ETH

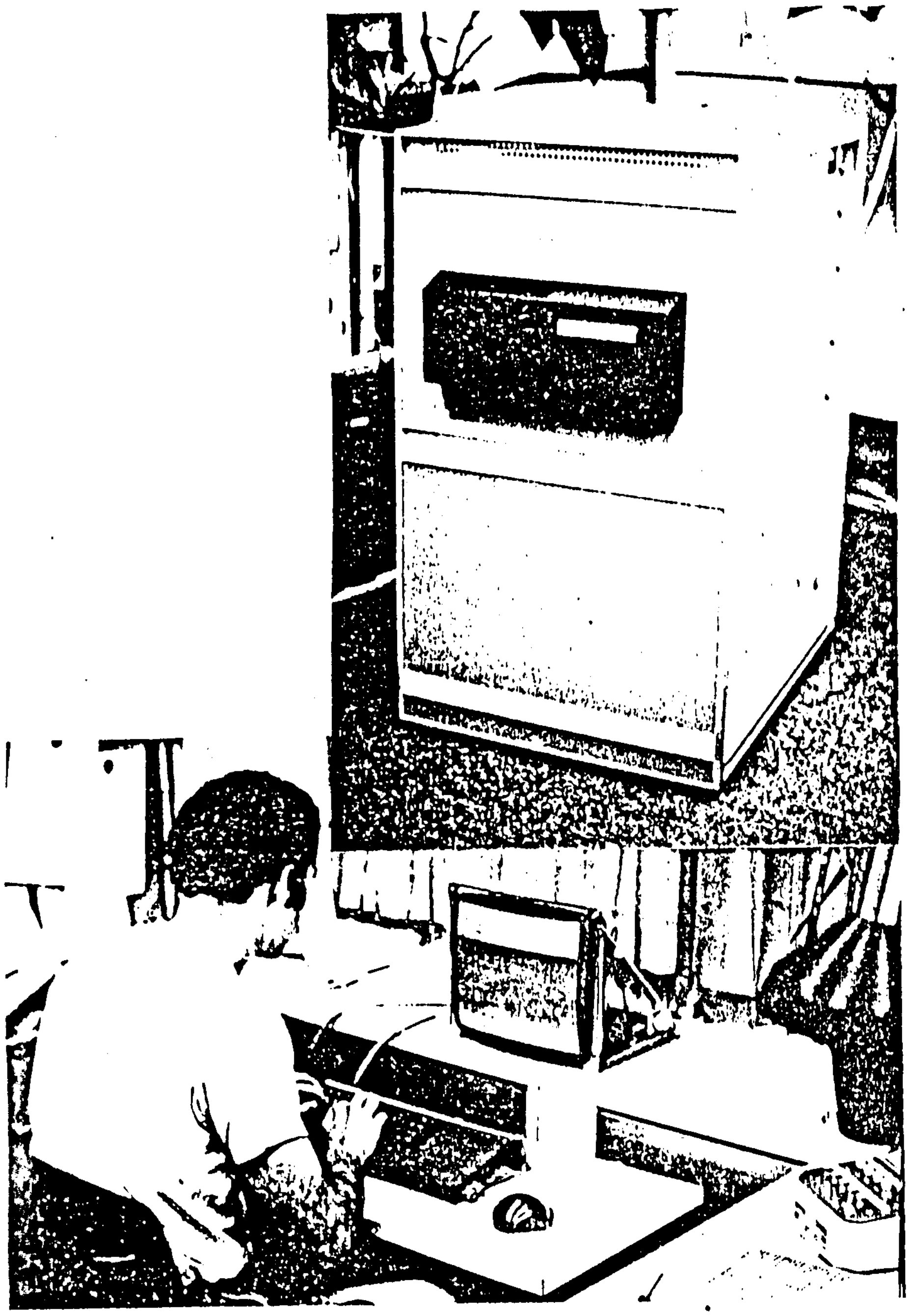
Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

N. Wirth

THE PERSONAL COMPUTER LILITH

April 1981 40



The Personal Computer Lilith

Abstract

The personal work station offers significant advantages over the large-scale, central computing facility accessed via a terminal. Among them are availability, reliability, simplicity of operation, and a high bandwidth to the user. Modern technology allows to build systems for high-level language programming with significant computing power for a reasonable price.

At the Institut fur Informatik of ETH we have designed and built such a personal computer tailored to the language Modula-2. This paper is a report on this project which encompasses language design, development of a compiler and a single-user operating system, design of an architecture suitable for compiling and yielding a high density of code, and the development and construction of the hardware. 20 Lilith computers are now in use at ETH.

A principal theme is that the requirements of software engineering influence the design of the language, and that its facilities are reflected by the architecture of the computer and the structure of the hardware. That the hardware should be designed according to the programming language, instead of vice-versa, is particularly relevant in view of the current trend towards VLSI technology.

Contents

Author's address:

N.Wirth
Institut fur Informatik
ETH
CH-8092 Zurich

1. Introduction

Software Engineering builds upon two pillars: methods and tools. Their interrelation is strong. In order to apply new methods effectively, we need the appropriate tools. In order to build supporting tools, we must master powerful methods. Much effort has been spent on improving our methods, particularly in programming, and many discussions and conferences have been devoted to the subject of furthering the state of the art by applying more effective methods. This includes a tendency towards the highly mathematical treatment of programming, the postulation of high-level languages for programming large systems, the method of structured programming, and the managerial aspects of organizing the work of programmers' teams.

All of these areas are important; they form the part of a whole. But perhaps none is as important as the adequate training of the individual team number into a habit of systematic thinking. No team can be successful without all its members being trained to regard programming as a highly logical and mathematical activity. And the success of a mathematical treatment rests largely on the use of an adequate notation, i.e. programming "language". The designer of algorithms might be content with the adequate notation and regard it as his only tool needed. However, our subject at large is not the design of algorithms or programs, but the design of machines. We must regard programming as designing machinery, for programs turn "raw hardware" into that machinery which fulfils the specified task.

Obviously a good notation must therefore be supported by an excellent implementation, just as a good mathematical framework for program design must be supported by an appropriate notation. Moving one step further, the notation's implementation must be supported by an appropriate computer system. The measure of its quality not only includes aspects of computing power, cost-effectiveness, and software support (the so-called programming environment), but also a simplicity and perspicuity of the entire system, a convenience to use the system, a high degree of availability, and - on the more technical side - a high bandwidth of information transfer between computer and programmer. The latter aspects point in the direction of a personal work station in contrast to the remote, time-shared, large-scale computing facility.

Fortunately, modern semiconductor technology has made it possible to implement a modern programming language with excellent support facilities on relatively small computers that are very inexpensive compared to conventional large computing installations. The fact that the mechanisms for sharing a computer - and in particular that for protecting the users from the mistakes of others - can be discarded, reduces a system's complexity drastically, and thereby improves both its reliability and perspicuity. The possibility to implement modern, high-level languages on relatively small non-shared computers was perhaps the most significant "discovery" during the last five years. A personal computer, programmable in such a language, constitutes, in my opinion, a necessary tool for the creative software engineer of the future.

2. Project history and overview

The decision to design and build a personal computer as motivated above was made in the fall of 1977, after the author had learned to appreciate the advantages of working with an Alto computer [1]. The project included the following principal parts [2]:

- design of the programming language Modula-2.
- implementation of a multipass compiler suitable for relatively small computers.
- development of a basic, single-user operating system, including a file system and a linking loader.
- design and implementation of a modern, flexible text editor taking full advantage of the computer's capabilities.
- implementation of a set of basic utility programs for file directory inspection, copying, renaming, deleting, and listing files.
- programming and implementing an appropriate set of library modules for file handling, access to peripheral devices - in particular the display - and storage management.
- designing a suitable machine architecture as ideal interface between compiler and hardware, and programming this architecture in microcode.
- design of the hardware capable of efficiently interpreting the microcode and supporting the desirable peripheral devices.
- building two prototypes of the designed hardware, and modifying them according to insight gained from the concurrent development of hard- and software.
- building a series of 20 computers, debugging and testing them.
- writing documentation and user manuals.

The language Modula-2 - the notation in which this system presents itself to the software engineer - was designed as a general system programming language [3]. The guiding principle was that this language would be the only language available on the computer. Especially, no assembler would be available, and hence, the language should be suitable for both high-level programming in a machine-independent manner and low-level programming of machine-particular aspects, such as device handling and storage allocation. In fact, the entire operating system, the compiler, the utility programs, and the library modules are programmed exclusively in Modula-2.

The compiler is subdivided into four parts. Each part processes the output of its predecessor in sequential fashion and is therefore called a pass. The first pass performs lexical and syntactic analysis, and it collects identifiers, allocating them in a table. The second pass processes declarations, generating the so-called symbol tables that are accessed in the third pass to perform the type consistency checking in expressions and statements. The fourth pass generates code. Its output is called M-code.

The operating system is conceived according to the concept of an "open" system [4]. It is divided into three principal parts, namely the linking loader, the file system, and routines for keyboard input and text output on the display. The file system maps abstract files (sequences of words or characters) onto disk pages and provides the necessary basic routines for creating, naming, writing, reading, positioning, and deleting files. Both, loader and file system present themselves to the Modula-2 programmer as modules (packages) whose routines can be imported into any program. Whenever a program terminates, the basic operating system activates the command interpreter which requests the file name of the next program to be loaded and initiated.

The computer as "seen by the compiler" is implemented as a microprogrammed interpreter of the M-code. The M-code is designed with the principal goals of obtaining a high density of code and of making the process of its generation relatively systematic and straight-forward. Although space is definitely the scarcer resource than time, a high density of code is desirable not only in the interest of saving memory space, but also for reducing the frequency of instruction fetches. A comparison between two different, but strongly related compilers revealed that M-code is shorter than code for the PDP-11 by a factor of almost 4. This surprising figure is clear evidence of the inappropriate structure of "conventional" computer instruction sets, including those of most modern microprocessors that were still designed with the human assembly language coder in mind.

The actual hardware consists of a central processing unit based on an Am2901 bit-slice unit, a multi-port memory with 128K words of 16 bits, a micro-code memory of 2K instructions implemented with PROMs, a controller each for the display, the disk, and a local network, and interfaces for the keyboard, a cursor tracking device called the mouse, and a V-24 (RS-232) serial line interface. The central processor operates at a basic clock cycle of 150 ns, the time required to interpret a micro-instruction. The most frequently occuring M-code instructions correspond to about 5 micro-instructions on the average.

The display is based on the raster scan technique using 594 lines of 768 dots each. Each of the 456'192 dots is represented in main memory by one bit. If the entire screen is fully used, its bitmap occupies 28'512 words, i.e. 22% of memory. The representation of each dot (picture element) in program accessible main memory makes the display equally suitable for text, technical diagrams, and graphics in general. In the case of text, each character is generated by copying the character's bitmap into the appropriate place of the entire screen's bitmap. This is done by software, supported by appropriate microcoded routines, corresponding to special M-code instructions. This solution, in contrast to hardware character generators, offers the possibility to vary the characters' size, thickness (boldface), inclination (italics) and even style. In short, different fonts can be displayed. This feature, which is particularly attractive for text processing, requires a substantial amount of computing power to be available in short bursts. The writing of a full screen, i.e. conversion of characters from ASCII code to correctly positioned bitmaps, takes about 1/4 second. Using a small font, a full screen may display up to 10'000 characters.

The disk used in this personal computer is a Honeywell-Bull D-120 cartridge disk with a capacity of 10 MBytes and a potential transfer rate of 720 kB/s, which results in an actual rate of 60 kB/s for reading or writing of sequential files. Disk sectors, each containing 256 Bytes, are allocated in multiples of 8 on the same track. Allocation is entirely dynamic, and hence no storage contraction processes are needed to retrieve "holes". The use of exchangeable cartridge disks in contrast to sealed (Winchester) disks has been considered as essential in order that a work station may be used by different people at different times without reliance on the existence of a network and a central file store.

The mouse is a device that transmits signals to the computer which represent the mouse's movements on the table. These movements are translated (again by software) into a cursor displayed on the screen. The accuracy of position is as high as the resolution of the screen, because the feedback from cursor to mouse travels via the user's eye and hand. The mouse also contains three pushbuttons (keys) which are convenient for giving commands while positioning the mouse.

The various principal parts of the projects were undertaken more or less concurrently. The team consisted of 8 (part time) people in the average (not counting the production of 20 machines), and was small enough to require neither management staff nor methods. The hardware was designed and built by three engineers (including the author), two computer scientists built the compiler, one the operating system, one implemented the microcode and most of the editor. The software effort was based on the use of a PDP-11/40 computer (with a 28K store) and was initiated with the development of a compiler for Modula-2 generating code for the PDP-11 itself. This "preliminary" compiler development constituted a significant part of the entire software effort, and resulted in a valuable software tool that had recently been released for distribution. It also made the development of the Lilith software quite independent from the progress of the hardware. Both the Modula-2 compiler for M-code, the operating system, and even the highly display-oriented editor were developed on the PDP-11, and the subsequent transport to the Lilith computer proved to be quite unproblematic due to programming in Modula-2. In fact, the untested compiler was transported and debugged (at least to an acceptable degree) in a few days only.

Whereas the software development could profit from our previous experience in designing compilers and programming in general, such was not the case in the hardware sector, as our institute had neither hardware expertise nor facilities. To gain experience and develop such facilities was, however, a prime challenge, and this project offered a welcome opportunity.

From the start it was planned to base the Lilith computer on the 2901 bit-slice processor, because one-chip processors available in 1977 did not offer the computing speed required for the efficient handling of the planned bitmap operations. This decision proved to be a good one. After 15 months of development, a first prototype was operational (without disk), proved to be too unreliable for extensive use, but confirmed the sensibility of the overall design. An additional year was needed to produce two identical prototypes which served to test the software that had been developed in the meantime. In the spring of 1980, a team was formed at the Department of Electrical Engineering of Brigham Young University in Provo, Utah, to build a series of 20 Lilith computers. This goal was achieved within 8 months by three graduating engineers and with the aid of student employees during the summer months. The cost per unit, not counting the development of the prototypes nor of organizing the production effort, but including labor and parts, in particular the 10MB disk, was about SFr 20'000.

In the meantime, a few important application programs were written at ETH, including a text editor, an editor for drawing circuit diagrams, and a window handler module. Some sample pictures illustrating their use are shown in Fig 1. They are printed with the same resolution as seen on the screen.

3. Modules and interfaces in Modula-2

Perhaps the most important criterion of a language for programming large systems is how well it supports program modularization. The earliest facilities introduced for effective program decomposition was the concept of locality, i.e. the restriction of the validity of names (identifiers) to welldelineated parts of the program, such as a block or a procedure. This concept was introduced by Algol 60 and adopted in Algol 68, PL/I, and Pascal, among others. The range of validity is called a name's scope. Scopes can be nested, and the rule is that names valid in the scope's environment are also valid inside it, whereas names declared within the scope are invisible outside. This rule immediately suggests a connection between the range of visibility (scope) of a name within the program text, and the time of existence of the object associated with the name: as soon as control enters the scope (procedure, block), the object must be created (e.g. storage must be allocated to a variable), and as soon as it leaves the scope, the object can be deleted, for it will no longer be visible. In spite of the tremendous value of this locality concept, there are two reasons why it is inadequate for large programs.

- there is a need to hide objects, i.e. to retain them while they are invisible. This calls for a separation of visibility and existence: visiblity as a property of names, existence as a property of objects.
- there is a need for closer control of visibility, i.e. for selection of particular names to be visible or invisible, in contrast to the "inheritance" of the total environment into a local scope.

In Modula-2, we have therefore added the structure of a module to the structure of the procedure. Both structures appear syntactically as almost identical, but are governed by different rules about visibility of local names and existence of the associated objects:

P1. An object declared local to a procedure exists only as long as the procedure remains activated.
M1. An object local to a module exists as long as the enclosing procedure remains activated.
P2. A name local to a procedure is invisible outside the text of that procedure, one visible in the environment is also visible inside the procedure.
M2. A name local to a module is visible inside the module, and outside too, if it appears in the so-called export list in the module heading. A name visible in a module's environment is visible inside that module only if it appears in its so-called import list.

From these rules, we can draw the following conclusion: A module itself has no "existence", since its local objects inherit their lifetime from the module's environment (procedure). Hence, the module is a purely syntactic structure acting like a wall enclosing its local objects and controlling their visibility by means of export and import lists. Modules therefore need not be instantiated; there are no instances of a module. The module is merely a textual unit.

A typical example of a module is the following:

```
MODULE m;
   IMPORT u,v;
   EXPORT p,q;

   VAR x: ...;
   PROCEDURE p(...);
      BEGIN ... x ... END p;
   PROCEDURE q(...);
      BEGIN ... x ... END q;
BEGIN ... u ... x ...
END m
```

This module owns three local objects: variable x and procedures p and q operating on x. It exports p and q and hides x by not exporting it. The body of the module serves to initialize x; it is activated when the environment of m is activated (created). This example is typical, because it shows how an object x can be hidden and how access from outside is restricted to occur via specific procedures. This makes it possible to guarantee the existence of invariant conditions on x, independent of possible errors in the environment accessing x via p and q. Such is the very purpose of modularization.

The typical purpose of a module is indeed to hide a set of interrelated objects, and the module is often identified by these objects, e.g. a table handler hiding the table, a scanner hiding the input stream, a terminal driver hiding the interface, or a disk system hiding the disk's structure and allocation strategy.

The module concept as described above had been introduced with the language Modula [5]. Modula-2 extends this concept in two important ways, namely by

- qualified export mode, and
- subdivision of a module into two textual parts, the so-called definition and implementation parts.

Qualified export serves to avoid clashes between identical identifiers exported from different modules into the same enclosing scope. If an identifier x is exported in qualified mode from a module m, then the object associated with x needs to be denoted as m.x. The qualified mode is therefore appropriate, if the writer of m does not know the environment of m. This is not the usual case for nested modules; individual members of a programming team more typically design modules that lie on the same level, namely the outermost, or global level (that may be considered as being enclosed in a universal and empty scope). It is this case that is particularly important in the design of large systems, where a better separation of the specification of import and export lists from the description of the actual objects is desirable.

Consequently, we divide a global module into two parts. The first is called a definition module; it contains the export list and specifications of the exported objects as far as relevant for the user (client) of this module to verify the adherence to language rules (in particular type consistency). A definition module also specifies the types of its exported variables and the parameter lists of its exported procedures. The second part is called the implementation module. It contains (usually) import lists and all the details that need not concern the client, such as the bodies of procedures. The notion of textually separate definition and implementation parts was pioneered by the language Mesa [6] and is here smoothly integrated with the module concept of Modula.

Example:
```
DEFINITION MODULE B;
   EXPORT QUALIFIED p,q;
   PROCEDURE p(...);
   PROCEDURE q(...);
END B.

IMPLEMENTATION MODULE B;
   FROM A IMPORT u,v;
   VAR x: ...;
   PROCEDURE p(...);
      BEGIN ... x ... u ... END p;
   PROCEDURE q(...);
      BEGIN ... v ... x ... END q;
BEGIN ... x ...
END B.
```

4. Coroutines and processes

With the design of the Lilith computer we did not follow the fashionable trend to design a system consisting of several co-operating concurrent processors, thereby avoiding one certain source of difficulties, namely their synchronization. The consequence for the language Modula-2 was that the concept of concurrent processes played a minor role only, whereas in Modula-1 it had been the major theme. The primary idea had been to distinguish the logical process from the physical processor, allowing implementations to choose their own mechanisms for allocating processors to processes. Logical processes are served by time-sharing the processors, which may well have different characteristics and capabilities. The processes are implemented as coroutines, and transfers of control between them are implied in statements that send signals or wait to receive signals, where the signal is an abstract notion represented as a data type. Each processor executes a sequence of coroutine segments, and the processor scheduling can well be hidden behind the primitive operations on signals. The principal difference between processes (as in Modula-1) and coroutines (as in Modula-2) is that the latter are explicitly identified whenever a transfer occurs, whereas processes are not, since transfers are implied by sending a named signal to some process which remains anonymous.

It is well in accordance with the premise of Modula-2 - namely to make primitives directly available to the programmer - to include coroutines instead of processes, because the latter are implemented by the former. As a consequence, Modula-2 implementations need no "run-time system" and no fixed, built-in scheduling algorithm. There exists no data type Signal, but instead transfer of control from a coroutine P to a coroutine Q is specified explicitly by the statement TRANSFER(P,Q). Here P and Q are variables of the primitive type PROCESS, whose actual values are pointers to the coroutines' workspace and state descriptors.

Furthermore, experience with Modula-1 showed the advisability of separating interrupt-driven from "regular" processes, because an interrupt signals a transfer of service among processors within the same process. A programmer may adopt this advice by supplying his own scheduling program. Modula-2 provides the appropriate mechanism for encapsulating such a user-defined scheduler in the form of its module structure. Naturally, such algorithms may also be provided in the form of library modules.

As an example we list a scheduler reflecting the simple round-robin algorithm. The module exports the data type Signal and the operators StartProcess, Send, and Wait, which correspond to the language facilities of Modula-1. The example excludes, however, the treatment of interrupt-driven processes. (Note that the type Signal is exported in opaque mode such that its structures is invisible to the importer.) Both Send and Wait imply a coroutine transfer. The primitive operation TRANSFER is, like the data type PROCESS, imported from the module SYSTEM, which typically contains low-level facilities. High-level programs should preferrably rely on the process concept as presented by such a ProcessScheduler module, rather than on named coroutines and explicit transfer of control.

```
DEFINITION MODULE ProcessSceduler;
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT QUALIFIED Signal, StartProcess, Send, Wait;

  TYPE Signal;
  PROCEDURE StartProcess(P: PROC; A: ADDRESS; n: CARDINAL);
  PROCEDURE Send(VAR s: Signal);
  PROCEDURE Wait(VAR s: Signal);
END ProcessScheduler.
```

```
IMPLEMENTATION MODULE ProcessScheduler;
  FROM SYSTEM IMPORT PROCESS, ADDRESS, NEWPROCESS, TRANSFER;

  TYPE Signal = POINTER TO ProcessDescriptor;
    ProcessDescriptor =
      RECORD ready: BOOLEAN;
        pr:    PROCESS;
        next:  Signal; (* ring *)
        queue: Signal; (* waiting queue *)
      END ;

  VAR cp: Signal; (* current process *)

  PROCEDURE StartProcess(P: PROC; A: ADDRESS; n: CARDINAL);
    (* start P with workspace A of length n  *)
    VAR t: Signal;
  BEGIN t := cp; NEW(cp);
    WITH cp^ DO
      next := t^.next; ready := TRUE;
      queue := NIL; t^.next := cp
    END ;
    NEWPROCESS(P, A, n, cp^.pr); TRANSFER(t^.pr, cp^.pr)
  END StartProcess;

  PROCEDURE Send(VAR s: Signal);
    (* resume first process waiting for s *)
    VAR t: Signal;
  BEGIN
    IF s # NIL THEN
      t := cp; cp := s;
      WITH cp^ DO
        s := queue; ready := TRUE; queue := NIL
      END ;
      TRANSFER(t^.pr, cp^.pr)
    END
  END Send;

  PROCEDURE Wait(VAR s: Signal);
    VAR t0, t1: Signal;
  BEGIN (* insert current process in queue s *)
    IF s = NIL THEN s := cp
    ELSE t0 := s;
      LOOP t1 := t0^.queue;
        IF t1 = NIL THEN
          t0^.queue := cp; EXIT
        END ;
        t0 := t1
      END
    END ;
    cp^.ready := FALSE; cp^.queue := NIL;
    t0 := cp; (*now find next ready process*)
    REPEAT cp := cp^.next;
      IF cp = t0 THEN HALT (*deadlock*) END
    UNTIL cp^.ready;
    TRANSFER(t0^.pr, cp^.pr)
  END Wait;

BEGIN NEW(cp);
  WITH cp^ DO
    next := cp; ready := TRUE; queue := NIL
  END
END ProcessScheduler.
```

Interrupts are transfers of control that occur at unpredictable moments. We can regard an interrupt as equivalent to a statement

```
        TRANSFER(interrupted, interrupting)
```

that is effectively inserted in the program wherever control happens to be at the moment when the external interrupt request is applied. The variable "interrupting" denotes the process that is destined to service the request, whereas the variable "interrupted" will be assigned the interrupted coroutine. The typical interrupt handler is a device driver coroutine of the following pattern; P and Q are variables of the primitive type PROCESS.

```
     PROCEDURE driver;
     BEGIN initialization;
        LOOP ...
           start device; TRANSFER(Q,P); ...
        END
     END driver
```

The driver process is created by the primitive statement

```
     NEWPROCESS(driver,wsp,n,Q)
```

which allocates the procedure "driver" and the workspace wsp of size n to this coroutine, now identified by Q. It is subsequently activated by the statement

```
     TRANSFER(P,Q)
```

which assigns the starting coroutine (e.g. the main program) to P. After initiation of a device operation the statement TRANSFER(Q,P), which symbolically stands for that part of the process which is executed by the device (i.e. another processor) actually returns control to P and assigns (the current state of) the driver coroutine back to Q. Termination of the device operation causes an interrupt signal which (if enabled) corresponds, as explained above, to an unwritten TRANSFER(P,Q). This signal again switches control back from the interupted to the driver (interrupting) routine.

Each interrupt signal - the Lilith computer offers 8 of them - is associated with its own variables P and Q at fixed locations. In order that further interrupts remain disabled while the processor executes the interrupt routine, drivers are typically declared inside a module with specified "priority" that causes interrupt inhibition up to that specified "priority" level.

This elegant conceptual unification of coroutine transfers and interrupt handling was made possible by an appropriately designed computer architecture and instruction set.

5. The operating system

The most noticeable aspect of the Lilith operating system Medos is its orientation towards a single user. It is devoid of any protection mechanism against malicious programs that could hamper another user's program. Since Medos is programmed in Modula, it benefits from the safety provided by Modula's type consistency and various run-time checks. Its safety features are "defensive", but certainly not invulnerable, considering the Modula's facilities for low-level programming offered to the brave programmer. In this regard, Medos follows the strategy of Pilot [9]. In a first, superficial look it can be regarded as a collection of modules that are imported by the current program (and its imported modules). Since a number of low-level modules (such as the file system) are used by virtually every program, they form a resident section. This set of modules consists of three main parts:

"Program"	- storage allocation, program loader
"Terminal"	- drivers for keyboard and display
"FileSystem"	- disk driver and file administration

The module Program exports the procedures

```
Call(name,sharedHeap,status)
AllocateHeap(size)
DeallocateHeap(size)
```

of which the first effectively represents the loader. The module administers the entire store as a stack and loads called programs sequentially. The remainder of the store is treated as data store. In this part, the data stack grows from one end and the heap from the other. The heap is used for variables that are allocated dynamically by calls of AllocateHeap and DeallocateHeap, which merely move the pointer that denotes the separation between data stack and heap. More sophisticated allocators can be programmed which, however, will also refer to these basic procedures.

If a program P calls the loader, the code and data segments of the loaded module Q (and of the modules imported by Q and not already present) are stacked on top of those of P. The set of segments thus loaded forms a new "level", one higher than that of P. The loader operates as a coroutine, and each new level of program is represented as a coroutine too. This slight misuse of the coroutine facility is justified by the convenience in which new sections of data and program (a level) can be administered, if described as a coroutine. Fig. 2 shows the storage layout and the implied transfers of control when a program is loaded from a caller at level 1.

The set of resident modules forms level Ø. Its main program is called the Sequential Executive Kernel. It invokes the loader which loads the command interpreter. This is merely a program that outputs a prompt character, reads a file name, and transmits the file identity to the kernel, which loads this file after the command interpreter has terminated and control is returned to level Ø. Loading of the main program usually requires the loading of further modules that are specified in import lists. Linking or binding of modules is simplified by the architecture of the Lilith computer such that it is performed directly upon program loading. Fig. 3 shows a typical sequence of programs, and how they occupy the store.

Since a program is loaded only after removal of the command interpreter, and because the command interpreter typically has ample time to process the slow input from the keyboard, it can be designed with additional sophistication. It can search the file table for program files whose names match with the input so far received and extend it as far as it is unambiguous. For example, if file names ABCD and ABCE are present, and no others starting with A, it may display both names after receiving "A?" and then allow continuation after receiving either D or E. This is a small but typical example of providing a convenient user interface without additional burden on the user's program.

The entire mechanism for loading and allocating is programmed exclusively in Modula-2; this includes the subtle point of changing our view of a program as data before to code after its loading. In Modula-2, this is possible without resorting to tricky programming and without the escape to small sections of assembly code.

The second principal part of the set of resident modules handles input from the keyboard and output to the display. This module is called Terminal. The input stream fetched by the procedure Read (contained in Terminal) flows through a switch that allows reading from a file instead of the keyboard. Because the command interpreter also calls Read, that file can even be a command file. The output stream, which is fed by calling the procedure Write, is fed to the low-level module TextScreen that simulates sequential writing and generates the bit pattern for each character according to a default font.

The module FileSystem constitutes the third major part of the resident system. Files are used for three main purposes:

- long-term storage of data on permanent, named files,
- communication among programs,
- secondary storage of data on temporary, unnamed files.

We distinguish between the naming and abstract definition of files as extendable arrays of elements (FileSystem) and the physical implementation of files on the disk (DiskSystem). The programmer refers to files through the module FileSystem which in turn calls procedures of the module DiskSystem hiding the details of their physical representation.

FileSystem exports the type File and operations on this type for opening (creating), naming, reading, writing, modifying, positioning, and closing files. Normally files are regarded as streams of either words or characters; writing occurs at the end of the stream only, and if writing is requested at some position other than the end, the file's tail is lost and deallocated. Although it is also possible to modify files, i.e. overwrite them, the abstraction of the stream is the preferred view of files.

The module DiskSystem implements Files on the Honeywell-Bull D-120 disk. It is designed according to the following main requirements:

- fast access, in particular if strictly sequential,
- robustness against hard- and software failures,
- accommodation of a large number of (mostly short) files,
- economical use of storage space.

The following scheme was chosen as a compromise between the various design objectives: Space is allocated in blocks of 2048 bytes. This results in a memory resident allocation table of 392 words (one per cylinder), each bit indicating whether or not its corresponding block is allocated to some file. Each block corresponds to 8 disk sectors, equally spaced on the same cylinder. A separate file, allocated at a fixed place, is called FileDirectory and consists of file descriptors. Every file is identified by the index of its (first) descriptor (= file number), and each descriptor contains a table of addresses of the blocks which constitute the file. Additionally, the descriptor specifies various properties of the file, such as its length, creation date, last modification date, whether it is permanent, protected, etc. Upon startup, the system reads the entire FileDirectory and computes the allocation table.

Unnamed files are released either by closing them or when the system is started. They are used as temporary files during execution of a program. For long term storage of data, a file has to be named. To administer permanent files, the module DiskSystem maintains another file (also placed in a fixed location) called the Name Directory. Each entry consists of a file name and the number of the associated file. The procedure Lookup(f,name,create) is used to search the name in the Name Directory and connects the file (if found) with the file variable f. The parameter "create" allows to ask for the creation and naming of a new file, if the specified name was not found (see Fig. 4).

A fourth, but effectivly hidden part of the resident system is called the Monitor. It contains two auxiliary processes that are used to monitor the third, namely the main process of the user. The auxiliary processes are called Clock and Trap (Fig. 5). Clock is invoked 50 times per second. It updates a variable called time, monitors the keyboard by polling, and buffers keyboard input, allowing for typing ahead.

Trap is invoked by various instructions detecting abnormal conditions, such as stack overflow, arithmetic overflow, index out of range, access to picture elements outside the specified bitmap, the standard procedure HALT, etc. The Trap process then may store the state of the main process (essentially a dump) on the disk for possible later inspection by a debugger program, and restarts the main process at the kernel level.

Typing the control character <ctrl>C is detected by Clock and causes an abortion of the main process in the same manner as a trap. Evidently, abnormal situations are here handled by coroutine transfers instead of an additional exception facility provided in the programming language. The auxiliary coroutine then regards the aborted coroutine as data (instead of as a program) and is thereby able to reset it to a state where continuation is sensible.

Fig. 6 shows the principal modules of Medos with arrows denoting calls of procedures. Usually, these arrows are identical to those denoting the import/export dependences among modules. Exceptions to this rule occur through the use of procedure variables.

6. Separate compilation of modules

For reasons of convenience and economy, large system programs need to be compiled in parts. It is only natural that these parts be the ones that from a logical point of view were designed as relatively independent units. The module is the obvious choice for the unit of compilation. Definition and implementation modules are therefore called compilation units.

The idea of compilation in parts is as old as Fortran and even assembler code. In high-level languages with data types the problem of partial compilation is of considerable complexity: we wish that type consistence checking is fully maintained across module boundaries. In fact, experience has shown that this is when it is most needed to avoid catastrophic errors. In order to recognize inconsistencies such as type mismatches, incorrect number or order of parameters, etc., as early as possible, they must be detectable by the compiler. The compiler therefore must have access to information about all imported objects. This is accomplished as follows [7]:

Assume that a module B depends on, i.e. imports objects from a module A. Therefore, module A has to be compiled first. During its compilation, th compiler generates, apart from a code file for A, a symbol file. Compilation of B subsequently accesses that symbol file. More accurately, program B can - according to the rules of the language - refer to information of A's definition part only. Thus, the symbol file is an extract only of the information available during compilation of A. Since only definition modules are capable of exporting, the symbol file is the result of compiling the definition module A, while code is the result of compiling implementation (or program) modules only.

This scheme - in particular the separation of definition and implementation parts - has important consequences for the manner in which systems are developed. A definition module constitutes the interface between its implementation part and its clients. Effectively the scheme forces the programmer to define interfaces first, for, whenever a definition module is (changed and) recompiled, all its importers (clients) have to be recompiled too. However, it is possible to change and recompile implementation modules without that far-reaching and costly consequence.

It should be noted that the consequences of this chronological ordering of compilations are less severe than might be anticipated due to the fact that the importers are usually implementation modules. Hence a change in a low-level module - a module that resides low in the hierarchical chain of dependencies - need *not* produce a chain reaction of recompilation up to all ultimate clients. The appropriate decomposition of a planned system into modules is nevertheless a most important aspect of competent programming. Often the decomposition has to be decided at an early stage when insight into many aspects of a system are still hazy. Its success therefore largely depends on the engineer's previous experience with similar tasks.

Learning how to deal effectively with a new facility offered by a programming language is a long-term process. The module facility forces the programmer team to make those decisions first that must be made first [8]. One lesson learned so far is that a module typically centers around a data structure, and that it is this data structure rather than the exported operations that characterize it.

7. The architecture of the Lilith computer

One of the challenges in designing a computer lies in finding a structure and an instruction set which yield a high density of code and a relatively simple algorithm for code generation. A premise of this project was that the computer had to be designed according to the language in which it was to be programmed. This resulted in a quite unconventional architecture. No attempt was made to make the instruction set suitable for "hand coding"; in fact, programming in machine code would be quite cumbersone, even if an assembler were available.

The Lilith computer is based on a stack architecture. Stack computers are by no means novel as such. Their history dates back to the early 60s with the English Electric KDF9 and the Burroughs B5000 as pioneers. The Lilith architecture adopts the stack principle without compromise, and its instruction set is chosen to obtain a high density of code requiring only straight-forward algorithms for instruction selection. The code is a byte stream. Each instruction consists of one or several bytes. The high density is achieved not only by implicit addressing of intermediate results in expressions, but mainly by the provision of different address lengths and suitable addressing modes. In order to explain these modes, we need to inspect the overall storage organization at run-time. In contrast to earlier stack computers, not only procedures play an important role, but also modules. The underlying premise is that objects local to the location of the present computation are accessed most fequently - and therefore require fast access by short instructions - whereas access to remote objects is relatively rare and requires less efficiency. Fast access is obtained by retaining "intermediate results" of address computations in fast registers (base address), in the expectation that they will be reused frequently, and that thereby their recomputation can be avoided. Several base address registers are used in the Lilith computer.

The origin of all address computations is a table with base addresses of all currently loaded data frames (see Fig. 7). A data frame is a contiguous area of store allocated to the (static) variables of a given module. By "module" we refer here and subsequently to compilation units; this excludes inner (nested) modules. Each loaded module has a number which is used as index to that frame table. The table resides at a fixed location and has a fixed length. The entry belonging to the module of which code is executed currently, is retained in the base address register G. It is the base address of "Global variables" in the sense of Algol or Pascal. G has to be reloaded whenever control transfers from one module to another. Data frames are static in the same sense that they are "permanent" for the duration of a program execution, with the (rare) exception of overlays performed by calls of the loader.

Data local to procedures are allocated in a stack which grows when a procedure is called and shrinks when it is terminated. Each coroutine (process) is allocated an area of the store, called a stack frame, when it is started, and which serves as the coroutine's workspace. The base address of the stack frame belonging to the coroutine currently under execution is stored in the register P, that of the last location allocated in this stack frame in register S, and the end of the workspace is designated by register H. P is used when a transfer from one coroutine to another coroutine occurs, S when a procedure is called or terminated. Each stack frame contains the hierarchy of data segments representing the variables local to the activated procedures. They are linked by the so-called dynamic chain of procedure activations. The base address of the last segment created is retained in register L (for Local data).

Local data are semi-dynamic in the sense that they are allocated for the duration of a procedure activation only. However, their addresses are determined by the compiler as offsets relative to the base address of their owner. Truly dynamic data are those allocated by explicitly programmed statements in an area of the store called heap. This storage area is managed by a utility module called Storage; these variables are accessed via pointer values. As in Pascal, pointers are bound to a given type, providing additional security in pointer handling.

Each loaded module owns a data frame and also a code frame, a contiguous area of store containing the code of all its procedures. The base address of the code frame of the currently executing module is retained in register F. Its value is used when calling a procedure, which is identified by a number used as index to a table containing the starting addresses of all procedures in a given module. This table resides in the header of the code frame. Using such an index instead of absolute addresses contributes to higher code density, particularly since procedure calls are very frequent instructions. The value of register F is changed whenever control transfers between modules. Jump addresses are relative to the F-register value.

8. The Lilith instruction set

Instructions consist of one or several bytes. They can be divided into four basic categories: Load and store instructions, operators, control instructions, and miscellaneous instructions:

The load and store instructions transfer data between memory (stack or heap) and the top of the stack, where they are accessed by operators. The top of the stack, where data are loaded as intermediate results (anonymous variables) is also called the expression stack. Load or store instructions require a single address only bacause the stack address is implicit; they are further subdivided according to the following criteria:

- data size: the transferred data are a word (16 bits), a double word, or a byte (halfword).
- addressing mode: local, global, external, stack, indexed, and immediate mode (the latter for load instructions only).
- address length: 4, 8, or 16 bit address (see Fig. 8).

The presence of different address lengths suggests that variables with frequent access be allocated with small offsets. Our present compiler does not perform any such optimization. The gain to be made does not appear to be overwhelming. The set of directly accessed (statically declared) variables is usually quite small, because structured variables are addressed indirectly.

The various addressing modes are defined as follows (m and n denote instruction parameters, and a the resulting address):

- Local mode: a = L+n, used for variables local to procedures.
- Global mode: a = G+n, used for global variables in the current module.
- Stack mode: a = s+n, where s is the value on top of the stack; mode used for indirect addressing and access via pointers.
- External mode: a = T[m]+n, T is the table of data frame addresses, m a module number; mode used for external variables imported from other modules.
- Indexed mode: a = s1 + k*s2, s1 is the array's base address, s2 the computed index (s1, s2 on stack), and k is a multiplier depending on the size of the accessed data type.
- Immediate mode: a = n. The loaded value is the parameter itself; mode used to generate constants.

The above explanations are given in this detail in order to show that the constructs defined in the programming language are strongly reflected by, i.e. have directly influenced, the design of the Lilith architecture. The beneficial consequence is not only ease of compilation, but simplicity of the linking loader. Whereas our Modula-2 system for the PDP-11 computer for good reasons requires a linker, such is not necessary for the Lilith implementation. A linker collects the code files of all required modules and links them together into an absolute (or relocatable) store image. This task can be performed directly by the loader, because it only has to insert module numbers (table indices) in instructions with external addressing mode.

The second category of instructions are the operators. They take operands from the top of the stack and replace them by the result. The Lilith instruction set includes operators for CARDINAL (unsigned), INTEGER (signed), double-precision, floating-point, BOOLEAN, and set arithmetic. It directly reflects the operations available in Modula-2.

The orientation towards a clean stack architecture also required a full set of comparison instructions which generate a BOOLEAN result. Distinct sets are provided for CARDINAL and INTEGER comparison. The distinction between CARDINAL and INTEGER arithmetic is partially due to the desire to be able to use all bits of a word to represent unsigned numbers, such as addresses. It would be of a lesser importance, if the wordsize were larger. However, our experience shows that it is desirable also from a purely logical point of view to declare variables to be non-negative, if in fact a negative value does never occur. Most of our programs require variables of the type CARDINAL, whereas the type INTEGER occurs only rarely. Although using 2's complement representation for negative values, addition and subtraction are implemented by the same hardware operations for both kinds of arithmetic, they differ in their conditions indicating overflow.

Control instructions include procedure calls and jumps. Conditional jumps are generated for IF, WHILE, REPEAT, and LOOP statements. They fetch their BOOLEAN operand from the stack. Special control instructions mirror the CASE and FOR statements.

Different calls are used for procedures declared in the current module and for those in other modules. For local procedures there exist call instructions with short 4-bit addresses, as they occur rather frequently. Calls for external procedures not only include an address parameter, but also a module number to be updated by the loader. Furthermore, an instruction is provided for so-called formal procedures, i.e. procedures that are either supplied as parameters or assigned to procedure variables.

There also exists an instruction for the transfer of control between coroutines. Various instructions may cause a trap, if the result cannot be computed. Such a trap is considered like an interrupt requested by the processor itself, and corresponds to a coroutine transfer with fixed parameters. The same mechanism is activated by the TRAP instruction (which corresponds to a HALT statement in Modula).

Arithmetic operators generate traps when unable to compute the correct result (e.g. overflow). Traps from CARDINAL and INTEGER arithmetic can be suppressed (masked) upon request; the programmer is then presumably aware that results are computed modulo 2^16. Also, load and store instructions generate a trap, if their address is NIL. This test requires a single micro instruction only. The routines for bitmap handling generate traps, if attempting to access data outside the specified bitmap. All these test are quite inexpensive (but not free).

A test for an array index lying within the specified bounds, or for a value to be within the subrange admitted by a variable, is more complicated. It requires two comparisons with arbitrary values. Therefore, the M-code contains an instruction for an "in range" test. The programmer may choose to omit these tests by selecting a compiler option that suppresses the generation of these test instructions.

These extensive checking facilities reflect our strong belief in designing an implementation (including the hardware) which properly supports a language's abstractions. For example, if the language provides the data type CARDINAL, its implementations should signal an error, if a negative result appears, just as it should signal an error, when a non-existing element of an array is identified. Omissions in this regard are to be considered as inadequacy in implementation. Nevertheless, the argument whether or not the experienced and conscientious programmer should be burdened with these "redundant" checks remains open. Our choice is to give the programmer the option to suppress at least the more expensive checks, at his own peril.

The category of miscellaneous instructions contains operators for reading and writing data on the input/output channels, and four instructions used for operating on bitmaps: The DDT instruction (display dot) writes a single dot at a specified coordinate, REPL replicates a bit pattern over a rectangle - a so-called block - in a given bitmap. The coordinates of this block are relative to the specified bitmap and are given in terms of dot coordinates rather than word addresses. The BBLT instruction (bit block transfer) copies a source block into a destination block. The DCH instruction (display character) copies the bitmap of a character (given its ASCII code) from a font table into a specified place of a bitmap.

The function of these bitmap instructions could well be coded in Modula-2 programs. Instead, they are included as single instructions represented by micro-coded routines. The primary reason is efficiency. The routines include checks against inconsistent parameters, such as blocks that do not fully lie within the bitmap. An essential detail is that they use the same convention about parameters as do regular procedures and operators: parameters are always passed via the stack. Modula-2 for Lilith offers a facility to use these instructions as if they were programmed as regular procedures. This uniformity of parameter passing has proved to be an invaluable asset.

Some analysis of representative programs reveals that M-code yields a significantly higher density of compiled code than do conventional instruction sets. Compared with the code compiled for the ubiquitous PDP-11, we obtained an improvement factor of 3.9. This implies that code for the same program occupies about one quarter of the memory space in the Lilith computer than in a PDP-11. This factor is noteworthy even in times of rapidly decreasing memory prices!

The principal contribution to this result stems from the short address fields. Dominant are the counts of load and store instructions; they address the stack implicitly and hence need only one address field. Access to local variables is most frequent; global variables are addressed about half as often, and external variables occur rarely. Jumps account for about 10% of all instructions, and procedure calls are about equally frequent. The following table displays percentage figures obtained from four programs (of different authors) for the most frequent instruction classes.

1-byte instr.	70.2	71.7	62.8	72.5
2-byte instr.	16.6	17.3	12.6	12.0
3-byte instr.	13.1	11.0	24.5	15.4
Load immediate	15.1	14.2	17.3	15.2
Load local	16.3	21.6	16.3	19.1
Load global	8.2	5.2	3.7	7.9
Load indirect	5.2	6.4	5.5	5.8
Store local	5.8	6.5	5.8	6.0
Store global	2.6	1.1	1.0	3.3
Store indirect	4.2	3.0	1.1	4.0
Operators	5.6	5.7	4.8	5.9
Comparators	3.9	4.4	5.6	3.7
Jumps	7.3	7.7	9.3	6.2
Calls	6.4	8.1	14.5	6.9
Total counts (100%)	11052	7370	7936	2814

Instructions are executed by a micro-coded program called the Interpreter, which may well be expressed in Modula; this algorithmic definition of the Lilith instruction set has proved to be extremely valuable as interface between the micro-programmer and the compiler designer.

9. The Lilith hardware structure

The following requirements determined the design of the hardware most significantly:

- fast implementation of the M-code interpreter, in particular of its stack architecture,
- the need for efficient implementation of the bitmap instructions which involve a large amount of bit pushing and

partial word accesses (bit addressing).
- high bandwidth between memory and display for continuous refreshing.
- the desire for a simple structure with a relatively large, homogenous store.
- ease of serviceability.

The computing power required by the bitmap instructions eliminated the choice of a one-chip processor. An even stronger reason against such a choice was the project's purpose to find a hardware architecture truly suitable for use with code compiled from a high-level language. The bit-slice processor Am2901 offered an ideal solution between a one-chip processor and the complete design of a unit built with SSI and MSI components. It allows for a basic instruction cycle that is about a fourth of a memory cycle (150 ns). This is a good relation considering the average amount of processing required between memory accesses.

The processor is built around a 16-bit wide bus connecting the arithmetic-logic unit (ALU) with the memory for transfer of data and addresses. Also connected are the instruction fetch unit (IFU), the disk and display controllers, and the interfaces to the standard low-speed I/O devices keyboard, Mouse, and serial V24 (RS232) line. Bus sources and destinations are specified in each micro-instruction by 4-bit fields which are directly decoded. The bus uses tri-state logic.

The refreshing of the full screen requires a signal with a bandwidth of 13 MHz, if interlacing and a rate of 50 half pictures per second is assumed. This implies that on the average one 16-bit word has to be fetched every 1.1 us, which implies that memory would be available to the processor about 50% of the time. This unacceptably low rate calls for a memory with an access path wider than 16 bits. It was decided to implement a 64-bit wide memory.

A third candidate for direct memory access is the instruction stream. Like the display, this port requires sequential reading only and therefore can benefit from a wide access path feeding an internal buffer. This organization reduces the average time that the memory is devoted to display and instruction fetching, i.e. where it is inaccessible to the data port of the main processor, to about 10%. The overall structure of the Lilith hardware is shown in Fig. 9. Its heart is the microcontrol unit (MCU) which contains the clock and controls the instruction stream.

9.1 The micro-control unit

The micro-control unit (MCU) consists primarily of a memory for the microcode, a micro- instruction register (MIR), an address incrementer, and some decoding logic. A micro- instruction consists of 40 bits; its formats are shown in Fig. 10. The micro-instruction address is a 12 bit integer, hence the memory may have at most 4K locations. Actually, only 2K are used and implemented as a read-only store (ROM). An additional 2K RAM may be supplied. Approximately 1K is used by initialization routines (bootstrap loader) and the M-code interpreter, and 1K is needed for the bitmap routines and the floating-point instructions.

Fig. 11 shows the structure of the micro-control unit. The next instruction's address is taken from one of several sources:

- the incrementer (normal case)
- an address stack (subroutine return)
- the current instruction (microcode jump)
- a table of addresses of routines which correspond to M-codes
- according to a pending interrupt request.

The addresses are generated by Am2911 bit-slice controllers which contain an incrementer and a short stack for subroutine return addresses. For jumps, the next address is supplied from sources external to the 2911. Conventional jumps take the address directly from the instruction register (MIR). Exceptions are the jumps to the start of the microcode routine representing the next M-code instruction. Here the address is taken from a ROM which maps the 8-bit M-code into a 12-bit address. This exception is signalled by a micro-instruction whose source field value causes the address to be selected from the map ROM. An exception to this exception occurs if an (unmasked) interrupt request is pending, in which case the next address is the fixed number assigned to the requesting line. Thereby the M-code sequence can be interrupted without requiring any additional micro-instructions, and the transition to the next micro-instruction routine is initiated by a single instruction at the end of each routine.

A tag bit of each micro-instruction determines whether it is to be interpreted as a regular or as a jump instruction. During execution of the latter the main processor is disabled. Jumps are conditional upon the state of the main processor's condition code register determined by the ALU's result computed during the previous cycle.

9.2 The arithmetic logic unit

The ALU's heart is a 2901 bit-slice processor. It contains the logic for integer arithmetic (addition) and for bit-parallel logical operations, and a set of 16 fast registers. Half of them are used for global state variables of the M-code interpreter, the others as work registers local to each microcode routine. The 2901 core is augmented by two facilities dictated by the requirements of the stack architecture and by the bitmap routines: a fast stack memory and a barrel shifter (Fig. 12).

The fast stack is a store of 16 locations (16 bit wide) and an address ·incrementer/decrementer. This memory holds the intermediate results during evaluation of expressions and statements, and must be regarded as logically being part of the (main) stack, but physically separate. Load instructions fetch data from the (main) stack in memory and push them onto the fast expression stack. Store instructions pop the expression stack and deposit data in main memory. As a consequence, each such instruction takes a single main memory cycle only. More precisely, data loaded from and stored into the main stack are transferred to and from a register in the 2901 processor itself, while during the same cycle this T-register is saved (or restored) into (from) the expression stack:

```
    Load:   push T onto stack; Bus -> T
    Store:  T -> Bus; pop stack into T
```

Operations such as addition, comparison, AND, OR, etc., can also be performed in a single cycle, because both operands are immediately accessible:

```
    Add:    T + top stack -> T; pop stack
```

The hardware represents a genuine stack in so far as the current stack top is the only accessible element, and that its address is inaccessible to the programmer. This address is generated by a 4-bit up/down counter and directly fed to a 16x16 high-speed RAM. A slight complication arises because address incrementation for a pop must occur before the data fetch, whereas the decrementing for a push must occur after the store. However, both address counting and data access must be performed during the same clock cycle. The solution is found in using an extra adder and to operate according to the following scheme:

```
    push:   DEC(x); S[x+1] := data
    pop:    INC(x); data   := S[x]
```

The circuit of the entire stack mechanism is shown in Fig. 13. It may be surprising that the fast stack has a depth of only 16. In practice, this proved to be ample. It should be noted that the compiler can keep track of the number of stack locations loaded, and hence no runtime stack overflow can occur, nor need it be monitored. The stack is empty after execution of each statement. In the case of function procedures, the expression stack has to be saved into the main stack before, and restored after the call. Special M-code instructions are provided for this purpose.

The barrel shifter is prefixed to the input lines of the 2901 processor. It allows the rotation of data by any number of bit positions between 0 and 15. Together with the logical instructions (AND, OR) it provides the necessary speed for partial word handling extensively used in all bitmap operations.

It is designed such that it can also generate masks of 0 to 15 bits in one cycle. The shift count (mask length) can either be taken from a field in the micro-instruction itself, or from a special 4-bit shift count register, also contained in the ALU.

9.3. The memory

The memory is built with 16K dynamic RAM chips distributed on four boards, each being organized as a 16K*32 block. For reading, 32 bits are accessed simultaneously from two of the four boards. Multiplexors select 8 of the 32 bits for output to the processor bus via the so-called CPU port. For writing, the same connection is used, and the data are fed to four chips in parallel, of which only one is enabled through the chip select signal. Fig. 14 shows the scheme for two boards; together they represent a 64K*16 bit memory for writing, or a 16K*64 bit memory for reading..

The choice of a 64-bit wide access path guarantees the necessary memory signal bandwidth, but it also poses significant electrical problems that should not be underestimated. Their mastery is an order of magnitude more difficult than the handling of conventional 8-bit microcomputer systems.

Processor and display operate asynchronously. Hence, an arbiter mechanism is needed for controlling memory access. It can easily be extended to accommodate several instead of only two ports. Each port is assigned a fixed priority, and the request from the source with highest rank among those pending is honoured. Fig. 15 shows the circuit used; it contains cascaded priority latches that retain posted requests. Also shown is the circuit used for the synchronization of a requestor (the CPU port as an example) and the memory, which operate on separate clocks. The priority latch is common to all ports, the other parts are individually replicated for each port. Fig. 16 shows the signal timing: If the port requests a memory cycle, the bus data, representing an address, are latched in the memory address register MAR, the port is marked busy, and the request is passed on to the arbiter. Unless a request with higher priority is present, the signal CPU.SEL goes high, indicating that the memory cycle now started belongs to the CPU port and MAR is gated to the address lines. When terminated, the signal CLR resets the busy latch, indicating to the polling CPU that its request has been served.

9.4. The instruction fetch unit

Instructions are fetched via a separate memory port controlled by the instruction fetch unit (IFU). This unit contains its own address registers (PC,F) and an 8-byte buffer. The buffer can be regarded as a small cache memory and is particularly effective because access is mostly sequential. Reloading occurs when either the buffer is empty, or when a new address is fed to the PC by a control instruction. The IFU contains its own address incrementer (the PC register is a counter) and an adder forming the sum of the PC and F values. This adder is 18 bits wide. A byte is fetched from the buffer and the address is incremented whenever the micro-controller executes a jump enabling the map ROM. Fig. 17 is a block diagram of the IFU.

9.5. The Mouse

The Mouse is a device to designate positions on the display screen. It operates on the principle that movements of the operator's hand on his desk are sensed, rather than on the recording of precise, absolute coordinates. A cursor is displayed (by appropriate programming) on the screen, changing its position according to the signals received from the Mouse. Hence, positioning of the cursor can be as accurate as the display's resolution allows, without requiring a high-precision digitizer device. The Mouse is also equipped with three pushbuttons (eyes) and is connected to the keyboard by a thin tail; hence its name.

The movements are transmitted via a ball to two perpendicular wheels, whose "spokes" are seen by a light sensor. The direction of their turning is perceived by sampling two signals received from spokes which are offset. If we combine the two binary signals and represent them as numbers to the base 4, the wheels' turning results in sample value sequences 0,2,3,1,0, ... or 0,1,3,2,0, ... depending on the sense of their rotation (see Fig. 18).

The interface for the Mouse contains two counters for the x- and y-coordinates. They are incremented or decremented whenever a transition of the input signals occurs as indicated by the two above sequences. A state machine registers the signal values sampled at two consecutive clock ticks; a ROM is used to map them into the necessary counting pulses.

9.6. The Monitor

The Monitor is an additional unit which is not present in the computer under normal circumstances, but for which nevertheless a permanent slot is reserved, such that it can be inserted any time. It represents a small computer of its own, and it has the capability to take full control over the Lilith processor. It is therefore used for servicing when the Lilith hardware fails, and it played a most crucial role during the entire development and debugging phases of the Lilith computer.

The Monitor's heart is a Motorola 6802 one-chip microprocessor, augmented by a 2K byte ROM and a 4K byte RAM, interface registers to the Lilith hardware, and a serial line interface to a terminal (UART). Its block diagram is given in Fig. 19. The Monitor can

- read the microinstruction register (MIR)
- supply the next microinstruction (disabling MIR)
- read the micro-program counter (2911)
- supply the next instruction address (disabling 2911)
- read the processor bus
- feed data to the processor bus
- disable the processor clock (halt)
- send clock pulses (single or multiple step)

For debugging and servicing, an elaborate set of programs was developed. In addition to a standard "operating system" residing in the ROMs, test programs can be loaded into the RAM from a terminal. We extensively used an HP 2645A terminal with tape cassettes as our program library store. When a new Lilith machine is to be tested, the Monitor is used to first test the MCU board, then to test the ALU board, thereafter the memory (in conjunction with MCU and ALU), then the IFU, and finally the interface boards. The Monitor not only made a front panel superfluous, but allowed the construction of the entire computer with the aid of only an oscilloscope and, very rarely, a small logic state analyzer.

9.7. The physical layout

The Lilith computer is designed to fit beside or underneath a table on which the 15"-display, the keyboard, and the mouse are placed. The cabinet has a height of 74 cm; it is 43 cm wide and 55 cm deep. The disk cartridge is accessible from the front.

The electronic components are placed on 10 boards housed in a rack with dimensions 42*35*30 cm. One board each contains the microcontrol unit, the arithmetic-logic unit, the processor part and interfaces to keyboard, mouse, and serial data line, the instruction fetch unit, the display interface and the disk interface. Four boards contain the main memory. Another board slot is reserved for a 2K*40 microcode RAM, one for the Monitor, and 5 slots are free for future experiments with other units or interfaces. This makes the computer suitable as an object for experimentation on the hardware as well as the software level.

The remaining space in the cabinet is taken by the disk drive and the power supply. Conventional linear power supplies were built after several disappointing experiments with modern switching power supplies that offer a much improved efficiency. They turned out to be unable to cope with the European 220 Volts.

10. Conclusions

The personal computer leads to an entirely new computing environment. Due to the high bandwidth of information between its user and his tool, a close interaction is possible that cannot be provided by a central, remotely accessed facility. The personal computer is much more than an "intelligent terminal", because it puts the computing power near the user. A particularly attractive feature is its constant availability, and consequently the owner's independence of a computing center's service hours.

Interactive usage is of particularly high value in the development of software, where text editing, compiling, and testing are the prime activities. In our experience, a personal computer increases the effectiveness of a competent software engineer by an order of magnitude. I stress the attribute "competent", for he needs the wisdom to leave his tool and retreat to quiet deliberations when deeper problems of algorithmic design appear. For the less competent engineer, the personal computer amplifies the danger of seduction to programming by trial and error ("hacking"), a method that is unacceptable in professional software engineering.

It has now become a widely accepted view that the software engineer's notational tool must be a high-level programming language. When large, complex systems are the objective, the tool must support modularization and the specification of interfaces. We have designed the language Modula-2, a more modern version of Pascal, with the principal addition of a module structure. Our implementation connects this feature with the facility of separate compilation. Separate compilation, however, is not independent compilation. On the contrary, the compiler must fully check the consistency of the separately compiled modules as if they were written as a single piece of text. The separation of global modules into definition and implementation parts makes it possible to define those aspects of a module that are significant for its clients apart from those that are private to its implementation. It reinforces the strategy of first breaking down a planned system into modules, then to define their interfaces with the goal to keep them "thin", and finally to let the members of the programming team implement the modules with relative independence.

The exclusive use of a high-level language makes it possible to design a computer architecture without regard of its suitability to assembler coding. The resulting architecture is organized around a stack. The instruction set is designed to provide a high density of code, largely due to the use of variable address length.

It is particularly attractive to design such an architecture and instruction set, if no conventional computer must be used for its interpretation. We have therefore also undertaken the design of a hardware system with the purposes to interpret this code efficiently and to accommodate the use of a high-resolution display. The latter requires a high memory bandwidth and bursts of fast computation. The implementation of a microcoded interpreter and the inclusion of a few special instructions for bitmap handling appears to be an ideal solution. These instructions correspond to microcoded routines that perform the necessary bit-pushing with greatest efficiency.

As an experiment to integrate the design of a programming language - the software engineer's notational tool - the development of its compiler and environment, the design of a computer architecture and instruction set, and the construction of the hardware - the software engineer's physical tool - the project has been successful and exciting. The resulting system is, of course, not without its deficiencies. Our consolation is that, if we did not know of items that should have been done differently, we would not have learned through our research. Also, the project had to be conducted with severe restrictions on manpower. This had the benefit that no significant management problems were encountered.

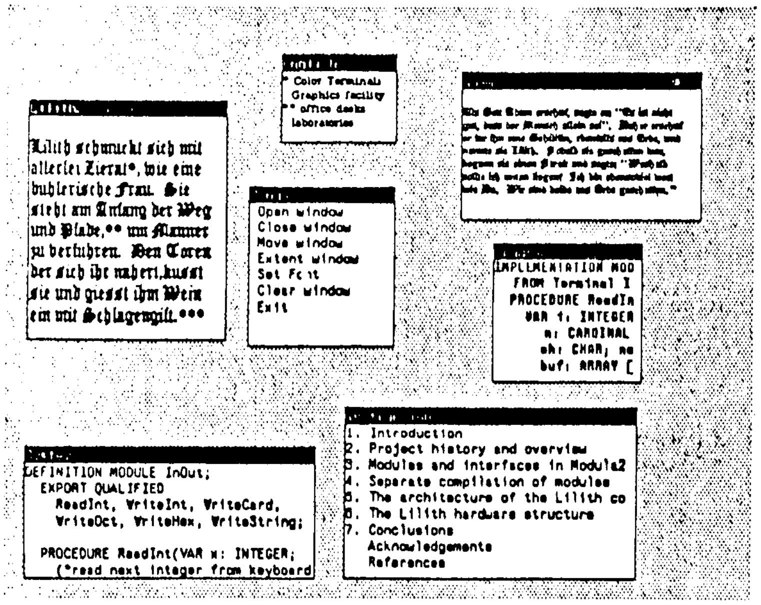
As far as the hardware is concerned, an additional constraint was the limited availability of modern technology. It was therefore decided to rely on commercially available TTL chips only, apart from MOS technology for the memory. The integrated, top-down design from software to hardware outlined by this project, is especially relevant in view of the future role of VLSI technology. Its unlimited possibilities require that the designer obtain new criteria guiding his objectives. The top-down approach crossing the soft/hardware boundary tells the hardware designer what is needed rather than the hardware customer what is available. An aspect of this project that we were unable to tackle was the design of LSI chips representing the essential units of the Lilith computer, incorporating the unconventional aspects of its architecture. The chip count (as well as power supply problems) could thereby have been reduced quite drastically. We hope that someone better equipped for the task will pursue this challenge.

Acknowledgements

I welcome the opportunity to thank all those who contributed to the success of this project, in particular those who suffered through setbacks and lost neither the sight of its goal nor the conviction of its worth. My thanks are due to the team members L.Geissmann, A.Gorrengourt, J.Hoppe, Ch.Jacobi, S.E.Knudsen, I.Noack, Ch.Richner, and W.Winiger at ETH Zurich, and L.Bingham, J.Nielson, and F.Ostler at BYU, Provo. I am particularly indebted to R. Ohran for his untiring enthusiasm and for not giving up even in moments of despair. The idea that such a project would be both worthwhile and feasible originated during my sabbatical leave at the Xerox Palo Alto Research Center in 1976/77, and I am most grateful to its staff members for many inspiring ideas. I thank Ch.Geschke, B.W.Lampson, J.Morris, and E.Satterthwaite for convincing me that complex software was not only needed but possible on small personal computers, and E.M.McCreight for sticking to his belief that it was possible to teach hardware design to a software type and for withholding his plentyful knowledge of all the difficulties to be encountered. Furthermore I am grateful to J.D.Nicoud of ETH Lausanne for his help in the design and for producing the Mouse. Finally I wish to acknowledge the financial support of ETH, making the entire endeavour possible.

References

1. C.P.Thacker, E.M.McCreight, B.W.Lampson, R.F.Sproull, D.R.Boggs. Alto: A personal computer. Xerox Palo Alto Research Center. Report CSL-79-11 (1979)
2. N.Wirth. A personal computer designed for a high-level language. In "Microcomputing", W.Remmele, H.Schecher, Eds., 115-134, Stuttgart, 1979.
3. N.Wirth. Modula-2. Institut fur Informatik, ETH. Report 36, 1980.
4. B.W.Lampson. An open operating system for a single-user machine. Revue Francaise d'Automatique. pp 5-18, Sept. 1975.
5. N.Wirth. Modula: A language for modular multiprogramming. Software - Practice and Experience, 7, 3-35. Jan.1977.
6. Ch.M.Geschke, J.H.Morris, E.H.Satterthwaite. Early experience with Mesa. Comm. ACM, 20, 8, 540-553, Aug. 1977.
7. L.Geissmann. Modulkonzept und separate Compilation in der Programmiersprache Modula-2. In "Microcomputing", 98-114, (see Ref.2)
8. H.C.Lauer, E.H.Satterthwaite. The impact of Mesa on system design. Proc. Int'l Conf. on Software Engineering, Munich, 174-182, IEEE (1979).
9. D.D.Redell et al. Pilot: An operating system for a personal computer. Comm.ACM 23, 2, 81-92 (Feb. 1980)



Lilith ſchmuckt ſich mit
allerlei Zierat*, wie eine
buhleriſche Frau. Sie
ſteht am Anfang der Weg
und Pfade,** um Manner
zu verfuhren. Den Toren
der ſich ihr nahert, kuſſt
ſie und gieſſt ihm Wein
ein mit Schlagengift.***

```
IMPLEMENTATION MOD
  FROM Terminal I
  PROCEDURE ReadIn
    VAR i: INTEGER
      n: CARDINAL
      ch: CHAR; ne
      buf: ARRAY [
```

```
DEFINITION MODULE InOut;
  EXPORT QUALIFIED
    ReadInt, WriteInt, WriteCard,
    WriteOct, WriteHex, WriteString;

  PROCEDURE ReadInt(VAR x: INTEGER;
    (*read next integer from keyboard
```

1. Introduction
2. Project history and overview
3. Modules and interfaces in Modula2
4. Separate compilation of modules
5. The architecture of the Lilith co
6. The Lilith hardware structure
7. Conclusions
   Acknowledgements
   References

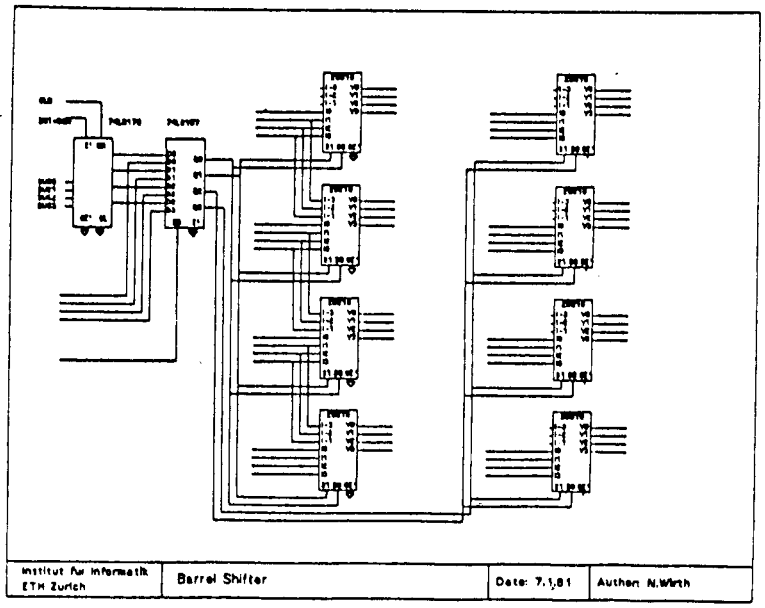


Fig. 1 Examples of displayed pictures in original screen resolution.

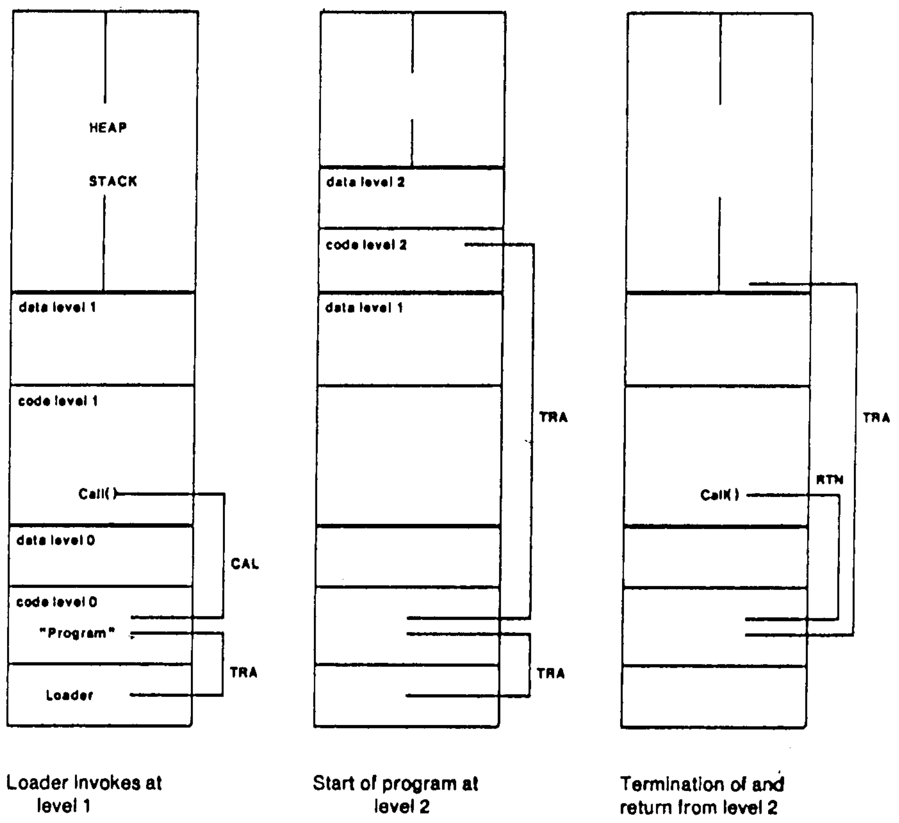


Fig.2. Loading of program at level 2

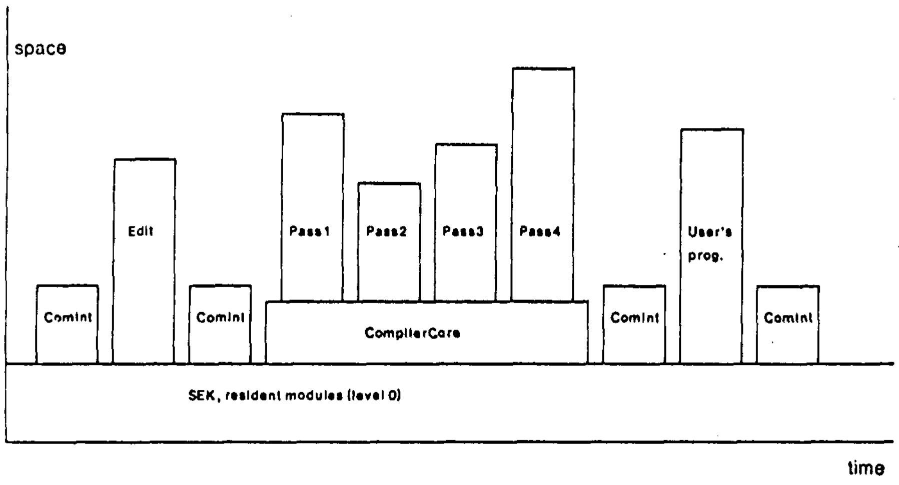


Fig.3. Typical sequence of program loadings

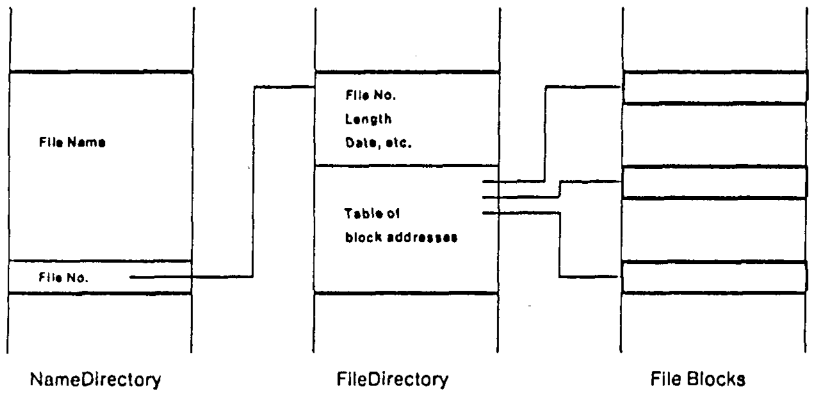


Fig.4. Storage layout on disk

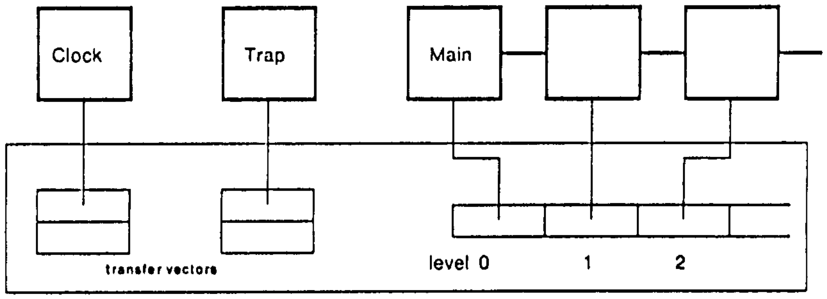


Fig. 5. The three processes of Medos

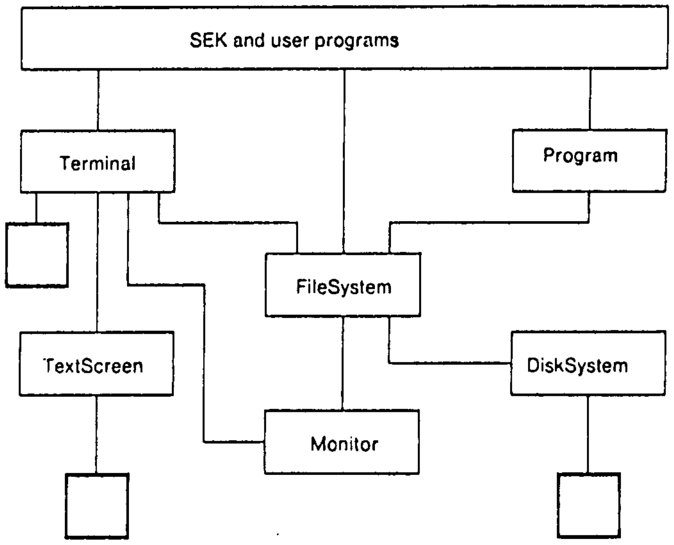


Fig. 6. Procedure call hierarchy

Registers:	L	pointer to local data segment
	G	pointer to global data segment
	S	pointer to top of stack
	H	pointer to stack limit
	F	pointer to current code frame
	PC	pointer to current instruction
	P	pointer to current process
	M	interrupt mask

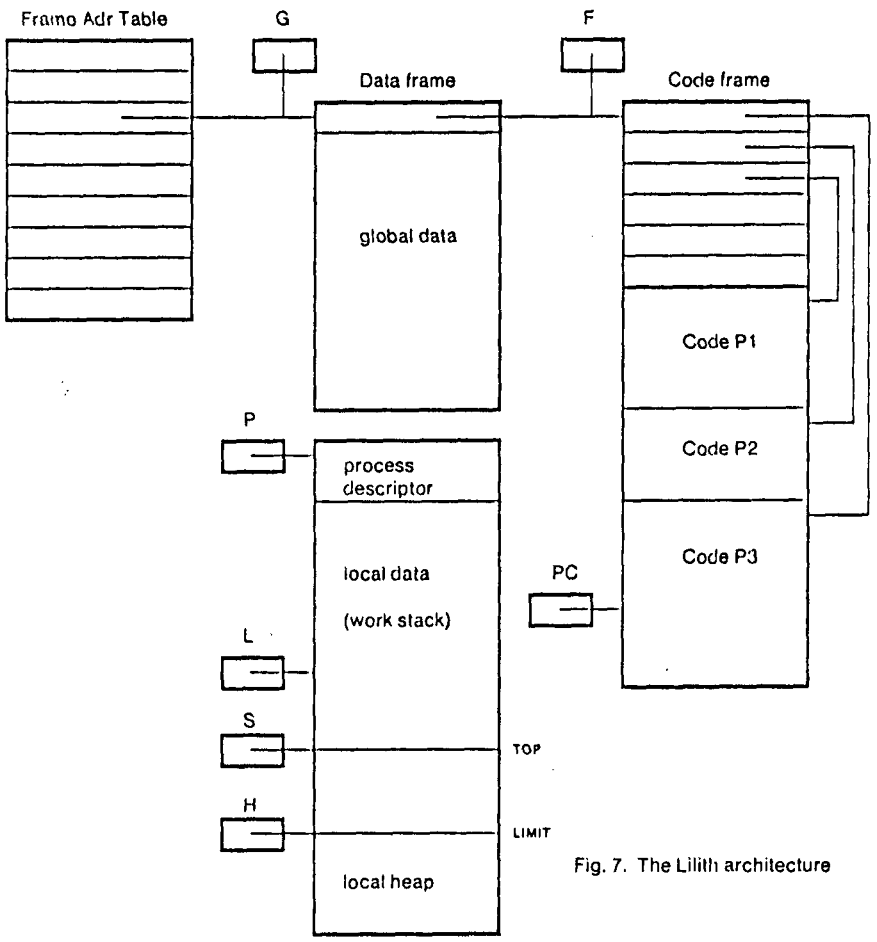
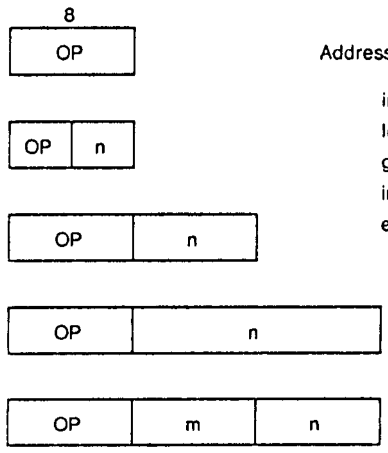


Fig. 7. The Lilith architecture



Addressing modes:

immediate	n
local	L + n
global	G + n
indirect	S + n
external	FrmTab[m] + n

Fig. 8. Lilith instruction formats

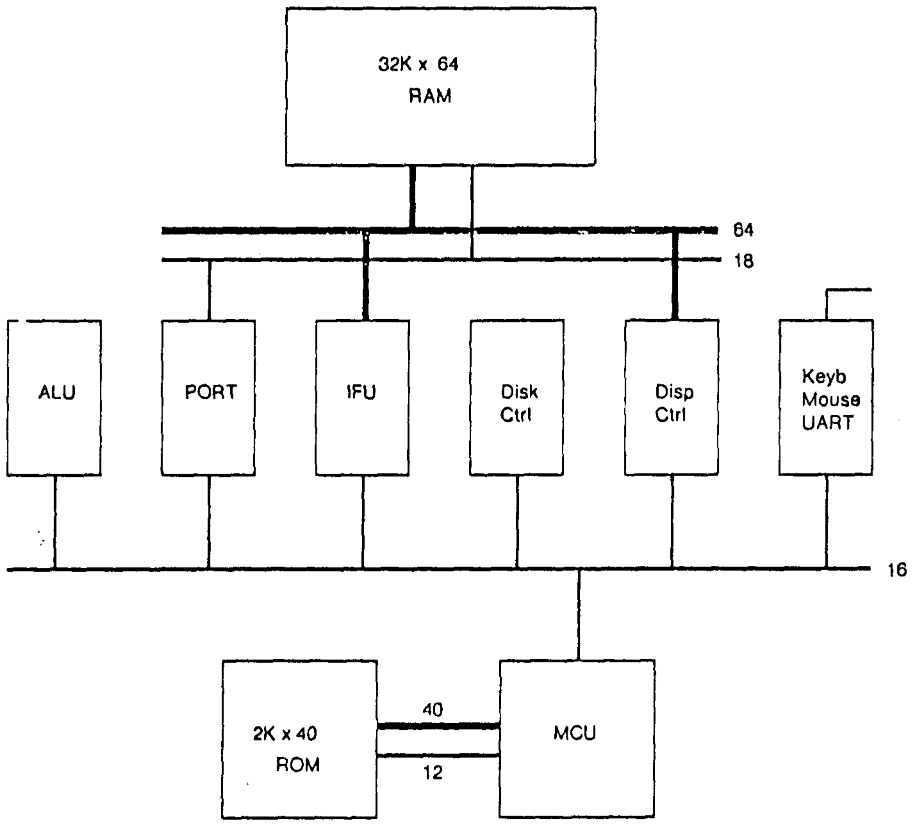


Fig. 9. Lilith hardware structure

1	3	3	3	2	4	4	2	4	1	3	1	1	8
1	Dst	Fct	RS	C	A	B	SM	SC	T	PC	S	1	Constant

1	3	3	3	2	4	4	2	4	1	3	1	1	4	4
1	Dst	Fct	RS	C	A	B	SM	SC	T	PC	S	0	BusDest	BusSrc

1	12	8	5	1	3	10
0	Jump address	CondMask		T	PC	

Dst, Fct, RS, A, B:	2901 control fields
PC:	2911 control field
SM, SC:	Shift mode and count
S:	Stack enable

Fig. 10. The micro-instruction formats

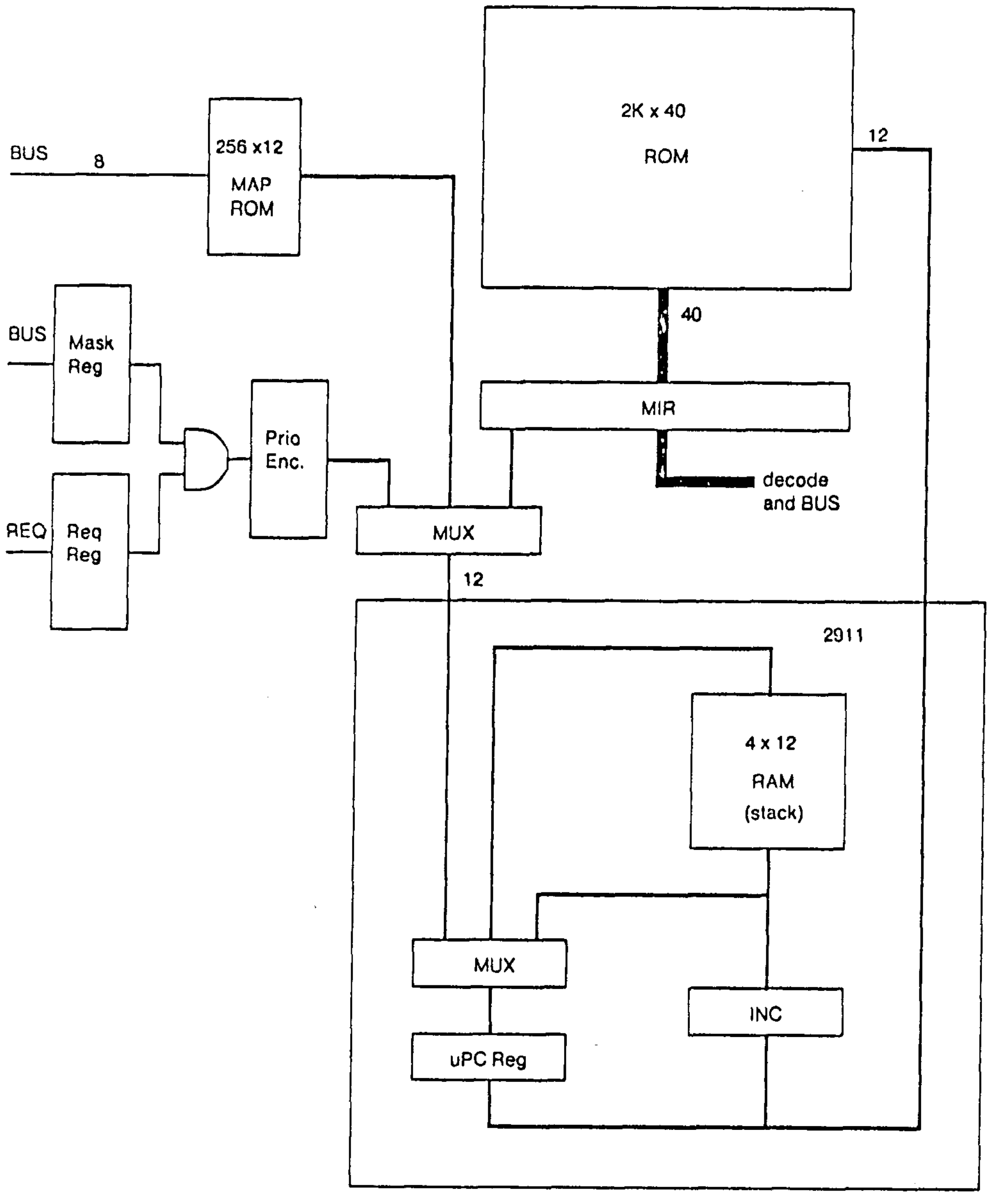


Fig. 11. The Micro-Control Unit

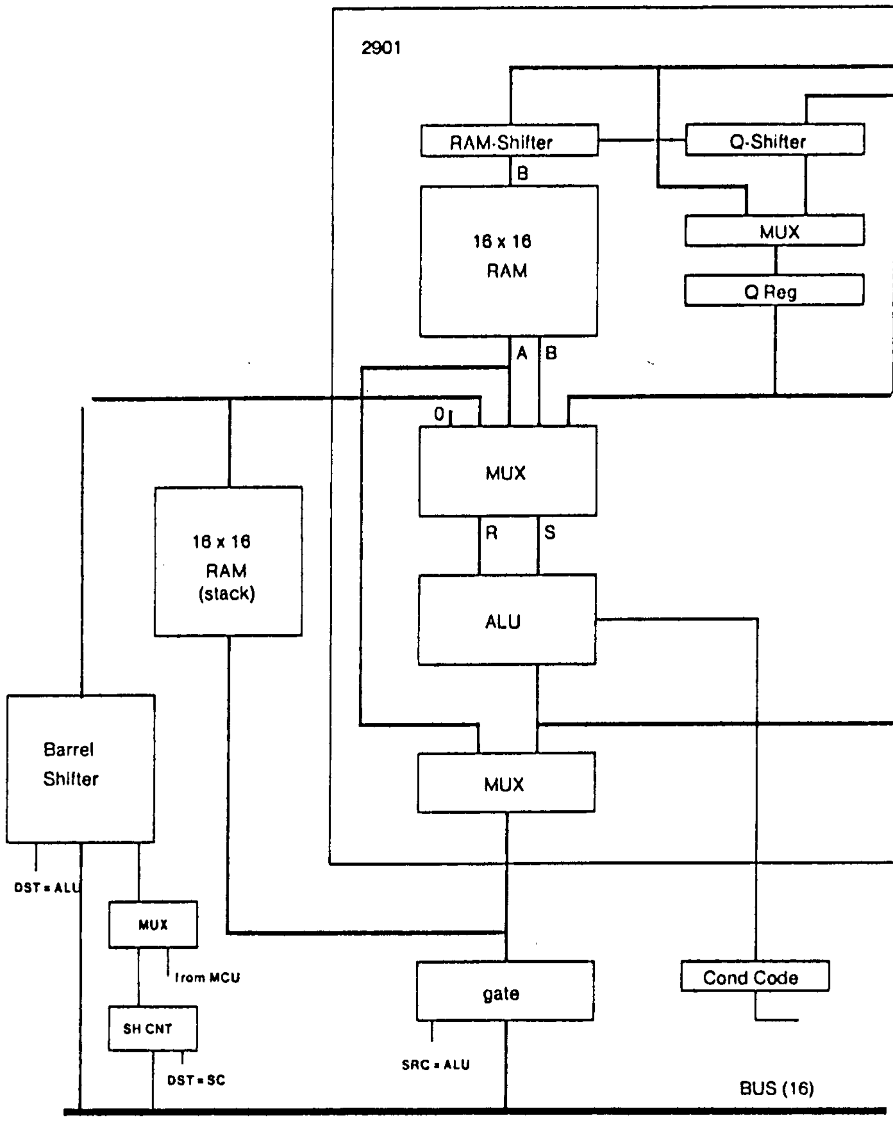


Fig. 12. The Arithmetic-Logic Unit

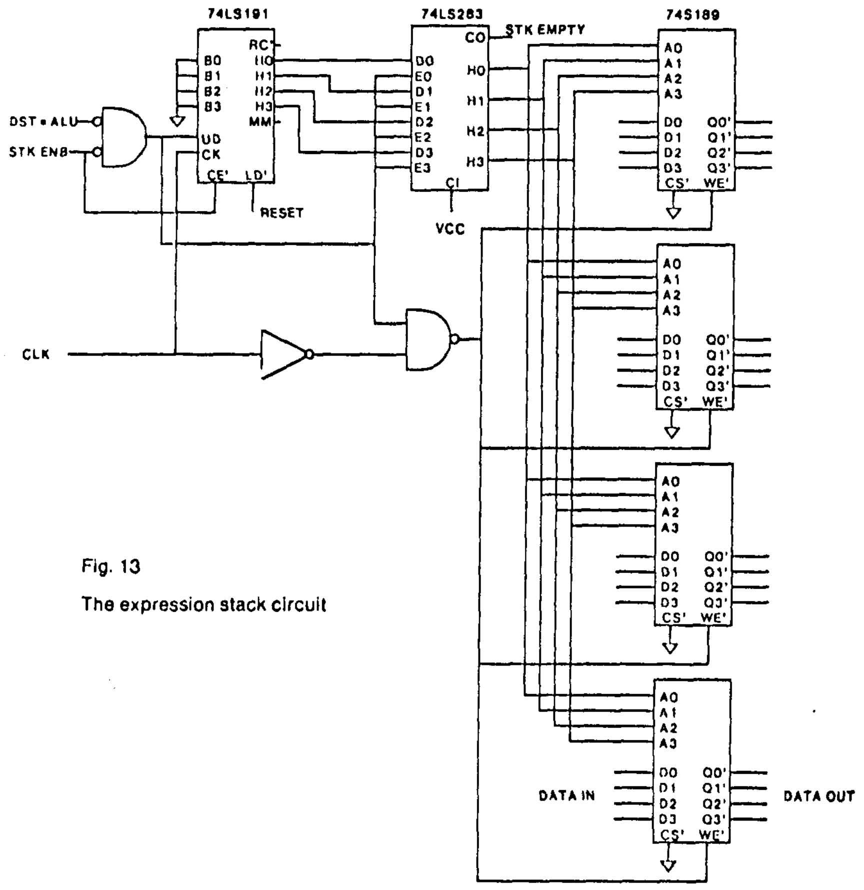


Fig. 13

The expression stack circuit

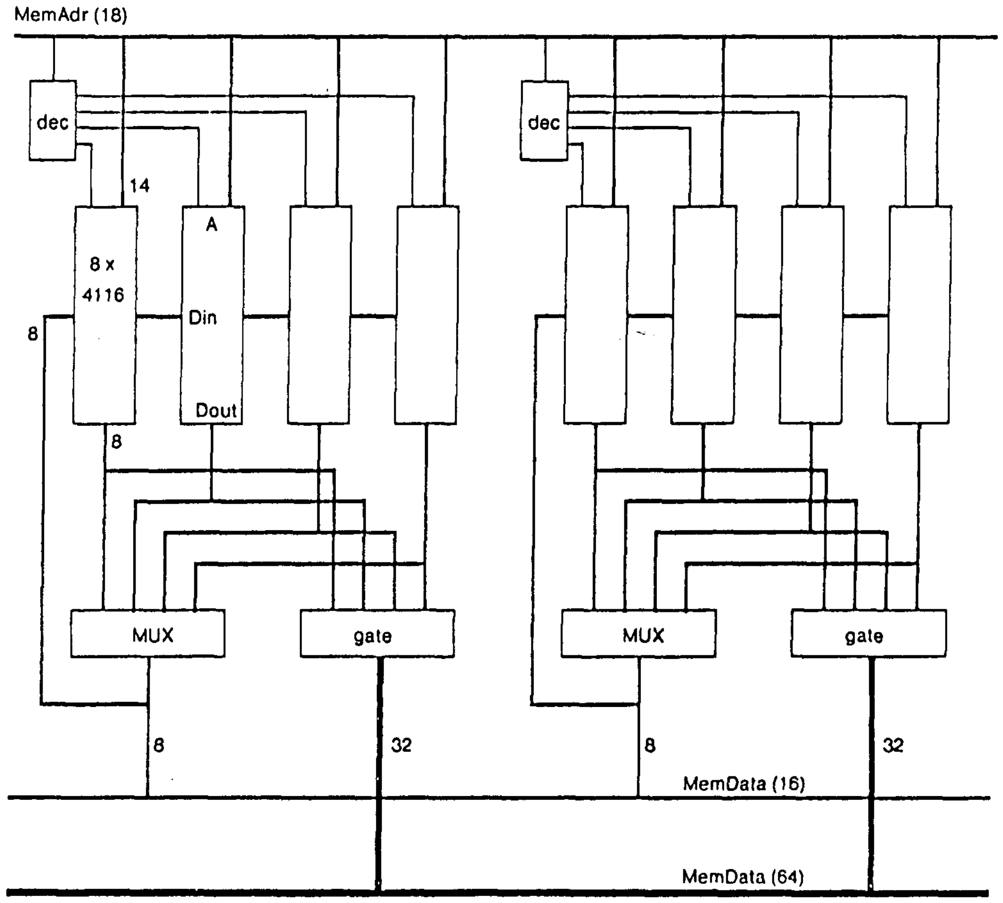


Fig. 14. Memory organization

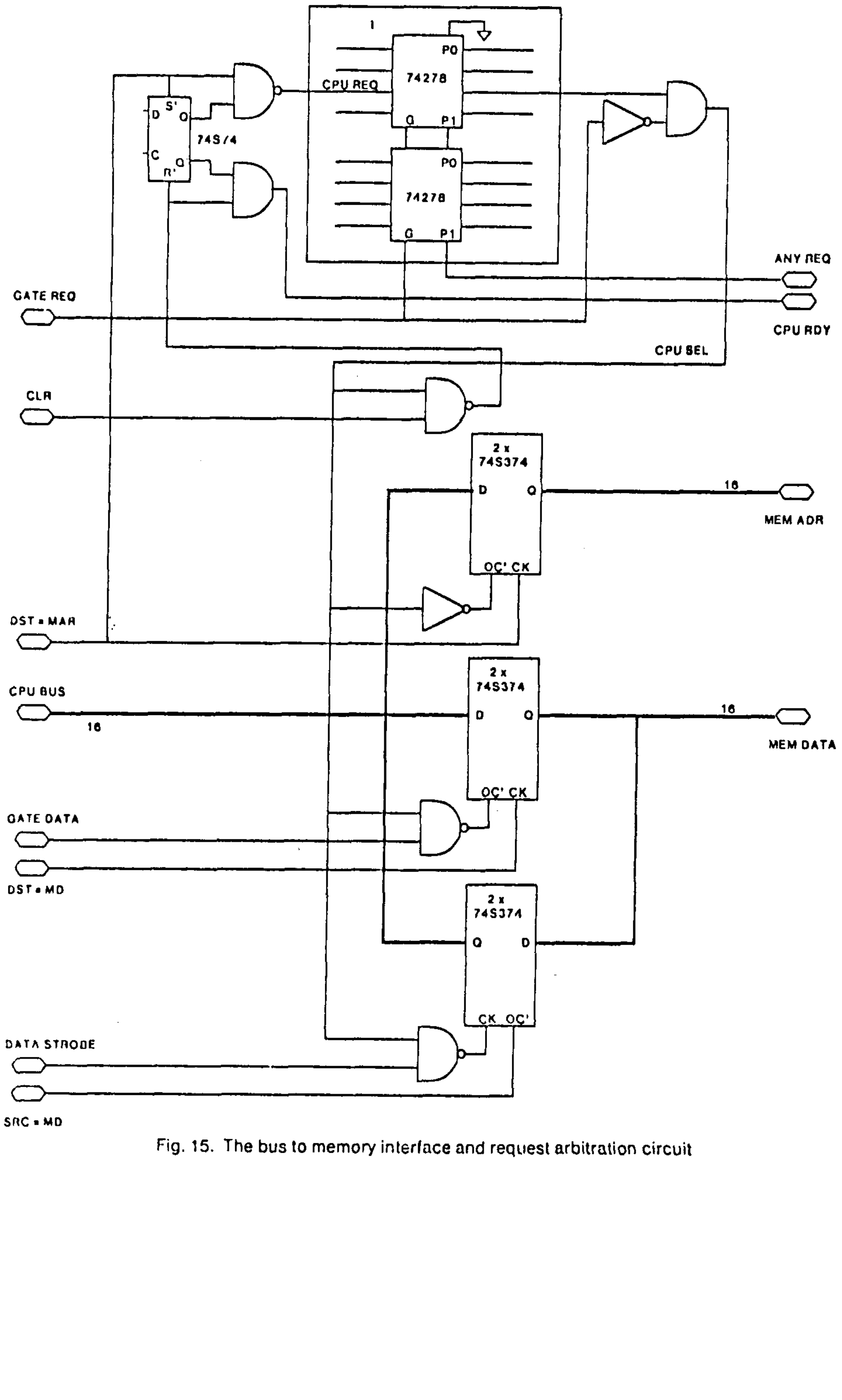


Fig. 15. The bus to memory interface and request arbitration circuit

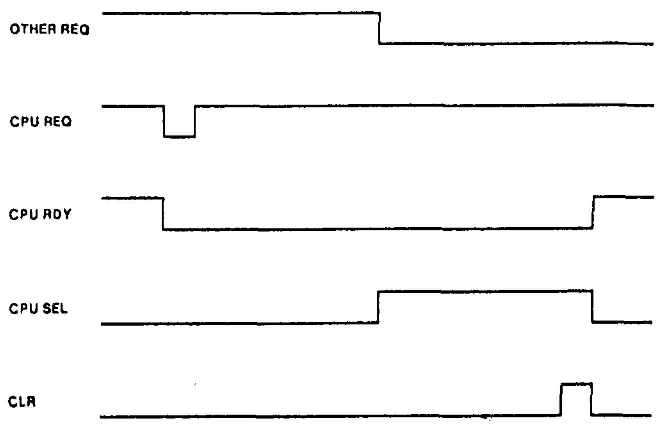


Fig. 16. Memory interface signal timing

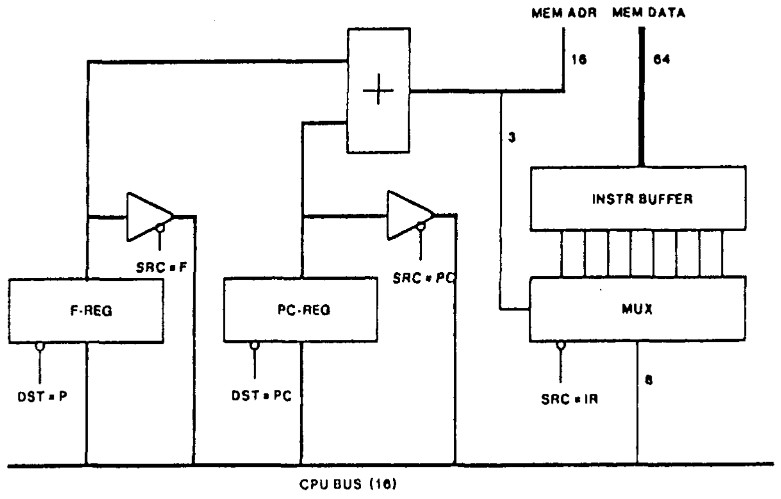


Fig. 17. The instruction fetch unit

Page 48 missing

Page 49 missing

Appendix 1

The M-code interpreter

The following Modula-2 program interprets M-code instructions and serves as a high-level definition of the Lilith computer's instruction set and architecture. A few comments are necessary to cover details that are not fully described by the program.

1. The array variables stk and code stand for the data and program stores respectively. We assume that on an actual computer they represent the SAME physical memory. The array indices then denote memory addresses. Access to the code involves the use of the base address F (and an 18-bit wide addition).

2. All checks against arithmetic overflow, storage overflow, and access with value NIL are omitted from the program in the interest of clarity and in order not to obscure the essentials of the interpretation algorithm.

3. Certain instructions are explained in loose English instead of precise Modula statements. Among them are the bitmap handling instructions, which actually constitute fairly complex algorithms, and also operations like shifts, packing, and unpacking, which are considered as primitives, and hence not to be defined contortiously in terms of even lower primitives.

4. The functions low(d), high(d), and pair(a,b) are introduced to denote selection of a part of a double word and construction of a double word. The functions Dtrunc and Dfloat denote conversion of floating-point values into double word integers and vice-versa. All these functions are NOT available in Modula-2. Also, sets of the form {m..n} are used, although proper Modula-2 does not allow expressions to be used within set constructors.

5. The detailed specification of I/O instructions is suppressed. It is considered not to be part of the general M-code definition, but should be allowed to vary among different implementations according to the available hardware. This is particularly true for the instructions DSKR, DSKW, SETRK used for accessing the disk.

6. The interrupt mechanism is described in a rather loose manner and requires additional explanation: At the start of each interpretation cycle, the Boolean variable REQ determines whether or not an interrupt request should be honoured. REQ means "at least one of the unmasked interrupt lines (numbered 8...15) is low". If we denote the request lines by the set variable ReqLines and the presence of a request on line i by

"NOT (i IN ReqLines)", then REQ can be expressed as

REQ = (ReqLines + Mask # {8 .. 15})

The value Mask is the union of the mask register M and a variable called DevMask (Mask = M + DevMask). This global variable allows a program (typically the operating system) to shut out any (or all) devices from interrupting. In the Lilith computer, DevMask is allocated in main memory at location 3. The value ReqNo determines the interrupt line whose request is being accepted. It determines the transfer vector used by the TRANSFER operation. The value ReqNo is defined as the maximum i such that "NOT (i IN ReqLines + Mask).

Table of instructions

	0	40	100	140	200	240	300	340
0	LI0	LLW	LGW	LSW0	LSW	READ	FOR1	MOV
1	LI1	LLD	LGD	LSW1	LSD	WRITE	FOR2	CMP
2	LI2	LEW	LGW2	LSW2	LSD0	DSKR	ENTC	DDT
3	LI3	LED	LGW3	LSW3	LXFW	DSKW	EXC	REPL
4	LI4	LLW4	LGW4	LSW4	LSTA	SETRK	TRAP	BBLT
5	LI5	LLW5	LGW5	LSW5	LXB	UCHK	CHK	DCH
6	LI6	LLW6	LGW6	LSW6	LXW		CHKZ	UNPK
7	LI7	LLW7	LGW7	LSW7	LXD	SYS	CHKS	PACK
10	LI8	LLW8	LGW8	LSW8	DADD	ENTP	EQL	GB
11	LI9	LLW9	LGW9	LSW9	DSUB	EXP	NEQ	GB1
12	LI10	LLW10	LGW10	LSW10	DMUL	ULSS	LSS	ALOC
13	LI11	LLW11	LGW11	LSW11	DDIV	ULEQ	LEQ	ENTR
14	LI12	LLW12	LGW12	LSW12		UGTR	GTR	RTN
15	LI13	LLW13	LGW13	LSW13		UGEQ	GEQ	CX
16	LI14	LLW14	LGW14	LSW14	DSHL	TRA	ABS	CI
17	LI15	LLW15	LGW15	LSW15	DSHR	RDS	NEG	CF
20	LIB	SLW	SGW	SSW0	SSW	LODFW	OR	CL
21		SLD	SGD	SSW1	SSD	LODFD	XOR	CL1
22	LIW	SEW	SGW2	SSW2	SSD0	STORE	AND	CL2
23	LID	SED	SGW3	SSW3	SXFW	STOFV	COM	CL3
24	LLA	SLW4	SGW4	SSW4	TS	STOT	IN	CL4
25	LGA	SLW5	SGW5	SSW5	SXB	COPT	LIN	CL5
26	LSA	SLW6	SGW6	SSW6	SXW	DECS	MSK	CL6
27	LEA	SLW7	SGW7	SSW7	SXD	PCOP	NOT	CL7
30	JPC	SLW8	SGW8	SSW8	FADD	UADD	ADD	CL8
31	JP	SLW9	SGW9	SSW9	FSUB	USUB	SUB	CL9
32	JPFC	SLW10	SGW10	SSW10	FMUL	UMUL	MUL	CL11
33	JPF	SLW11	SGW11	SSW11	FDIV	UDIV	DIV	CL10
34	JPBC	SLW12	SGW12	SSW12	FCMP	UMOD		CL12
35	JPB	SLW13	SGW13	SSW13	FABS	ROR	BIT	CL13

```
36   ORJP   SLW14   SGW14   SSW14   FNEG    SHL     NOP     CL14
37   ANDJP  SLW15   SGW15   SSW15   FFCT    SHR     MOVF    CL15
```

Reserved locations:

```
   0   (F-register of module 0)
   1   (initialization flag of module 0)
   2   (string pointer of module 0)
   3   device mask
   4   P-register
   5   saved P-register
   6   boot flag
  16,17   trap vector
  20,21   interrupt vector for line 8 (clock)
  22,23   interrupt vector for line 9 (disk)
  ...
  36,37   interrupt vector for line 15
  40..177 data frame table
```

```
MODULE Interpreter;  (*N.Wirth, Ch.Jacobi; Feb.81*)

  CONST tlc      = 16B;             (*trap location adr*)
      dft        = 40B;             (*data frame table adr*)

  VAR  (*global state variables*)
      PC:  CARDINAL;                (*program counter*)
      IR:  CARDINAL;                (*instruction register*)
      F:   CARDINAL;                (*code frame base address*)
      G:   CARDINAL;                (*data frame base address*)
      H:   CARDINAL;                (*stack limit address*)
      L:   CARDINAL;                (*local segment address*)
      S:   CARDINAL;                (*stack pointer*)
      P:   CARDINAL;                (*process base address*)
      M:   BITSET;                  (*process interrupt mask*)
      REQ: BOOLEAN;                 (*interrupt request*)
      ReqNo: CARDINAL;              (*request number, 8..15*)

    (*auxiliary variables used over single instructions only*)
      i, j, k: CARDINAL;
      sz, adr, low, hi: CARDINAL; (*used in FOR, ENTP, PCOP*)
      sb, db, sbmd, dbmd, fo: CARDINAL;  (*display handling*)
      x, y: REAL;

      stk:  ARRAY [0..177777B] OF CARDINAL;    (*data store*)

  MODULE InstructionFetch;
    IMPORT F,PC;
    EXPORT next, next2;

    VAR code: ARRAY [0..77777B] OF [0..255];
  PROCEDURE next(): CARDINAL;
  BEGIN
    INC(PC); RETURN code[4*F+PC-1]
  END next;

  PROCEDURE next2(): CARDINAL; (*get next two code bytes*)
  BEGIN
     INC(PC, 2); RETURN code[4*F+PC-2]*400B + code[4*F+PC-1]
  END next2;
END InstructionFetch;

MODULE ExpressionStack;
  EXPORT push, pop, Dpush, Dpop, empty;

  VAR sp: CARDINAL;
      a: ARRAY [0..15] OF CARDINAL;   (*expression stack*)

  PROCEDURE push(x: CARDINAL);
  BEGIN a[sp] := x; INC(sp)
  END push;

  PROCEDURE pop(): CARDINAL;
  BEGIN DEC(sp); RETURN(a[sp])
  END pop;

  PROCEDURE Dpush(d: REAL);
  BEGIN a[sp] := high(d); INC(sp); a[sp] := low(d); INC(sp)
  END Dpush;

  PROCEDURE Dpop(): REAL;
  BEGIN DEC(sp,2); RETURN pair(a[sp], a[sp+1])
  END Dpop;

  PROCEDURE empty():BOOLEAN;
  BEGIN RETURN sp = 0
  END empty;

BEGIN sp := 0;
END ExpressionStack;

PROCEDURE mark(x: CARDINAL; external: BOOLEAN);
  VAR i: CARDINAL;
BEGIN i := S;
  stk[S] := x; INC(S);  (*static link*)
  stk[S] := L; INC(S);  (*dynamic link*)
  IF external THEN
    stk[S] := PC+100000B ELSE stk[S] := PC
  END ;
  INC(S,2); L := i
END mark;

PROCEDURE saveExpStack;
```

```
      VAR c: CARDINAL;
    BEGIN c := 0; (*expression stack counter*)
      WHILE NOT empty() DO
        stk[S] := pop(); INC(S); INC(c);
      END ;
      stk[S] := c; INC(S)
    END saveExpStack;

    PROCEDURE restoreExpStack;
      VAR c: CARDINAL;
    BEGIN DEC(S); c := stk[S];
      WHILE c>0 DO
        DEC(c); DEC(S); push(stk[S])
      END
    END restoreExpStack;

    PROCEDURE saveRegs;
    BEGIN saveExpStack;
      stk[P  ] := G;  stk[P+1] := L;
      stk[P+2] := PC; stk[P+3] := CARDINAL(M);
      stk[P+4] := S;  stk[P+5] := H+24;
      (* stk[P+6] is reserved for error code *)
      (* stk[P+7] is reserved for error trap mask *)
    END saveRegs;

    PROCEDURE restoreRegs(changeMask: BOOLEAN);
    BEGIN
      G := stk[P];    F := stk[G];
      L := stk[P+1]; PC := stk[P+2];
      IF changeMask THEN M := BITSET(stk[P+3]) END ;
      S := stk[P+4]; H := stk[P+5]-24;
      restoreExpStack
    END restoreRegs;

    PROCEDURE Transfer(changeMask: BOOLEAN; to, from: CARDINAL);
      VAR j: CARDINAL;
    BEGIN
      j := stk[to]; saveRegs; stk[from] := P;
      P := j; restoreRegs(changeMask)
    END Transfer;

    PROCEDURE Trap(n: CARDINAL);
    BEGIN
      IF NOT (n IN BITSET(stk[P+7])) THEN
        stk[P+6] := n;
        Transfer(TRUE, tlc, tlc+1)
      END
    END Trap;

  BEGIN (* readBootFile *)
    P := stk[4]; restoreRegs(TRUE);
    LOOP
      IF REQ THEN Transfer(TRUE, 2*ReqNo, 2*ReqNo+1) END ;
```

```
IR := next();
CASE IR OF
 ØB .. 17B: (*LIØ - LI15 load immediate*)  push(IR MOD 16) |

20B: (*LIB  load immediate byte*)  push(next()) |

22B: (*LIW  load immediate word*)  push(next2()) |

23B: (*LID  load immediate double word*)
     push(next2()); push(next2()) |

24B: (*LLA  load local address*)   push(L+next()) |

25B: (*LGA  load global address*)  push(G+next()) |

26B: (*LSA  load stack address*)   push(pop()+next()) |

27B: (*LEA  load external address*)
     push(stk[dft+next()]+next()) |

30B: (*JPC  jump conditional*)
     IF pop() = Ø THEN PC := PC + next2()
                  ELSE INC(PC,2)
     END  |

31B: (*JP   jump*)  PC := PC + next2() |

32B: (*JPFC  jump forward conditional*)
     IF pop() = Ø THEN PC := PC + next() ELSE INC(PC) END  |

33B: (*JPF  jump forward*)   PC := PC + next()  |

34B: (*JPBC  jump backward conditional*)
     IF pop() = Ø THEN PC := PC - next() ELSE INC(PC) END  |

35B: (*JPB  jump backward*)  PC := PC - next()  |

36B: (*ORJP  short circuit OR *)
     IF pop() = Ø THEN INC(PC)
                  ELSE push(1); PC := PC+next()
     END |

37B: (*ANDJP  short circuit AND *)
     IF pop() = Ø THEN push(Ø); PC := PC+next()
                  ELSE INC(PC)
     END |

4ØB: (*LLW  load local word*)     push(stk[L+next()])  |

41B: (*LLD  load local double word*)
     i := L+next(); push(stk[i]); push(stk[i+1])  |

42B: (*LEW  load external word*)
```

```
      push(stk[stk[dft+next()]+next()]) |

 43B: (*LED  load external double word *)
      i := stk[dft+next()]+next();
      push(stk[i]); push(stk[i+1]) |

 44B .. 57B: (*LLW4-LLW15*)   push(stk[L + (IR MOD 16)]) |

 60B: (*SLW  store local word*)    stk[L+next()] := pop() |

 61B: (*SLD  store local double word*)
      i := L+next(); stk[i+1] := pop(); stk[i] := pop()  |

 62B: (*SEW  store external word*)
      stk[stk[dft+next()]+next()] := pop() |

 63B: (*SED  store external double word *)
      i := stk[dft+next()]+next();
      stk[i+1] := pop(); stk[i] := pop() |

 64B .. 77B: (*SLW4-SLW15  store local word*)
      stk[L+(IR MOD 16)] := pop()  |

100B: (*LGW  load global word*)    push(stk[G+next()]) |

101B: (*LGD  load global double word*)
      i := next()+G; push(stk[i]); push(stk[i+1]) |

102B .. 117B: (*LGW2 - LGW15  load global word*)
      push(stk[G + (IR MOD 16)]) |

120B: (*SGW  store global word*)   stk[G+next()] := pop() |

121B: (*SGD  store global double word*)
      i := G+next(); stk[i+1] := pop(); stk[i] := pop()  |

122B .. 137B: (*SGW2 - SGW15  store global word*)
      stk[G + (IR MOD 16)] := pop() |

140B .. 157B: (*LSW0 - LSW15  load stack addressed word*)
      push(stk[pop()+(IR MOD 16)]) |

160B .. 177B:  (*SSW0 - SSW15  store stack-addressed word*)
      k := pop(); i := pop()+(IR MOD 16); stk[i] := k |

200B: (*LSW  load stack word*)
      i := pop() + next(); push(stk[i]) |

201B: (*LSD  load stack double word*)
      i := pop() + next(); push(stk[i]); push(stk[i+1]) |

203B: (*LXFW  load indexed frame word*)
      k := pop() + pop()*4; push(stk[k]) |
202B: (*LSD0  load stack double word*)
      i := pop(); push(stk[i]); push(stk[i+1]) |

204B: (*LSTA  load string address *)  push(stk[G+2]+next()) |

205B: (*LXB  load indexed byte*)
      i := pop(); j := pop(); k := stk[j + (i DIV 2)];
      IF i MOD 2 = 0 THEN push(k DIV 400B)
                     ELSE push(k MOD 400B)
      END |

206B: (*LXW  load indexed word*)
      i := pop()+pop(); push(stk[i]) |

207B: (*LXD  load indexed double word *)
      i := 2*pop()+pop(); push(stk[i]); push(stk[i+1])  |

210B: (*DADD  double add.  Subsequent operators for double
        words denote unsigned fixed-point arithmetic,
        although the program shows REAL operands*)
      y := Dpop(); x := Dpop(); Dpush(x+y) |

211B: (*DSUB  double subtract*)
      y := Dpop(); x := Dpop(); Dpush(x-y) |

212B: (*DMUL  double multiply*)
      j := pop(); i := pop(); (* x := i*j *) Dpush(x) |

213B: (*DDIV  double divide*)
      j := pop(); x := Dpop();
      (* k := x DIV j; i := x MOD j *) push(i); push(k) |

216B: (*DSHL  double shift left*)
      x := Dpop(); (*shift x left 1 bit*) Dpush(x) |

217B: (*DSHR  double shift right*)
      x := Dpop(); (*shift x right 1 bit*) Dpush(x) |

220B: (*SSW  store stack word*)
      k := pop(); i := pop()+next(); stk[i] := k |

221B: (*SSD  store stack double word*)
      k := pop(); j := pop(); i := pop()+next();
      stk[i] := j; stk[i+1] := k |

222B: (*SSD0  store stack double word*)
      k := pop(); j := pop(); i := pop();
      stk[i] := j; stk[i+1] := k |

223B: (*SXFW  store indexed frame word*)
      i := pop(); k := pop() + pop()*4; stk[k] := i |
224B: (*TS  test and set*)
      i := pop(); push(stk[i]); stk[i] := 1 |

225B: (*SXB  store indxed byte*)
      k := pop(); i := pop(); j := pop() + (i DIV 2);
      IF i MOD 2 = 0 THEN
           stk[j] := k*400B + (stk[j] MOD 400B)
      ELSE stk[j] := (stk[j] DIV 400B) * 400B + k
      END |

226B: (*SXW  store indexed word*)
      k := pop(); i := pop()+pop(); stk[i] := k |

227B: (*SXD  store indexed double word*)
      k := pop(); j := pop(); i := 2*pop()+pop();
      stk[i] := j; stk[i+1] := k |

230B: (*FADD  floating add*)
      y := Dpop(); x := Dpop(); Dpush(x+y) |

231B: (*FSUB  floating subtract*)
      y := Dpop(); x := Dpop(); Dpush(x-y) |

232B: (*FMUL  floating multiply*)
      y := Dpop(); x := Dpop(); Dpush(x*y) |

233B: (*FDIV  floating divide*)
      y := Dpop(); x := Dpop(); Dpush(x/y) |

234B: (*FCMP  floating compare*)
      x := Dpop(); y := Dpop();
      IF x > y THEN push(0); push(1)
        ELSIF x < y THEN push(1); push(0)
        ELSE push(0); push(0)
      END |

235B: (*FABS  floating absolute value*) Dpush(ABS(Dpop())) |

236B: (*FNEG  floating negative*)  Dpush(-Dpop()) |

237B: (*FFCT  floating functions*)  i := next();
      IF i=0 THEN Dpush(FLOAT(pop()))
        ELSIF i=1 THEN Dpush(DFloat(Dpop()))
        ELSIF i=2 THEN push(TRUNC(Dpop()))
        ELSIF i=3 THEN Dpush(Dtrunc(Dpop(), pop()))
      END |

240B: (*READ*)    i := pop(); k := pop();
                  (* stk[i] := input from channel k *) |

241B: (*WRITE*)   i := pop(); k := pop();
                  (* output i to channel k *) |
242B: (*DSKR  disk read*) |

243B: (*DSKW  disk write*) |

244B: (*SETRK  set disk track*) |

245B: (*UCHK*) k := pop(); j := pop(); i := pop(); push(i);
      IF (i < j) OR (i > k) THEN Trap(4) END |

247B: (*SYS  rarely used system functions*) |

250B: (*ENTP  entry priority*)
      stk[L+3] := CARDINAL(M); M := {0..next()-1} |

251B: (*EXP  exit priority*)   M := BITSET(stk[L+3]) |

252B: (*ULSS*) j := pop(); i := pop();
      IF i < j THEN push(1) ELSE push(0) END  |

253B: (*ULEQ*) j := pop(); i := pop();
      IF i <= j THEN push(1) ELSE push(0) END |

254B: (*UGTR*) j := pop(); i := pop();
      IF i > j THEN push(1) ELSE push(0) END  |

255B: (*UGEQ*) j := pop(); i := pop();
      IF i >= j THEN push(1) ELSE push(0) END |

256B: (*TRA  coroutine transfer*)
      Transfer(BOOLEAN(next()), pop(), pop()) |

257B: (*RDS  read string*)   k := pop(); i := next();
      REPEAT
        stk[k] := next2(); INC(k); DEC(i)
      UNTIL i < 0  |

260B: (*LODFW  reload stack after function return*)
      i := pop(); restoreExpStack; push(i) |

261B: (*LODFD  reload stack after function return*)
      i := pop(); j := pop(); restoreExpStack;
      push(j); push(i) |

262B: (*STORE*)  saveExpStack |

263B: (*STOFV  store stack with formal procedure on top*)
      i := pop(); saveExpStack; stk[S] := i; INC(S) |

264B: (*STOT  copy from stack to procedure stack*)
      stk[S] := pop(); INC(S) |

265B: (*COPT  copy element on top of expression stack*)
      i := pop(); push(i); push(i) |
266B: (*DECS  decrement stackpointer*)  DEC(S) |

267B: (*PCOP  allocation and copy of value parameter *)
      stk[L+next()] := S;
      sz := pop(); k := S+sz; adr := pop();
      WHILE sz>0 DO
          stk[S] := stk[adr]; INC(S); INC(adr); DEC(sz)
      END |

270B: (*UADD*)   j := pop(); i := pop(); push(i+j) |

271B: (*USUB*)   j := pop(); i := pop(); push(i-j) |

272B: (*UMUL*)   j := pop(); i := pop(); push(i*j) |

273B: (*UDIV*)   j := pop(); i := pop(); push(i DIV j) |

274B: (*UMOD*)   j := pop(); i := pop(); push(i MOD j) |

275B: (*ROR*)    j := pop(); i := pop() MOD 16;
      (* k := j rightrotated by i places *) push(k) |

276B: (*SHL*)    j := pop(); i := pop() MOD 16;
      (* k := j left shifted by i places *) push(k) |

277B: (*SHR*)    j := pop(); i := pop() MOD 16;
      (* k := j right shifted by i places*) push(k) |

300B: (*FOR1  enter FOR statement *)
      i := next(); (* =0: up; >0: down *)
      hi := pop(); low := pop(); adr := pop();
      k := PC + next2();
      IF ((i = 0)  AND (low <= hi)) OR
         ((i # 0)  AND (low >= hi)) THEN
        stk[adr] := low;
        stk[S] := adr; INC(S); stk[S] := hi; INC(S)
      ELSE (* don't execute the FOR loop *)
        PC := k
      END |

301B: (*FOR2  exit FOR statement *)
      hi := stk[S-1]; adr := stk[S-2];
      sz := INTEGER(next()); (* step range -128..+127 *)
      k := PC + next2(); i := stk[adr]+sz;
      IF ((sz >= 0) AND (i > hi))
        OR ((sz <= 0) AND (i < hi))
      THEN (* terminate *) DEC(S,2)
      ELSE (* continue *) stk[adr] := i; PC := k
      END |

302B: (*ENTC  enter CASE statement*)
      PC := PC + next2(); k := pop();
      low := next2(); hi := next2();
      stk[S] := PC + 2*(hi-low) + 4; INC(S);
      IF (k >= low) AND (k <= hi) THEN
          PC := PC + 2*(k-low+1)
      END;
      PC := PC + next2() |

303B: (*EXC  exit CASE statement*) DEC(S); PC := stk[S] |

304B: (*TRAP*)  i := pop(); Trap(i) |

305B: (*CHK*) k := pop(); j := pop(); i := pop(); push(i);
      IF (INTEGER(i) < INTEGER(j)) OR
         (INTEGER(i) > INTEGER(k)) THEN Trap(4) END  |

306B: (*CHKZ*)
      k := pop(); i := pop(); push(i);
      IF i>k THEN Trap(4) END  |

307B: (*CHKS  check sign bit*)
      k := pop(); push(k);
      IF INTEGER(k) < 0  THEN Trap(4) END |

310B: (*EQL*)  j := pop(); i := pop();
      IF i = j THEN push(1) ELSE push(0) END  |

311B: (*NEQ*)  j := pop(); i := pop();
      IF i # j THEN push(1) ELSE push(0) END  |

312B: (*LSS*) j := pop(); i := pop();
      IF INTEGER(i) < INTEGER(j) THEN
        push(1) ELSE push(0)
      END  |

313B: (*LEQ*) j := pop(); i := pop();
      IF INTEGER(i) <= INTEGER(j) THEN
        push(1) ELSE push(0)
      END |

314B: (*GTR*) j := pop(); i := pop();
      IF INTEGER(i) > INTEGER(j) THEN
        push(1) ELSE push(0)
      END  |

315B: (*GEQ*) j := pop(); i := pop();
      IF INTEGER(i) >= INTEGER(j) THEN
        push(1) ELSE push(0)
      END |

316B: (*ABS*)  push(ABS(INTEGER(pop())))  |

317B: (*NEG*)  push(-INTEGER(pop()))  |
320B: (*OR*)    j := pop(); i := pop();
                push(CARDINAL(BITSET(i)+BITSET(j))) |

321B: (*XOR*)   j := pop(); i := pop();
                push(CARDINAL(BITSET(i)/BITSET(j))) |

322B: (*AND*)   j := pop(); i := pop();
                push(CARDINAL(BITSET(i)*BITSET(j))) |

323B: (*COM*)   push(CARDINAL({0..15}/BITSET(pop()))) |

324B: (*IN*)    j := pop(); i := pop();
                IF i > 15 THEN push(0)
                  ELSIF i IN BITSET(j) THEN push(1)
                  ELSE push(0)
                END |

325B: (*LIN  load immediate NIL*)   push(177777B) |

326B: (*MSK*)   j := pop() MOD 16; push(CARDINAL({0..k-1})) |

327B: (*NOT*)   i := pop();  push(CARDINAL({15}/{j})) |

330B: (*ADD*)   j := pop(); i := pop();
                push(CARDINAL(INTEGER(i) + INTEGER(j))) |

331B: (*SUB*)   j := pop(); i := pop();
                push(CARDINAL(INTEGER(i) - INTEGER(j))) |

332B: (*MUL*)   j := pop(); i := pop();
                push(CARDINAL(INTEGER(i) * INTEGER(j))) |

333B: (*DIV*)   j := pop(); i := pop();
                push(CARDINAL(INTEGER(i) DIV INTEGER(j))) |

334B: (*MOD*)   j := pop(); i := pop();
                push(CARDINAL(INTEGER(i) MOD INTEGER(j))) |

335B: (*BIT*)   j := pop() MOD 16; (* k := {j} *) push(k) |

336B: (*NOP*) |

337B: (*MOVF  move frame *) i := pop();
      j := pop()+pop()*4; (*18 bits*)
      k := pop()+pop()*4; (*18 bits*)
      WHILE i>0 DO
        stk[k] := stk[j]; INC(k); INC(j); DEC(i)
      END |

340B: (*MOV  move block*)
      k := pop(); j := pop(); i := pop();
      WHILE k>0 DO
        stk[i] := stk[j]; INC(i); INC(j); DEC(k)
```

```
       END |

341B: (*CMP  compare blocks*)
       k := pop(); j := pop(); i := pop();
       IF k=0 THEN push(0); push(0)
       ELSE
         WHILE(stk[i] # stk[j]) AND (k > 0) DO
           INC(i); INC(j); DEC(k)
         END;
         push(stk[i]); push(stk[j])
       END |

342B: (*DDT  display dot*)
       k := pop(); j := pop(); dbmd := pop(); i := pop()
       (* display point at <j,k> in mode i inside
          bitmap dbmd *) |

343B: (*REPL  replicate pattern *)
       db := pop(); sb := pop(); dbmd := pop(); i := pop()
       (* replicate pattern sb over block db inside
          bitmap dbmd in mode i *) |

344B: (*BBLT  bit block transfer*)
       sbmd := pop(); db := pop(); sb := pop();
       dbmd := pop(); i := pop()
       (* transfer block sb in bitmap sbmd to block db
          inside bitmap dbmd in mode i *) |

345B: (*DCH  display character*)
       j := pop(); db := pop(); fo := pop(); dbmd := pop()
       (* copy bit pattern for character j from font fo
          to block db inside bitmap dbmd *) |

346B: (*UNPK  unpack*) k := pop(); j := pop(); i := pop();
       (*extract bits i..j from k, then right adjust*)
       push(k) |

347B: (*PACK  pack*)
       k := pop(); j := pop(); i := pop(); adr := pop();
       (*pack the rightmost j-i+1 bits of k into positions
         i..j of word stk[adr] *) |

350B: (*GB  get base adr n levels down*)
       i := L; j := next();
       REPEAT
         i := stk[i]; DEC(j)
       UNTIL j=0;
       push(i) |

351B: (*GB1  get base adr 1 level down*)  push(stk[L]) |

352B: (*ALLOC  allocate block*)
       i := pop(); push(S); S := S + i;
```

```
        IF S > H THEN S := pop(); Trap(3) END |

    353B: (*ENTR  enter procedure*)
        i := next();  S := S+i;
        IF S > H THEN S := S-i; Trap(3) END |

    354B: (*RTN  return from procedure*)
        S := L; L := stk[S+1]; i := stk[S+2];
        IF i < 100000B THEN PC := i
          ELSE G := stk[S]; F := stk[G]; PC := i - 100000B
        END |

    355B: (*CX  call external procedure*)
        j := next(); i := next();
        mark(G, TRUE); G := stk[dft+j];
        F := stk[G]; PC := 2*i; PC := next2()  |

    356B: (*CI  call procedure at intermediate level*)
        i := next(); mark(pop(), FALSE);
        PC := 2*i; PC := next2()  |

    357B: (*CF  call formal procedure*)
        i := stk[S-1]; mark(G, TRUE);
        j := i DIV 400B; G := stk[dft+j];
        F := stk[G]; PC := 2*(i MOD 400B); PC := next2()  |

    360B: (*CL  call local procedure*)
        i := next(); mark(L, FALSE);
        PC := 2*i; PC := next2()  |

    361B .. 377B:  (*CL1 - CL15  call local procedure*)
        mark(L, FALSE); PC := 2*(IR MOD 16); PC := next2()
    END
  END (*LOOP*)
END Interpreter.
```

Appendix 2

A benchmark test for Modula-2 implementations

In order to provide a basis for measuring and comparing the efficiency of implementations of the language Modula-2, a benchmark program is proposed. It measures selectively various specific language features. Instead of relying on a built-in timing mechanism (which depends on an underlying operating system and quite likely impedes the program's portability), the program merely counts the number of times certain statements are executed. Computation is monitored and interrupted by the human operator equipped with a stop watch. Each test is selected by typing its identifying character (a - o); the end of the test is signalled by typing any character. Further details are to be derived from the program listing.

The following figures have been measured for the Lilith, the PDP-11/40, and the Xerox Alto 2 computers. (On the Alto, the program was translated into Mesa). The timing period is 1 minute for each test. Implementors of Modula-2 are encouraged to apply this test fully or partially to their system and to let us know their results.

	facility	Lilith	PDP-11/40	Alto 2
a	empty REPEAT loop	321	184	
b	empty WHILE loop	334	185	116
c	empty FOR loop	422	230	172
d	CARDINAL arithmetic	187	84	54
e	REAL arithmetic	130		
f	sin, exp, ln, sqrt	87		
g	array access	109	54	32
h	same with bounds tests	89	11	26
i	matrix access	197	93	44
j	same with bounds tests	164	21	36
k	call of empty procedure	144	37	40
l	with 4 parameters	94	29	32
m	copying arrays	63	11	56
n	access via pointers	125	66	54
o	reading a disk stream	207		36

```
MODULE Benchmark;
  (*$T-
    a: empty REPEAT loop
    b: empty WHILE loop
    c: empty FOR loop
    d: CARDINAL arithmetic
    e: REAL arithmetic
    f: standard functions
    g: array of single dimension
    h: same as g but with index tests
    i: matrix access
    j: same as i but with index tests
    k: call of empty, parameterless procedure
    l: call of empty procedure with 4 parameters
    m: copying arrays (block moves)
    n: pointer chaining
    o: reading of file  *)

  FROM Storage IMPORT ALLOCATE;
  FROM Terminal IMPORT Read, BusyRead, Write, WriteLn;
  FROM InOut IMPORT WriteCard;
  FROM FileSystem IMPORT
       File, Lookup, ReadWord, Reset, Response;
  FROM MathLib0 IMPORT sin, exp, ln, sqrt;

  TYPE NodePtr = POINTER TO Node;
      Node = RECORD x,y: CARDINAL; next: NodePtr END ;

  VAR A,B,C: ARRAY [0..255] OF CARDINAL;
      M: ARRAY [0..99],[0..99] OF CARDINAL;
      m: CARDINAL; head: NodePtr;

  PROCEDURE Test(ch: CHAR);
    VAR i,j,k: CARDINAL;
        r0, r1, r2: REAL; p: NodePtr;

    PROCEDURE P;
    BEGIN
    END P;

    PROCEDURE Q(x,y,z,w: CARDINAL);
    BEGIN
    END Q;

  BEGIN
    CASE ch OF
      "a": k := 20000;
           REPEAT
             k := k-1
           UNTIL k = 0 |

      "b": i := 20000;
           WHILE i > 0 DO
```

```
          i := i-1
        END |

"c":  FOR i := 1 TO 20000 DO
      END |

"d":  j := 0; k := 10000;
      REPEAT
        k := k-1;  j := j+1; i := (k*3) DIV (j*5)
      UNTIL k = 0 |

"e":  k := 5000; r1 := 7.28; r2 := 34.8;
      REPEAT
        k := k-1; r0 := (r1*r2) / (r1+r2)
      UNTIL k = 0  |

"f":  k := 500;
      REPEAT r0 := sin(0.7); r1 := exp(2.0);
             r0 := ln(10.0); r1 := sqrt(18.0); k := k-1
      UNTIL k = 0 |

"g":  k := 20000; i := 0; B[0] := 73;
      REPEAT
        A[i] := B[i]; B[i] := A[i]; k := k-1
      UNTIL k = 0 |

"h":  (*$T+*) k := 20000; i := 0; B[0] := 73;
      REPEAT
        A[i] := B[i]; B[i] := A[i]; k := k-1
      UNTIL k=0  (*$T-*)   |

"i":  FOR i := 0 TO 99 DO
        FOR j := 0 TO 99 DO
          M[i,j] := M[j,i]
        END
      END |

"j":  (*$T+*)
      FOR i := 0 TO 99 DO
        FOR j := 0 TO 99 DO
          M[i,j] := M[j,i]
        END
      END (*$T-*) |

"k":  k := 20000;
      REPEAT
        P; k := k-1
      UNTIL k = 0  |

"l":  k := 20000;
      REPEAT
        Q(i,j,k,m); k := k-1
      UNTIL k = 0  |
      "m": k := 500;
           REPEAT
             k := k-1; A := B; B := C; C := A
           UNTIL k = 0  |

      "n": k := 500;
           REPEAT p := head;
             REPEAT p := p^.next UNTIL p = NIL;
             k := k-1
           UNTIL k = 0 |

      "o": k := 5000;
           REPEAT
             k := k-1; ReadWord(f,i)
           UNTIL k = 0;
           Reset(f)
    END
  END Test;

  VAR ch,ch1: CHAR;
      n: CARDINAL;
      f: File;
      q: NodePtr;

BEGIN Lookup(f,"anyFile", FALSE);
  head := NIL; n := 100;
  REPEAT q := head;
    NEW(head); head^.next := q; n := n-1
  UNTIL n = 0;
  Write(">"); Read(ch);
  WHILE ("a" <= ch) & (ch < "p") DO
    Write(ch); WriteLn; n := 0;
    REPEAT n := n+1; Test(ch);
      IF (n MOD 50) = 0 THEN WriteLn END ;
      Write("."); BusyRead(ch1)
    UNTIL ch1 # 0C;
    WriteCard(n,6); WriteLn; Write(">"); Read(ch)
  END ;
  Write(14C)
END Benchmark.
```