vs. digital, binary vs. decimal

In the 1960s, the computing world consisted of two camps: the analog and the digital world. It was by no means clear to which the future would belong. Typically, mathematicians belonged to the digital, electrical engineers to the analog camps. Digital computers were exact, but required very many expensive components, whereas engineers were used to live with approximate results, correct to perhaps 3 or 4 decimal digits. In the analog world, only addition and integration (over time) is easily achieved, whereas multiplication and division are difficult or almost impossible. Today, the then heated controversy over analog vs. digital has completely vanished. This is not only due to the enormous reduction in price of digital circuitry, but primarily because analog values were very hard, if not

impossible, to store. Computers are now not so much used to compute, but rather to store data. As a consequence, analog computers have died out.

Remains to be noted that digital is actually an illusion. The physical world is analog except deep down on the atomic level of quantum physics. Even stored values are often not digital (not binary). Modern memories (2016) consist of cells capable of distinguishing between 8 (analog) values.

As an aside, the adjective digital stems from the Latin word *digis* (digitis) meaning finger. This suggests that digital computers count by holding up fingers like first year pupils. The adjective *analog* reflects the fact that a computed value is represented by an analogous quantity, typically a voltage. It would be much more sensible to distinguish between discrete and continuous, rather than digital and analog.

Within the digital community, there existed a further schism. It divided the world into binary and decimal camps. Binary computers - which are the rule today - represent integers as binary numbers, i.e. as a sequence of *binary digits* (bits), the one at position n with weight 2^n. Decimal computers represent them as sequences of *decimal digits*, the one at position n with weight 10^n, each decimal digit being encoded by 4 bits. Large companies offered both kinds of computers at considerable expense, although evidently the binary representation is definitely more economical. The reason behind this separation lay in the financial world's insistence that computers produce in every case exactly the same results as calculation by hand would, even if erroneous. Such errors may happen in the case of division.

The dilemma was "solved" in 1964 by IBM's System 360 by offering *both* number representations by a large instruction set.

First Programming Languages (1957-62)

Constructing long sequences of instructions had been a tedious and error-prone activity. A remedy was first presented by IBM with the language **Fortran**, standing for *Formula Translator*. The translator, a program later to be called *compiler,* was developed by a group under the guidance of John Backus in 1957. The goal was to replace assembler code by mathematical formulas, by variables and expressions. Whereas in assembler code every single instruction had to be listed, now this could be expressed more appealingly in a single line (punched card). For example, the instructions to add two variables and store the result

```
LOD    X
ADD    Y
STO    Z
```

this would be expressed in Fortran as

```
Z = X + Y
```

FORTRAN was oriented to input from punched cards. They postulated a line of 80 characters (capital letters only), with columns 73-80 reserved for card numbering (useful if you dropped the card deck!). A C in column 7 meant that the card

contained a comment. There were no declarations and no data types, except that a variable's name beginning with I, J, ..., N denoted an integer, all others floating-point (real) numbers. FORTRAN featured subroutines. They were to be compiled independently, and no checking of parameter consistency was performed. Fortran introduced the notion of program libraries (mostly for mathematical functions).

Fortran proved to be a remarkable feat. The techniques of translation had to be invented from scratch. Fortran remained the Standard tool for decades to come. Particularly research laboratories and universities, where numeric calculations were dominant, stuck to Fortran for decades to come, although more powerful languages had emerged. As time went on, they had amassed huge amounts of programs, now called *software*, and invested so much effort, that it seemed impossible to depart from Fortran. This led to the phenomenon *legacy software*.

The first alternative to Fortran was presented in 1958 by a group of European scientists. Their language was called **Algol,** standing for *Algorithmic Language*. It was revised by a group of 13 European and American scientists and was announced at the IFIP (International Federation of Information Processing Societies) Congress in Paris in 1960. This language became known as *Algol 60* and was supposed to become *the* international standard. It was felt that such an important concept was not to be left to a single commercial company. The following highlights were the hallmark of Algol 60:

1. The language was defined by a concise report, edited by Peter Naur. The basis was a formal notation to specify the structure (syntax) of the language.

2. The notation for program texts was in a free format, merely as sequences of characters, or rather basic symbol of the language. These symbols constituted its vocabulary. There was no reference to 80-column punched cards.

3. Programs were given a structure by the grammar (or syntax) of the language. This syntax was defined by a set of productions (or derivation rules), based on a strictly defined meta-notation called BNF (Backus-Naur form).

4. The vocabulary consisted of (capital and lower-case) letters, digits, and special symbols (such as + - < > ; etc.), and reserved words (such as **begin, end**).

5. Every variable was to be declared in a declaration.

6. Every variable, constant, or expression was given a *type*. The standard data types were Integer, Real, and Boolean.

7. The type Boolean specified the two logical values True and False, and logical operators.

8. Expressions could contain sub expressions. Their syntax was recursive.

9. Procedures (in Fortran called subroutines) could be recursive.

10. A sequence of statements could be expressed as a begin-end *block* with declarations. Variables declared within a block were considered as local to this block. The concept of *locality* proved to be of utmost importance in programming methodology.

11. The language definition was *machine-independent.*

It was the structured description of the language, the syntax, that promoted the term *language*, although it is actually a misnomer. A language is spoken, a programming language is not. The term *formalism* would have been more appropriate, but the term language persists to the present day.

Algol, just as Fortran, was oriented to the needs of numerical computation. This in contrast to commercial applications such as accounting and book-keeping. But while the continuing growth of the user community was assured for Fortran due to the support of a mighty industrial player, that of Algol never became large. Algol remained confined to the academic communities, mainly in Europe, and remained largely unknown in the US, although it was far superior

To cater for the needs of the commercial world, the language *Cobol* was invented in 1962 (Common Business Oriented Language, Jean Sammet of IBM) and promoted by the US Department of Defense (DoD). By academics it was considered as far too verbose, trying to accommodate users disinclined to mathematical conciseness. Cobol obtained a very large user community. Just as computers with binary vs. decimal arithmetic had separated the academic from the commercial world, so did now the languages.

Another notable development, was the language **Lisp** (List Processing, McCarthy, 1962). In contrast to all others, this language was very terse. Not repetition but recursion was its distinctive property. All elements of data were nodes of a dynamic data structure. These nodes were either numbers or (short) names, or pairs referencing two descendant nodes. Thereby (recursive) structures, such as lists or trees of arbitrary complexity could be generated during program execution (i.e. dynamically)

The amazing property of Lisp was its utmost simplicity. There existed only a single data type, appearing in two variants. The first was a pair of pointers to two other elements. The second was a value, typically a number or a few letters. There existed only a few basic functions allowing to construct arbitrary lists and trees. A Lisp-program was a single function application, which in turn called other (programmer-defined) functions Lisp was therefore said to be a *functional language*:

CONS (x, y)	generates a new element pointing to the pair x and y.
CAR(x)	the first element of the pair x
CDR(x)	the second element of the pair x
ATOM(x)	telling whether x is a pair or an atom (number)

This property brought attention to the problem of storage allocation and, mainly, storage reclamation to the foreground. The solution lay in automating storage recovery (reuse of storage of no longer accessible data). It was soon to become known as *garbage collection*.

Oddly enough, Lisp became known as the language of artificial intelligence, a subject that came up during this time, encompassing many applications that were of a non-numerical nature. But there was no "artificial intelligence" in the language

itself. Nevertheless, the artificial intelligence community staunchly adhered to Lisp. One distinctive feature may be accountable for this fact: LISP-code itself was represented by LISP data structures, and therefore a generated structure could be regarded as an interpretable program. Hence, LISP programs could extend and alter themselves, a rather dangerous facility.

Although Algol never became a widely used language, its influence was profound in the fields of language design and implementation. This was due to its precise, formal definition. A flurry of activities started in academia, and languages proliferated in all directions. They were mostly local developments that soon vanished from the scene again. The new subject of *computer science* emerged mostly due to language design and programming. Languages hardly fitted into mathematics nor electrical engineering departments. Of particular relevance was the subject of parsing formal texts, the substance of compilers. Parsers could now be derived directly from the syntax specifications of a language, and were no longer ad-hoc, empiric algorithms. The two principles of top-down and bottom-up parsing became heavy competitors. Unfortunately, it was believed that a syntax could well be defined without giving consideration to parsing, and as a result rather sophisticated parsing schemes emerged. This was unfortunate, because with few exceptions a language could be given a syntax that could be treated by a simple and efficient parser.

Some efforts attempted to formalize not only the syntax, but also the semantics (meaning). The formal definition of semantics, however, is an elusive subject, and therefore these efforts were condemned to fail. Still, all these activities gave computer science a touch of academic respectability and maturity, counteracting the view that CS was primarily an arena of practitioners.

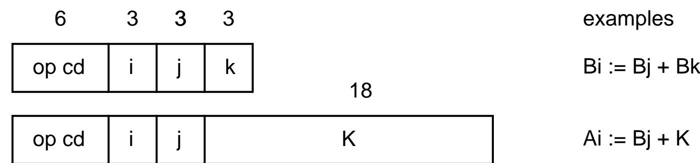
The first Reduced Instruction Set Computer (RISC) (1962)

From 1950 to 1970 the National Laboratories promoting the development of atomic weapons (Los Alamos, Livermore) came up with insatiable demands of computing power. Their calls for faster computers presented massive challenges to computer designers, and their financial resources promised enormous challenges. The goal was, simply speaking, to develop a super-computer.

The company that first picked up the challenge was CDC (Control Data Corporation, Minneapolis), and its chief designer Seymour Cray. After the CDC 1604, he undertook a radical departure from the ongoing trend, which had led to computers with more and more complicated and large instruction sets. His idea was to provide only a few, basic instructions, all executable with greatest speed. Apart from this, he relied on the new component family with ECL-technology (emitter-coupled logic), which provided very fast transistors, but with the disadvantage of requiring large amounts of current (energy). As a consequence, his main challenge became not designing logic, but cooling the heated transistors. This computer was built before integrated circuits (chips) had become available.

The CDC 6000 architecture is briefly sketched as follows. The memory consists of 262144 (= 2^{18}) 60-bit words, and the processor contains three sets of 8 registers.

The data registers X0 - X7 are 60 bits wide, the address registers A0 - A7, and the index registers B0 - B7 are 18 bits wide. Instructions are either 15 or 30 bits long, 2, 3, or 4 fitting into a 60-bit word.



The computer used 1-complement representation of integers. This allowed faster sign inversion, but provided two forms for zero, giving rise to several unpleasant problems.

A clever oddity is the rule that assignments to an A-register have a side-effect on the corresponding X-register. Assigning an address to an A-register was the only way to transfer data from and to memory. (This "feature" was loved by compiler builders).

Ai := a	→	Xi := mem[a]	for i < 6	load
Ai := a	→	mem[a] := Xi	for i >= 6	store

Compiler construction (1960 -)

Around 1962 language design and compiler implementation had become important topics in Computer Science. The publication of Algol in 1960 with its rigorous definition of its syntax gave rise to strong efforts in the development of parsing methods. It was recognized that compilers must be driven by their parser, decomposing the text into its structural entities, and that then code was to be generated for the individual entities. Given a complex syntax, parsing was to be efficient.

Soon two basically different methods emerged, the top-down and the bottom-up methods. The direction is that of the traversal of the syntax tree of the text to be parsed. Hence, the top-down method always starts with "program" as the parsing goal, and then subdividing this goal into sub goals.

This intuitively obvious method is somewhat limited and unsuitable for certain kinds of grammars. A more powerful method is the bottoms-up parsing, which simply reads text and recognizes structural units, and then composes higher units out of subunits, until the goal "program" is reached. This method is table-driven. The tables can be generated automatically from the given syntax.

Whereas early compilers were essentially, but not systematically top-down parsers, subsequent developments favored the bottom-up method, because it admitted more complex grammars, which seemed to be advantageous for the ever growing complexity of new languages such as PL/1 (IBM) and Pascal.

But the argument is a fallacy. As the language designer has many freedoms, he must not ignore the problems of parsing, but choose the syntax so that parsing is a simple as possible. Evidently, this recommendation was gallantly ignored,

particularly after parser generators had become available. A syntax seemed to be God-given, and it was up to the implementers to cope with possible difficulties.

However, the real crux of compiler construction lies in code generation which resists systematization until the present days. What made the task so difficult was the evident mismatch of given computer architectures and language features. Computers had been designed with Fortran in mind, and their instruction sets were oriented towards the language elements that Fortran carried. But Algol 60 and PL/1 changed the scene drastically. A few example must suffice.

The first, and perhaps minor challenge was the handling of complicated expressions with only 1 or 2 registers available. This topic became easier with the later advent of computers with banks of registers, and much effort went into optimizing the code. A much greater challenge was the handling of procedures with local variable. These allowed the reuse of storage without the programmer's having to be aware of it. The admission of recursion in Algol made it possible that several incarnations of local variables had to be allocated (dynamically, at run-time). Therefore variables had to be addressed via a register designating the relevant set. All this gave rise to the use of a stack (of local variable frames) supported by dedicated registers holding base addresses. But such were not available of the prevalent computers.

Therefore, these facilities had to be implemented by software, sharing the few registers for different purposes, and thereby reducing execution speed. As a result, Algol with its generality and with recursion was said to be intrinsically inefficient and inferior to Fortran. And then, numeric computations did not require recursion. Hence, Fortran was for "the real world", whereas Algol appeared as a luxury for tinkerers.

The B-5000 (1963)

It was to be hoped that the innovative design of Algol would extend its influence on computer development. And it did. Algol turned out to be difficult, if not impossible, to implement efficiently and to make it competitive with Fortran concerning execution speed. Already in the early 1960s some computers appeared that catered to Algol's specific features, emphasis given to the implementation of general expressions and of recursive procedures.

In order to translate expressions of Algol's generality into sequences of simple instructions, they are typically converted from infix to postfix form:

```
x + y                 x y +
x + y + z             x y + z +
(x+y) * (z-w)         x y + z w - *
```

This allows a straight-forward evaluation, if operands can be stacked (pushed onto a first-in-last-out stack). Then operators simply replace operands by the result. Computer designers therefore accommodated compiler designers by providing a push-down stack in place of a register array. Effectively, the stack is an array with implied up-down counter used as index, and the net effect is that register numbers can be omitted in the code. This leads to denser code.

The idea of a (hardware) stack appeared as very attractive, and both the British GE KDF9 and the Dutch Electronica X8 implemented it. The Burroughs B5000 also adopted the concept, implementing the top two elements as registers. Transfer of lower elements of the stack to and from memory was automated by a fairly complex scheme.

It is noteworthy that the B5000 used the same encoding for integers and real numbers with a 39-bit mantissa, a 6-bit exponent, and a base B = 8, integers with exponent = 0. The program code consists of 12-bit syllables, 4 to a word.

In the long run, the idea of the expression stack did not catch on and disappeared form the scenery. Direct register numbers in the code seemed preferable, and sophisticated optimization algorithms make the best of it.

The most challenging feature of Algol, however, were recursive calls of procedures. Consider

```
PROCEDURE P(x: INTEGER);
BEGIN
    IF x > 0 THEN P(x-1) END
END P
```

For every (recursive) call of F a new incarnation of x is created and must be allocated. The same would hold for local variables. The evident solution is a stack of frames, each holding parameters, local variables, and the return address. As a consequence, every variable cannot be accessed by a simple, static address, but by an address, to which the base address, the address of the containing frame, is added. This base must be available from an index register, to be initialized upon call of the procedure (and restored on exit).

A nasty problem with existing main frame computers was the acquisition of the return address (also to be deposited in the frame). The worst solution was that of the CDC 6000 super computer. Its call instruction deposited the return address in the memory location preceding the call address (i.e. in front of the procedure's first instruction. This scheme writes into the program, and thereby makes the code non-reentrant. Clearly an unacceptable solution.

The efficient and elegant implementation of recursion requires a suitable addressing mode with specific address registers (stack pointer, frame pointer). The B5000 offers these ingredients. As a result, however, a call instruction constitutes a fairly complicated scheme with several accesses to memory, requiring complex circuitry and time.

But Algol had some further hard nuts in store. A major stumbling block was its so-called name-parameter. It postulated the literal replacement of the formal parameter by the actual parameter. The best example for its use is the following procedure (here translated into the syntax of Pascal):

```
PROCEDURE Sum(i, n: INTEGER; x REAL): REAL;
    VAR s: REAL:
BEGIN s := 0;
    FOR i := 0 TO n-1 DO s := x + s  END ;
    RETURN s
```

```
END Sum;
```

Given further variables

```
k: INTEGER;
a: ARRAY 100 OF REAL;
```

the sum of the elements of *a* can be expressed by the function call

```
Sum(k, 100, a[k])
```

rendering the FOR statement into

```
FOR k := 0 TO 100-1 DO s := a[k] + s  END
```

This implies that for every addition the actual parameter a[k] must be evaluated, and that therefore it must be called like a (parameter-less) function. The feature complicates compilation of procedure calls enormously. The B-5000 pushed this complication from the compiler (software) into hardware: Instructions for fetching a value must first inspect (at run-time) whether an operand is a variable or a function call. Discrimination was based on a bit in every operand, indicating whether it is a simple variable or a descriptor (of the function).

The B5000 introduced the concept of descriptors and is said to have a *descriptor architecture*. In particular, every array is represented by an array descriptor, just as procedures are represented by procedure descriptors. The array descriptor also contains the array index bounds, and an array access tests the index to be within these bounds. The net effect of the descriptor scheme was that almost every variable access required an indirection, and therefore more time.

In most cases, computer engineers had little notion about languages and compilers, and language designers had little knowledge about hardware design and even less about compiling. They were even proud of this and believed that true innovation could occur only by ignoring technical constraints. The result was a gap between software and hardware. The B5000 designers tried to bridge this gap courageously. But in hindsight they went too far. They implemented language features that proved to be not quite realistic, nor truly needed. As a consequence, the B5000 computer, delivered in 1964, one of the first to Stanford University, became a very complex, costly device. It could efficiently handle sophisticated Algol features, but it could hardly compete with conventional machines for everyday applications.

The notion of a family, and the 8-bit byte (1964)

IBM's System 360 was announced in 1964. It brought two other innovations. The first was the notion of a computer *family*. Up to this time, every computer had its own structure and performance; it was sort of unique. Now System 360 consisted of many incarnations of one and the same instruction set. Each incarnation, called *model*, had its distinct performance figures, size and price. But all of them looked the same to the programmer. In this sense they formed a family; they featured the same *architecture.* This is where the word architecture in connection with computers appeared.

It goes without saying that the different models had very different implementations of the hardware, although they were handling the same software. For the first time, the technique of emulation was used extensively. The smaller models used the novel technique of micro-programs. The genuine hardware always interpreted the same program, namely an interpreter of the 360 instruction set. The interpreter was defined in so-called microcode. This micro-code rested in a small but very fast microcode memory, typically a read-only memory implemented in a proprietary technology. The consequence was that some of the 360's instructions were fast, others slow in comparison. It was possible to include some very complex and hard to understand instructions. Some of them even included loops in the micro-code.

The second novelty was that the smallest individually addressable unit in memory was not the word, but the *byte*, and that this byte consisted of 8 bits. So far, that unit was considered as consisting of 6 bits, and the word length of all computers were multiples of 6. An immediate negative consequence had been the limitation of character sets (ASCII and IBM's EBCDIC) to 64 characters. The extension to 256 characters was a welcome benefit.

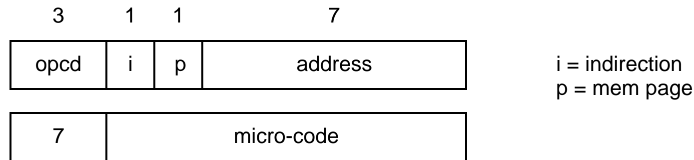
This revolutionary step also dispensed with "variable length data computers", such as the ubiquitous IBM 1401, which featured string instructions, which operated on sequences of 9-bit bytes, the 9th bit acting as a string termination indicator.

Minicomputers, and a time-sharing system (1965)

In the shadow of the large main frame monsters a new generation of computers emerged around 1965: *Mini computers*. The first was the PDP-1 of DEC (Digital Equipment Corp.), later to be followed by the very popular PDP-8. They were mounted on standard 19" racks for typical use in laboratories, and they were built with discrete bipolar junction transistors and magnetic core memories. Their word-length was 18 bits (PDP-1), 12 bits (PDP-8) and 16 bits (HP 2116). Their cycle time lay around 2 us. Every instruction needed 1 or 2 cycles. For the time, this was considered very fast. Their instruction sets contained load and store instructions, logical operations, single-bit shifts, and add and subtract. Multiplication and division was to be programmed. Their memory sizes being small, typically 4K or 8K words, compilers for high-level languages were out of the question. They were connected to a typewriter (teletype) for input and output.

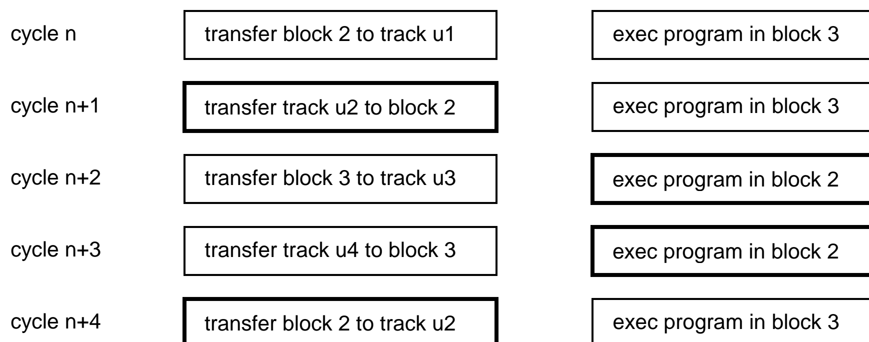
The minicomputers' distinct attraction was their direct accessibility. They were not hidden in computation centers, but allowed interactive use. The old scheme of sign-up hours had returned which, inherently, is inefficient.

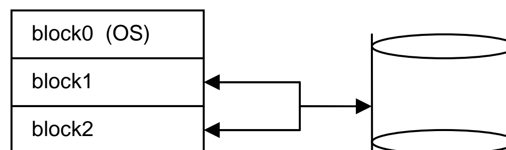
As an example, we sketch the structure of the popular PDP-8. Its memory consisted of magnetic cores, 4096 words, 12 bits long. The computational unit contained the accumulator, also 12 bits long. Instructions came in (essentially) two variants. Memory reference instructions and operators in micro-instruction format.



The memory reference instructions include loading, storing, adding, logical *and*, and jumps. This description makes it clear that such minicomputers were apt to be coded "by hand" with the aid of an assembler. Compilers were out of the question; the resulting code would be too long and inefficient. The memory was small enough to ensure that only small programs could be loaded, typically for controlling equipment and data acquisition.

A most remarkable novelty was the first *time-sharing system* with the goal of allowing fully interactive use without monopolizing the computer. This system was developed by Dennis and van Horn at MIT. In cooperation with DEC they devised an augmented PDP-1. It was beefed up by a drum store with 16 tracks of 4K words in each. Furthermore, the main store was enlarged to 3 blocks of 4K words. The first section was used for a small "operating system" .The system allowed 16 programmers to work on the single computer quasi-simultaneously. It operated in time slices of 33 ms duration. In every slice, three actions occurred: A track was read alternately from the drum into main store section 2 (or 3) , the same main store section was dumped onto the appropriate drum track, and the processor executed the program in main store section 3 (or 2). This allowed every programmer to obtain the impression that a regular PDP-1 was at his exclusive disposal. The scheduling of the processor, i.e. its allocation to an individual user, was quite simple. Users being statically given a place in a ring, the next time-slice was given to the next user in the ring, except, if he was waiting for input or output.





Probably the first such system was installed at Stanford University in 1964 in an experimental lab for teaching programming (Prof. Suppes). In order to facilitate

textual input and control features, a special key was included in the keyboard of each station. It later became ubiquitous and called the CTRL-key.

This idea of time-sharing was simple and fascinating. Soon such systems became a normality. Manufacturers of main frames caught on to the idea and extended their products accordingly (IBM 360/67, GE 645). But they believed that software was omnipotent, and did not restrict themselves to a fixed time slice scheme, nor to fixed memory allocation for every user. Individual, dynamic resource allocation was their vision, memory sizes according to every user's needs, and scheduling individually according to priorities and resource availability. A side-effect was the emergence of memory management units in the hardware of these multi-user computers. This was pioneered by the giant Atlas computer in Manchester.

These were high-flying goals. Nobody had ever designed systems of this complexity. The difficulties had been vastly underestimated, Yet, delivery times had been promised to eager customers. The dire situation was met by employing huge crowds of system programmers, resulting in overwhelming management and communication problems. It ended in what was (in 1968) called the *software crisis*, documented very aptly by Fred Brooks' book *"The mythical man month"*, culminating in the wise insight that "adding man power to a late project makes it later".

Compilers together with OS/360 became what was (perhaps) the first of the software giants, systems that did not rest on coherent design elements, built by armies of programmers, and in their entirety understood by nobody, and as a result of questionable reliability. Failures had been pre-programmed. Updates became a monthly routine.

Software, that is operating system, compilers, and libraries, were delivered by the computer manufacturers like an accessory. As software grew to huge dimensions and generated large cost, the process of "unbundling" started. Software became an object of its own, and its pricing was separated from that of hardware. This occurred not without intervention from legal courts. Software companies emerged. They generated programs usable on many different brands of computers and founded a market of its own.

Second generation languages, and Pascal (1965-1970)

The first of what I call second generation languages was PL/1 created by IBM, or rather a consortium of user organizations. What IBM had done to hardware, was to be done also to software: The unification of the scientific and the commercial computing communities by merging Cobol and Fortran. This would have to be an enormous undertaking with very many actors inserting their concepts and ideas. The result was published in 1965 soon after the announcement of the 360 computer family, when implementation of PL/1 was already under way.

While the Fortran, Cobol, and now PL/1 bandwagons were picking up speed, the academic community with its idea of a clean language resting on solid principles formed a group of scientists under the auspices of IFIP, forming the *Working Group 2.1* in 1963. Its goal was to promote the discipline of programming in

general and of designing a successor of Algol 60 in particular. This group of about 40 people met about every half year at various places spread over Europe and the USA.

Obviously, such a large group was ill-suited to produce a coherent design. There were long and irksome debates about various language features and constructs to be deleted from or added to Algol 60. There were obviously questionable, ill-defined items in Algol 60, but although agreement on their deletion was unanimous, their replacement would cause long arguments. The principal "speakers of the house" were Aad van Wijngaarden from the Mathematisch Centrum in Amsterdam and Fritz Bauer from the Technical University in Munich. In heated discussions the emotions flew high and the antagonists even tore their collars.

A particular source of lengthy arguments were procedure parameters. Algol 60 featured two, the value and the name-parameter. Some argued that the name-parameter was a distinctive feature, even a hallmark of Algol 60, others that it inherently degraded the efficiency of any implementation and was of little use (see chapter on B5000).

There were other peculiarities of Algol 60 that caused puzzlement and heated discussion. Only one of them is exhibited by the following pathological but legal piece of program:

```
PROCEDURE P (q, b);
  BOOLEAN b; PROCEDURE q;
BEGIN INTEGER n;
  PROCEDURE Q; n := n + 1;
   n := 0;
  IF b THEN q(~b, Q);
  Write(n)
END ;
```

Which sequence of numbers would the statement P(P, TRUE) generate? 0, 1 or 1, 0?

After several meetings it became evident that on one side there were "theorists" intent on proposing a design satisfying everybody, based on powerful, generalized foundations, and on the other side "practitioners" with experience in implementing languages, fearing that if a sensible agreement could not be found soon, grounds would be lost forever. After several years, the Group split. The "theorists" pursued what became called *Algol Y*, while the "practitioners" headed for a more modest *Algol X*. The latter was implemented on a IBM 360/50 under the guidance of this writer at Stanford University. It became known under the name *Algol W* in 1966, and used at many universities in Europe and the US. It was less than a monster, but still too heavy.

The Algol Y faction continued its work (mainly in Amsterdam and Vancouver), hoping for a release in a few months. But the difficulties remained, and what was cautiously called Algol 68 was finally implemented in 1972. As far as the computing

world at large was concerned, it proved to be a still birth. What was supposed to become another milestone (after Algol 60), had become a millstone.

Liberated from the burden of reaching compromises within the Group. this writer went along on his own. He designed the language *Pascal* , published in 1970. It gained world-wide acceptance due to its relative completeness and simplicity. Still being a language without the facility of separate compilation of parts of systems and of linking them when needed, it nevertheless satisfied the basic needs of *structured programming* and was ideally suited for teaching this discipline. Efforts to facilitate the porting the compiler to many other (non-IBM) computers helped the spreading of Pascal.

Already around 1966 E.W. Dijkstra (also a member of WG 2.1) published a document entitled *Structured programming*. He had been the designer of the first Algol compiler that implemented Algol fully, including recursion, a hot topic at the time, as recursion was believed to inherently lead to inefficient implementations. Pascal was to implement structured programming, to be the exponent and natural tool to express this metaphor. It extended the concept of structure from statements to data.

Algol 60 featured the conditional statement (if-then-else) and (for repetitions) the for statement with a large variety of options. Still, it was the only way to express iteration, except by explicit jumps, by **go to** statements. These were the major facility for concocting unintelligible programs. They had been the target of an arduous indictment by E.W. Dijkstra in his famous letter titled *Goto statements considered harmful.* As a reaction, Pascal featured a while- and a repeat statement (and a case statement) catering to this critique. But the go to remained; it would have been a shock to programmers at large to have to cope without jumps. Or so it was believed.

In the realm of data structures, Algol offered only the array, a structure with all elements of the same type. Pascal added the record, a unit with elements of possibly different types called fields, the set, and the file (sequence). The key property was that such structures could freely be nested. Not only arrays of arrays (matrices) were possible, but also array of records, and records with array structured fields. These structures were declared as *types* in harmony with integers, real numbers, and Boolean values.

Yet, the wide-spread acceptance of Pascal occurred only six years after its publication. Then Pascal was riding on the back of the new wave of micro-computers, whose cost had come down to be affordable by schools and homes. In order to facilitate porting the compiler to other computers, a hypothetical computer was postulated, and our compiler generated code, so-called P-code, for it. An emulator was simple to program in assembler code for any other architecture. This method contributed significantly to the spread of Pascal.

Our compiler had been sent to (among many others) the University of San Diego (UCSD), and to W. Kahn, who had founded the Borland company. Both embedded the compiler in a system for PCs with a simple "operating system", text editor, and a debugger. Together these components allowed for a very fast "turn-around"

cycle of writing, compiling, testing, and debugging. Apart from this, the system was sold on a floppy disk for some $50. This was the key to its success.

Thus Pascal reached masses of people who had not been "corrupted" by complex industrial products; who did not first have to "un-learn" old ideas. Only later the impact became apparent that Pascal had exerted in countries outside Europe and USA, particularly in Russia and China.

At the same time as Pascal (1970), the language C was designed by D. Ritchie at Bell Labs. When he once confided to me that Pascal and C had come out surprisingly similar, in many ways identical. I could hardly agree, and I still consider C to be an assembler code embellished by a certain, less than elegant syntax. It claimed to attach data types to variables. But what is their usefulness, if compilers do not check for consistency, if type information is treated like comments that may simply be disregarded? I did and still do consider the language C as a (Turing Award -sanctified) curse of computing. C may have been necessary at the time of Fortran dominance; the curse was its world-wide, indiscriminate adoption.

In fact, we had implemented the (second) Pascal compiler in Zurich using a similar language as C (Scallop by Max Engeli). After completion of a workable part, the compiler was translated by hand (by R. Schild) to Pascal itself for boot-strapping, The auxiliary initial version was thereafter happily discarded for ever. C, however, gained world-wide attention.

Micro Computers (1975)

Microcomputers mentioned in the preceding paragraph were the visible result of the immense progress made in semiconductor technology, in particular of the efforts on miniaturization. Discrete transistors had been replaced by integrated circuits containing themselves many transistors, all produced in the same manufacturing step in a silicon foundry. Fairchild was leading the way, followed by Intel, National Semiconductor, Motorola, and others. First, there was the (short-lived) RTL technology (Resistor-Transistor Logic), soon to be super ceded by TTL technology (Transistor-Transistor Logic). Packages of ICs successfully became standardized founding the era of *TTL chips*, the series 74xxx. (Only the military had their own, more expensive cookies, the 54xxx series). Also, a single supply voltage of 5V belonged to the standard (only at the beginning augmented by -5V and 12V for memory chips). This was definitely one of the most successful standardization efforts in industrial history.

The building blocks of circuits were no longer transistors and resistors (and occasionally a capacitor), but elementary circuits, such as gates, multiplexers, decoders, adders, register arrays, and buffers. The heart of these micro computers was a single chip of a novel dimension of complexity, a complete, small computer, incorporating a simple arithmetic/logical unit, a set of data registers, an instruction register and a program counter, that is, a complete control unit.

The (not quite) first microprocessor chips featured a data path of only 8 bits. The prominent samples were the Intel 8080, the Motorola 6800, and the Rockwell 6502. Memory chips became available, first with 1K bits, then followed soon by 4K,

16K and even 64K (1980). As a result of this development it became relatively easy to design and build small computers for modest amounts of money. Upgrades of microprocessors followed soon, in particular the Motorola 6809 with a 16-bit internal ALU.

A special line of microcomputers appeared soon thereafter (1975). They were complete computer systems on a single chip, and they became known as *micro-controllers,* to be used mostly in *embedded systems*. They consisted of a simple ALU, a control unit, and a small amount of static memory (SRAM). They also contained on-chip (programmable) program memory. In early versions, this memory was writable only once (PROM), later versions contained erasable memory (EPROM). The most successful ones came from Intel (8048, 8051) and they were soon manufactured by the millions, driving down the cost to the order of a dollar and entering cars, refridgerators, and television sets.

Correctness concerns

The tremendous increase of computing power clearly resulted in rapidly growing demands on software which became larger and more intricate, but still being produced by old methods and teams. Gradually it became admitted that programming was a difficult intellectual activity, and that sometime had to be done to curtail difficulties, that a discipline was required.

The leading voice in this new insight was E.W. Dijkstra, well-known for his slashing of the Go to statement, leading to undisciplined program structures. His first contribution was to establish a style of structured programming, implying go-to-less programs. But he also tackled the and categorized problems arising with the new discipline of concurrent programming, where several programs run at the same time, communicating via shared variables. He showed that these problems could not be handled using assignments present in established languages. He presented his famous scenario of two (or more) sequential processes, each with a *critical section*, and with the rule that never more than one process was to execute its critical section. At first sight an obvious solution is the following, using two state variables:

```
VAR q0, q1: BOOLEAN;  (*qi means: Process Pi is in its critiacl section*)

PROCESS P0;
REPEAT ...
    IF ~q1 THEN q0 := TRUE; CS; q0 := FALSE END ;  ...
END P0

PROCESS P1;
REPEAT ...
    IF ~q0 THEN q1 := TRUE; CS; q1 := FALSE END ;  ...
END P1
```

A long controversy ensued, alternating with new proposals how to solve the problem and proves that the proposals were wrong. It became clear that a new instruction was needed that offered a test and an assignment in an atomic fashion. i.e. without letting another process interfere. Dijkstra postulated his *semaphores* with primitives

P(s) wait until s > 0, then decrement s
V(s) increment s

After it became fully recognized that programming was a difficult activity, it also became clear that programming languages, being the basic, mathematical formalism, had to be specified with rigor and accuracy. Algol had been the first language defined with a rigorous syntax specification. But its semantics were less clear, and sometimes even diffuse, certainly not appropriate for the application of formal, logical proofs.

The first step towards formally defined semantics was made by R. Floyd at Stanford in 1968. He created the notion of *assertion*. These are predicates involving the program's variables, conditions that were to hold whenever program execution reached the place where the assertion stood. Every statement S was to be preceded by an assertion P (precondition) and followed by an assertion Q (post condition) satisfied after S was executed. It was thought that Q could then be computed from P and S by a program analyzer or theorem prover. The semantics (meaning) of a statement S was defined by the triple {P} S {Q}. Assignment can now be formally specified by

P(y) x := y {P(x)

Soon afterwards, C.A.R. Hoare published his seminal paper entitled "Axiomatic definition of programming languages". Here the key idea was that the semantics of composite statements could be derived from the semantics of their components. As an example, the semantics of the while statement for repetitions were specified as follows:

Given {P} **S** {P}, where P is a predicate, then for the while statement

{P} WHILE b DO **S** END {P & ~b)

holds. P is called its *loop invariant* . The second part of a proof about repetitive statements must show that progress is guaranteed, that is, that the repetition will ultimately terminate by invalidating *b*.

E.W. Dijkstra then refined the theory and introduced the notion of *predicate transformer*. Given a statement S and a predicate P, S would transform P into Q, i.e. Q = F(P, S), where F is a Boolean function. It turned out that it was more useful to define the transformer as P = F'(S, Q), i.e. to work backwards from the result to the precondition when proving program correctness: Postulating a desired result, and then working backwards to find out which states would be acceptable as starting points of the computation.

Dijkstra was a leading member of the WG 2.1. and an ardent advocate of a scientific, mathematical, rigorous approach to programming. He was a strong critic of empirical approaches, of designing by trial and error, and an emphatic advocate of programming as mathematical engineering. Well-known remains his skepticism about program testing. He noted that testing can show the presence of errors, but never prove their absence. Another harsh dictum of his was "Do not give industry what it wants, but what it needs". He also chided that many people mix up conventional with convenient.

Although it was soon realized that correctness proving was an arduous task, the influence of this development was strong on the discipline of programming. It was an incentive to keep languages simple and to proceed in programming always with correctness concerns in mind. But all his statements did not win Dijkstra admirerers only.

The problems with correctness proofs are three-fold. First, the proofs become equally long, tedious, and perhaps error-prone as the programs themselves. Second, specifications of a program's pre-condition and post-condition are usually very complex, almost as complex as the programs themselves, or even worse. Third, most programmers lack the facilities to reason about formal logic to a degree required here. Nevertheless, this work on correctness proving exerted a distinct influence on how programs are now developed: With their proof in mind.

Meanwhile, programming, or rather *software engineering* as it became called, had become a business of its own, independent of hardware on which programs rested. The term unbundling, created in legal disputes in the 1970s, had become the way of life. Companies like Microsoft were the visible result.

The beginning of the computing age (1975)

Micro computers quickly spread into homes and schools and made computing a "household activity". Still, they had to be considered as toys. They were insufficiently powerful for the task of serious work. The first truly useful personal computer I encountered during a sabbatical year at the Palo Alto Research Center (PARC) of Xerox, was the *Alto* (B. Lampson, Ch. Thacker, E. McCreight).

The Alto did not use one of these 8-bit microprocessors, but was built with discrete components of the 74Sxx series of chips. The ALU's heart were 4 ALU chips 74S181 with a propagation delay of less than 100 ns. The data paths were 16 bits wide, and the memory had a capacity of 64K 16-bit words. The Alto was equipped with a 2-MByte cartridge disk store. It featured a bit-mapped display, 808 bits high and 606 dots (bits) wide, and a novel pointing device called mouse with three buttons. This latter combination allowed to directly program every dot on the entire display field. This was in stark and visible contrast to the conventional displays with fixed character sets, 80 characters per line and 25 lines on the page. Texts could be displayed with individually designed character patterns, the basis for the use of many fonts in regular, italic, and bold styles. Arbitrary graphics could be displayed mixing texts, line graphics, and small pictures. These features opened an entirely new world for programmers and computers. The fundamental innovation was *interactivity*.

Equally important was the fact that the memory size was sufficiently large to accommodate compilers for high-level languages for system programming. At Xerox, the language *Mesa* covered these needs. It represented a big extension of Pascal with several features deemed necessary to express particular facilities of the Alto hardware. Mesa definitely belonged to the second, if not third generation of programming languages. Like the Alto and its mouse, it was Xerox-proprietary.

The implementation of Mesa rested on Mesa byte code. The compiler generated byte code (similar to Pascal's earlier P-code) and the Alto was *micro-programmed* to interpret its byte code. This interpreter resided permanently in a special, very fast microcode memory. This scheme allowed for a rather dense program code, and only through this design large programs could be fitted into the still rather limited main memory.

Another hallmark of the Alto was that all workstations were connected through a network, the 3-MHz Ethernet, a single wire bus. The concept of *servers* emerged. There was a server, a dedicated Alto, for a first laser printer, and one for a large, common file store.

What so far had been possible only on large-scale mainframes, was now feasible on a personal, that is, *not shared* computer. The new scheme was such a radical departure from the common computing environment that it is fair to call it *the beginning of the (modern) computing era.* After all, every one of the millions of computer users now use personal computers (laptops) whose ancestor is the Alto. Hardly anybody remembers or imagines the way computers were used before 1975, namely through card decks or slow lines connected to terminals. The difference between then and now is enormous.

Lilith (1978)

Recognizing this difference, I simply could not imagine how to return from my sabbatical to the conventional computing environment, in which one communicated with a main frame computer over a thin wire with 300 b/s sitting in front of a dumb terminal. But how could I avoid this dire fate? As a workstation of this caliber was unique, and because it could not be bought, the only way out was to build one. But this was a difficult decision to make. I had to build up a team and a workshop with electronic equipment. I was lucky in succeeding in both endeavors, and within only two years in 1978 a prototype of a workstation was in operation. We called it *Lilith*, according to Adam's first wife, who, after having been kicked out of paradise, seduced men at night, just as our Lilith seduced researchers to stay at work in the evenings and over weekends. It was a truly exciting project with concurrent development of hardware and software by a team of only 4 - 7 people.

The hardware of Lilith was a 16-bit computer built (mostly) of Shottky TTL parts. Lilith consisted of about 8 boards in a 19" rack. The heart was the processor (board) with 4 bit-slice chips Am2901. They constituted the ALU with 16 registers and the usual arithmetic and logical operations. We augmented them with an external stack memory and a barrel shifter. Lilith was to be a computer with a short stack for the evaluation of expressions. This fast memory was built of 4 74S189 chips, 16x4 RAMs. The barrel shifter was an essential feature, necessary to perform fast bitmap operations, placing data at any bit position rather than at word boundaries within only one clock tick. The shifter consisted of 8 Am 25S10 chips. The main memory was built with 64 16K-bit DRAM chips.

Like the Alto, Lilith was micro-coded. That is, the externally visible computer was represented by a single program interpreting M-code, code generated by the

Modula compiler. This microcode consisted of 40-bit micro-instructions residing in the microcode memory consisting of five 2K x 8 EEPROMs (2716). In every clock cycle (7 MHz, 140 us), one micro-instruction is decoded and executed by the Am2901 processor chips. The control unit, generating the address of the next instruction consisted of 3 Am2911 sequencer chips, accommodating addresses of 12 bits. Microcode was the most convenient solution to implement a relatively complex instruction set with a modest amount of hardware. The instruction set of Lilith was indeed quite complex. This was due to the need of M-code being dense, again in order to store complex programs in a relatively small main memory (64K words). One contribution to the density of M-code came from Lilith's expression stack organization, but the other, more significant contribution came from load and store instructions with different address lengths or operand (constant) lengths. They came in versions with addresses of lengths 4, 8, and 16. Extensive program analysis had shown that about 80% of instructions had addresses less than 16 (4 bits), because they referenced local variables. Analysis also showed that our M-code was much shorter than code for then popular microprocessors, shorter by a factor of 1.5 against the NS32000, of 2 for the Motorola 68000, and of 2.5 for the Intel 8086. After all, this was significant, because at the time memory was (still) a scarce resource.

A particular advantage of micro-processing was that instructions could be of very different complexity. The shortest and fastest ones, like load and store, took only two or three micro-cycles (i.e. less than half a microsecond), whereas others, such as those for display operations for drawing lines and characters, were themselves short programs including loops. As a result, Lilith showed a fantastic performance in displaying lines and characters on the bit-mapped display.

However, this had only been achieved with some trickery. With 1 bit/pixel (no color!) a display page consisted of 808x606 = 489648 pixels = 30603 words. With a refresh frequency of 50 Hz, this left an access time of (less than) 0.65 us per word, which would have (more than) monopolized the memory by the display. The trick consisted of providing aside of the regular 16-bit read port, a second read port of 64 bit width for the display processor. With this solution, the display would block the memory for only about 25% of time.

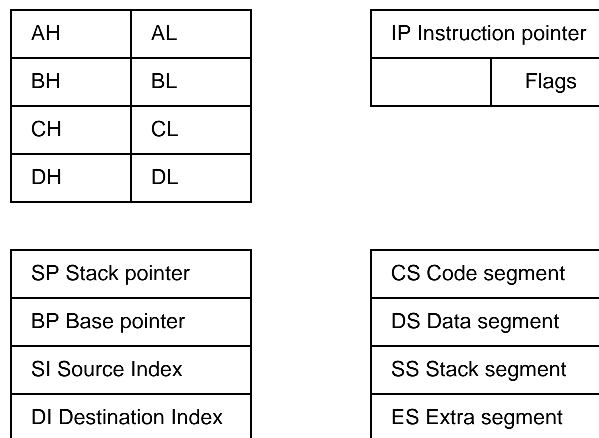
Second generation of microprocessors

In hindsight it is staggering what extra efforts and complications had to be devised and endured, because the current technology was not quite sufficient. Also the engineers at Xerox underwent the same fate, but knowingly and willingly. They knew that in a few years much more powerful technology would be available. If they succeeded in building an advanced personal computer (although at high cost) that would be much cheaper in the foreseeable future, they could develop advanced system technology and appropriate software at the present time, gaining a distinct advantage over competitors later on.

This "philosophy" stood behind the computer Dorado, built with ECL technology in 1984/5. Emitter-coupled logic was very fast, but consumed a very large amount of power, requiring extensive cooling. But circumstances changed, and the famous

Dorado never delivered what the designers had hoped. A consequence was the departure from the concept of your own, personal computer on or under the table, pioneered at PARC. All Dorados had to be stuffed into an air-conditioned room and were connected with individual cables throughout the building - not a bus like Ethernet - to the users.

However, while Alto, Dorado, and others were designed at Xerox, the rest of the world also moved forward. The companies which had pioneered the 8-bit micro-processors had now moved to 16-bit, and soon even to 32-bit processors *on a single chip*. They did not follow the path to faster, higher-current semiconductor technologies, but rather to the low-power CMOS FET technologiy. Thus emerged in 1978 the 8086 processor chip by Intel (whose successors dominate the market until today), and the 68000 by Motorola.



In particular the Intel part suffered severely under the scarcity of on-chip resources, resulting in a large variety of special-purpose registers and addressing tricks. The latter was due to the restriction of 16-bit word and the demand for 20-bit addresses. This gave rise to the horrid scheme of memory segments and segmented addressing, a pain for compiler designers and a source of many inefficiencies. The later 68000 was a much cleaner design, but the Intel part had already gained the market. The 68000 became the core of the famous Apple Macintosh, the toy with the small display that looked like a toaster.

Modula-2 (1979)

Towards 1975 a distinct need for an update of Pascal lay in the air. I felt so in particular upon encountering Mesa. Hence, just as Mesa was the language for the Alto, I designed Modula-2 for the Lilith. The main inspiration came from Mesa. But Mesa was already far too complex. I felt that Modula should be closer to the spirit of Pascal, and in particular should be suitable for teaching and use in courses on system design.

For this purpose, it had to offer facilities for *low-level programming*, i.e. for dealing with hardware entities and machine-dependent features. This notion was greatly

facilitated by the new concept of modules, already present in Mesa. *Modules* allow to hide local variables and procedures, prohibiting access from other modules, i.e. to *encapsulate* them. Systems would now consist of a hierarchy of modules, client modules at a higher level importing service modules at lower levels. It was no longer tolerable that programs would be described in one piece of monolithic text. Development by teams of programmers had become mandatory, and languages had to cater for this. The heading of modules constituted an interface specification, revealing which objects would be accessible form client modules. Of crucial importance was that these objects would be checked for type consistency, just as it was done for local entities.

To implement these new requirements proved to be a significant challenge. A simple solution was to "include" the source of a module B by the source of module A, if A was to be compiled. But this was not only considered inefficient, but also insecure, because B could have undergone alterations since A had been designed. Nevertheless, commercial systems adopted this deficient solution. Our solution was to let the compiler not only generate a code file, but also a symbol file, containing the compiled specifications of the exported entities.

However, Modula had grown into a language of considerable size and complexity. This appeared to be the inevitable alley that languages and systems would have to pass in order to meet the growing number of demands and expectations. The world was doomed to become more complicated.

Another language emerging from C underwent the same fate: C++. Its ominous name already hints at an over-sized monster. It passed through various versions as time went on, each time including new feature, and each time becoming bigger and heavier and more difficult to master. However, like C it gained wide-spread acceptance and became the favorite for industrial software managers. Instead of being leaders, educational institutions followed blindly. Along with this went the emergence of huge software "libraries", mutating programming into the art of selecting (more or less) appropriate packages to be fitted together into a (more or less) coherent whole. The excuse for big libraries was that programmers must no longer stand on the others' feet, but on their shoulders.

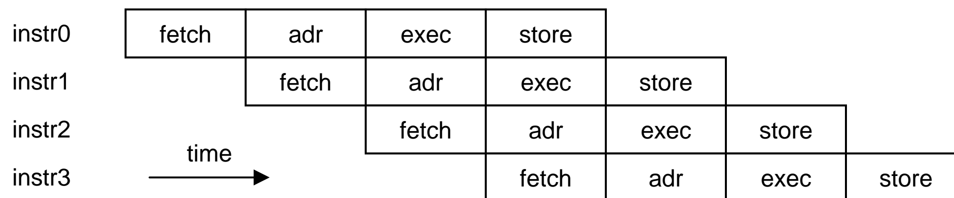
Reduced Instruction Set Computers (RISC) (1985)

Also on the hardware front, the world became more complex. Semiconductor technology had made vast progress. It had become possible to place several millions of transistors on one chip. Reduced size led to less signal propagation time, and to higher speed. Designers were seduced to make use of the armies of available transistors and include features that previously had been deferred to software. A direct consequence were more on-chip registers, virtual addressing, memory management units, and floating-point units. Apart from this, also memories became bigger and faster, up to 1 MB per chip around 1985. Another way to use up transistors was to offer a complex instruction set including string instructions and those for decimal and for floating-point arithmetic. And yet another was to offer a selection of addressing modes. The most advanced in this trend was the 32000 series of National Semiconductor. Not only did it offer a set of address

length (similar to Lilith), but a large set of modes, including one for linking modules. Besides, it was the first processor with 32-bit data paths. But compared to conventional design with discrete components, computers based on these modern chips (also Intel 80286 and Motorola 68000), were considered slow. The question to which technology the future would belong was an open one.

The decision was made by the advent of *Mosfe*t: Metal oxide semiconductor field effect transistor. The gate is made conductive by electrons. In bipolar junction transistors they are inserted by current injected into the gate layer between source and drain, in FETs by applying a voltage to a layer isolated from the source-drain channel, i.e. by a field effect. Today (2015) practically all transistors are FETs, and the bipolar era has come to its end.

On the side of architecture, a complete revolution was taken shape. The trend towards ever higher complexity was stopped. The new RISC excelled by simplicity. Typically a processor now consisted of a set of registers (32 bits), and the instruction set was minimized to simple logical and add/subtract instructions. Even multiplication and division were left out. The rationale behind this departure from conventions was that every instruction should require a single clock cycle and be very fast. Code density became of negligible importance, because memory was available in large amounts. If every instruction took a single cycle only, it became possible to execute several instructions concurrently, each one in 4 - 5 steps: Fetching, address computation (an addition), execution, and storing the result. This concept was known as *pipelining*, and it promotes speed substantially.



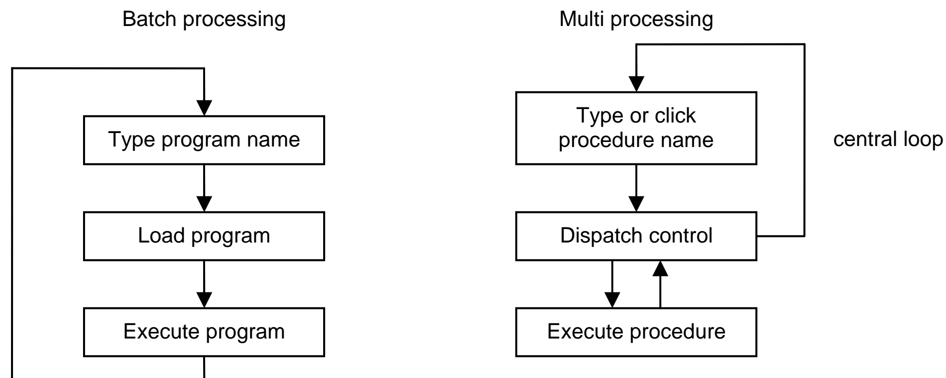
This scheme was proposed in academia (J. Hennessey at Stanford and D. Patterson at Berkeley) around 1985. Start-up companies were the result: MIPS and SPARC. In England similar developments evolved: Acorn at Cambridge, leading to the ARM architecture. The ARM was the proponent that survived. best However, as time went on, new versions appeared, each featuring new facilities such as miniaturization of transistors would allow. Surprisingly, it could retain its label even after its instruction set was practically replaced (by 16-bit "thumb" instructions).

A new paradigm of computing (1985)

When micro-computers emerged around 1975, they were still used in the old way: batch processing: Programs were activated one after another. A new paradigm was pioneered in the Xerox Palo Alto Reserach Laboratory. The new facility of high-resolution, bit-mapped displays called for a more flexible use of the screen. So-called viewers, later windows, became available, and it was evident, that an individual process should be attached to every viewers. The computer now allowed

the user to switch with every command from one viewer (process) to another. The availability of a pointing device (mouse) facilitated this new paradigm: A mouse click would replace the typing of an entire command.

Typing as such became less important. It was recognized that what was hitherto typed, appeared somewhere in a viewer already. Thus, pointing at this text made re-typing superfluous. One may claim that the viewer concept revolutionized computer usage. It was a direct consequence of the new paradigm of object-oriented programming: procedures (processes) can be attached to data structures This paradigm had already been introduced in the language Simula (1965), and later by Smalltalk (1976), and the viewer concept is its most famous application. It was perfected in Xerox' Cedar System and taken over in the Oberon system. Here the notion of textual unit of program (the module) became clearly separated from the unit of program execution (the procedure) contributing significantly to conceptual clarity.



This entire progress was ultimately possible only due to the enormous increase in memory size. When a procedure had been executed, it was not disposed from memory, as was done in batch processing, but it was kept in memory to be quickly accessible in a later call.

Oberon, the result of simplification

This brings us into the 1990 years. I felt that continued complexification had reached an alarming state. To curtail this cancerous growth had become more and more urgent, as systems had reached a size and weight under which they might soon collapse, as nobody would fully understand these monsters, but rather equate complex with powerful

Reduction of complexity was the guiding principle behind the design of the language *Oberon* . Quite obviously, Modula-2 was too complicated, and therefore laborious to implement. Also, it had not quite reached the goal of being truly computer-independent, a prerequisite for any language that claimed to be "higher-level". Oberon marked a significant step towards this difficult, but crucial and unique goal. The key to achieving it was the rigorous restriction to essential features, and the discarding of all "bells and whistles", a genuine exercise towards

simplicity. But despite frugality, Oberon was to be a powerful, general-purpose language in the tradition of Pascal and Modula. The result was a surprisingly small language (which, in 2007) was revised again to become even more frugal.

This guideline, however, was not just an esoteric idea. It was a necessity. It was decided to implement not only a compiler, but also an entire, self-contained operating system along the lines of *Cedar*, existing at the Xerox PARC facility in Palo Alto, a system that marked a radical departure from conventional, batch-processing systems.Oberon was to be catering for full interactivity with a high-resolution, bit-mapped display and a mouse. Compiler and operating system were implemented by two people only (this author and J. Gutknecht) in their spare-time over almost two years. Naturally, we were forced to concentrate on what was considered essential. The successful implementation of the entire system in its own language proved that the remaining features were sufficient, and that actually a simple language is more suitable for a complex system than one which is part of the problem rather than of the solution.

However, a single feature not present in Modula was added to Oberon: *Type extension*. The Algol - Modula line represents *static* typing. The type of a constant, variable, or function is visible from the program text alone, without executing the program. Type inconsistencies can therefore always be checked by the compiler. This rigid scheme was to be slightly relaxed. Through type extension it becomes possible to declare hierarchies of types, and to construct at run-time data structures with elements of different, although related types. This is the key to object-oriented programs; Oberon contains all the ingredients for object-oriented programming, but no more. Type checking at run-time could be realized very efficiently.

Object-orientation had become the one popular innovation in the realm of software. It had taken a long time since its origin was laid by the language *Simula* (Dahl & Nygaard) in 1967. It became better known (in the US) by the languages *Smalltalk* (Kay) in 1976 (implemented on the Alto), and *Object-Pascal* (Tesler, 1980). Smalltalk went all the way: everything was to be an object. You cannot add two numbers x and y. The proper way of looking at this problem is to consider x as an object which contains a method to add y to itself. Genius or perversion?

In 1995 Sun Microsystems presented its language *Java*, fully 6 years after Oberon. It incorporated much of the "philosophy" of Oberon, but, alas, chose the style and syntax of C. Around 2000 Microsoft released its language C# as a strong competitor of Java, and Google followed in 2007 with its language *Go*, even more strongly following (the 18 years old) Oberon. The crux with these languages, which all became wide-spread due to strong industrial support, is their size and complexity. The ambition to provide everything to everybody prevailed and let them grow into complex bodies difficult to master.

On the hardware front, the development was similar. The multitude of processor architectures of earlier decades has vanished. Only a few architectures prevail, mainly those of Intel and ARM. Engineers are pushed to make use of the abundance of available transistors. One way is to provide several, even many,

processors on the same chip, another to include large cache memories, and yet another to provide interfaces to external devices, such as to networks, or digital to analog and analog to digital converters. Also here complexity grows without bounds. It becomes harder and harder to recognize the original core and to identify the basic principles among the myriad of gadgets and gismos.

The unbelievable success of computers is mostly due to the incredible advances in semiconductor fabrication. Processors are now available with immense power, and memories with vast capacity. As in every other field of endeavor, abundance at low cost invariably leads to wasteful design. This entails not only waste, but poor products of declining quality. In particular software engineering now seems to be the eldorado of splashing and wastefulness.

Dijkstra once claimed that it is the foremost duty of the software engineer to fight (home-grown) complexity like the devil every minute. The same is now true also for the hardware engineer.