Diss. ETH No 7195

Code Generation and the Lilith Architecture

Dissertation

submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by

Christian Jacobi

Dipl. Math. of the Swiss Federal Institute of Technology
born August 10, 1951
Citizen of Zürich (Canton of Zürich)

Accepted on the recommendation of
Prof. Dr. N. Wirth
Dr. P. Schulthess

1982

Abstract

The Lilith computer is particularly suited to execute Modula-2 programs. The instruction set is chosen to reflect the needs of the compiler. Compilations, therefore, are easy and produce dense and fast code. There are special purpose high-level instructions as well as quite primitive operations.

The Lilith computer uses a stack for the evaluation of expressions. These evaluations are done with the use of a small hardware *expression* stack. The hardware expression stack permits arithmetic operations without memory access. It allows to combine the simplicity of a stack computer with the efficiency of a computer with general registers. The compiler guarantees that the expression stack never overflows. Lilith is also a stack computer in another sense; it allocates data segments of procedures on a stack located in main memory.

The memory is subdivided into several areas. Local data, global data, and code are accessed relative to specific registers. These registers are updated on execution of a call or return instruction. Data of external modules are accessed indirectly through pointers in the module table. Relative addressing allows usage of short offsets. Over 95% of all *load* instructions need an offset of less than 12. For these instructions the offset is directly encoded in the instruction byte.

Some language constructs suggest the provision of corresponding machine instructions. The CASE and the FOR statements suggest corresponding *case* and *for* instructions. Most instructions do not correspond to Modula structures in such a simple manner. However, the influence of Modula in defining the overall behaviour of otherwise general instructions is pervasive.

In structured programming languages short-distance jumps occur much more frequently than long-distance jumps. The compiler performs a jump optimization. For most jumps a short-address instruction is generated.

Defining our own instruction set allowed simple code generation. In spite of this, programming the code generation part of the compiler required more work than usual, since a lot of thought went into defining the instructions. As a result, simple code generation and efficient object code was achieved. The code of the Lilith computer (M-code) is more than twice as dense as the code of the well-known PDP-11.

Zusammenfassung

Mit dem Projekt, einen eigenen Rechner zu entwerfen, haben wir drei Hauptziele verfolgt. Wir wollten forschen, uns selber weiterbilden und als Resultat auch noch ein gutes Werkzeug zur Softwareherstellung erhalten. Wir sind konsequent nach einem Top-Down Verfahren vorgegangen. Zuerst wurde die Programmiersprache Modula-2 entworfen. Dies ist die einzige Programmiersprache, welche wir auf dem Lilith Rechner verwenden. Die Architektur des Rechners ist derart festgelegt, dass der Compiler einfach wird, aber trotzdem sehr effizienten Code erzeugt. Schliesslich wurde die Hardware darauf ausgelegt, dass diese Architektur effizient microcodiert werden konnte.

Stack Rechner erlauben einfache Code Generierung, gelten sonst aber nicht als effizient. Der Lilith Rechner hat einen kleinen Hardware *Expression*-Stack. Stackoperationen sind dadurch in einem einzigen Maschinenzyklus möglich und benötigen keine Speicherzugriffe. Der Expression-Stack erlaubt, die Einfachheit einer Stack Maschine mit der Effizienz einer Register Maschine zu kombinieren. Der Compiler stellt sicher, dass der Expression-Stack nicht überläuft.

Auf den Speicher wird auf strukturierte Weise zugegriffen. Lokale Daten, globale Daten eines Moduls und Instruktionen werden relativ zu entsprechenden Registern adressiert. Diese Register werden bei Prozeduraufrufen oder Rücksprüngen automatisch nachgeführt. Die Daten externer Module können erreicht werden, indem über die Modul-Tabelle zugegriffen wird. Die Adressierung relativ zu den entsprechenden Registern erlaubt kurze Adressen (*Offsets*). Auszählungen haben ergeben, dass über 95% aller *Load*-Instruktionen einen Offset kleiner als 12 benötigen. Für diese Instruktionen ist der Offset im Instruktionsbyte codiert.

Für gewisse Sprachelemente lohnt es sich, entsprechende Maschineninstruktionen zu definieren. Die *CASE* Anweisung wird in die entsprechende *Case*-Instruktion compiliert. Analoge Spezialinstruktion sind für die *FOR* Anweisung reserviert. Meistens sind aber die M-Code Instruktionen nicht so offensichtlich auf Modula-2 zugeschnitten. Die Eigenschaften der Sprache Modula-2 wiederspiegeln sich mehr in der Gesamtstruktur des Instruktionssatzes als in den einzelnen Instruktionen.

Kurze Sprünge treten wesentlich häufiger auf als Sprünge über weite Strecken. Die Länge eines Programmes kann wesentlich reduziert werden, wenn für kurze Sprünge auch kurze Instruktionen generiert werden. Der Modula Compiler führt eine Sprungoptimierung durch. Diese Sprungoptimierung ist wesentlich einfacher als entsprechende in der Literatur erwähnte Optimierungen. Sie basiert auf den Tatsachen, dass Sprunginstruktionen relativ addressiert werden, und dass für strukturierte Anweisungen sowohl das Sprungziel wie auch die Absprungstelle Bestandteil derselben Anweisung sind.

Die Definition eines geeigneten Instruktionssatzes hat zu einer relativ einfachen Codegenerierung geführt. Trotzdem steckt dadurch im Codegenerierungsteil des Compilers mehr Arbeit als üblich. Als Resultat entsteht eine einfache Codegenerierung, die dennoch äusserst effizienten Code erzeugt. Der Code für den Lilith Rechner (M-Code) braucht weniger als die Hälfte des Speicherplatzes des entsprechenden PDP-11 Codes.

Table of contents

1. Introduction

The Lilith computer [Wir1] was built with three main goals in mind: computer science research, experience in hardware design and producing modern computers for future use. A top down approach was followed: first the programming language Modula-2 (henceforth called Modula) was defined. Next, the Lilith computer was designed to be programmed in Modula, exclusively. The architecture of the Lilith computer was chosen so that the compiler can be simple and still produces short and fast object code. Finally, the hardware was designed to allow efficient microcoding of the desired architecture.

Modula grew from PASCAL [Wir2], where its data structures and structured statements originated. From MODULA [Wir3] comes a more modern syntax, in which every structure ends with an explicit termination symbol, and a powerful module concept. The module concept is influenced by Mesa [MMS]. This concept allows partition of large programs into individual modules which can be separately compiled [Gei]. A definition module specifies an interface which is completely realized in a corresponding implementation module. Any program using the interface refers to the definition part only. Modula includes a coroutine concept which allows the programming of user-tailored schedulers or device drivers. The language is essentially machine independent, but a predefined module exists which contains some machine dependent features. A powerful feature of Modula is its procedure type. Procedures may be freely assigned to variables of the corresponding procedure type; calling the variable is equal to calling the assigned procedure.

The first compiler for Modula was written for the PDP-11 [PDP] computer. It stems from the compiler for MODULA [Le], but it is reduced to five passes. The first pass performs lexical analysis and syntax checking. The second pass processes the declarations. The third pass does type checking for the statements. Passes four and five perform code generation for expressions and statements. For the Lilith computer, the code generation was completely rewritten and requires only one pass.

The division of the compiler into several passes was done mainly because of memory limitations on the PDP-11. The passes communicate through the symbol table, a few global variables, and interpass files. A pass reads the interpass file, except pass 1 which reads the source, accumulates some information in the symbol table, and writes another interpass file. The interpass files retain the syntactic structure of the original Modula program and are scanned with the method of recursive descent compiling. The final code-generation pass scans the entire statement part of the program. It needs no scanning with respect to imports and declarations. The compilation of expressions is simplified, since the evaluation of constant expressions is already done before code generation. This thesis will treat the code generation

aspects of the compiler only. Usually the code generation part of a compiler is heavily influenced by the machine architecture; for Lilith, the reverse is also true.

Early computers were designed with the machine language programmer in mind. A lot of special cases are obvious to the human assembly programmer. Compilers generate code in a different way. A compiler is easy to write if its early code generation decisions need not be reconsidered. This can be achieved, if the target machine is defined with regularity and in an orthogonal manner [Wul]. If something is done one way in one place, it ought to be done the same way everywhere. Also, it should be possible to combine orthogonal concepts arbitrarily.

Another principle to simplify compiler decisions is to allow only one way to do a thing. It seems to be no advantage to make the machine simpler and thereby eliminate some choices; the compiler could simply ignore them. However, not to implement unnecessary hardware features reduces complexity and omits the time and space used to select between alternatives.

Some current computer architectures like the PDP-11 or the VAX [VAX] allow both easy assembler programming, and fairly easy code generation for compilers. Usually this compromise is paid for by obtaining less then optimal code density. It is easy to generate simple code, but generating optimal code is still hard. Computers like the Burrough 5000 [Bur] are designed to be programmed in high-level languages exclusively. If only a trusted compiler is allowed to generate code, some security constraints can be checked by the compiler instead of at runtime.

Architectures which support only one programming language can avoid a lot of instructions. However, such an architecture can only be useful if the chosen programming language is powerful enough. Other architectures have some special instructions designed to implement some high level language constructs, these instructions sometimes look simply like an add-on feature.

General stack computers are believed to simplify code generation, but execute programs less efficiently. Register-machines allow some optimizations in register usage. However, this is possible only if a sufficient number of registers are available. It is the author's experience that the 8 registers of the PDP-11 are of no real help. Four registers are constantly used for fixed purposes of hardware and compiler dictated structures, one register must be held free to allow special operations *(eg. MOD)* which need two adjacent registers. The number of remaining registers is not sufficient to avoid operand fetching, since any useful value has been overwritten again. On the other hand, a compiler optimizing the use of three registers is not less complex as a compiler optimizing the use of 16.

Some architectures try to speed up stack operations with cache memory. Caching is

still more time consuming than accessing a register, since a memory request is to be prepared, and a decision not to use the memory must be made. The HP 3000 [HP] computer introduces a special cache which is only used for the top four elements of the stack. The Mesa architecture [JW] introduces an evaluation stack, which works with register speed and needs no special connection to the memory.

A good encoding of instructions primarily saves memory space for programs. However, to fetch fewer instruction bytes also reduces the time necessary for memory accesses. J. Wade [WS] describes an algorithm to find the optimally compact encoding of instructions, but he ignores implementation costs. C. Foster [FG] recognizes that most instructions are used only in connection with very few other instructions. He proposes that every instruction has a restricted number of successors and an escape instruction, which is used if another instruction follows. A. Tanenbaum [Tan] proposes an instruction set which is based on an empirical study of programs. This instruction set uses byte encoding and is designed to be decoded with minimal hardware effort. In spite of not being optimal, he gets very dense code with minimal instruction decoding time.

The Lilith architecture is heavily influenced by the Mesa [TCLSB] architecture and is a combination of the best of these aspects. This combination makes the Lilith architecture new and remarkable. It is designed to execute a high-level language; there is no assembler. Some special high-level instructions implement exactly the behaviour of a corresponding modula statement, and also the global aspects reflect the language. Most features are accessible in exactly one way. It is a stack machine, which uses a memory stack for procedure calls and a fast evaluation stack for expressions. The instructions, most of which use one byte only, are well encoded.

One of the most exciting aspects is the direct mapping of separately compiled modules onto the Lilith architecture. The direct consequence, to define an instruction to address data of imported modules, is of only minor importance. However, the consequences on the overall structure of the architecture are crucial. Mainly the access of global data of the current module and the procedure call mechanism are designed with respect to separate compilation. The partition of memory segments mirrors the partition of the program into modules.

Still, the Lilith architecture is quite simple. Some of its few complexities come from being a real computer and not just a study of theoretical aspects. The architecture is not always as simple as possible, but complexity is accepted only where it results in substantial optimizations.

The architecture was defined in parallel with the code generation of the compiler. The distribution of the complexity between compiler and instruction set is carefully planned. For example overflow of the evaluation stack is recognized at compile

time; there is no test for this at run time. Optimization of jump instructions is done by the compiler, whereas for the CASE statement a more complex instruction can be generated, which needs no optimization.

Several people are involved in the Lilith project. We will concentrate on work mentioned in this thesis.

Niklaus Wirth guided the whole Lilith project; together with him, the author designed the instruction set. Niklaus Wirth, Richard Ohran and Jirka Hoppe have developed the hardware. Werner Winiger did the micro-programming. Leo Geissmann programmed the compiler-passes one to three. Svend E. Knudsen worked on pass one early in project. The author did the early programming of code generation for the PDP-11 computer, which later, was taken over by Anton Gorrengourt. Code generation for the Lilith computer was completely done by the author. Further, the author made the measurments to document the code density and the benchmark measurments.

Svend E. Knudsen designed the operating system; Werner Winiger the Editor. The author programmed the screen and window-handler packages and coordinated the work on the debugger.

The main innovative concepts of the Lilith architecture which are due to the author's work are:

The overall structure of Lilith to represent programs. One single frame address table serves to access procedures from separate compiled modules. The frame address table contains the addresses of the data frames; the pointer to a code frame is part of the corresponding data frame. Support of separate compilation by the architecture has proven to be very important.

The procedure call mechanism, which is very efficient in most (simple) cases but needs additional overhead for rare complex calling situations. Further, the typical distinction of local and exported procedures is ommitted.

The Modula oriented high-level instructions (FOR, CASE, short circuit and/or branches). However, such instructions are also useful primitives for compilation of other high-level languages.

1.1. Highlights of the architecture of the Lilith machine

The expression stack allows arithmetic operations on a small but fast hardware stack. It has the simplicity of stack machines without the penalty of additional memory requests.

All computational operations are stack operations with known type and implicit overflow test. No additional instructions are needed for these tests and for tests against NIL access.

A special instruction fetch unit allows separate byte access to the code, in spite of the machine being word addressed. The instruction fetch hardware fetches the code from a separately specified code frame. The code address space is not limited by the program counter.

The carefully planned encoding of the most frequently used instructions results in very short code. The typical frequency of the instructions is known, since they are generated by a single compiler.

Relative addressing to module and procedure data results in short offsets. It also fully supports separate compilation with independent code and data frames. Procedures and modules are referenced by numbers, not addresses. This simplifies the work of the linker; it allows linking at load time. Further, it results in shorter code.

The completeness of the instructions with respect to Modula allows for every Modula construct to be compiled into few (mostly one) powerful instructions. Many Modula features have their exact representation in machine instructions.

The instruction set is not exclusively Modula oriented; it has a complete set of operators, even if not all are now used for Modula. Shift, rotate, double precision and device-accessing instructions have not been forgotten. The Modula language has the ability to execute special machine instructions and allow for extensions or modifications of the language without redefining the machine.

Lilith is a single user machine; no protection mechanism is needed. In case of severe errors, rebooting the machine is always possible. There are no priviledged instructions which the compiler cannot generate normally.

The mouse, the raster scan display, which displays main memory with its bitmap, font structures and instructions are important aspects of the machine but need not be supported by the compiler; they are therefore not discussed in this paper.

1.2. Short aside on optimality and optimization

Optimization is most often done for optimization of execution time. At least as important is an optimization of memory space. On minicomputers, memory is limited, whereas time is available.

From a more general viewpoint, only costs are minimized. Therefore, minimizing the programer's work is considered more important.

Real time applications show a special need for optimality: either the computer is fast enough or it fails completely.

Sometimes it is better to buy a faster computer than to optimize too much.

Where can optimization be done?

Optimality may be achieved in many places. The solution of a problem with a computer is done on several levels. Any of these levels are candidates for improvement. Any translation from one level to the next lower level can be used for optimization. Typically, the levels at which a problem is considered are the following:

Choice of the algorithm
Programming the algorithm
Peculiarities of the programming language
Translation of the programming language into machine code
Peculiarities of the machine code
Execution of the machine code (processor speed)

Typically, the term optimization is used for optimization of translation of the programming language into machine code. We should not forget the other possibilities, however.

Sometimes an optimization is made superfluous because the next lower level does not offer more than one choice for the translation. In fact, this can have advantages. This reduction in choice also results in a reduction in complexity, and it may result in memory savings and faster execution.

A classification of compiler optimizations

Register optimization

The CDC Pascal compiler [Amm] remembers which variable or address is loaded in a register and can sometimes avoid the reloading of this value. The compiler

assigns to any register a value function which shows the probability that the register's content may be used a second time. A more sophisticated optimization would also try to avoid register stores and keep the value in a register only. A data flow analysis can replace the probabilistic assignment of a register number.

Span-dependent optimizations

Since short-distance jumps occur much more frequently than long-distance jumps, the code length can be reduced considerably when short addresses are used for short-distance jumps.

Variables may be reordered in a way that more frequently used variables are accessed with shorter instructions.

Pre-evaluations

The simplest optimization of this type is the evaluation of constant expressions at compile time or the replacement of multiplication by a specific constant with shifts.

The CDC Pascal compiler replaces multiplications by some constants with two shifts and an addition. Which operations should be replaced depends on the relative speed of the instructions.

We consider elimination of common sub-expressions and moving computations out of loops as optimizations of this type. Replacing the multiplication by the FOR loop control variable in the FOR loop with additions is called *reduction in strength.*

Any of these optimizations may be done on a local or a more global basis and at different places in the compilation process. Also, an important criterion for locality is whether the optimized environment includes loops and conditional statements.

Programming languages which support abstract data types show new optimization possibilities. Abstract data types are often handled with very short procedures. The procedure call may need more code than the whole procedure body. A compiler with very global optimization could try to generate inline code for such procedures.

We prefer local optimizations and the optimizations which are done while the structure of the compiled program is still known to the compiler. Other compilers often use later phases for optimizations.

A typical optimization on already generated code, is *peephole optimization* [Kee]. The instructions are analyzed in a very small context and some combinations of instructions are replaced by others. Such a replacement may allow further optimizations in late optimization passes.

Some general remarks

Several authors distinguish between machine-independent and machine-dependent optimization. However, machine-independent optimizations often needs be repeated on a machine-dependent level.

Sometimes it is not clear at all what is optimization and what is simply "not generating foolish code". Measuring the reward of optimization depends greatly on what is chosen as the starting point. The poorer the non-optimized code, the better are the results of an (so-called) optimization. When results of optimizations are considered, it is very important in which order the optimizations are implemented (or better, measured).

The simple optimizations usually result in big benefits. Further improvements of the code need much more sophisticated algorithms; to generate really optimal code is not worth the enormous costs.

It is a good idea to concentrate on optimization of user-inaccessible features and to leave optimization, which the programmer can do, to the programmer; at least as no automatic program generators are considered.

What is done for the Lilith computer

On Lilith, all levels are candidates for optimizations. The language Modula is defined in such a way that simple constructs are easy to program. Inefficient features are not built in but, rather, must be explicitly programmed. The language contains constant expressions explicitly. Special standard procedures INC and DEC are introduced. Most identical subexpressions occur in a very simple context and are replaced by use of the INC and DEC procedures. Thus, hypothetical optimization to recognize identical subexpressions looses most of its advantages.

The compiler performs an optimization for short and long jumps. Except for the EXIT statement, all statements are compiled with the shortest possible jump instructions.

Shifts and masks are generated for multiplications and divisions with constants, if their values allow it.

Constant expressions are evaluated, even if they are not in the declaration part.

The compiler omits any addition of zero, multiplication by one, and code generation for branches of IF statements which can never be executed because of constant conditions. However, these topics are not thought of as an optimization, but are implemented to allow parametrizations of programs.

A large effort is spent on an optimal instruction set. The encoding of the instruction set is an obvious place where optimization is done. At least as important are the special instructions to implement some statements in an optimal way and the benefits of an adequate global structure.

2. Overall structure and procedure call

In any language which allows procedures to be activated recursively, it is an appropriate mechanism to use a stack allocation strategy for local variables. Each activated procedure reserves a *procedure segment* on the top. At the beginning of each procedure segment is a *procedure mark.* The procedure mark saves the information used to return from the procedure activation. We will call this stack used for procedure segments a *data stack.*

Modula allows coroutines, therefore, each coroutine needs its own data stack.

Note: The Mesa architecture [JW] allocates procedure segments in a heap. Using a heap has the advantage of using all available memory in a big pool. The program does not crash if one coroutine produces a stack overflow. As long as memory is available, no stack overflow occurs. On the other hand, using separate stacks for each coroutine allows to avoid storage overflows in critical system coroutines. More important, using a stack mechanism is much faster than using heap management. Allocating and deallocating elements in a heap is certainly more complicated than on a stack.

2.1. The expression stack

The Lilith computer is a stack computer. All expression evaluation is done on a special stack, called the *expression stack.*

Stack computers are known to simplify compilation of expressions considerably. But usually stack computers are less efficient than computers with general registers. Pushing the operands and popping results of expressions doubles the memory references. The Lilith expression stack is a small hardware stack; accessing the expression stack requires no memory references. While no good register optimization is considered, the expression stack allows faster expression evaluation than a computer with general registers would. Using a stack scheme, specifying register numbers may be omitted.

The expression stack is more efficient than a stack in main memory and speeding up the memory access with a cache. Caching mechanisms are typically intransparent to the processor, which "sees" normal memory requests. The special hardware stack is much simpler and is directly accessed by the arithmetic unit.

The hardware expression stack is implemented with a 16 word stack and a separate register. The top element of the expression stack is kept in the register; the hardware stack contains the next element down to the bottom element. This allows simultaneous access to the two top elements of the expression stack. On a push operation, the old top-of-stack contents (in the register) is pushed onto the hardware stack and the register is loaded with the new value. Arithmetic operations take the first operand from the register, pop the second operand from the hardware stack, and reload the register with the result. Finally, the pop operation delivers the value

from the register, which will then be reloaded with the value popped from the hardware stack. Such operations may all be done in one machine cycle (micro-cycle).

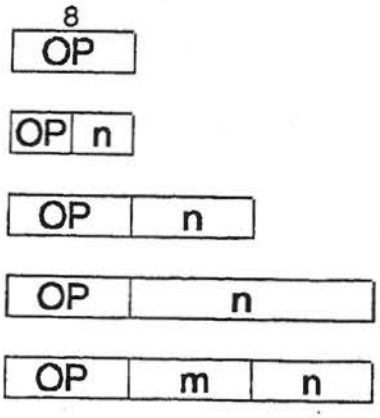
The compiler has to check the expression stack so that it never overflows. In spite of this check being a burden to the compiler, it eliminates the need of runtime checking for stack overflow in arithmetic expressions. To enable the compiler to count the values loaded on the expression stack, at procedure entry the stack-pointer must have a known value.

There is exactly one expression stack. On coroutine transfer (including interrupts) the expression stack is saved onto the data stack of the interrupted coroutine and restored from the data stack of the resumed coroutine. However, it is a compiler convention that on executing a transfer the expression stack is empty. This does not apply to interrupts.

2.2. Addressing of data

stack	ES[top]
immediate	n
local	M[L + n]
global	M[G + n]
indirect	M[ES[top] + n]
external	M[FrameTable[m] + n]

ES expression stack
M memory
m,n Consider the instructions and their formats:



"Addressing Modes"

The memory is organized in 16-bit words. It is subdivided into several areas. Data of a procedure or a module is allocated in a segment. Variables are accessed relative to the origin of the segment. For indirect addressing, absolute addresses are used. These absolute addresses however, are never part of an instruction. Absolute addresses are generated at run time . The code itself is independent of the location of the segment. Since absolute addresses are stored in a word, 65,536 (64k) words may be used to store data.

Local data are addressed via the L-register, which points to the local data segment. Every procedure invocation allocates a new data segment on the stack of the running coroutine. The dynamic link of the procedure segment points back to the segment of the calling procedure.

Data of intermediate level procedures are still accessed relative to their data segments. First, the start address of the segment is loaded on top of the expression stack (by executing the get base (GB) instruction). The data themselves are then addressed relative to the base, found on top of the expression stack.

Global data of the module currently under execution are addressed via the G-register.

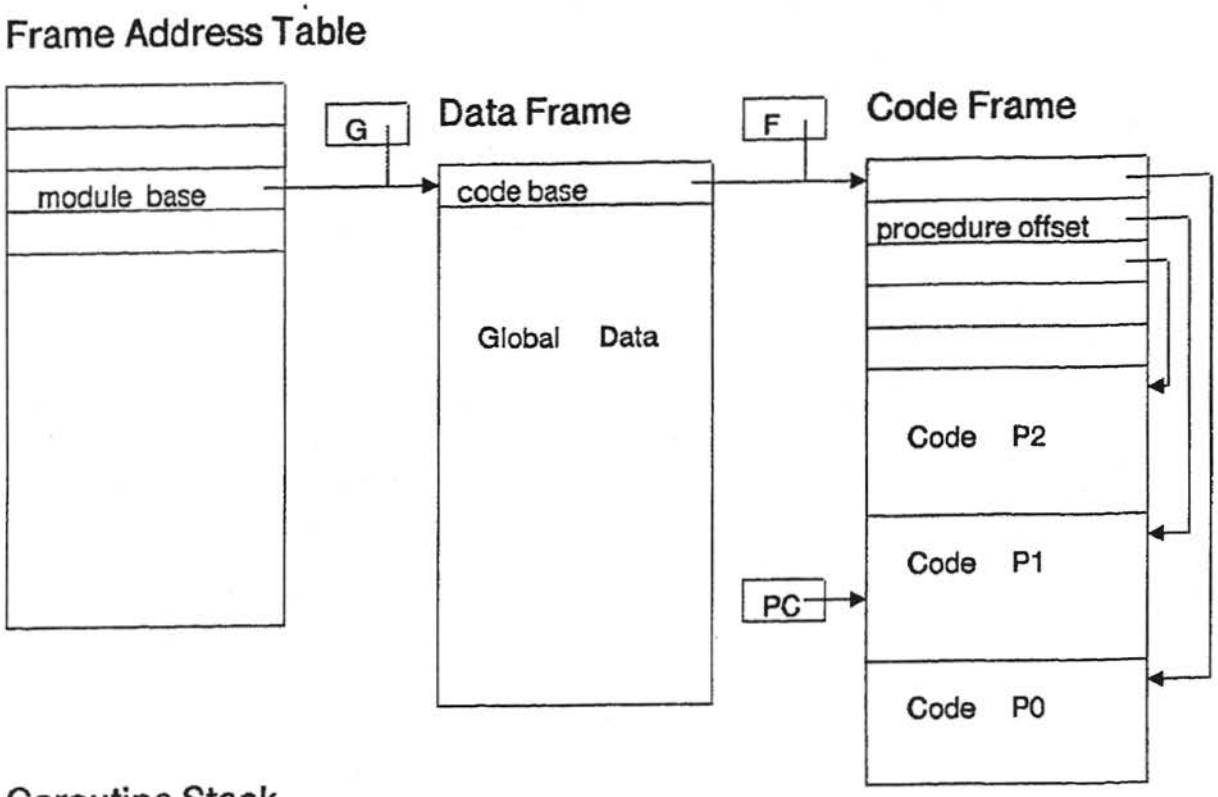
Global data of other modules are accessed via the frame address table. This table is accessed by the instructions which access external data. Example: The load external address (LEA) instruction may be used to load the address of an imported variable onto the expression stack.

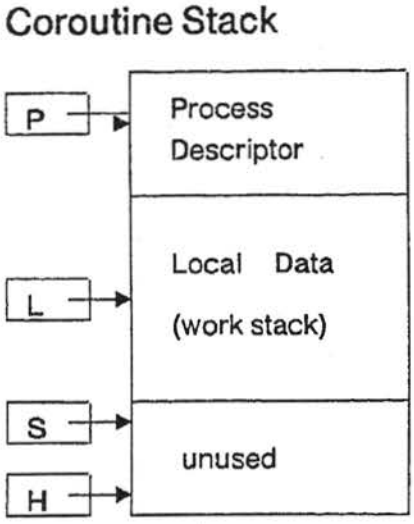
Variable parameters and dynamic memory (variables obtained with NEW) are addressed by their absolute addresses. These absolute addresses (i.e pointers) are stored in ordinary memory locations; The addresses themselves are accessed relative to the origin of some segment, like other data. There is no instruction which accesses absolute-addressed data directly.

Value parameters are treated like local variables.

Structured data are usually accessed indirectly. A pointer to the structure is stored at a known (at compile time) offset in the data segment. This pointer is loaded on the expression stack like the value of a simple variable. Elements of the structure are accessed with an offset to this pointer.

The architecture needs three kinds of storage areas: code frames, data frames, and stacks. The remaining memory is used for other purposes like heaps, fonts or bitmaps. Our Lilith computers are equipped with 128k words of memory. The normal instructions to access data are designed for only 64k. The upper 64k is accessed by special instructions and by instructions which allow access to fonts and bitmaps.





Registers

L	pointer to local data segment
G	pointer to global data segment
S	pointer to top of stack
H	pointer to end of stack
F	pointer to current code frame
PC	offset to current Instruction (byte)
P	pointer to current process descriptor
M	interrupt mask

"Global Structure"

2.3. Global data of modules and code

The global variables of each (separate) module are represented as a so-called *data frame.* This area is allocated by the loader. The base addresses of all loaded frames are assigned (by the loader) to a fixed area of the store, called the *frame address table.* The frame address table can contain up to 256 entries, since it is indexed with byte values.

However, the debugger can handle at most 229 entries; this will be explained later.

The G-Register contains the base address of the data frame which belongs to the module containing the currently executed procedure. The G-register value must be

changed whenever a transition occurs from one (separately compiled) module to another. This is done automatically by the firmware when a call instruction is executed.

The first three words of a data frame are reserved:

Word 0: pointer to the code frame (F-register)
Word 1: initialization flag
Word 2: pointer to an area used for string literals

The subsequent words hold the global data of the module. All global variables are addressed relative to the data frame origin.

The pointer to the code frame is crucial for the machine architecture. This pointer is the only information indicating where the code of a module is allocated.

The initialization flag is a private convention of the compiler. The compiler requests the loader to initialize this word to zero. The module initialization procedure sets this flag to one with the test and set (TS) instruction before initializing the module. If the flag shows that the module has already been initialized, the initialization procedure returns.

Note: A module tries to initialize any module it imports. There may be several attempts to initialize a module. Therefore, it must check to assure its initialization code is executed only once.

The pointer to the string template area is implicitly used by the load string address (LSTA) instruction. String templates are used for passing literal strings as actual parameters, because the procedure must receive an address.

The F-register points to the base address of the code of the module which contains the procedure currently executed. Code is accessed relative to the F-register. The hardware adds the F-register to the program counter (PC) on any instruction fetch. The code of each separately compiled module is represented as such a segment, called a code frame. The base address of a code frame is assigned (by the loader) to word 0 of the corresponding data frame, and the one which is currently under execution is also contained in the F-Register. The F-Register value is loaded from the data frame whenever the G-Register is changed.

A module contains a number of procedures. The entry points (offsets to F) of these procedures are stored in the first words of the code frame of the module; location n contains the offset to the procedure n. (The compiler uses procedure 0 for the module initialization). The further locations contain the actual code of the procedures.

The hardware interprets the content of the F-Register as the address of a double-word. The program counter is an offset to the F-register; the hardware performs an addition, which results in an 18-bit address of an instruction byte. This

allows the code to be loaded anywhere in a 128k word memory. The length of a single code frame is restricted to 32k bytes.

Note 1: The only two programs which need to know this behaviour are the loader (for initialization of the code frame pointers) and the debugger. (The debugger inspects the codeframes to get knowledge about procedure numbers).

Note 2: It is an implication of separate compilation that modules must be initialized. The PDP-11 Modula system takes a different approach for the initializing of modules. The compiler generates only one call instruction per module for the initialization of other modules. The linker inserts an initialization procedure address into the call instruction. It is the linker which detects the order for module initialization and the linker selects which module initialization must be called from which module.

This PDP-11 initialization scheme needs less space at runtime, since every module initializes at most one other module. On Lilith, however, the linking may be postponed until the module is loaded. If on Lilith the loader were to take care of the module initializations, the algorithm to detect the order would need more memory space than the compiler generated algorithm which tries to initialize every imported module. In addition, the Lilith mechanism does not need the complexity (at link time) of the PDP-11 mechanism.

2.4. Local data of procedures

Data segments for local data are allocated on the stack of the current coroutine whenever a procedure is called. This data stack is distinct from the expression stack; the data stack is in the main memory, and its size is limited only by the memory area available for the coroutine. The L-register points to the data segment of the current procedure. The first words of any procedure data segment are used for a *procedure mark.* The S-register points to the top of the stack; its value gives the base address where a new segment is created. The S-register is also incremented whenever more memory is reserved for local data; therefore the area between the L-register and the S-register is the most recent data segment. The stack area limit is assigned to the H-Register. The Lilith uses the H-register to check against stack overflow on procedure entries or data allocation.

The L-Register points to the most recent data segment or procedure mark. It is therefore the base address of variables local to the currently executed procedure.

To allow unstacking of data segments a link is needed. It is called the *dynamic link* and chains every data segment to its immediate predecessor in the data stack.

Modula allows, like other block structured languages, to access data of intermediate level procedures. The GB instruction loads the base address of an intermediate procedure data segment. To get this base address, a second link chain is maintained, which links the data segments the way the compiler sees the situation. These links are called *static links.* The get base (GB) instruction follows the static link and pushes a base address onto the expression stack.

Another, well known method to get intermediate procedures' data segments is a

display, which has been invented by Dijkstra [Dij]. A display speeds up the access of intermediate-level data since one indirection allows to get any intermediate level base address. However, a display must be updated on both procedure call and procedure return. A static link only needs to be set up at procedure call time. On procedure return, the top static link is forgotten together with the data segment of the released procedure.

The frequency of accesses to intermediate level data is drastically reduced in Modula. Because of the module structure procedures are local to modules and use more global data instead of intermediate data. This behaviour tells us that a static link is better suited to Modula than a display. We prefer to speed up procedure calls.

The PDP-11 Modula implementation (and the original Modula-1 implementation [Le]) use a display for access of intermediate level data but speeds up the construction of the display: The compiler checks for each procedure if its data is accessed by other, local procedures. Only if this is the case the procedure modifies the entry in the display.

Note: We modified the compiler to count the procedures which need the display to be maintained. The code generation pass itself had only three such procedures.

For Lilith, because of the rare use of intermediate levels, we preferred to omit the compiler complexity of checking data access by inner procedures and to use the static link method. As a further consequence the maximum nesting level of procedures is increased to the maximum level the GB instruction may handle, and is not restricted to the size of a display.

The procedure mark is four words long. The fields in the procedure mark are the following:

Word 0: Static link
Word 1: Dynamic link
Word 2: Return address and external flag
Word 3: Reserved for interrupt mask

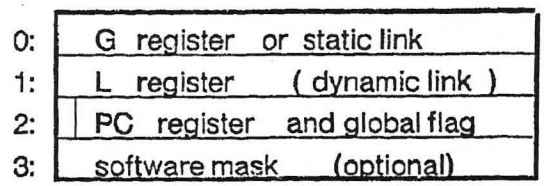
The procedure mark is created by the execution of a procedure call instruction; the procedure return instruction removes the mark. The enter priority (ENTP) and exit priority (EXP) instructions use the mark for storing the interrupt mask.

Several stacks of data segments may exist. Every coroutine has its own stack for its procedure invocations.

Note: Word three of a procedure segment (in the procedure mark) is reserved for the interrupt mask. The instructions to handle priority use this word implicitly. For procedures without priority, this word is not used. Through the absence of a one byte instruction for loading this word, the architecture suggests its reservation. However, the decision not to define an instruction to access this word [LLW3] was taken only after it was known that the address generation part of the compiler does not distinguish whether a procedure is declared with priority or not. We do not consider this to be optimal.

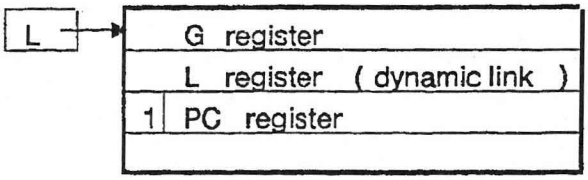
2.5. Procedure calls

General format of the procedure mark



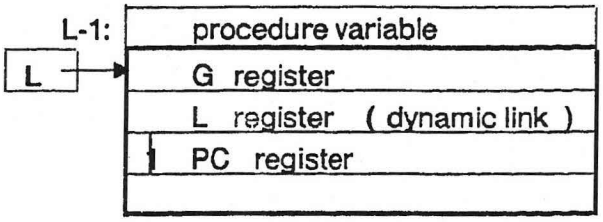
the software mask is saved and used by the code of the procedure if necessary

called as external procedure



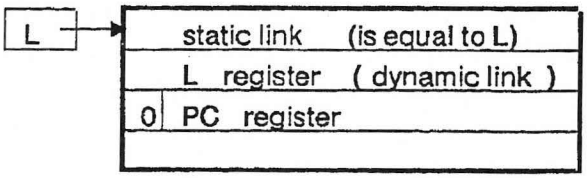
CX mod proc

called as formal procedure



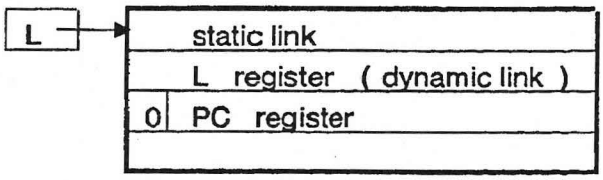
CF the procedure variable is saved and eliminated by the calling environment of the procedure

called as local procedure



CLn
CL proc

called as intermediate procedure



CI proc

"Procedure mark formats"

The procedure call mechanism is very important for implementing high-level programming languages. For this reason, newer computer architectures usually support additional, specific call instructions. The Lilith computer supports only those call instructions needed by the high-level language, no alternate, more primitive subroutine call mechanisms exist. The call instructions are completely embedded in the overall structure of the machine.

For parameter transmission the expression stack is used; either the value or the address of the parameter is loaded onto the expression stack. Likewise, function results are returned on the expression stack.

Therefore, the general call mechanism is extremly simple.

The call instruction establishes the procedure mark on the data stack. Space for local data is allocated by the ENTR instruction, which increments the stack pointer. The ENTR instruction is the first instruction of each procedure. Once space is allocated, the parameters are copied from the expression stack into their proper memory locations. The last instruction of a procedure is a return instruction (RTN).

Since the same procedure may be called either as a local, an external or a formal procedure, the return instruction checks a flag in the mark indicating whether the G and F-Registers must be reset or not. This flag is automatically set by the call instruction.

Notes: The ENTR instruction implicitly tests for stack overflow. This instruction is even used if the segment size is 0. The call instructions do not test for stack overflow. The memory for the procedure mark is always available (because internally the H-register value has been decremented). The overflow tests on allocation of a procedure mark and on allocation of local data are combined in one single test.

Since one bit of the return address is used as a flag, actual code frames should not exceed the length of 32k bytes.

There are four different call instructions:

CL call local procedure (also used for global procedures in the same separately compiled module).

CX call external procedure. The F and G-Registers are reset.

CI call of procedure at intermediate levels (neither local nor global).

CF call formal procedure (procedure declared as a variable).

CL, CX, and CI have a procedure number as an argument. CX has the module number (the index of the module in the frame address table) as an additional argument. These arguments are part of the instructions. The additional argument of CI is the base of a data segment (static link), which is put on the expression stack by a preceding GB instruction.

CF is explained in the following chapter about procedure variables.

For most procedure calls the scheme used is very simple. However, procedure variables or functions (or a combination of both) lead to more complex situations. The complexity of special cases is the price for the efficiency of the code in the more frequent cases.

Function calls

It is very natural to return function results onto the expression stack. However, some difficulties with the expression stack occur. The compiler needs complete knowledge of the number of values stored in the expression stack. Therefore, on function calls the expression stack is saved on top of the data stack before the parameters are loaded onto the expression stack and the function is called. The STORE instruction serves that purpose.

After return from the function, the function result is on the expression stack. The saved contents of the expression stack must be restored, but the function result must remain the top value of the expression stack. The *load after function* (LODFW) instruction is used to reload a saved expression stack below the top word which is not modified. (For double word function results the LODFD instruction is used).

Most often function calls occur in simple assignment statements where the expression stack is empty. The compiler recognizes these cases and omits generation of the STORE and LODFx instructions.

Procedure variables

Modula allows additional flexibility with respect to procedures. Procedures may be objects that can be assigned to variables. Such variables may be called like ordinary procedures. Procedure types are used rarely but they constitute a powerful facility.

For calling procedure variables, the call formal (CF) instruction is used. CF has its arguments, the module number and the procedure number, packed in one word, on the data stack, **not** the expression stack.

When the compiler detects a formal procedure call, code for loading the procedure descriptor (on the expression stack) is generated before the code for loading the parameters. The expression stack is a real stack, i.e. data elements below the top cannot be accessed. Therefore, the STOT instruction has been introduced. It stores the procedure descriptor from the expression stack to the top of the data stack. Only after the procedure descriptor is moved are the parameters loaded onto the expression stack, as for explicit procedure calls. The CF instruction reads the stored procedure descriptor on the data stack and calls the procedure. The procedure descriptor is removed after the return with a DECS instruction.

The reason why the CF instruction does not automatically remove the procedure descriptor from the data stack is found in the debugger. The procedure and module number of a procedure called by a CF instruction is found by inspecting the procedure descriptor below the procedure mark. Theoretically, the debugger could

determine the procedure number with the help of a load map. If, at some later date, exceptions were to be implemented, it should be possible to recognize the procedure number without additional file access.

For function procedure variables, both the care taken for function procedures and the care for procedure variables must be combined. Both the unused contents of the expression stack and the procedure descriptor must be copied from the expression stack to the data stack. On most occurrences of such calls the expression stack was empty before, and no problems occur. The STOT and DECS instructions may be freely combined with the returning of a function result.

In very few cases a function procedure variable is called with the expression stack not empty. Combining the instructions in an orthogonal way would require the following calling sequence:

```
STORE (store expression stack before)
Load procedure descriptor
STOT
Load parameters
CF
DECS
LODFW (reload expression stack below function result)
```

This instruction sequence would work perfectly well. However, the compiler cannot generate it. The compiler does not recognize the moment when the evaluation of the formal procedure descriptor starts, it cannot generate the STORE instruction in time. The *store expression stack with function variable* (STOFV) instruction is invented to help. It stores the expression stack with a procedure descriptor on top in the correct order. The calling sequence is modified to

```
Load procedure descriptor
STOFV (stores expression stack and procedure descriptor)
Load parameters
CF
DECS
LODFW (reload expression stack below function result)
```

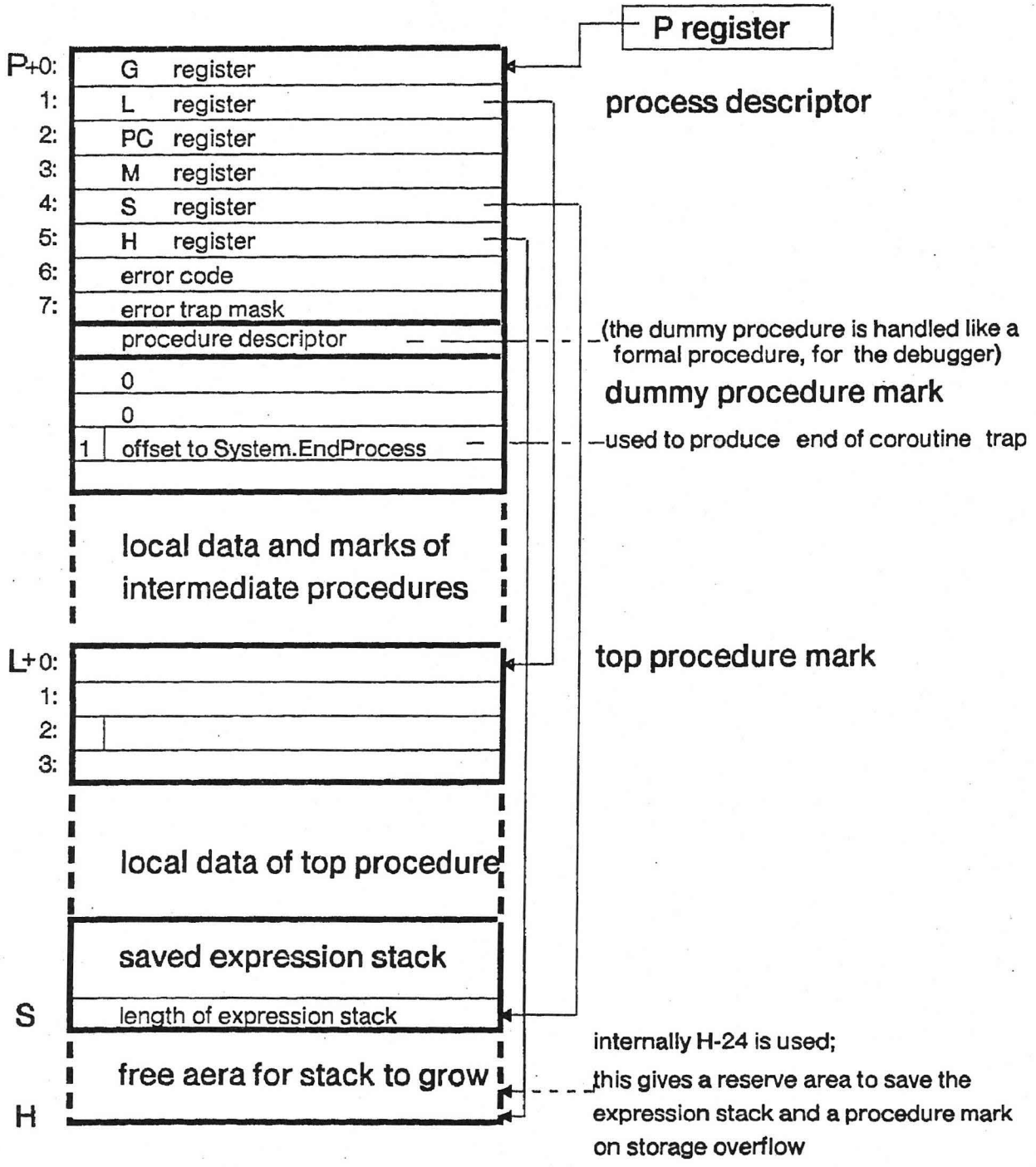
This sequence can easily be generated by the compiler.

Modula allows only global procedures to be used as procedure variables. However, they may be declared in another module. This language restriction is very important. If intermediate level procedures had to be stored in descriptors, it would be necessary to store also their environment (static link). But much worse would be the obligation to check whether the environment is still active when the procedure is called.

Note 1: Another language restriction could replace the current restriction with the similar result of disabling the call of procedure variables with non-existing environments: Procedures are compatible to procedure types declared in the same or in higher levels only. (The type declaration is important, not the variable declaration.)

Note 2: The numerical values of the procedure call instructions are defined with regard to the debugger: A return address points to the next instruction to be executed. The debugger recognizes the procedure number by inspecting the preceeding code byte. It has to distinguish the different call instructions of different lengths. The call instructions have opcodes greater than 345B. Because these numbers are bigger than module or procedure numbers, the debugger can decide if a byte is the opcode of a call, a module number, or a procedure number.

2.6. Coroutine calls



"Data segment (stack) associated with each coroutine"

Modula supports a coroutine mechanism to implement multiprogramming and processes. Coroutines are executed sequentially. The processor is switched between coroutines by an explicit call of TRANSFER.

Note: It is unfortunate that the coroutine type is called PROCESS in Modula.

Any program may consist of several coroutines, each of them associated with its own stack. Whenever a coroutine is initiated, an area of storage is allocated (as specified by the programmer). This area constitutes the data stack containing the data segments of the procedures.

At the bottom of the stack an area for a process descriptor is reserved. This process descriptor is used to preserve the processor state while the processor executes another coroutine. The P-register points to the process descriptor of the coroutine under execution. (The process descriptor of the coroutine currently under execution contains undefined values in the locations used to preserve the register contents; the register contents are saved when the coroutine releases the processor).

The following table shows the assignment of the words in a process descriptor.

Word 0: the G-register
Word 1: the L-register
Word 2: the program counter (PC)
Word 3: the software mask (M-Register)
Word 4: the stackpointer (S-Register)
Word 5: the stack limit (H-Register + 24)
Word 6: used for the error code
Word 7: used for the error trap mask

Modula variables of type PROCESS are implemented as pointers to process descriptors.

Coroutine transfer

We represent a coroutine transfer in Modula by the standard procedure

PROCEDURE TRANSFER(VAR source, destination: PROCESS)

A transfer from one coroutine to another is caused by a TRA instruction and constitutes a context switch. The instruction has three arguments:

A Boolean, which determines if the interrupt mask should be switched or kept; this boolean is part of the instruction code itself. We will explain this flag in the next chapter.

Two addresses, on the expression stack, of pointers to coroutines. The PROCESS variable *destination* is the coroutine to be resumed. The PROCESS variable *source* receives the suspended coroutine.

Note: The TRA instruction first fetches the destination and thereafter stores the source. This order is relevant if source and destination are stored in the same variable.

Coroutines are created by the procedure

```
PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS; n: CARDINAL;
                     VAR p1: PROCESS);
```

P: denotes the procedure which constitutes the coroutine
A: is the base address of the workspace used by the coroutine
n: is the size of this workspace
p1: is the newly created coroutine.

This procedure does not use special instructions; it is itself written in Modula (in module System) and is loaded together with the program. However, the compiler compiles calls of this procedure directly; i.e. it knows the procedure number. The absolute linker has to guarantee that module System is loaded.

Building a coroutine stack with its process descriptor is easy as long as the termination of the coroutine is not considered. When it terminates, this is the end of a program. To interpret this behaviour, the coroutine stack is built in such a way that its initialization procedure already has a return address. This return address then points to a procedure (also in module System) where a TRAP instruction will be executed (to tell the operating system the coroutine has ended). The implementation of this procedure must consider the fact that after the return the L-Register is undefined, so the procedure must not contain local data.

2.7. Interrupts and System initialization

If the hardware provides coroutines as a basic facility, an interrupt can take the form of a coroutine transfer. In fact, it can easily be represented and interpreted as equivalent to the operation

```
TRANSFER(interrupted, interrupthandler)
```

where *interrupthandler* is the routine to be resumed and where the pointer to the interrupted routine is assigned to *interrupted.* A pair of memory locations, (interrupted and interrupthandler,) is associated with each source of an interrupt signal. Moreover, an action identical to that generated by an interrupt signal can also be achieved by execution of certain instructions. These interrupts are called *traps.* One such pair of memory locations is associated with traps.

Traps occur for so-called *hard errors* (errors which can destroy program control), namely, range checks, NIL access, and stack overflow. The cause of the trap is stored, for inspection by the trap handler, in a word of the process descriptor. So-called *soft errors* are overflow and division by zero. Soft errors which occur either because of overflow of signed or unsigned arithmetic may be masked off with a mask

bit in the process descriptor. The mask bits are checked only when an overflow occurs. (These overflows are made maskable, since they cannot destroy the code or change variables).

Two kinds of coroutines exist and should not be confound:

A scheduler switching *independent* coroutines *(processes)* has to switch the complete context including the processor priority. Device drivers also have to switch the complete context.

The other kind are coroutines used to switch control of execution as part of *one single* program *(process)*. A context switch between coroutines which belong to the same process should not modify the processor priority.

The distinction of the two context switch methods is done by the compiler. The flag of the TRA instruction denotes the choice of the transfer method:

Calls of TRANSFER from modules with priority are assumed to do scheduling, the processor priority will be switched.

A call of TRANSFER from a module without priority will not switch the processor priority.

Note: Implementing interrupts as a coroutine transfer is not self-evident. The PDP-11 interrupt mechanism executes an interrupt without exchanging the coroutine data stack. The Modula runtime system needs a very tricky method for transforming the interrupt into a coroutine transfer. [For PDP-11 assembler programmers: In the Modula implementation, the interrupt vector points to the STACK of the driver coroutine, on top of which is a JSR R2 xx instruction. On an interrupt, the runtime system (at location xx) gets control. The stack-pointer points to the interrupted coroutine, R2 points to the interrupt handler.]

Masking interrupts

We consider two distinct concepts for masking interrupts:

The device status: An interrupt from a device is enabled or disabled, because its handler is loaded or the device is physically present. We use the memory word at absolute address 3 to denote if interrupts should be accepted.

Priority: The process priority may be changed to postpone interrupts. This is used when program control enters a critical region. The priority is stored in the procedure mark.

The priority is set by the compiler, which takes into account module priority; the device status is handled by the operating system. The operating system is not allowed to modify the priority field of process descriptors (simply because it does not know all process descriptors of the user program).

The ENTP instruction enters a priority region; it stores the old priority in word 3 of the procedure mark. EXP resets the priority to its stored value.

However, our hardware knows only one concept for disabling interrupts, a mask register. The microcode has to map both the priority and the device status into one hardware priority mask. Whenever the priority is modified, the hardware priority mask is set to the intersection of the priority and the device status. Modification of the device status must be done on the highest priority level; access to the device status should be atomic. Only when the priority region is left does the EXP instruction force the modified device status to be reconsidered for setting the hardware priority mask.

The Lilith has priority levels 0 to 15. However, since it is also possible to run without priority, 17 priorities are to be considered. Traps use priority 7; priorities 8-15 are used for interrupt lines.

Note: With ENTP the priority is not allowed to decrease; calling a procedure with low priority from within a procedure with higher priority is considered to be an error. This test can be used to prevent some procedures from being called from within drivers. It may serve as a cheap method to force mutual exclusion in some operating system routines.

Interrupts can be masked off by setting priority, but traps cannot. Furthermore, traps cause a microprogram loop if they are masked out by the device status word.

System initialization

When the reset button is pressed, the microprogram stops the current coroutine, initializes its process descriptor, and writes a pointer to the process descriptor into a fixed memory location.

Note: On a cold start of the machine saving a process descriptor results in saving garbage. However, when the machine was running before the reset, this operation is used to save execution information, if a dumpfile is produced.

According to the next input from the keyboard, the microprogram saves the whole memory onto the dumpfile and loads (one of) the bootfiles. At a specific memory location, the microcode reads a value which is interpreted as a pointer to a process descriptor. This coroutine is resumed; it is the program loaded at initial boot (e.g. the operating system). This program is assumed to have masked off all interrupts by the initial values of the device mask, until it installs the appropriate drivers.

The machine does not completely supports all features of Modula; a few features are implemented by a runtime system. The compiler assumes a module *System* to be linked to the operating system. This module *System* implements the standard procedure NEWPROCESS. The behaviour at the end of a coroutine must be implemented by software; doing it by firmware would require a special check at every procedure return, to recognize the end of a coroutine. The module System is

loaded at absolute memory location zero. This is however, not absolutely necessary, but a convention used inside the module.

See the appendix about *Special memory locations.*

3. Representation of data

3.1. Simple types

Any operation in Modula requires operands of a certain type. There are different instructions which support all standard types.

Signed and unsigned arithmetic is used to implement the Modula types INTEGER and CARDINAL; instructions for arithmetic operations on these types imply an overflow check. The compiler need not generate instructions for the overflow check.
Note: It is possible to disable the traps for INTEGER or CARDINAL overflow by setting the trap mask in the process descriptor.

The floating-point instructions support the standard type REAL. REAL variables are stored in 32 bits. The machine further supports double (multi) precision unsigned arithmetic, but the compiler does not use it. Most load or store instructions exist also in a double word version.

Compare instructions load a Boolean value onto the stack. No condition code register is used, this simplifies the code generation. This also explains why overflows are detected implicitly and generate traps. The "short circuit" Boolean AND and OR are implemented with the ANDJP and ORJP instructions, respectively. These instructions do not necessarily result in more efficient code; however, they simplify the code generation considerably.

For pointers or addresses a 16-bit absolute address is used. Access to NIL pointers must cause a trap. Since the test for NIL should always remain enabled, it is done in firmware. On every access to an user generated address (e.g pointers), a test is introduced. Note that doing the tests by firmware on all candidate instructions (LSW, LSA, ...), introduces too many tests. However, these tests need no memory at all, and need no instruction fetch time. We consider this implicit test to be less costly than having special instructions included in most, but not all, cases.

We further use a reserved value for NIL: the address with the biggest possible value (for 16-bit addresses: 177777B). The firmware verifies that an address is not equal to NIL, and it checks further that no cardinal overflow occurs in address calculation. This overflow detection is mainly important when accessing structured data. To access a component, an offset is added to the variable address; an illegal NIL address would be modified and could no more be detected by comparing its value with NIL.
It is further possible for the software to introduce a value nearly-NIL (e.g. 177776B). When memory is initialized to this value, access of non-initialized pointers to records will cause an address error trap. Such an initialization test could not be done with a NIL initialization, since a NIL initialization would hide non-initialized pointers in programs which check explicitely for NIL values.

3.2. Arrays and other structures

Assume the declaration of an array:

```
a: ARRAY [m..n] OF T
```

The array will be accessed indirectly. A word is allocated in the data stack which contains a pointer to the actual array variable. We distinguish the three cases where elements of type T require one, two, or several words. Strings constitute a separate case and are described in the next section. The following tables show what code is generated to access arrays:

a[i] (array element occurring within an expression)

Single word arrays	double word arrays	multi word arrays
load addr of a	load addr of a	
load i	load i	
load m	load m [suppressed if m=0]	
SUB	SUB "	
*	* [* here the runtime test is inserted]	
LXW	LXD	

a[i] := x

load addr of a	load addr of a	compute address of a[i]
load i	load i	like below.
load m	load m	load address of x
SUB	SUB	load TSIZE(T)
*	*	MOVE
load x	load x	
SXW	SXD	

Compute address of a[i]

load addr of a	load addr of a	load addr of a
load i	load i	load i
load m	load m	load m
SUB	SUB	SUB
*	*	*
ADD	COPT	load TSIZE(T)
	ADD	MUL
		ADD

For the case where the lower index is 0, the compiler omits the subtraction. Furthermore, other instruction sequences are optimized whenever possible. For example, multiplication by the size is replaced by a shift whenever possible. When constant indices are used, the compiler does part of the address arithmetic at compile time. Depending on the range type, either signed or unsigned arithmetic has to be used for the index calculations. The table ignores these differences.

The code sequences shown above do not include run time tests. These tests look the same for all cases in the table. The instruction CHKZ checks for i <= n where i is the index and n is the number of array elements (-1). The instruction removes n but leaves i on the expression stack. This instruction allows detection of indices which are too large, indices which are too small will cause a trap on the preceding subtraction.

Virtual origins might be used to avoid the subtraction. We preferred not to use that method because of its complexity. With virtual origins the address arithmetic should be allowed to legally wrap around. Because memory allocators do not know about usage of allocated areas, addresses point to the effective start address. If an address is converted to a pointer, a correction for the virtual origin must be done; etc.

The check and unsigned check (CHK, UCHK) instructions test m <= i <= n and remove m and n from the stack. If the condition is not satisfied, a trap is initiated. The compiler does not generate these instructions for array bound checks, since the array bound check mechanism with the subtraction and the CHKZ instruction is more efficient. However, these instructions are useful to check assignments to subrange variables.

If an array or a record is a component of another structure, then it is directly embedded in that structure, i.e. the structure does not contain a pointer to the component. The load stack address (LSA) instruction is defined to add the offset of such a component to the variable address. If the offset is to big to be stored in a byte the compiler uses regular unsigned arithmetic.

The base address is then computed by using either the load stack address (LSA) instruction or using regular cardinal arithmetic.

Strings

Arrays of characters (strings) are stored as packed arrays, two characters per word. The two cases of loading or storing an element of a character array are compiled with the instructions LXB and SXB. Because of the packing, it is not possible to get the *address* of a character in a string.

When strings are used as parameters, the procedure must get an address. Therefore, literals must be generated. The compiler reserves an area for such string templates. Word 2 of the module's data frame is a pointer to that template area. The load string address (LSTA) instruction is used to load the pointer to that template area and to add the offset to the corresponding string.

The read string (RDS) instruction is defined to implement string assignment statements. The compiler does not remember the context when it scans a string constant. It does not recognize if the string constant occurs as an actual parameter or in an assignment statement. Therefore, it cannot generate the RDS instruction.

Allocation of memory for structures

Blocks of storage on the stack are allocated by the instruction ALOC.

The code sequence

```
load n
ALOC
```

allocates n words on the data stack, and leaves the address of the allocated block on the expression stack. A test for stack overflow is included; it may induce a trap. A normal store instruction stores the base address at its proper location in the procedure segment.

For structured variables of a global module another method must be used. Allocation must be done at compile time; the loader needs to know the size of the data frame of the module. At module initialization time, pointers to all global structures are computed and stored at their proper locations. This must be done before any other action is taken, especially before initialization of imported modules. Thus, on circular imports the data may be not initialized, but the pointers to the structured variables are.

Still another method is used for allocation of memory for structured value parameters. The parameter copy instruction (PCOP) does all necessary work. It allocates the memory area, stores the base address at its proper location in the procedure frame, and copies the parameter.

3.3. Compile-time representation of data

Compiling an expression requires the compiler to keep track of what the nature of the expression is. For an ideal machine, it could be assumed that all used variables are loaded on the stack. Because the Lilith machine is not the ideal stack machine, we use an attribute record in the compiler. Another reason for the compiler to distinguish between different attribute modes is optimization. However, the attributes used for the Lilith compiler look simple when compared to those of other compilers (e.g. PDP Modula, CDC Pascal [Amm]).

The type *ArithmeticType* is defined, as being a type which denotes different classes of machine instructions for doing arithmetic in Modula.

```
TYPE ArithmeticType = (signed, unSigned, bitwise, floating, logical);
```

The type AtMode is used to define the way in which an operand is loaded or stored.

```
TYPE AtMode =
  (globalMod, localMod, loadedMod, addrLoadedMod, externalMod,
   indexMod, byteIndexMod, doubleIndexMod, absolutMod,
   constantMod, procedureMod, stringConstMod, doubleConstMod,
   stringTemplateMod, illegalMod);
```

The following table shows which instructions are generated to load a variable for the different values of AtMode.

AtMode	corresponding Instruction
globalMod:	LGW, LGD
localMod:	LLW, LLD
loadedMod:	none
addrLoadedMod:	LSW
externalMod:	LEW
indexMod:	LXW
byteIndexMod:	LXB
doubleIndexMod:	LXD
absolutMod:	LIW addr; addr : = 0 --> addrLoadedMod
constantMod:	LIn, LIB, LIW
procedureMod:	CL, CX respectively LIW for assignments
stringConstMod:	RDS (not generated by the current implementation)
doubleConstMod:	LID
stringTemplateMod:	LSTA addr; addr : = 0 --> addrLoadedMod
illegalMod:	-

The attribute which describes an expression appears as follows:

```
TYPE Attribut =
   RECORD
    typtr: Stptr;
    CASE mode: AtMode OF
      globalMod, localMod, addrLoadedMod, externalMod,
      absolutMod, stringTemplateMod:
       addr: CARDINAL; (*for addrLoadedMod this is an additional offset*)
       CASE AtMode OF
          externalMod: (*only variables, no constants*)
           moduleNo: CARDINAL |
          localMod, addrLoadedMod:
           CASE BOOLEAN OF (*used for dynamic array parameters*)
            TRUE: dynArrLevelDiff, dynArrOffset: CARDINAL
           END
       END |
      constantMod:
       CASE BOOLEAN OF
          TRUE:  value:  CARDINAL |
          FALSE: iValue: INTEGER
       END |
      doubleConstMod:
       CASE BOOLEAN OF
          TRUE:  r1, r2: CARDINAL |
          FALSE: r: REAL
       END |
      stringConstMod:
       strgPtr: Stringptr |
      procedureMod:
```

```
    procPtr: ldptr (*for module number use procPtr↑.globmod↑.modnum*) |
    loadedMod, indexMod, byteIndexMod, doubleIndexMod:
  END;
END (*Attribut*);
```

This attribute type is much simpler than the attribute types used in either the PDP-11 compiler or the CDC Pascal compiler. A lengthy part of its declaration is used for (compile-time) type conversions and dynamic arrays. Its essential parts are only the type, the mode, an offset (for some modes) and a value (for constants and module numbers). Introducing module numbers is a consequence of separate compilation.

To get the module numbers, the compiler uses the following conventions:

variables: there is the explicit field *moduleNo* in the corresponding variant of the attribute.

constants: a copy of the imported constant is made; the import is no longer visible to the code generation.

string constants: a pointer to the actual value is given; for imported string constants the current implementation makes a copy.

procedure names (assigned to formal procedures): a pointer to the actual procedure declaration entry is given; this entry includes the module number.

The attribute variables are mainly used by the following code generation procedures:

```
PROCEDURE Load(VAR fat: Attribut);
  (* Generate code to load the value of fat onto the expression stack *)

PROCEDURE LoadAddr(VAR fat: Attribut);
  (* Generate code to load the address of fat onto the expression stack *)

PROCEDURE Store(VAR fat: Attribut);
  (* Generate code to store the value from the expression stack onto fat *)
```

These procedures are well known and are already used in the Pascal compiler of U. Ammann [Amm], where these procedures handle register assignment. In the Modula compiler, these procedures have a very regular structure. The procedures consist of one CASE statement for the various modes. Each case consists of simple actions only. Below, the structure of the procedure Load is shown. This skeleton discards checks for expression stack overflow.

Note: Beside the CASE statement for the different modes, the procedure contains an IF statement to check if the offsets are representable in a byte. However, this is only necessary because the code generation optimizes access of components of structured types.

```
PROCEDURE Load(VAR fat: Attribut);
  (* Generate code to load the value of fat onto the expression stack *)
BEGIN Assert(SizeType(fat)<=2);
```

```
WITH fat DO
  CASE mode OF

    localMod, globalMod, externalMod, addrLoadedMod:
     IF addr< = 255 THEN
      CASE mode OF

        globalMod:
         IF SizeType(fat) = 1 THEN EmitPacked(LGW, addr)
         ELSE Emit(LGD); Emit(addr) END |

        externalMod:
         IF SizeType(fat) = 1 THEN Emit(LEW); MarkVarAddr(fat)
         ELSE Emit(LED); MarkVarAddr(fat) END |

        localMod:
         IF SizeType(fat) = 1 THEN EmitPacked(LLW, addr)
         ELSE Emit(LLD); Emit(addr) END |

        addrLoadedMod:
         IF SizeType(fat) = 1 THEN EmitLSW(addr)
         ELSE
           IF addr = 0 THEN Emit(LSD0)
           ELSE Emit(LSD); Emit(addr) END
         END;
      END
     ELSE LoadAddr(fat); Load(fat)
     END |

    absolutMod:
     EmitLI(addr);
     IF SizeType(fat) = 1 THEN Emit(LSWn) ELSE Emit(LSD0) END |

    constantMod: EmitLI(value) |

    procedureMod:
     Emit(LIW); LinkageMark;
     Emit(procPtr↑.globmodp↑.modnum); Emit(procPtr↑.procnum) |

    loadedMod: (* skip *) |

    indexMod:
     IF SizeType(fat) = 1 THEN Emit(LXW)
     ELSE (* size 2 is possible ! *)
       Emit(UADD); Emit(LSD0)
     END |

    byteIndexMod: Emit(LXB); Assert(SizeType(fat) = 1) |

    doubleIndexMod:
```

```
        IF SizeType(fat) = 2 THEN Emit(LXD)
        ELSE (* size 1 is possible ! *)
          UConstMul(2); Emit(LXW);
        END |

      doubleConstMod: Emit(LID); Emit2(r1); Emit2(r2)

    ELSE (*illegalMod, stringConstMod, stringTemplateMod*) CompilerError
    END;
    mode := loadedMod
  END
END Load;
```

The two procedures

```
PROCEDURE PreAssign(VAR fat: Attribut);
PROCEDURE Assign(VAR desAT, expAT: Attribut);
```

are used to compile assignments. The procedure PreAssign prepares an attribute so that a subsequent value may be loaded and assigned with the procedure Assign. This preparation is necessary since the attribute for the destination need not be such that a corresponding store instruction exists. The compiler generates such attributes when it postpones the code generation in the hope for possible optimizations. E.g. if the compiler scans record fields, it delays the addition of offsets to an already loaded address. If the compiler needs to add further offsets, it adds them at compile time and generates code for an addition only once. This optimization may cause offsets to be out of the range of existing store instructions. PreAssign checks that either the offset is in the range of existing store instructions, or it will generate code to add the offset to the address.

Assign generates the code for the assignment

desAT := expAT.

The many self-checking calls of procedures like CompilerError or Assert may be surprising. Additional modes like illegalMod are introduced and the code generation itself runs with all runtime checks enabled. The reason for this apprehension stems from the experimental environment where the machines were developed. Any hardware error should cause the compiler to trap or to write an error message. Under no circumstances should an object code file be produced which does not correspond to the Modula source program. The achieved confidence in the compiler is much more important than the lost compile speed. The compiler has proved to be very reliable.

The WITH statement

The designator in a WITH statement must be evaluated only once. The compiler reserves a memory word in the local data segment to store the address resulting from evaluation of the designator. This word is accessed with the normal instructions for local data. However, the code generating pass examines, whether reproducing the address is correct and more efficient than to save it. It uses an attribute variable to remember the designator (and how to reload it) for use local to the WITH statement.

4. Compiling control statements

4.1. Statements with specific high level instructions

To design an instruction set well suited for compiling high-level languages is a real challenge. The temptation to optimize special features is great. The decision of which language features should be supported by special instructions must be carefully weighed. Primary candidates for memory (and time) optimizations are those instructions generated by the compiler. Compiler generated instructions may occur thousands of times in memory, although the user may never suspect their generation by the compiler.

The three main criteria to include a Modula-oriented instruction were:

Time savings
Memory savings
Simplicity of code generation

By using one single instruction for complex but important language features, instruction fetch time and memory can be saved. The memory savings are always worthwhile, even though memory is getting cheaper. In addition to the gains in memory space, more dense code leads to shorter instruction fetch time. We also felt that the simplicity of code generation is an important goal.

The literature offers methods to analyze relative occurrence of different instructions. Instruction pairs occurring most often are candidates for new instructions. We prefer the simpler solution to look into the compiler; instructions which are generated as sequences by the compiler, form tuples which occur frequently. Optimizing such instruction tuples directly eliminates (most of) the needs for additional peephole optimization.

Our experience is that code generation may be simplified through a machine with an appropriate instruction set. However, defining a new instruction set is much more work than generating code for a machine with a baroque instruction set. The total amount of work is not minimized by designing one's own instruction set; however, the reward are nice structures and nearly optimal code.

Remark: many of the complications are caused by the need of optimizations.

Second remark: it is much more effective to have an instruction set well-suited to high-level languages, than to have some special instructions.

Special instructions created for the operating system, like for disk access, need not be optimized with respect to space requirements. They will occur rarely. It is valuable to make certain operating system instructions fast. The operating system has the right to demand special functions (with regard to IO).

4.1.1. The CASE statement

Most computers have some kind of an indexed jump instruction. We designed our indexed jump instruction in such a way that it does everything that needs to be done for executing a Modula CASE statement. The "jump" table is placed after the code resulting from the statement sequences of the different cases. The compiler simply generates the code. At the end, the compiler updates the case instruction; it does not need to move the code.

An explicit enter case (ENTC) instruction is generated at the beginning of the case statement. The enter case instruction contains an offset which points to the jump table. If the case expression value is outside the range of the jump table, the enter case instruction jumps to an ELSE part. Other non-existing case labels cause the compiler to introduce jump addresses to an ELSE part into the jump table; such an ELSE part is treated like the other cases. If no explicit ELSE part is provided, a trap is generated as ELSE part.

The end of the CASE statement is also the end of the jump table. This end address is pushed onto the (data) stack when the enter case instruction is executed. The exit case (EXC) instruction pops the address and performs the jump to the end. This results in a short (1-byte) jump (return) instruction and in a simplification of the compiler.

The statement

```
CASE exp OF
  c1: s1 |
  c2: s2 |
    ....
  cn: sn
  ELSE se
END
```

is compiled into the code sequence shown below; the ENTC instruction computes the end address at runtime: end := tab+2*(high-low+4), where low is the smallest case-label value and high the largest.

```
      code(exp)
      ENTC tab
c1:   code(s1)
      EXC
c2:   code(s2)
      EXC
      ...
cn:   code(sn)
      EXC
els:  code(se)
      EXC
```

```
tab:  low
      high
      els
      c1
      c2
      ...
      cn
end:
```

The difference between the EXC instruction and the method used for EXIT statements, a jump with its address encoded in the instruction stream, reflects also the difference between the CASE and LOOP statements. For the LOOP statement, time savings are considered important. However, CASE statements have several cases. The saving of code space gets priority over speed optimization. The price of this memory saving is the additional memory access to put the end address onto the stack.

4.1.2. The FOR statement

The FOR statement is a typical candidate to require a specially tailored instruction. Many computers have special instructions to optimize the FORTRAN DO-loop. These instructions usually serve exactly the FORTRAN language and its compiler; the instructions are hardly usable for another language. (e.g. the SOB instruction on the PDP-11 computer [PDP] is not too specific, but it forces the compiler to keep a dummy control variable in a register). Our solution might cause the same problems to implementors of other languages. However, for Modula it is well suited.

The VAX supports the FOR loop with an add compare and branch (ACB) instruction; However, this instruction does not work properly if arithmetic overflow occurs.

Why use special instructions?

- A FOR loop could also be programmed using IF and WHILE statements, so it could be compiled into that sequence. The special FOR instructions are surprisingly complicated. We find, however, that the special FOR instructions are preferrable in spite of their complexity. The FOR loop is the fastest executed loop on the Lilith machine, mainly because it needs only a minimal number of instructions to establish the loop.

- The loop range could include the maximum value of the control variable type. Should the FOR statement cause arithmetic overflows or should the arithmetic wrap around the maximal value? Special FOR instructions can solve this problem. This is particularly severe for small integer ranges.

Consider the examples

```
FOR i := 2 TO 1 DO s.. END
FOR i := minint TO maxint DO s END
FOR i := maxint-(n*3)-2 TO maxint-1 BY 3 DO ... END
```

The first two examples show that a FOR statement can be executed 0 to maximum range times, a variation of maximum range + 1 executions.

The third example shows a case which should also be handled without overflow.

Once the goals of specific FOR instructions are defined, defining their exact behaviour causes no further problems. The specific instructions result in simple code generation. A FOR1 instruction is placed at the loop entry and a FOR2 instruction handles the end of the loop.

The statement

```
FOR i := e1 TO e2 BY c DO statementsequence END;
```

is compiled into the following sequence of instructions.

```
        load address of i
        load e1
        load e2
        FOR1 direction  (0 for up; >0 for down), b
a:      statementsequence
        FOR2 c, a
b:
```

If the FOR1 instruction allows to enter the loop, it moves the three loop parameters from the expression stack to the data stack. The FOR2 instruction fetches the three loop parameters from the data stack and, at loop exit, it removes them from the data stack. The step of the loop is a constant; for efficiency it is not put onto the stack but into the code sequence.
However, allowing constant steps only might be a little too restrictive.

The compiler has to know the nesting depth of FOR statements at any point in the program. If an EXIT statement introduces an additional exit from a FOR loop, the FOR loop parameters must be removed from the data stack. (The compiler generates three DECS instructions; no special instruction is defined, since it occurs quite rarely).

Length of code generation part for the FOR statement in the compiler:

Lilith:	50 lines	(26 essential lines; 20 lines for run and compile time error message 4 lines comment)
PDP-11:		
Pass 4:	125 lines	(105 essential lines; 10 lines for compile time error messages; 10 lines comment)
Pass 5:	21 lines	(17 essential lines; 4 lines comment)

However, the PDP FOR statement is also implemented for cardinals > maxint, and no runtime subrange checks are needed.

Run time benchmark: (counting the executions done within the same time interval)

	Lilith	PDP-11/40	Alto-2	MC68000
WHILE loop	334	185	116	275
FOR loop	422	230	172	320
Gain of FOR loop	26%	24%	48%	16%

Lilith takes its gain from the elimination of memory accesses in the computations of the control variable and in eliminating some execution fetch time.

The PDP-11 can eliminate memory cycles too, by holding more information in the registers. This results not only in reducing memory requests, but also in reducing code length by register addressing mode.

The Alto machine is programmed in Mesa. This language, like Pascal, does not allow FOR statement steps other then 1 OR -1; a special machine instruction would not need a specification of the step, the ending value of the FOR statement is always reached exactly and no wrap around or overflow can occur.

This measurement tells us that the Language designers of Pascal and of Mesa were wise not to include steps in the FOR statement. This allows faster code for the simple case while complicated cases must (and can) be hand-programmed with IF and WHILE statements.

4.2. Statements compiled to simple instructions and jump optimization

In a structured programming language, short-distance jumps occur much more frequently than long-distance jumps. The code size can be considerably reduced, if short addresses can be used for short jumps. To accomplish this, a short jump distance has to lead to a short instruction. This is realized in many computers with a jump address relative to the program-counter, that is, the jump instructions use addresses relative to the address of the instruction itself. However, how do we determine whether or not a short instruction will suffice if the jump's direction is forward?

Modula has no GOTO statement. All program execution control is expressed by well structured IF, WHILE, etc. statements.

Given that restriction we know that jumps will only occur as a result of the compilation of control statements. Since the location of the jump and the destination of the jump are part of the same statement, we know that we can treat each of them separately without regard to its context, nor to its components. In addition, using program counter relative instructions gives us a free hand to move the code of components up or down in store without change. This allows us to compile nested statements attractively by the use of recursion.

An unfortunate violation of the well-structured method to generate jumps is the EXIT statement. Renouncing the use of short jumps for the EXIT still allows usage of the advantages given from the language structure. EXIT statements occur significantly rarer than the other control structures which need generation of jumps. We accept the restriction to keep the algorithm simple and optimize only the frequent cases.

4.2.1. The REPEAT statement

Implementing an optimal jump instruction for the REPEAT statement is refreshingly simple. It requires a single, conditional backwards jump. The jump distance is known when the instruction is emitted. Hence it is known whether a short (2-byte) or a long (3-byte) instruction is required.

4.2.2. The WHILE statement

The WHILE statement has the following form

```
WHILE exp DO s END
```

The resulting code is

```
a:  code(exp)
    JPC b
    code(s)
    JMP a
b:  ...
```

We know that short jumps are much more frequent, and therefore leave space for a short instruction whenever its eventual length is unknown. This means that the subsequent block of code will have to be moved forward by one location if it later turns out that a long jump is required. We present the necessary steps in terms of a top-down compilation routine in familiar style. pc is the global counter for the code; startPC, jumpOutPC, and forwardOffset are local variables of that routine; N is the maximal jump distance available to the short version of the jump instruction.

```
IF sym = "WHILE" THEN GetSymbol;
  startPC := pc; Expression; jumpOutPC := pc;
  Emit(JPFC); Emit(0);
  StatementSequence;
  forwardOffset := pc-jumpOutPC + 1;
  IF (pc-startPC)>(N-1) THEN (*backjump is long*)
    IF forwardOffset>(N-1) THEN
      (*two long jumps*)
      MoveCode(jumpOutPC + 2, pc-1, 1);
      Insert(jumpOutPC, JPC); Insert2(jumpOutPC + 1, forwardOffset + 2)
    ELSE
      (*only backjump is long*)
      Insert(jumpOutPC + 1, forwardOffset + 1)
    END;
    Emit(JP); Emit2(startPC-pc)
  ELSE (*two short jumps*)
    Emit(JPB); Emit(pc-startPC);
    Insert(jumpOutPC + 1, forwardOffset)
  END;
  GetSymbol
END
```

Emit2 or Insert2 means to handle 2 bytes.

4.2.3. The IF statement

The IF statement presents a much more formidable case because it generates forward jumps that "cross" each other. Since in Modula-2 a whole cascade of cases can be obtained, we record the locations of the incomplete jump instructions in a list structure.

The statement

```
IF e1 THEN s1
ELSIF e2 THEN s2
  ...
ELSIF en THEN sn
ELSE s
END
```

is compiled into the code

```
        code(e1)
        JPC c2
        code(s1)
        JMP ex
c2:     code(e2)
        JPC c3
        code(s2)
        JMP ex
c3:     ...
        code(en)
        JPC el
        code(sn)
        JMP ex
el:     code(s)
ex:
```

After the statement is compiled, a pass is made over the generated code in the reverse direction to determine where short jumps have to be replaced. Only in a third step is the actual code moved, and the jump addresses inserted.

For a complete reference of this optimization see Appendix 6.

4.2.4. The LOOP and EXIT statements

The LOOP and EXIT statements are quite exceptional. The EXIT causes control to leave the enclosing LOOP statement. The EXIT statement is not syntactically part of the LOOP statement. The compiler must discover and take into account the environment of the EXIT statement.

The LOOP statement is much like the REPEAT statement. The jump addresses for EXIT's, however, are only known after the entire LOOP statement has been analyzed. We decided not to optimize EXIT statements. All EXIT statements must be compiled into a forward jump. When EXIT statements are compiled, the address of the jump instruction is entered in a table which belongs to the innermost LOOP statement. These tables are handled in the code emitting module; they need updating whenever the the code is moved.

Note: If an EXIT statement jumps out of a FOR loop, it must adjust the top of the data stack. (The WITH statement cause no problems. If an address is to be remembered, it is stored in the local data segment and not on top of the stack.)

4.2.5. Keeping track of code moves

Two methods for moves of code by the compiler are used:

Rule for the simple case:

> The generating procedure does everything itself; it may move the code, it marks and updates locations, and it does the neccessary bookkeeping itself whenever it moves a piece of code. This scheme is used for IF, WHILE etc statements.

More general rule, used for EXIT statements and locations where linker fixups are to be requested:

> The code emitting module keeps track of the move. Both, the marking of locations and the update is done through calls of emitting procedures. No other module is involved in the code moving.
>
> *mark:* to remember the address of a location where later something must be updated. The recorded address must change when the code is moved.
>
> *update:* to insert something into a marked location. The address of the location may have been changed since it had been marked.

The mark and update procedures are implemented in the same module as the code moving procedure. The code moving procedure keeps track of marked addresses.

Using these two methods, any compiler part may move the code which is generated by other parts without further respecting their bookkeeping needs. The code generated by one compiler part may be moved by others without notice.

The following module CodeSys shows the simplified code emitting module

```
MODULE CodeSys;
 (*everything mentioned here is exported*)

 VAR pc: INTEGER; (*exported read only*)

 PROCEDURE Emit(i: WORD);
 PROCEDURE Emit2(i: WORD);

 PROCEDURE Insert(at: CARDINAL; i: WORD);
 PROCEDURE Insert2(at: CARDINAL; i: WORD);

 PROCEDURE MoveCode(from, to, displacement: CARDINAL);
  (* decrements pc if necessary *)
```

```
PROCEDURE MarkLong(VAR lastPc: CARDINAL);
 (* insert the program counter in a "table". lastPc must be the (provisional)
   program counter of the last mark of the used "table",
   or 0 to start a new "table" *)

PROCEDURE UpdateLong(lastPc: CARDINAL);
 (* Update "table" of long jumps.
   the MarkLong, UpdateLong pair is used for EXIT statements *)

PROCEDURE LinkageMark;
 (* enter pc in the table of linkage marks; it is updated implicitely *)

END CodeSys
```

The complete CodeSys module contains some initialization routines and further pairs of mark and update; however, they all use the same basic principles. In spite of general information hiding ideas the simple mechanism for track keeping with cardinals avoids separate list structures. Such a module is only used by a limited number of code generation clients which can be assumed to be correct.

We will not describe completely the trivial (but verbose) mechanism of how the marks have to be updated on moves. However, a compile time saving optimization complicates the algorithm. The optimization relies on:

All moves are forward.

The IF statement leads to sequences of moves with non overlapping areas which are ordered.

All nonupdated marks are kept internally in an ordered sequence, using the same order as the nonoverlapping areas of IF statements. Whenever a move is called, the last move position and the internal position of the mark information is remembered. On the next move the compiler checks first to see if it can start the search for addresses and mark information at the remembered position. Most often this is the case and the search may continue at that position.

4.2.6. The value of the short and long jump optimization

Similar optimizations lead to substantial savings of memory in the Modula compiler version for PDP-11 machines (10%); however, the savings for the Lilith machine are only about 3%. The PDP instructions have a length of 1 or 2 words. The difference in the Lilith machine is only 2 versus 3 bytes. We feel that an optimization of 3% is not important in a special program; however, when generated by the compiler, those 3% *are* important, because *every* program will gain 3%. This limitation of the

optimization had been predicted; the relative number of jump instructions gives an upper limit to the gains of this optimization.

The other reason this optimization is less valuable on Lilith than on the PDP-11, is that it competes with other optimizations; for example special instructions for the FOR or CASE statements.

Short and long jump optimization is frequently mentioned in the literature (e.g. [Rob]). Most authors present algorithms which perform all possible optimizations; these algorithms are quite complex. Our algorithm does not consider modification of instructions in statements which are already scanned. It is based on the fact that jump addresses are always relative and that for structured statements the location of the jump and the destination of the jump are part of the same statement.

Note: Jump instructions for EXIT statements and addresses for linker fixups are updated after the corresponding statement is scanned, but no opcodes need to be modified.

Another often mentioned optimization is peephole optimization. We prefer optimization at a time when the program's structure is known. It is not forbidden, however, that a subsequent peephole optimization can be performed on the output of the compiler.

Counting the long jumps in some actually compiled object programs yields the following results:

> The complete code generation part of the compiler compiles itself with only 2 statements (beside EXIT statements) needing long jump range.
>
> Note: not generating long jumps at all would be an awful restriction. If programmers were forced to use smaller local procedures in non-adequate cases, they would have to use more intermediate level variables.
>
> The long jumps, which could theoretically be replaced by short ones, but which our simple algorithm ignores, amount to less than 0.1% loss of memory space in the code (rare EXIT statements).

The jump optimization needs surprisingly few lines in the compiler: Mainly the code in Appendix 6, and the code for performing the code moves and the necessary updates. However, this code leads to a substantial increase of the complexity of the module interfaces.

Optimizations omitted:

> Jump to jump instructions are changed. This could also be done by a peephole optimization. Such an optimization would speed up the programs, but it would not save space.
>
> Code which can never be executed is omitted. This is done for IF statements with constant Boolean expressions; it could be improved to eliminate code after an EXIT statement. This, of course, could not speed up programs but would save space. To be fair, eliminating never executed code in IF-statements is not

considered an optimization. It allows to include debug statements in the program without introducing concepts like *conditional compiling.*

Currently our compiler performs a number of valuable optimizations omiting the more exotic and difficult ones. Special optimizations which cannot be done in source level and optimizations which happen to well structured programs are preferred.

4.3. High-level instructions versus simple instructions

Since the whole architecture is defined in view of Modula, we were surprised how few instructions are introduced to map specific Modula concepts to the machine. Other instructions are introduced to implement a Modula concept without being explicitly the result of compiling a specific statement.

Instructions for formal procedures, processes, data allocation, parameter copying, accessing arrays, and the index tests are well suited for Modula. However, they would certainly also be useful for any other high-level language. These instructions do not result in isolated high level features but they are all organically related to each other and fit into the framework of a consistent architecture. Such instructions are therefore not described in this chapter but, rather, in the earlier chapter *Overall structure and procedure call.*

The FOR statement requires a special control structure of which the jump is only a minor part. We provide an extra machine instruction that performs the loop index counting in addition to the jump control logic.

The CASE statement needs a table with the offsets to the variants. At the end of each case variant, a jump to the end of the CASE statement is needed. The compiler generates an exit case (EXC) instruction (1-byte) for this jump, saving even more memory than with optimized jumps. For these specific instructions, no special versions with byte offsets are defined.

The other control statements do not request specific instructions. For example, consider again the statement

```
WHILE exp DO s END
```

which is compiled into

```
a:  code(exp)
    JPC b
    code(s)
    JMP a
b:  ...
```

The jump and jump conditional instructions exactly match the while statement. If specific instructions were requested, we could call such instructions WHILE1 and WHILE2, but they would still act as jumps and conditional jumps. We might say that FOR and CASE are standard constructs of any high-level language. A matter of opinion.

5. The encoding of instructions

A good instruction encoding reduces the memory space required for object programs. Generating shorter programs not only saves memory, but also results in less execution time to fetch the instructions. Some mathematical methods allow very dense coding of the instructions. Such methods are described by C. Foster [FG], J. Wade [WS], and others. The proposal of A. Tanenbaum [Tan] considers hardware constraints more fully. He proposes a byte encoding. This encoding is not optimal in a mathematical sense but is a good compromise which can be built with real hardware.

The Lilith instruction encoding is similar to the proposal of Tanenbaum. However, the instruction set is more complete. The number of different formats for instructions is limited since each variant needs hardware support. Our instruction fetch hardware delivers the four least significant bits of the instruction byte in a special register of the micro machine.

In fact, not all of the 256 different opcodes are necessary instructions. This allows us to encode the more frequently used instruction sequences into one byte. More specifically, the most often used constant operands are encoded into the instruction byte itself. Hence, a complete byte for the operand alone is not needed.

It is very remarkable that on a word addressed machine the instructions start on byte boundaries. To enable this byte instruction stream, the instruction fetch unit reads the instructions from a special memory port and not from the usual 16 bit memory port for data.

The code of the Lilith computer has a significantly higher density than the code for most conventional instruction sets. Compiling Forest Basket's *puzzle* benchmark program [Bas] shows that the Lilith has the most dense code, with the exception of the Mesa [JW] architecture. This is mainly due to the short address fields. Also, comparisons of longer programs show that on Lilith the code needs less than half as much memory as the same program on the PDP-11. This measurement was made in a fair manner. Our PDP-11 Modula compiler generates an excellent code and also uses short branch instructions. Moreover, the PDP-11 is known as a Pascal friendly machine.

The following table shows figures about the distribution of instruction length. The numbers stem from 30000 inspected instructions, the editor, the windowhandler, and the codegeneration pass of the compiler. The table shows that the encoding is well done.

1-byte instructions	20747	69.3%
2-byte instructions	6289	21.0%

3-byte instructions	2746	9.2%
Others	135	0.5%
Total	29917	100.0%

To really judge the density, the instructions with several variations in length must be compared.

	1-byte	2-byte	3-byte instructions
Load immediate	3094	1031	322
Load and store global	1827	862	-
Load and store local	6509	161	-
Load and store indirect	2658	195	-
Jumps (without FOR, CASE)	-	2172	150
Calls	473	415	1492
(Total 29917 instructions)			

The table shows that the short versions of most instructions are preferred. The best results are the figures for accessing local data. The other lines are also quite impressive. However, the call instructions do not seem to be optimal. The density of the instructions could be further improved if a call external instruction with procedure number encoded in the opcode were introduced. The figures for load immediate show that small constants occur much more frequently than large ones. This is especially remarkable since most ASCII values need a two byte instruction.

A similar measurement is made by G. Mc. Daniel [Dan]. This paper also allows comparison of the behavior for different possible encodings of the instructions.

The next table shows which subject the instructions belong to. With it we can see that local variables are handled more frequently than globals. Stores occur less frequently than loads. Compare this table to the corresponding table in *Lilith: A Personal Computer for the Software Engineer* [Wirl].

		%
Load immediate	4560	15.2
Load address	840	2.8
Load local	5213	17.4
Load global	2098	7.0
Load indirect	1834	6.1
Load indexed	224	0.7
Load external	601	2.0
Store local	1555	5.2
Store global	591	2.0
Store indirect	1019	3.4
Store indexed	369	1.2
Store external	175	0.6
Operators	1885	6.3
Comparators	1260	4.2

Jumps	1953	6.5
Short circuit AND/OR	324	1.1
FOR/CASE	333	1.1
Calls (without returns)	2380	8.0
Others (aloc, rtn, entr,)	2703	9.0
Total	29917	100.0

It is no surprise that loading local values and load immediate are the most frequent instructions. It might be a consequence of structured programming that 8% of all instructions are calls. Some special instructions like the FOR, the CASE and the short circuit AND/OR do not occur frequently. However, these instructions are worth beeing defined either because of speed or because they reduce the compiler complexity. Generally, the complex instructions occur much less frequently than the simpler ones.

It is sufficient to consider the static distribution of the instructions to determine the memory needs. However, speed is dependent on the dynamic distribution. We anticipated that the dynamic distribution of most instructions would be much like the static and tried to verify this with an appropriate experiment.

The dynamic instruction execution frequency is measured by interpreting the program. The interpreter counted the instructions. External calls to procedures of other, not measured, modules were not interpreted but the procedures were really called. This resulted in statistics on the dynamic occurrence of the instructions in the measured modules only.

Note. In interactive use, the processor loops in a busy waiting loop for input. Our method allows to avoid counting those instructions.

This interpreter is too slow to interpret a reasonable sample. G. Mc. Daniel uses an other method to get the dynamic instruction frequency of Mesa programs. He inserted a counter in the microcode.

Some figures of the dynamic instruction frequency are shown in Appendix 4.

Our dynamic measurements showed enormous differences for similar instructions. This proved to be a peculiarity of the inspected programs. We were not willing to concentrate so much on measurements with large program samples, as G. Mc. Daniel and R. Sweet [SS] did. These two papers analyze the Mesa instruction set. The first paper analyzes the static instruction frequency, the second paper analyzes dynamic instruction frequencies. It is important to us, that their main conclusions from analyzing dynamic instruction frequencies show the same importance of short addressing for the first few local and global variables as ours. The access distribution of these variables shows no significant difference between the static and the dynamic measurements.

The Mesa instruction set uses short instructions for the first variables of procedures

and modules, as does the Lilith instruction set. The Mesa compiler rearranges the order of the variables so that the most frequently accessed variables get the first memory locations. The Modula compiler performs no such rearrangements. Our measurements show that variables, which cannot be addressed with short instructions, are used rarely. However, if less offsets would lead to the short instruction format, the value of such a rearrangement is increased.

The good encoding of instructions has further benefits than just measured quantitative improvements: Because most instructions use one byte only, many more complex instructions can be omitted. If an operation can be programmed with two primitive one byte instructions, there is no need for a more complex two byte instruction executing the same operation.

6. Conclusions

Designing an architecture and an instruction set is an intellectual challenge. A significant portion of the success of the Lilith project is due to the elegance and simplicity of the chosen architecture.

Specific high-level instructions have been used in object machine architectures for a long time. In addition to single instructions, the general architecture was designed explicitly with the compiler and its high-level language in mind. It is important that the Lilith architecture was designed concurrently with the compiler.

The architecture is defined to support the module structure of Modula. While the module concept is fundamental for the language, it is equally important to support modularization by the architecture of the machine. A program's subdivision into modules does not only introduce the need to access external objects, but introduces a new concept for globals objects. To be global is no more an absolute property. Each module has its own global objects, which may be accessed while the module's procedures are executed.

We defined special instructions which directly support compilation of specific Modula elements. Some of these instructions just simplify the code generation task. Other special instructions are tailored to specific constructs of the language (FOR, CASE); they replace long sequences of other instructions and result in space and time savings.

Most control statements of Modula are quite simple, they are compiled with jump or conditional jump instructions. The compiler performs an optimization to use short address fields for these instructions. The jump optimization algorithm is simple, because it knows the structure of the Modula statement which required the jumps.

Because of the style of modular programming, data are distributed to several procedures and modules. Every program unit owns only relatively few data elements, which can be accessed with small offsets. Encoding small offsets into the opcode byte of some instructions reduces code length considerably. The largest space savings of the Lilith instruction set are due to this encoding of the instructions.

The architecture is a compromise; it does not necessarily lead to an ideal compiler. It tries to find an optimum between fast and short code and a simple compiler structure. Complications, which do not result in better code, are eliminated; but whenever better code can be generated, the additional compiler complexity is accepted.

In general we have chosen a good compromise between generality and optimality for

Modula. However, it is also unwise to make too many concessions to the compiler. We learned a lot about balancing between general and special purpose instructions.

The benchmark test shows that the Lilith is relatively fast. The results are better than what would be expected when only processor speed is considered. This improvement is therefore achieved through the quality of the architecture.

Defining the architecture with a Modula program which simulates the Lilith has proven to be very useful. This Modula program served as an interface between compiler writer and microcode programmer. The microcode programmer is not in the best position to define instructions because he does not see the detailed needs of the compiler. Sometimes during this project, the microprogrammers did not wait for a clear definition of the microcode in the Modula interpreter. This always caused trouble for the compiler.

The compiler has proved to be very robust. The partitioning of the compiler into four passes helped to separate code generation from the rest. However, the idea that the code generation pass should be the only machine dependent pass is only of very limited value, since addresses must be determined before code can be generated. It would be overly puristic to introduce an additional pass only to separate these machine dependent functions from the machine independant ones.

Note: The compiler is not only robust but also very reliable. Since its first internal use (more than two years ago) only three errors in code generation have been detected.

6.1. What can be done better next time

Global aspects

There are not enough free opcodes. Further extensions could need more additional instructions than there are free opcodes. Introducing a second map-rom and a corresponding indexed-jump micro instruction in hardware could allow the introduction of an escape instruction which executes faster.

Virtual addresses are needed; the current access of the upper memory bank is a patch. A whole chapter about virtual addresses follows.

The machine is sometimes not interruptable for excessive time intervals (bitmap instructions).

Disabling traps for arithmetic overflow should be done on a module basis instead of on a process basis. The current solution is useful for main programs, but system modules cannot know if they cause traps or not. A possible solution would be to reserve a bit in a fixed location per module, e.g. in the code frame. On overflow the module number serves to get the corresponding trap disable bit.

Features not demanded by the language Modula are neglected in the instruction set. Bit-operations like a constructor `BITSET{n..m}` with `n` and `m` being either expressions or variables cannot be generated easily. Also, instructions for sets of more than 16 elements are missing. The instructions for multiprecision cardinal arithmetic are useful to program a package, however, a set of double precision integer operators would sometimes be useful too.

An exeption mechanism could be included which is more specific than generating traps. Introduction of an exeption concept to Modula is not yet been prepared. Such an exeption concept is particularly worthwhile for implementing remote procedure calls in a transparent way, as proposed by Nelson [Nel]. When an exception is raised, the machine has to follow backwards the dynamic link chain until a procedure is found which catches the exception. We propose an exeption mechanism similar to the one of Mesa [MMS]: The exeption is caught by a construct which is syntactically bound to the call statement. An alternate solution is to introduce a suffix to the statement part of the procedure which is executed when an exception occurs. This alternative is found in the programming language Ada. The Mesa solution reflects the programmer's intention better. However, the full Mesa *signal* mechanism is too general.

Reading the initial bootstrap program from disk by microcode introduces device dependency into the microcode. Using a read only memory to get an initial loader could allow more flexibility.

Modula specific instructions

The FOR and CASE instructions are defined for signed arithmetic only. They should be improved to include unsigned arithmetic, then they would serve better the needs of Modula. However, they could also be defined in a more general way.

The MOD instruction for signed operands is missing.

Encoding of instructions

The CX instructions should have special opcodes for small procedure numbers. The measurements have shown that external procedures are called as often as local procedures, which we did not expect. It is not possible to use opcodes to encode module numbers. They must all be encoded in the same way i.e. in a byte, since the loader (or linker) must update them.

Note: The Mesa instruction set allows to call an external procedure with a one-byte instruction, but it needs one more indirection.

IO and Screen-IO instruction opcodes do not need to be encoded in a single byte.

Virtual Addresses, a proposal

Memory has proved to be a crucial point on Lilith, as on any computer. On Lilith, there may be a need for more memory, for more flexibility in using memory, or for memory management.

The missing flexibility is a severe restriction for system programs. Swapping, protection, or just increasing memory size are less important considerations.

Since data addresses are absolute it is not possible to move the used storage. This results in a severe restriction for system modules like the file system or the window handler: they are not allowed to use dynamic storage. In Medos terminology, a module may only ask for memory when it belongs to the top level.

Reason for the restriction:

The dynamic storage of the top level *(user)* program must be returned when the program terminates. The storage handler should separate dynamic memory according to the owner programs. The two obvious methods for separation do not work:

If one big heap for all levels is used, the *user* programs may cause a fragmentation of the memory which cannot be cleared later by the *system* programs.

If several distinct memory segments are used, the heaps are all of fixed size. If a heap overflows, it cannot be moved into a larger free memory area.

Terminology

Virtual addresses:

> Virtual addresses used in programs do not denote physical memory addresses but. A mapping function exists from virtual to phisical addresses.

Virtual memory:

> Parts of the memory may be swapped onto a secondary storage device. The memory units swapped out are either *segments* or *pages*. Pages have a fixed size, segments are memory parts with a size determined by the running program. Virtual memory requires virtual addresses.

Typical relation about address space:

> Virtual address space >= Virtual memory space >= Physical memory space

Proposal for Lilith

We consider virtual addresses to be important, but shall not consider the problem of virtual memory. Neither do we discuss paging and protection mechanisms.

Any virtual address implementation on Lilith should consider the module structure. The frame table with all pointers to data frames and the G-register with the pointer to the data frame of the current global module suggest the main idea: all memory mapping functions are done through the frame table. When registers (G, L, P, ..) are loaded, the mapping is to be already done. The register then points to real memory, not to virtual memory. This allows to ommit a time consuming memory request for the mapping function (from virtual to physical addresses) in most cases.

Not less important for this proposal, is to keep the 16-bit words for data and addresses. The proposal is so designed that only one additional bitslice is used. This limits real addresses to 20 bits, which we consider enough for a simple personal computer. This enlarges the (addressable) memory by a factor of 16.
If this memory is still too small, it is also possible to use two additional bitslices, but then please consider also other ideas like 32-bit machines and multiprogramming.

The frame table also serves for memory mapping. Consider the first 256 entries of the frame table to point to global data frames. The further entries may be used for heaps and bitmaps etc. A frame pointer points to a continuous area of memory which has a real address divisible by 16.

We introduce 3 kinds of addresses: virtual addresses, P-relative addresses, and G-relative addresses. Real addresses do not need to be visible to the program. The L-register cannot be used as a frame pointer because there are too many procedure invocations. Also, the alignment condition for frame pointers prevents the use of frame pointers in the L-register. To allow short (16-bit) addresses in process stacks, the P-register is used.

The virtual addresses are used for pointers and for parameter passing. Structured data may be accessed with frame relative (G or P) offsets. Conversion from a relative address to a virtual address means:

Load the 16-bit relative address (the offset)
Load the (16-bit) index

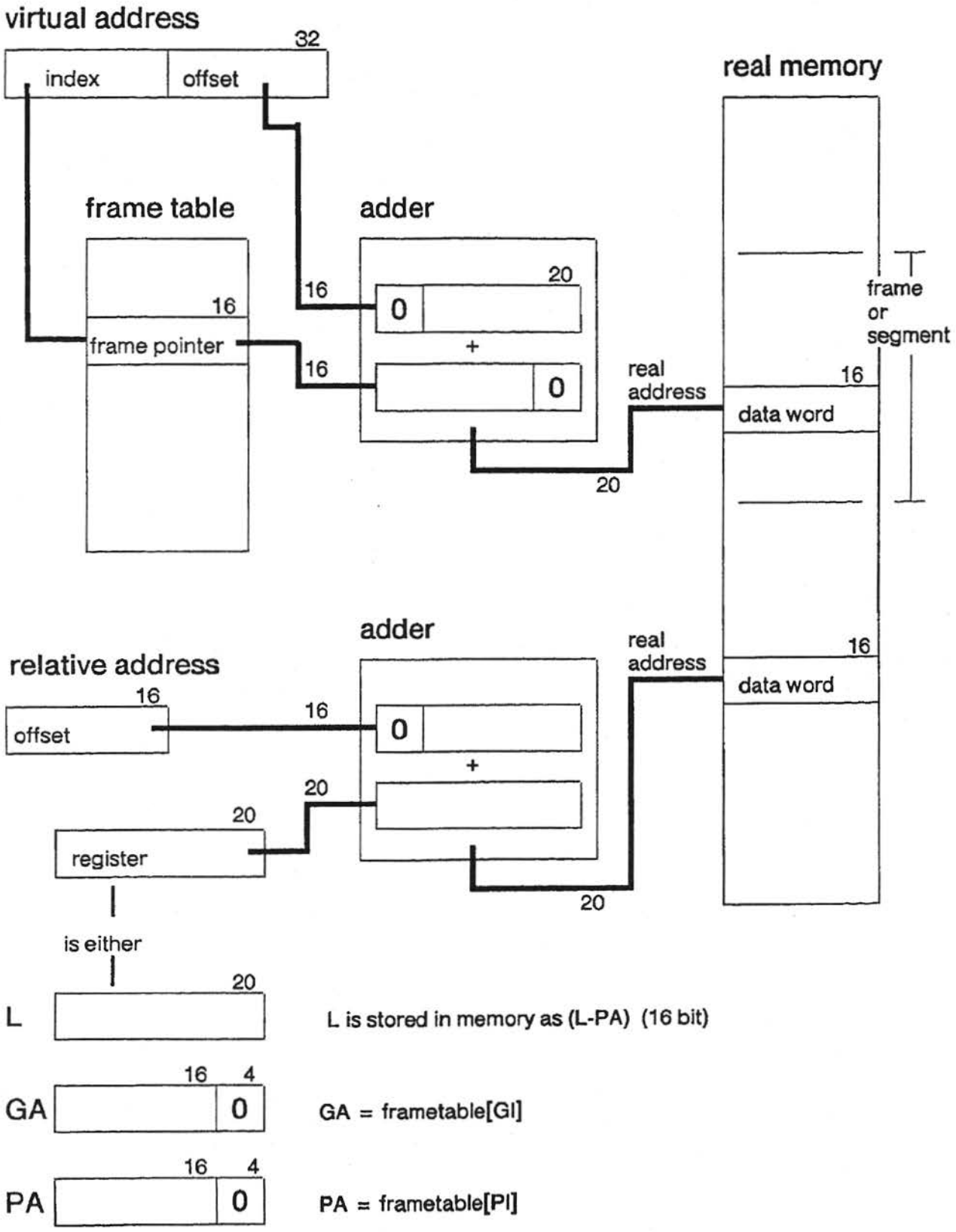
Two additional registers are used:

The G-register is split into a GI- and a GA-register. Also the P-register is split into a PI- and a PA-register.

The GI-register contains the module number. It is used for address conversion and is saved in the procedure marks on external calls. The GA-register contains the corresponding frame pointer and is used to access G-relative data. Similarly, the PI-register identifies processes and the PA-register is used to access P-relative data.

The "I" versions of the registers replace the old registers, however they are changed to be indices to the frame table. The "A" versions of the registers contain the

corresponding entries of the frame table, expanded to be 20 bit addresses. The "A" register versions are transparent to the program.



"Virtual Addresses"

New LGI and LPI instructions are introduced. These instructions load either the module number or the index of the current process on the expression stack (the GI or the PI register respectively).

It is worthwile to use relative addresses. Relative addresses need only one word of memory, to load a relative address only one memory request is required. Conversion from relative to virtual addresses is mostly done without memory requests: Most often an LGI or LPI instruction replaces the access to the frame table.

For storing static and dynamic links, the PA-register is subtracted from the L-register, yielding a relative address.

The first word of a module's data frame points to the corresponding code frame. It seems obvious to represent it by a frame table index; for speed reasons we might replace it by a frame pointer. We think the speed gain is worth the additional complexity of the loader.

List of modified instructions

LSW0-LSW15	only LSW0-LSW7, but in virtual, P-relative and G-relative mode
SSW0-SSW15	same
LSW n, LSD n, LSD0	all 3 address modes
SSW n, SSD n, SSD0	all 3 address modes
LXFW, SXFW, MOVF	are no longer needed
LSTA	is G-relative
LXB, LXW, LXD	all 3 address modes
SXB, SXW, SXD	all 3 address modes
GB, GB1	is P-relative
ALOC	is P-relative
TRA	uses index in frame table (PI-Register)
LLA	virtual and P-relative mode
LGA	only virtual mode
LSA	relative (P, G is same) and virtual mode
LEA	no more needed
TS	virtual
PCOP	actual parameter address is virtual, local address P-relative
LIN	Load immediate NIL (32-bit; offset is 177777B)
LIN1	Load immediate -1
LGI	new
LPI	new
DDT, DCH, REPL, BBLT	Fonts and Bitmaps are denoted by indices, not frame pointers. The hardware has some words with offsets 0-n not used, but reserved for the software. Use 2 byte opcodes (to get free opcodes).
READ, WRITE, DSKR, DSKW, SETTRK	use 2 byte opcode (to get free opcodes).
LLW4-LLW15	replace by LLW4-LLW10 (to get opcodes)
SLW4-SLW15	same

For efficiency reasons, the most important instructions dealing with G-relative or P-relative addresses may be defined, but the less frequent instruction may require explicit conversion from relative addresses to virtual addresses.

Instructions causing storage overflow should be completely cancelled. The operating system can then enlarge the process stack, repeat the instruction, and continue execution.

Rationale:

We include the concept of virtual addresses in the processor's design instead of relegating it to a special memory management unit. This allows reduction of most accesses to a segment table (the frame table) and also reduces the necessary hardware components. (This advantage is also realized with the iAPX 286 processor in its so-called *Protected Virtual Address Mode.*)

Acknowledgements

I wish to thank Prof. N. Wirth for conceiving and guiding the Lilith project and for supervising this thesis. I thank S. E. Knudsen for his readiness for discussions. He, L. Geissmann and A. Gorrengourt participated in the compiler work. A special thanks to W. Winiger for microprogramming the Lilith architecture as it was defined by a Modula program and to R. Ohran for building the hardware. I thank F. Ostler for checking my English formulations; however, for any further errors I myself am responsible. Finally, I thank Dr. P. Schulthess for accepting to be my co-advisor and for his careful reading of my manuscripts.

References

[Amm]

U. Ammann. Die Entwicklung eines Pascal-compilers nach der Methode des Strukturierten Programmierens. *ETH Dissertation 5456, 1975.*

We learned quite a lot from this Pascal compiler.

[Bas]

F. Baskett. Puzzle: an informal compute-bound benchmark. *Widely circulated and run.*

This benchmark is used to compare the Lilith with some other computers.

[BS]

Miles A. Barel, John P. Strait. PERQ QCode Reference Manual. *Three Rivers Computer Corporation, Sept 3, 1980.*

The PERQ is another son of the Alto computer, the QCode is derived from the P-code.

[BKMRS]

David W. Best, Charles E. Kress, Nick M. Mykris, Jeffrey D. Russel and William J. Smith. An Advanced-Architecture CMOS/SOS Microprocessor. *IEEE Micro, Vol 2, No 3, Aug 1982, p10-26.*

This architecture may be compared to Lilith in many aspects.

[Bur]

The operational characteristics of the processors for the Burroughs B 5000. *Burroughs Corporation, Nov. 1961.*

The Burroughs computers were well suited for high-level languages a long time ago.

[Dan]

Gene McDaniel. An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies. *Symposium on Architectural Support for Prog. Lang. and Operating Sys. Palo Alto, Mar. 1982, p167-176.*

Analyzing dynamic instruction frequencies shows similar results than static instruction frequencies; Since the occurrences of instructions are parametrized, the paper contains relevant information for our purposes.

[Dij]

E. W. Dijkstra. Recursive Programming. *Numerische Mathematik 2, 1960, p312-318.*

This paper introduces the stack mechanism for handling of local data of procedures; further, it introduces the so called "display".

[Dis]

Walt Disney. Mickey Mouse.

Excellent literature, also different topics beyond computer science.

[FG]

Caxton C. Foster, Robert Gonter. Conditional interpretation of Operation Codes. *IEEE Transactions on computers, C 20, No 1, Jan 1971, p108-111.*

Every instruction has only very few successor instructions. A short encoding is used which allows only these few successor instructions and an escape instruction to occur after each instruction.

[Gei]

Leo Geissmann. Implementation of Separate Compilation in the Modula-2 Compiler. *Proceedings of the German Chapter of the ACM 11, Teubner, July 1982, p15-27.*

More information about the compiler.

[HP]

HP 3000 Computer Systems, General Information Manual. *Hewlett Packard, June 1981.*

The HP 3000 is a stack machine. It has a hardware implementation for the top elements of the stack.

[Jac]

Ch. Jacobi. Evaluation of the Lilith architecture in view of compiling Modula programs. *Proceedings of the German Chapter of the ACM 11, Teubner, July 1982, p67-80.*

Subset of this thesis.

[JW]

R. Johnson, J. Wick. An Overview of the Mesa Processor Architecture. *Symposium on Architectural Support for Prog. Lang. and Operating Sys. Palo Alto, Mar. 1982, p20-29.*

Mesa is implemented on the Alto computer. We learned a lot from Mesa.

[Kee]

W. M. McKeeman. Peephole optimization. *Comm. ACM, Vol 8, No 7, 1965, p443-444.*

A simple optimization idea. However, we do not perform a peephole optimization.

[Le]

Van Kiet Le. The Module: A Tool For Structured Programming. *ETH Dissertation 6153, 1978.*

This is about MODULA-1, not Modula-2. Shows distribution of long and short range jumps.

[MMS]

James G. Mitchell, William Maybury, Richard. E. Sweet. Mesa Language Manual. Version 5.0. *Xerox Palo Alto Research Center, CSL-79-3, April 1979.*

Mesa influenced the definition of Modula.

[NAJNJ]

K. V. Nori, U. Ammann, K. Jensen, H. H. Naegeli, Ch. Jacobi. The Pascal-P Implementation Notes. *Pascal - The Language and its Implementation, Edited by D. W. Barron, John Wiley.*

The P-code is another high-level language instruction set. The main goal of the Pascal-P project is portability, not most efficient execution.

[Nel]

B. J. Nelson. Remote Procedure Call. *Xerox Palo Alto Research Center, CSL-81-9, May 1981.*

We use remote procedure calls only as an example to speculate about the necessary power of an exception mechanism.

[PDP]

PDP 11 Processor Handbook. *Digital Equipment Corporation, 1973.*

We sometimes compare the code-generation for the PDP-11 and for the Lilith. The PDP-11 was used to bootstrap the Lilith compiler.

[Rob]

Edward L. Robertson. Code Generation and Storage Allocation for Machines With Span-Dependent Instructions. *ACM Trans. on Programming Languages and Systems, Vol. 1, No. 1, July 1979, p71-83.*

Handles short and long jumps. Shows that the problem is hard to solve.

[SS]

Richard E. Sweet, James G. Sandman, Jr. Empirical Analysis of the Mesa Instruction Set. *Symposium on Architectural Support for Prog. Lang. and Operating Sys. Palo Alto, Mar. 1982, p158-166.*

They put much more efforts in the measurements than we were able to.

[Tan]

Andrew S. Tanenbaum. Implications of Structured Programming for Machine Architecture. *Comm. ACM, Vol 21, Num 3, March 1978, p237-246.*

Excellent paper, independently describes a machine architecture with some ideas also found on the Lilith architecture. He analyzes a big sample of programs and proposes an instruction set which leads to dense code.

[TCLSB]

C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. R. Boggs. Alto: A personal computer. *Xerox Palo Alto Research Center, July 11 1979.*

The Alto computer may be called the father of the Lilith machine.

[VAX]

VAX Architecture Handbook. *Digital Equipment Corporation, 1981.*

Successful architecture, it is much more general than our approach; But it contains too many features for a personal computer.

[Wir1]

Niklaus Wirth. Lilith: A Personal Computer for the Software Engineer. *Proceedings of the 5th international conference on software engineering (1981) 2-17* and *Microcomputer System Design. Lecture Notes in Computer Science 126 Springer, 1982, p349-397.*

More general description of the Lilith project. It includes language definition, operating system overview, compiler and architecture overviews, and a description of the hardware structure.

[Wir2]

Niklaus Wirth. Programming in Modula-2. *Springer, 1982.*

Language definition.

[Wir3]

Niklaus Wirth. Modula: a language for modular multiprogramming. *Software - Practice and Experience, 7, 1977, p3-35.*

Language definition of MODULA-1.

[Wir4]

Niklaus Wirth. The programming language PASCAL. *Acta Informatica 1, 1971, p35-63.*

Modula gots most of its principles from Pascal.

[Wir5]

Niklaus Wirth. A solution to the problem of compiling short and long jumps. *Internal memo, 1978.*

Our short and long jump optimization is based on this memo.

[Wul]

William A. Wulf. Compilers and Computer Architecture. *Computer, Vol 14, No 7, July 1981, p41-47.*

Excellent paper, comes to different conclusions than we do.

[WS]

James F. Wade, Paul D. Stigall. Instruction design to minimize program size. *Proceedings 2nd Symp. on Comp. Arch, Jan 1975, p41-44.*

They find optimal encoding for memory saving, however, this is done with ignoring hardware costs.

Appendix 1: 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	ESC	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

The undefined opcodes in the instruction set:

```
 21B, 334B: reserved for use by the compiler
214B, 215B: reserved for use for arithmetic

237B second byte >3: reserved for floating arithmetics
246B second byte  0: reserved for debugging new instructions
                 >0: reserved for supporting special hardware,
                     extensions
247B second byte >5: reserved for operating system needs
```

Appendix 2: Special memory locations

The following memory locations are assumed by the hardware (i.e. the micro-program).

0..2	reserved; may be fixed locations of a data frame
3	device mask
4	P-Register used for initialization
5	saved value of P-Register on reset (for dumping)
...	7 free locations (reserved for the operating system)
16B, 17B	trap vector
20B, 21B	interrupt vector for line 8
22B, 23B	interrupt vector for line 9
...	
36B, 37B	interrupt vector for line 15
40B	frame address table
...	usable memory
177777B	reserved; NIL points to that location

The first three memory words have no special hardware functions. They are reserved and may be used if software assigns a data frame to location **0**, since the first three words of a data frame have hardware and compiler dictated purposes. The current compiler version assumes module *System* to be loaded on memory location **0**.

Appendix 3: The M-code interpreter

The following Modula-2 Program is an extension of the Appendix in "Lilith: A Personal Computer for the Software Engineer" [Wirl].

Its purpose is to document how the actual implementation on the Lilith was made. The version in "Lilith: A Personal Computer for the Software Engineer" is better suited for introduction, because the traps and other details there are omitted. Use this version as a reference what is exactly microprogrammed. (However, here trap for storage overflows undo the instruction which caused the overflow, on microcoded versions this is not implemented).

The array stk and code stand for the data and program stores respectively. In the actual computer they represent the same physical memory. The array indices denote memory addresses. Access to the code involves the use of the base address F (and an 18-bit wide addition).

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. Also, sets of the form {m..n} are used, although proper Modula-2 does not allow expressions to be used within set constructors. "Sets" of the form {i..i-1} are considered as {}.

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. However, to achieve portability, different devices should either be programmed identically or different instructions with different opcodes should be used. The SYS instruction may be used for future devices.

```
MODULE Interpreter;    (* N. Wirth, Ch. Jacobi Feb. 81  *)
  CONST
      NILL       = 177777B;        (*NIL*)
      devstatadr =  3;             (*devicestatus address*)
      tlc        = 16B;            (*trap location adr*)
      dft        = 40B;            (*frame table adr*)

      (*Trap Error Numbers*)
      (*------------------*)

      end        =  0;  (*end*)
      illegalChk =  1;  (*illegal instruction*)
```

```
    prioChk    =  2;  (*priority error*)
    storageChk =  3;  (*storage overflow*)
    rangeChk   =  4;  (*range violation*)
    addrChk    =  5;  (*NIL access or invalid address*)
    realOvfl   =  6;  (*floating point overflow*)
    cardOvfl   =  7;  (*cardinal overflow (maskable)*)
    intOvfl    =  8;  (*integer overflow  (maskable)*)
 (* software assignments:
    funcErr    =  9;  (*function return error [software]*)
    halt       = 10;  (*halt called [software]*)
    assertErr  = 11;  (*assertion error [software]*)
 *)

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 base address*)
    S:   CARDINAL;              (*stack pointer*)
    P:   CARDINAL;              (*process base address*)
    M:   BITSET;                (*software priority mask*)
    MSK: BITSET;                (*hardware interrupt mask*)
    REQ: BOOLEAN;               (*interrupt request*)
    ReqNo: [8..15];             (*request number*)
    overflow: BOOLEAN;          (*overflow (condition code)*)

  (* auxiliary variables;
     used over single instructions only *)
    i, j, k, sz, adr, low, hi: CARDINAL;
    b: BOOLEAN; lm: BITSET;
    i1, istep, ihi: INTEGER;
    fromea, toea: CARDINAL;        (*framepointers, 18 bit*)
    sb, db, sbmd, dbmd, fo: CARDINAL;  (*display handling*)
    x, y: REAL;   (*double precision unsigned fixed point*)

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

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

  VAR code[0]: ARRAY [0..777777B] OF [0..255];(*2↑18 bytes*)
      (*code[0]..code[377777B] shares memory with stk*)

  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, exsz;

    CONST exsz = 16;
    VAR sp: CARDINAL;
        a: ARRAY [0..exsz-1] OF CARDINAL; (*expression stack*)

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

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

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

    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;

  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      (*return address and external flag*)
      stk[S] := PC+100000B ELSE stk[S] := PC
    END;
    INC(S,2);             (*reserved for interrupt mask*)
    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]);
    MSK := M+BITSET(stk[devstatadr])
  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 (n IN {cardOvfl,intOvfl})
       AND (n IN BITSET(stk[P+7])) THEN
      (*the trap is masked out*)
      (*the expression stack pointer must be correct; the
        mask check may be implemented inline, instead here*)
    ELSE stk[P+6] := n;
      IF 7 IN BITSET(stk[devstatadr]) THEN LOOP END (*!*) END;
      Transfer(TRUE, tlc, tlc+1);
    END;
  END Trap;

  PROCEDURE Init;
    VAR bootflag: CARDINAL;
  BEGIN
    (* saveRegs; *) stk[5] := P;  (*allows debugging*)
    (* read a key from keyboard *)
    (* depending on the key: dump the memory onto the disk *)
    (* read the boot file according to key and set bootflag *)
    stk[6] := bootflag;
  END Init;

BEGIN (*main*)
  Init;
  P := stk[4]; restoreRegs(TRUE);
  LOOP
   IF REQ THEN Transfer(TRUE, 2*ReqNo, 2*ReqNo+1) END;
   IR := next();
   CASE IR OF
     0B..17B: (*LI0 - LI15 load immediate*)  push(IR MOD 16) |

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

    21B: (*reserved for future instruction [compiler]*)
         Trap(instrChk) |

    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());
         IF overflow THEN Trap(addrChk) END |
```

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

30B: (*JPC  jump conditional*)
     IF pop() = 0 THEN PC := PC + next2()
                  ELSE INC(PC, 2)
     END |

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

32B: (*JPFC  jump forward conditional*)
     IF pop()=0 THEN PC := PC + next() ELSE INC(PC) END |

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

34B: (*JPBC  jump backward conditional*)
     IF pop()=0 THEN PC := PC - next() ELSE INC(PC) END |

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

36B: (*ORJP  short circuit OR *)
     IF pop()=0 THEN INC(PC)
     ELSE push(1); PC := PC+next()
     END |

37B: (*ANDJP  short circuit AND *)
     IF pop()=0 THEN push(0); PC := PC+next()
     ELSE INC(PC)
     END |

40B: (*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: (*LSW0  load stack addressed word*)
      k := pop();
      IF k=NILL THEN Trap(addrChk)
      ELSE push(stk[k])
      END |

141B..157B: (*LSW1 - LSW15  load stack addressed word*)
      push(stk[pop()+(IR MOD 16)]);
      IF overflow THEN Trap(addrChk) END  |

160B: (*SSW0  store stack-addressed word*)
      k := pop(); j := pop();
      IF j=NILL THEN Trap(addrChk)
      ELSE stk[j] := k
      END |

161B..177B:  (*SSW1 - SSW15  store stack-addressed word*)
      k := pop(); i := pop()+(IR MOD 16);
      IF overflow THEN Trap(addrChk)
      ELSE stk[i] := k
      END |

200B: (*LSW  load stack word*)
      i := pop()+next(); push(stk[i]);
      IF overflow THEN Trap(addrChk) END |
```

```
201B: (*LSD  load stack double word*)
      i := pop()+next(); push(stk[i]); push(stk[i+1]);
      IF overflow (* on either addition*) THEN
        Trap(addrChk) END |

202B: (*LSDO  load stack double word*)
      i := pop(); push(stk[i]); push(stk[i+1]);
      IF (i=NILL) THEN Trap(addrChk) END |

203B: (*LXFW  load indexed frame word*)
      k := pop() + pop()*4 (*18 bits*);
      push(stk[k]);
      IF overflow(*18 bits*) THEN Trap(addrChk) END |

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 overflow THEN Trap(addrChk)
      ELSIF 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]);
      IF overflow THEN Trap(addrChk) END |

207B: (*LXD  load indexed double word *)
      i := 2*pop()+pop();
      IF overflow(*any op*) OR (i=NILL) THEN Trap(addrChk)
      ELSE push(stk[i]); push(stk[i+1])
      END |

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

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

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

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

```
214B, 215B: (*reserved [multiprecision instructions]*)
      Trap(instrChk) |

216B: (*DSHL  double shift left; [multiplication by 2]*)
      x := Dpop(); (* shift x left 1 bit *) Dpush(x)   |

217B: (*DSHR  double shift right; [division by 2]*)
      x := Dpop(); (* shift x right 1 bit *) Dpush(x)  |

220B: (*SSW  store stack word*)
      k := pop(); i := pop()+next();
      IF overflow THEN Trap(addrChk)
      ELSE
        stk[i] := k
      END |

221B: (*SSD  store stack double word*)
      k := pop(); j := pop(); i := pop()+next();
      IF overflow OR (i=NILL) THEN Trap(addrChk)
      ELSE
        stk[i] := j; stk[i+1] := k
      END |

222B: (*SSD0  store stack double word*)
      k := pop(); j := pop(); i := pop();
      IF i=NILL THEN Trap(addrChk)
      ELSE
        stk[i] := j; stk[i+1] := k
      END |

223B: (*SXFW  store indexed frame word*)
      i := pop();
      k := pop() + pop()*4; (*18 bits*)
      IF overflow(*18 bits*) THEN Trap(addrChk)
      ELSE
        stk[k] := i
      END |

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 overflow THEN Trap(addrChk)
        ELSIF 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();
      IF overflow THEN Trap(addrChk)
      ELSE
        stk[i] := k
      END |

227B: (*SXD  store indexed double word*)
      k := pop(); j := pop(); i := 2*pop()+pop();
      IF overflow(*any op*) OR (i=NILL) THEN Trap(addrChk)
      ELSE
        stk[i] := j; stk[i+1] := k
      END |

230B: (*FADD  floating add*)
      y := Dpop(); x := Dpop(); Dpush(x+y);
      IF overflow THEN Trap(realOvfl) END |

231B: (*FSUB  floating subtract*)
      y := Dpop(); x := Dpop(); Dpush(x-y);
      IF overflow THEN Trap(realOvfl) END |

232B: (*FMUL  floating multiply*)
      y := Dpop(); x := Dpop(); Dpush(x*y);
      IF overflow THEN Trap(realOvfl) END |

233B: (*FDIV  floating divide*)
      y := Dpop(); x := Dpop(); Dpush(x/y);
      IF overflow (*OR zerodivide*) THEN
        Trap(realOvfl) END |

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()))
                       (*trap on overflow*)
        ELSIF i=3 THEN Dpush(Dtrunc(Dpop(), pop()))
        (*...*)
        ELSE Trap(instrChk)
```

```
      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  check j <= i <= k *)
      k := pop(); j := pop(); i := pop(); push(i);
      IF (i<j) OR (i>k) THEN Trap(rangeChk) END  |

246B: (*ESC*)  (*escape*)
       i := next();
       IF i=0 THEN
         (*jump to high micro ram;
           used for debugging instructions*)
       ELSIF i IN {1..3} THEN (*printer instructions*)
       ELSE (* extensions; unknown behavior, eg. boot *)
       END |

247B: (*SYS  rarely used system functions: dump, boot,...*)
      i := next();
      IF i=0 THEN i := pop()       (* boot machine; boot#i*)
      ELSIF i=1 THEN               (* dump *)
      ELSIF i=2 THEN push(P)       (* read P register *)
      ELSIF i=3 THEN               (* set H register *)
        i := pop(); H := i-24; stk[P+5] := i;
      ELSIF i=4 THEN push(H+24)    (* read H register *)
      ELSIF i=5 THEN
        (* i := version code; 13.2 -->1*) push(i)
      ELSE Trap(instrChk)
      (* further cases reserved for operating system *)
      END |

250B: (*ENTP  entry priority*)
      i := next();
      (* lm := {0..i-1}; *)
      IF NOT (lm >= M) THEN Trap(prioChk)
      ELSE
        stk[L+3] := CARDINAL(M);
        M := lm;
        MSK := BITSET(stk[devstatadr])+M
      END |
```

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

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 INTEGER(i) < 0  |

260B: (*LODFW  reload expression stack but save function
               result value on top *)
      i := pop(); restoreExpStack; push(i) |

261B: (*LODFD  reload expression stack but save double
               word function result value on top *)
      i := pop(); j := pop(); restoreExpStack;
      push(j); push(i) |

262B: (*STORE  save expression stack*)
      IF S>H-(exsz+1) THEN
        PC := PC-1; Trap(storageChk);
      ELSE  saveExpStack  END |

263B: (*STOFV  store expression stack to data stack and
               put formal procedure variable on top*)
      IF S>H-(exsz+1) THEN
        PC := PC-1; Trap(storageChk)
      ELSE
        i := pop();
        saveExpStack; stk[S] := i; INC(S)
      END |

264B: (*STOT  store (from expression stack) to top of data
              stack; used for formal procedure*)
```

```
      IF S>=H THEN PC := PC-1; Trap(storageChk)
      ELSE
        stk[S] := pop(); INC(S)
      END |

265B: (*COPT  copy element on top of expression stack*)
      i := pop(); push(i); push(i) |

266B: (*DECS  decrement stackpointer*)  DEC(S) |

267B: (*PCOP  parameter copy; storage allocation and copy
              of value parameter*)
      stk[L+next()] := S;
      sz := pop(); k := S+sz;
      IF overflow OR (k > H) THEN
        PC := PC-2; Push(sz);
        Trap(storageChk)
      ELSE
        adr := pop();
        WHILE sz>0 DO
          stk[S] := stk[adr]; INC(S); INC(adr); DEC(sz)
        END
      END |

270B: (*UADD*)  j := pop(); i := pop(); push(i+j);
      IF overflow THEN Trap(cardOvfl) END |

271B: (*USUB*)  j := pop(); i := pop(); push(i-j);
      IF overflow THEN Trap(cardOvfl) END |

272B: (*UMUL*)  j := pop(); i := pop(); push(i*j);
      IF overflow THEN Trap(cardOvfl) END |

273B: (*UDIV*)  j := pop(); i := pop(); push(i DIV j);
      IF j=0 THEN Trap(cardOvfl) END |

274B: (*UMOD*) j := pop(); i := pop(); push(i MOD j);
      IF j=0 THEN Trap(cardOvfl) END |

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

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

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

300B: (*FOR1  entry FOR statement *)
      IF S>=H THEN PC := PC-1; Trap(storageChk)
```

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

301B: (*FOR2  exit FOR statement *)
      ihi := INTEGER(stk[S-1]); adr := stk[S-2];
      istep := INTEGER(next());
      (*sign extend; range -128..+127*)
      IF istep>127 THEN istep := istep-256 END;
      k := PC+next2();
      i1 := INTEGER(stk[adr])+istep;
      IF overflow OR ((istep>=0) AND (i1>ihi))
                  OR ((istep<=0) AND (i1<ihi))
      THEN (* terminate *) DEC(S,2)
      ELSE (* continue *)  stk[adr] := i1; PC := k
      END |

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

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

304B: (*TRAP*)
      i := pop(); Trap(i MOD 16) |

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

```
306B: (*CHKZ  check i <= k *)
      k := pop(); i := pop(); push(i);
      IF i>k THEN Trap(rangeChk) END  |

307B: (*CHKS  check sign bit*)
      k := pop(); push(k);
      IF INTEGER(k)<0 THEN Trap(rangeChk) 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*) i := pop(); push(ABS(INTEGER(i)));
      IF i=100000B THEN Trap(intOvfl) END |

317B: (*NEG*) i := pop(); push(-INTEGER(i));
      IF i=100000B THEN Trap(intOvfl) END |

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())))
      (* is equal  push(-pop()-1) *) |

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(NILL) |

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

327B: (*NOT*)  push(CARDINAL({15}/BITSET(pop()))) |

330B: (*ADD*)  j := pop(); i := pop();
      push(CARDINAL(INTEGER(i) + INTEGER(j)));
      IF overflow THEN Trap(intOvfl) END  |

331B: (*SUB*)  j := pop(); i := pop();
      push(CARDINAL(INTEGER(i) - INTEGER(j)));
      IF overflow THEN Trap(intOvfl) END  |

332B: (*MUL*)  j := pop(); i := pop();
      push(CARDINAL(INTEGER(i) * INTEGER(j)));
      IF overflow THEN Trap(intOvfl) END  |

333B: (*DIV*)  j := pop(); i := pop();
      push(CARDINAL(INTEGER(i) DIV INTEGER(j)));
      IF (j=0) OR ((j=177777B) AND (i=100000B)) THEN
        Trap(intOvfl)
      END |

334B: (*reserved for future instruction [compiler]*)
      (* e.g.  (*MOD*)  j := pop(); i := pop();
              push(CARDINAL(INTEGER(i) MOD INTEGER(j)))
      *)
      Trap(instrChk) |

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

336B: (*NOP*) |

337B: (*MOVF  move frame *)
      i := pop();
      fromea := pop()+pop()*4; (*18 bits*)
      toea   := pop()+pop()*4; (*18 bits*)
      IF overflow(*18 bits*) THEN Trap(addrChk)
      ELSE
```

```
        WHILE i>0 DO
          stk[toea] := stk[fromea];
          INC(toea); INC(fromea); DEC(i)
        END
      END |

340B: (*MOV  move block*)
      k := pop(); j := pop(); i := pop();
      IF (j=NILL) OR overflow(*on i+k*) THEN
        Trap(addrChk) (*source may wrap around*)
      ELSE
        WHILE k>0 DO
          stk[i] := stk[j]; INC(i); INC(j); DEC(k)
        END
      END |

341B: (*CMP  compare blocks*)
      k := pop(); j := pop(); i := pop();
      IF overflow (* on computing j+k or i+k*) THEN
        Trap(addrChk)
      ELSIF 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;
         may cause a Trap(addrChk) *) |

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

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;
         may cause a Trap(addrChk) *) |

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; update block db;
          may cause a Trap(addrChk) *) |

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 overflow OR (S > H) THEN
        S := pop(); Push(i); PC := PC-1;
        Trap(storageChk)
      END |

353B: (*ENTR  enter procedure*)
      i := next();  S := S + i;
      IF overflow OR (S > H) THEN
        S := S - i; PC := PC-2; Trap(storageChk)
      END |

354B: (*RTN  return from procedure*)
      S := L; L := stk[S+1]; i := stk[S+2];
      IF i < 100000B THEN (*local*) PC := i
      ELSE (* external *)
        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*)
          k := stk[S-1]; mark(G, TRUE);
          j := k DIV 400B;
          G := stk[dft+j]; F := stk[G];
          PC := 2*(k 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()

    ELSE Trap(instrChk)
    END
  END (*LOOP*)
END Interpreter.
```

Appendix 4: Distribution of instructions

The following table shows the static distribution of instructions of a big sample. The binary code of the editor, the windowhandler and the code generation pass of the compiler have been concatenated and analyzed. A program used to check the compiler output has been modified to count the instructions and to write a table. Finally, this table was analyzed with a statistics program to get the figures in this thesis.

LI0	815	2.7	LGW	643	2.1	LSW	149	0.5	FOR1	50	0.2
LI1	884	3.0	LGD	0	0.0	LSD	0	0.0	FOR2	50	0.2
LI2	198	0.7	LGW2	212	0.7	LSD0	0	0.0	ENTC	35	0.1
LI3	87	0.3	LGW3	176	0.6	LXFW	13	0.0	EXC	198	0.7
LI4	239	0.8	LGW4	116	0.4	LSTA	8	0.0	TRAP	121	0.4
LI5	165	0.6	LGW5	89	0.3	LXB	92	0.3	CHK	0	0.0
LI6	78	0.3	LGW6	102	0.3	LXW	131	0.4	CHKZ	405	1.4
LI7	56	0.2	LGW7	116	0.4	LXD	1	0.0	CHKS	74	0.2
LI8	44	0.1	LGW8	143	0.5	DADD	0	0.0	EQL	615	2.1
LI9	162	0.5	LGW9	63	0.2	DSUB	0	0.0	NEQ	204	0.7
LI10	71	0.2	LGW1	110	0.4	DMUL	0	0.0	LSS	12	0.0
LI11	32	0.1	LGW11	119	0.4	DDIV	0	0.0	LEQ	6	0.0
LI12	29	0.1	LGW12	74	0.2		0	0.0	GTR	12	0.0
LI13	23	0.1	LGW13	34	0.1		0	0.0	GEQ	6	0.0
LI14	30	0.1	LGW14	37	0.1	DSHL	0	0.0	ABS	2	0.0
LI15	83	0.3	LGW15	64	0.2	DSHR	0	0.0	NEG	9	0.0
LIB	1031	3.4	SGW	219	0.7	SSW	46	0.2	OR	16	0.1
	0	0.0	SGD	0	0.0	SSD	0	0.0	XOR	0	0.0
LIW	322	1.1	SGW2	93	0.3	SSD0	0	0.0	AND	53	0.2
LID	0	0.0	SGW3	48	0.2	SXFW	3	0.0	COM	4	0.0
LLA	194	0.6	SGW4	37	0.1	TS	25	0.1	IN	34	0.1
LGA	160	0.5	SGW5	34	0.1	SXB	88	0.3	LIN	211	0.7
LSA	472	1.6	SGW6	29	0.1	SXW	38	0.1	MSK	0	0.0
LEA	6	0.0	SGW7	26	0.1	SXD	1	0.0	NOT	63	0.2
JPC	41	0.1	SGW8	20	0.1	FADD	0	0.0	ADD	24	0.1
JP	109	0.4	SGW9	14	0.0	FSUB	0	0.0	SUB	42	0.1
JPFC	1161	3.9	SGW10	14	0.0	FMUL	0	0.0	MUL	0	0.0
JPF	458	1.5	SGW11	8	0.0	FDIV	0	0.0	DIV	3	0.0
JPBC	43	0.1	SGW12	11	0.0	FCMP	0	0.0		0	0.0
JPB	141	0.5	SGW13	14	0.0	FABS	0	0.0	BIT	19	0.1
ORJP	119	0.4	SGW14	9	0.0	FNEG	0	0.0	NOP	250	0.8
ANDJP	205	0.7	SGW15	15	0.1	FFCT	0	0.0	MOVF	4	0.0
LLW	116	0.4	LSW0	625	2.1	READ	7	0.0	MOV	67	0.2
LLD	34	0.1	LSW1	224	0.7	WRITE	6	0.0	CMP	0	0.0

LEW	601	2.0	LSW2	242	0.8	DSKR	0	0.0	DDT	2	0.0
LED	0	0.0	LSW3	189	0.6	DSKW	0	0.0	REPL	6	0.0
LLW4	1399	4.7	LSW4	84	0.3	SETRK	0	0.0	BBLT	5	0.0
LLW5	935	3.1	LSW5	65	0.2	UCHK	2	0.0	DCH	4	0.0
LLW6	929	3.1	LSW6	90	0.3	ESC	0	0.0	UNPK	0	0.0
LLW7	631	2.1	LSW7	19	0.1	SYS	2	0.0	PACK	2	0.0
LLW8	396	1.3	LSW8	40	0.1	ENTP	3	0.0	GB	13	0.0
LLW9	300	1.0	LSW9	17	0.1	EXP	3	0.0	GB1	81	0.3
LLW10	164	0.5	LSW10	16	0.1	ULSS	111	0.4	ALOC	89	0.3
LLW11	117	0.4	LSW11	17	0.1	ULEQ	96	0.3	ENTR	501	1.7
LLW12	82	0.3	LSW12	20	0.1	UGTR	155	0.5	RTN	642	2.1
LLW13	31	0.1	LSW13	5	0.0	UGEQ	43	0.1	CX	1492	5.0
LLW14	50	0.2	LSW14	23	0.1	TRA	0	0.0	CI	9	0.0
LLW15	29	0.1	LSW15	9	0.0	RDS	0	0.0	CF	33	0.1
SLW	45	0.2	SSW0	389	1.3	LODFW	87	0.3	CL	406	1.4
SLD	64	0.2	SSW1	182	0.6	LODFD	0	0.0	CL1	115	0.4
SEW	175	0.6	SSW2	122	0.4	STORE	87	0.3	CL2	31	0.1
SED	0	0.0	SSW3	95	0.3	STOFV	0	0.0	CL3	21	0.1
SLW4	395	1.3	SSW4	38	0.1	STOT	72	0.2	CL4	22	0.1
SLW5	293	1.0	SSW5	20	0.1	COPT	244	0.8	CL5	34	0.1
SLW6	273	0.9	SSW6	19	0.1	DECS	78	0.3	CL6	45	0.2
SLW7	167	0.6	SSW7	19	0.1	PCOP	47	0.2	CL7	28	0.1
SLW8	127	0.4	SSW8	17	0.1	UADD	786	2.6	CL8	13	0.0
SLW9	78	0.3	SSW9	11	0.0	USUB	502	1.7	CL9	16	0.1
SLW10	45	0.2	SSW10	11	0.0	UMUL	114	0.4	CL10	22	0.1
SLW11	29	0.1	SSW11	18	0.1	UDIV	30	0.1	CL11	35	0.1
SLW12	17	0.1	SSW12	14	0.0	UMOD	14	0.0	CL12	11	0.0
SLW13	5	0.0	SSW13	9	0.0	ROR	1	0.0	CL13	17	0.1
SLW14	12	0.0	SSW14	5	0.0	SHL	87	0.3	CL14	16	0.1
SLW15	5	0.0	SSW15	4	0.0	SHR	82	0.3	CL15	14	0.0

The first row, a), of the following table shows the same data as in the preceding table grouped according instruction classes. It is the big sample and it corresponds also the figures found in the chapter, *The encoding of the instructions.* Rows b) and c) show the instruction frequency of Forest Basket's puzzle benchmark [Bas], also found in an apendix of this paper. Row b) is the static, whereas row c) is the dynamic instruction frequency of the benchmark. These two rows show mainly the pecularities of the program; the sample is too small.

	a)	%	b)	%	c)	%
Load immediate	4560	15.2	222	28.5	134297	2.4
Load address	840	2.8	63	8.1	23516	0.4
Load local	5213	17.4	34	4.4	1809652	32.0
Load global	2098	7.0	114	14.6	221038	3.9
Load indirect	1834	6.1	2	0.3	3992	0.1
Load indexed	224	0.7	12	1.5	910397	16.1

Load external	601	2.0	0	0.0	0	0.0
Store local	1555	5.2	11	1.4	62002	1.1
Store global	591	2.0	11	1.4	2014	0.0
Store indirect	1019	3.4	32	4.1	4022	0.1
Store indexed	369	1.2	7	0.9	3997	0.1
Store external	175	0.6	0	0.0	0	0.0
Operators	1885	6.3	90	11.5	161594	2.9
Comparators	1260	4.2	2	0.3	27905	0.5
Jumps	1953	6.5	13	1.7	826273	14.6
Short circuit AND/OR	324	1.1	2	0.3	540906	9.6
FOR/CASE	333	1.1	100	12.8	824530	14.6
Calls (without returns)	2380	8.0	28	3.6	21433	0.4
Others (aloc, rtn, ent,..)	2703	9.0	37	4.7	73512	1.3
Total	29917	100.0	780	100.0	5651080	100.0

Appendix 5: Benchmark Tests

We will not introduce new benchmark programs. Using existing benchmark programs allows comparison with other computers for which data is available.

Forest Basket's Puzzle benchmark

Time	Bytes	Inst.	Machine	language	comments
2.3	4882	644	S-1	Pascal	
3.0	1214	955	Dorado	Mesa	
3.5	2080	293	VAX 11/780	C	pointers, not subscripts
4.4			DEC 2060	Pascal	pointers, not subscripts
5.4	3829	851	DEC 2060	Pascal	
6.8	4630	1029	DEC 2060	Pascal	with checking
7.5	4744	1052	IBM 158	Pascal	
8.9	2500	646	MC68000	C	pointers, not subscripts
9.0	2130		VAX 11/780	C	
13	2500		DEC 11/34	C	pointers, not subscripts
26	1214	955	Dolphin	Mesa	
26.5	2500	646	MC68000	C	
28.5			PERQ	Pascal	with checking
38	4744	1052	IBM 4331	Pascal	
42	3829	851	MAXC1	Pascal	
42	4630	1029	MAXC1	Pascal	with checking
50	1214	955	Alto I	Mesa	
53	1214	955	Alto II	Mesa	
16	1369	851	Lilith	Modula	with checking
14	1259	779	Lilith	Modula	no checking
10	1259	779	Lilith(1982)	Modula	no checking
25	2794		PDP-11/40(EIS)	Modula	no checking

The modula compiler does not recognize nor optimize FOR statements like

```
FOR i := 0 TO 0 DO ... END
```

which are executed only once.

We have introduced a WITH-statement which does not change the algorithm, but influences the generated code. We did not touch the lengthy initializations.

The time used is measured with a stop watch because the system clock on Lilith may be inexact.

```
MODULE Puzzle;  (* Forest Basket's Puzzle benchmark *)
                (* translated to Modula-2           *)
  FROM OutTerminal IMPORT WriteLn, WriteC;
  FROM Terminal IMPORT WriteString;

  CONST
    d           =  8;
    size        =  d*d*d-1;
    classMax    =  3;
    typeMax     =  12;

  TYPE
    PieceClass =  [0..classMax];
    PieceType  =  [0..typeMax];
    Position   =  [0..size];

  VAR
    pieceCount:    ARRAY PieceClass OF [0..13];
    class:         ARRAY PieceType OF PieceClass;
    pieceMax:      ARRAY PieceType OF Position;
    puzzle:        ARRAY Position OF BOOLEAN;
    p:             ARRAY PieceType OF
                     RECORD r: ARRAY Position OF BOOLEAN
                     END;
    m, n:          Position;
    i, j, k:       [0..13];
    kount:         CARDINAL;

  PROCEDURE Fit(i: PieceType; j: Position): BOOLEAN;
    VAR k: Position;
  BEGIN
    WITH p[i] DO
      FOR k := 0 TO pieceMax[i] DO
        IF r[k] AND puzzle[j+k] THEN
          RETURN FALSE
        END;
      END
    END;
    RETURN TRUE
  END Fit;

  PROCEDURE Place(i: PieceType; j: Position): Position;
    VAR k: Position;
  BEGIN
    WITH p[i] DO
      FOR k := 0 TO pieceMax[i] DO
        IF r[k] THEN puzzle[j+k] := TRUE END
      END;
    END;
    DEC(pieceCount[class[i]]);
```

```
    FOR k := j TO size DO
      IF NOT puzzle[k] THEN RETURN k END
    END;
    WriteString("puzzle filled"); WriteLn;
    RETURN 0
  END Place;

  PROCEDURE Remove(i: PieceType; j: Position);
    VAR k: Position;
  BEGIN
    WITH p[i] DO
      FOR k := 0 TO pieceMax[i] DO
        IF r[k] THEN puzzle[j+k] := FALSE END;
      END
    END;
    INC(pieceCount[class[i]]);
  END Remove;

  PROCEDURE Trial(j: Position): BOOLEAN;
    VAR i: PieceType;
        k: Position;
  BEGIN
    FOR i := 0 TO typeMax DO
      IF (pieceCount[class[i]] <> 0) THEN
        IF Fit(i, j) THEN
          k := Place(i, j);
          IF Trial(k) OR (k = 0) THEN
            WriteString("piece"); WriteC(i+1, 0);
            WriteString(" at"); WriteC(k+1, 0); WriteLn;
            INC(kount);
            RETURN TRUE
          ELSE Remove(i, j)
          END
        END
      END
    END;
    INC(kount);
    RETURN FALSE
  END Trial;

BEGIN
  WriteString("time"); WriteLn;
  FOR m := 0 TO size DO puzzle[m] := TRUE END;
  FOR i := 1 TO 5 DO
    FOR j := 1 TO 5 DO
      FOR k := 1 TO 5 DO puzzle[i+d*(j+d*k)] := FALSE END
    END
  END;
  FOR i := 0 TO typeMax DO
    FOR m := 0 TO size DO p[i].r[ m] := FALSE END
```

```
  END;

  FOR i := 0 TO 3 DO
    FOR j := 0 TO 1 DO
      FOR k := 0 TO 0 DO p[0].r[i+d*(j+d*k)] := TRUE END
    END
  END;
  class[0] := 0;
  pieceMax[0] := 3+d*1+d*d*0;

  FOR i := 0 TO 1 DO
    FOR j := 0 TO 0 DO
      FOR k := 0 TO 3 DO p[1].r[i+d*(j+d*k)] := TRUE END
    END
  END;
  class[1] := 0;
  pieceMax[1] := 1+d*0+d*d*3;

  FOR i := 0 TO 0 DO
    FOR j := 0 TO 3 DO
      FOR k := 0 TO 1 DO p[2].r[i+d*(j+d*k)] := TRUE END
    END
  END;
  class[2] := 0;
  pieceMax[2] := 0+d*3+d*d*1;

  FOR i := 0 TO 1 DO
    FOR j := 0 TO 3 DO
      FOR k := 0 TO 0 DO p[3].r[i+d*(j+d*k)] := TRUE END
    END
  END;
  class[3] := 0;
  pieceMax[3] := 1+d*3+d*d*0;

  FOR i := 0 TO 3 DO
    FOR j := 0 TO 0 DO
      FOR k := 0 TO 1 DO p[4].r[i+d*(j+d*k)] := TRUE END
    END
  END;
  class[4] := 0;
  pieceMax[4] := 3+d*0+d*d*1;

  FOR i := 0 TO 0 DO
    FOR j := 0 TO 1 DO
      FOR k := 0 TO 3 DO p[5].r[i+d*(j+d*k)] := TRUE END
    END
  END;
  class[5] := 0;
  pieceMax[5] := 0+d*1+d*d*3;
```

```
FOR i := 0 TO 2 DO
  FOR j := 0 TO 0 DO
    FOR k := 0 TO 0 DO p[6].r[i+d*(j+d*k)] := TRUE END
  END
END;
class[6] := 1;
pieceMax[6] := 2+d*0+d*d*0;

FOR i := 0 TO 0 DO
  FOR j := 0 TO 2 DO
    FOR k := 0 TO 0 DO p[7].r[i+d*(j+d*k)] := TRUE END
  END
END;
class[7] := 1;
pieceMax[7] := 0+d*2+d*d*0;

FOR i := 0 TO 0 DO
  FOR j := 0 TO 0 DO
    FOR k := 0 TO 2 DO p[8].r[i+d*(j+d*k)] := TRUE END
  END
END;
class[8] := 1;
pieceMax[8] := 0+d*0+d*d*2;

FOR i := 0 TO 1 DO
  FOR j := 0 TO 1 DO
    FOR k := 0 TO 0 DO p[9].r[i+d*(j+d*k)] := TRUE END
  END
END;
class[9] := 2;
pieceMax[9] := 1+d*1+d*d*0;

FOR i := 0 TO 1 DO
  FOR j := 0 TO 0 DO
    FOR k := 0 TO 1 DO p[10].r[i+d*(j+d*k)] := TRUE END
  END
END;
class[10] := 2;
pieceMax[10] := 1+d*0+d*d*1;

FOR i := 0 TO 0 DO
  FOR j := 0 TO 1 DO
    FOR k := 0 TO 1 DO p[11].r[i+d*(j+d*k)] := TRUE END
  END
END;
class[11] := 2;
pieceMax[11] := 0+d*1+d*d*1;

FOR i := 0 TO 1 DO
  FOR j := 0 TO 1 DO
```

```
        FOR k := 0 TO 1 DO p[12].r[i+d*(j+d*k)] := TRUE END
      END
    END;
    class[12] := 3;
    pieceMax[12] := 1+d*1+d*d*1;

    pieceCount[0] := 13;
    pieceCount[1] := 3;
    pieceCount[2] := 1;
    pieceCount[3] := 1;
    m := 1+d*(1+d*1);
    kount := 0;
    IF Fit(0, m) THEN n := Place(0, m)
    ELSE WriteString("error 1"); WriteLn END;
    IF Trial(n) THEN
      WriteString("success in"); WriteC(kount,0);
      WriteString(" trials"); WriteLn;
    ELSE
      WriteString("failure"); WriteLn;
    END;
    WriteString("time"); WriteLn;
END Puzzle.
```

A benchmark test for Modula implementations

This is the Program found in the Appendix of [Wirl]. We added some more machines in the table. The benchmark just executes some instructions in a loop. The test figures denote how many times the loop is repeated in one minute.

The PERQ data was obtained indirectly. On some machines the speed of such small loops may depend on alignment conditions of the code.

Bare speed *(loop executions in one minute)*

	facility	Lilith	PDP11/40 (EIS)	Alto 2	MC68000 8 MHz	VAX 11/750 compat.	PERQ	Lilith faster
a	empty REPEAT loop	321	184		280	267	309	422
b	empty WHILE loop	334	185	116	275	262	308	419
c	empty FOR loop	422	230	172	320	341	254	528
d	CARDINAL arithmetic	187	84	54	107	116	124	218
e	REAL arithmetic	130				4		144
f	sin, exp, ln, sqrt	87						99
g	array access	109	54	32	76	81	106	136
h	same with bound test	89	11	26	61	19	71	111
i	matrix acces	197	93	44	128	135	158	249
j	same with bound test	164	21	36	108	36	130	198
k	call of empty procedure	144	37	40	128	53	93	187
l	with 4 parameters	94	29	32	79	41	69	123
m	copying arrays	63	11	56	53	11	110	96
n	access via pointers	125	66	54		97	50	162
o	reading a disk stream	80		36				94

We do not know why on the PERQ the c) test is slower than the b) test. On Lilith the a) test is slower than the b) test, probably because of alignments of the code. We did not expend any effort to tune the tests.

For checking pure hardware speed the puzzle test is better suited. This test allows comparison of single topics.

The next table is normalized. Every test figure shows the percentage execution speed compared to the Lilith. However, this normalization is not made with the idea of comparing of the figures of computer x with the figures of the Lilith. The figures should be inspected row-wise. Speed differences for different topics on the same computer show architectural differences between the inspected computer and the Lilith.

Normalized *(% of Lilith's computation of the same test)*

facility	Lilith	PDP11/40 (EIS)	Alto 2	MC68000 8 MHz	VAX 11/750 compat.	PERQ	Lilith faster
a empty REPEAT loop	100	57		87	83	96	131
b empty WHILE loop	100	55	35	82	78	92	125
c empty FOR loop	100	54	41	76	81	60	125
d CARDINAL arithmetic	100	45	29	57	62	66	117
e REAL arithmetic	100				3		111
f sin, exp, ln, sqrt	100						114
g array access	100	46	29	70	74	97	125
h same with bound test	100	12	29	69	21	80	125
i matrix acces	100	47	22	65	69	80	126
j same with bound test	100	13	22	66	22	79	121
k call of empty procedure	100	26	28	89	37	65	130
l with 4 parameters	100	31	34	84	44	73	131
m copying arrays	100	17	89	84	17	175	152
n access via pointers	100	52	43		78	40	130
o reading a disk stream	100		45				118

Two Liliths are compared, the last column is a Lilith with faster memory. Speeding up the memory improved the Lilith about a factor 1.26, however it also changed the order. Copying arrays is improved very similar to the memory speed increase, CARDINAL arithmetic and REAL arithmetic are improved much less. Indeed, this is not surprising.

The array copy test shows surprising figures on the Alto 2 and on the PERQ. These machines will use a single instruction to copy arrays, as Lilith does. These figures suggest that on the (older) Lilith the memory is too slow.

The PDP-11 array bound check, and the array copying are compiled into trap instructions, resulting in relatively short code, showing clearly the speed penalty.

The Alto 2 has quite efficient pointers.

On the MC68000, CARDINAL arithmetic seems to be the less efficient.

The VAX 11/750 is measured in compatibility mode. In that mode REAL arithmetic seems not to be supported. The numbers look similar to the PDP-11/40, maybe more than really necessary, because the PDP compiler is used.

Generally, on the competitor machines, the three loops a)-c) are executed relatively more efficiently than arithmetic, procedure calls or array access. This shows that the optimizations of more complex subjects on the Lilith wins more than bare speed on loops. The bare speed of instruction fetch played also a big role on the Lilith, it has a special instruction fetch unit in hardware. However, other micro-programmed computers also have hardware mechanisms to improve the instruction fetch.

Appendix 6: Jump optimization for IF statements

The following program part shows the optimization of the jump instructions for the IF statement. The scanning of the program text is simplified for clarity. The chapter "Statements compiled to simple instructions and jump optimization" describes this algorithm in a more general way. See also [Wir5].

The compiler does not generate code for branches which never can be executed because of constant expressions. This optimization is omitted in the following program, the jump optimization is complex enough.

These declarations are local to the procedure which compiles the IF statement. If the IF statements are nested, each IF statement gets its own instance of these variables.

```
TYPE
  ListPtr = POINTER TO JumpRecord;
  JumpRecord =
    RECORD
      nextPtr: ListPtr;
      nextJump: CARDINAL;
      endJump: CARDINAL
    END;
VAR
  lat: Attribut;
  lp, jumpList: ListPtr;
  shift, curPC, c: CARDINAL;
```

The code generation for the IF statement

```
IF sym = "IF" THEN
  (* analysis of the IF statement *)
  jumpList := NIL;
  REPEAT
    GetSymbol; Expression;
    NEW(lp); lp↑.nextPtr := jumpList; lp↑.nextJump := pc;
    Emit(JPFC); Emit(0);
    StatementSequence; (*termination with "END", "ELSIF", "ELSE"*)
    lp↑.endJump := pc; jumpList := lp;
    IF sym <> "END" THEN Emit(JPF); Emit(0) END;
  UNTIL sym <> "ELSIF";
  IF sy = "ELSE" THEN GetSymbol;
     (* the occurence of an "ELSE" can be detected
       by jumpList↑.endJump<>PC *)
     StatementSequence (*termination with "END"*)
  END;
  GetSymbol; (* skip endsy *)

  (* compute total displacement; pass the IF statement
    in the reverse direction *)
  shift := 0; lp := jumpList;
  WHILE lp<>NIL DO
```

```
      (* test for the jump to the end *)
      (* c gets the length of the jump instruction to the end *)
      IF (pc + shift-lp↑.endJump) = 0 THEN c := 0
      ELSIF (pc + shift-lp↑.endJump)>(255 + 1) THEN INC(shift); c := 3
      ELSE c := 2 END;
      (* test for the jump to the next test or else part *)
      IF (c + lp↑.endJump-lp↑.nextJump)>(255 + 1) THEN INC(shift) END;
      lp := lp↑.nextPtr
    END;

    (* fix jump addresses and perform necessay code moves *)
    lp := jumpList; curPC := pc;
    WHILE lp<>NIL DO
      (* handle the jump to the end *)
      (* c gets the length of the jump instruction to the end *)
      IF lp↑.endJump<curPC THEN
        IF shift>0 THEN MoveCode(lp↑.endJump, curPC-1, shift) END;
        IF (pc-shift-lp↑.endJump)>(255 + 1) THEN
          Insert(lp↑.endJump + shift-1, JP);
          Insert2(lp↑.endJump + shift, pc-(lp↑.endJump + shift));
          DEC(shift); c := 3
        ELSE
          Insert(lp↑.endJump + shift + 1, pc-(lp↑.endJump + shift + 1));
          c := 2
        END
      ELSE c := 0 (*no else part*)
      END;
      curPC := lp↑.nextJump;
      (* handle the jump to the next test or else part *)
      IF shift>0 THEN MoveCode(lp↑.nextJump, lp↑.endJump-1, shift) END;
      c := c + lp↑.endJump-lp↑.nextJump-1;
      IF c>255 THEN
        Insert(lp↑.nextJump + shift-1, JPC);
        Insert2(lp↑.nextJump + shift, c + 1);
        DEC(shift)
      ELSE
        Insert(lp↑.nextJump + shift + 1, c);
      END;
      lp := lp↑.nextPtr
    END
  END
```

Curriculum vitae

On August 10, 1951 I was born in Zürich, Switzerland. After six years of primary school and two years of secondary school, I was admitted in 1967 to the Kantonale Oberrealschule Zürich, which I left in 1971 with the Maturität.

At the Swiss Federal Institute of Technology Zürich I studied Mathematics from 1971 until I obtained, in 1976, the Diploma in Mathematics. Since then I have been an assistant at the Institute for Informatics of the Swiss Federal Institute of Technology.