DRAFT PROJECT PLAN

Date: 27 August 1987
From: Don MacLaren
Dept: DECwest Eng.
Phone: (206) 865-8730
MS: ZSO
ENET: DECWET::DON

Subj: DECwest Compiler Project, Description and Plan

NOTE: In this document, PRISM refers to the mainline 64-bit PRISM architecture, not the 32-bit uPRISM.

As part of PRISM project, the DECwest compiler group is producing a highly optimizing compiler for both C and Pillar, which is a new systems programming language whose development is also part of the project. The compiler group is also responsible for some related PRISM utilities, a highly optimizing VAX compiler for Pillar, and some compilers used in the initial development of PRISM hardware and software.

The purpose of this document is to provide people outside the compiler group with the information they need to understand the project and to produce related documents such as business plans and formal project plans. The document covers the PRISM language strategy, DECwest languages and related software, properties of the new compiler, and the schedule through V1 of the Pillar compiler. The document will be updated as required during the project.

If you want to be on the distribution list for updates or you want a copy of this document, please send mail to DECwet::Pillar. Questions about the content of the project plan should be directed to Darryl Havens or Don MacLaren.

There is no formal product management for the compiler project, but it is closely related to the compute server project. For that, contact Cathie Richardson in regards to business product management and Terry Morris in regards to technical product management.

CONTENTS

# 1 MILESTONES AND SCHEDULE

The milestones listed in this section are the points at which significant software items are available outside the compiler group in stable form. To be available in this sense, software must have passed the applicable test system and, by noon Pacific time of the specified date, must be in the DECwest cluster directory used for software distribution. Documentation must have been dispatched so that it will arrive at the ZK mailroom in time for delivery on the specified day.

As a result of the redirection of the PRISM project, the compiler group's original implementation plan has been revised to get the production PRISM Pillar compiler as quickly as possible. The schedule given here runs only through delivery of that compiler. The schedules for PRISM C, VAX Pillar, the utilities, and the advanced code-optimization features will all be determined later.

The schedule allows for reasonable support of the SIL compilers. It requires timely availability of the PRISM simulator, linker, and librarian, all running on VAX/VMS. Apart from this there are no significant dependencies on other groups. There are no resources to spare within the compiler group.

MILESTONES:

1. SIL V1.0, Feb. 18, 1987. The SIL compilers generate code for VAX or PRISM. PRISM code is packaged in VAX object modules. The reference manual distributed with this compiler is out of date. The SIL User Manual covers mainly the SIL command and the structure of programs and modules. The compiler comes with a set of examples.

2. SIL V2.0. Nov. 87. This version of SIL adds:

   - Structured exception handling including the necessary VAX runtime support.
   - An option to probe and capture arguments for entries to the MICA executive.
   - Some minor language improvements.
   - A SIL Reference Manual and an expanded SIL User Manual.

3. Pillar Reference Manual, Rev. 1.0. Dec. 1987. This is a clean and complete Pillar reference manual. The second Pillar language review begins at this point.

4. SPASM. December 1987. SPASM is the PRISM assembler. This version runs on VAX/VMS but generates true PRISM object modules. It will have a simple macro facility if this is required by the schedule for development of PRISM hardware diagnostics. In the long run, the Pillar compile time facility (CTF) will serve as a SPASM's macro facility.

5. Close Pillar language review. March 1988, about 3 months after distribution of the manual. After this date, no new comments will be accepted. However, there will be a chance

to comment on language changes proposed as a result of the review.

6. Pillar language freeze. April 1988, about one month after the review closes. All language extensions, changes, and clarifications are recorded in the notes file.

7. Pillar V1. June 6, 1988. This is the first production Pillar compiler. It runs on VAX/VMS generating PRISM object modules.

## 2 DEVELOPMENT TEAM

Compiler group:

| | | |
|---|---|---|
| Don MacLaren | -- | DECwet::DON |
| Tony Ercolano | -- | DECwet::ERCOLANO |
| John Hamby | -- | DECwet::HAMBY |
| Darryl Havens | -- | DECwet::HAVENS |
| Lois Hayes | -- | DECwet::HAYES |
| Gary Kimura | -- | DECwet::KIMURA |
| Jay Palmer | -- | DECwet::PALMER |
| Lu Anne Van de Pas | -- | DECwet::VANDEPAS |

Technical Writers:

| | | |
|---|---|---|
| Helen Custer | -- | DECwet::CUSTER |
| Liz Hunt | -- | DECwet::HUNT |
| Bill Muse | -- | DECwet::MUSE |

## 3 PRISM LANGUAGE STRATEGY

This section describes the original PRISM language and compiler strategy worked out between DECwest and SDT. The recent high level events have disrupted the PRISM project, but original language and compiler strategy is still the right strategy. It is compatible with marketing PRISM as a compute server, as a work station, as a complete system with one or more operating systems, and as an architecture with either or both word sizes.

PRISM programming is done in high level languages. Four language products are planned for FRS: C, FORTRAN, Pascal, and Pillar. Additional languages will be provided in a second wave. All PRISM languages share a common language environment that makes it easy for a user to mix languages in a single application. The key components of this common environment are

1. The PRISM calling standard.

2. The PRISM object language.

3. A standard (as yet, unnamed) for general language-independent compiler features such as the form of command options and the arrangement of listings.

4. The PRISM debugger, which is based on the new debug symbol table (DST) architecture.

5. The PRISM performance and coverage analysis utility (PCA).

6. The language sensitive editor, LSE.

7. The source code analyzer, SCA.

Under the compute server approach, the user interfaces to the debugger and PCA are on VAX (integrated with the VAX versions), while LSE and SCA are the VAX versions of these utilities.

Pillar and C are the systems programming languages for PRISM. DEC is using Pillar because of its significant advantages in the areas of type checking, program structure, and integration with the DEC software environment. Most customers will use C for systems, because it is a standard language, it is practical for systems programming, and it has good portability when programmers are careful in their use of the language. Customers will perceive the Pillar and C compilers to be of the same quality -- the highest possible. They will not feel forced to use Pillar.

In addition to the language products, BLISS and a new assembler, SPASM, are also available for use within DEC. Customers can obtain these only by special arrangement. BLISS is used for porting existing software from VMS to PRISM. SPASM is used only in special contexts where normal programming language concepts do not apply.

Pillar, C, and SPASM are implemented by DECwest. FORTRAN, PASCAL, and Bliss are implemented by SDT.

## 4 PILLAR

Pillar is a high-level systems programming language for use on 32- and 64- bit Digital Equipment systems. The Pillar design emphasizes general features for high-level programming: modules, data type declarations, control structures, and the use of procedures. Examples of Pillar features are:

- A flexible module structure in which information that is not logically part of a definition module can be hidden in a separate implementation module even though it is needed at compile time.

- A treatment of data types, with roots in Pascal, that provides flexible types (parametric types) to describe dynamically sized data and records with variants.

- The sort of type escapes necessary for systems programming, but with some safeguards.

- Inline procedures that can be defined in modules.

- Four distinct modes for parameters: input, output, input-output, and bind, this last mode being for the unusual case of an argument that must be addressed in its original

storage.

- Parameters with matching extents: the extents of the parameter's data type are determined by the extents of the actual argument.

- Structured exception handling.

When compared with low-level systems programming languages, Pillar has the following advantages:

- Software is more portable, because hardware dependencies are isolated in declarations and small procedures. Accidental hardware dependencies are avoided.

- Code is easier to read and maintain. This was emphasized in the design of Pillar's syntax.

- Program development is faster because the compiler detects more errors, and the language provides explicit help in difficult areas, such as exception handling.

- Pillar yields the fastest object code for the PRISM architecture.

Although system independent in most respects, Pillar has been designed to take full advantage of DEC's software technology. For example, exception handling and messages are provided in a way that extends the existing VAX/VMS capabilities. Also, specific hardware features are supported via system dependent modules built into the compiler. The PRISM Pillar compiler has PRISM specific modules for scalar operations (e.g.; shift instructions), vector operations, and privileged operations.

## 4.1 Pillar Development

Pillar has been developed as a fundamental part of the PRISM project to implement the PRISM executive and PRISM software components that operate above the executive (compilers,linkers, runtime libraries and such). The first draft Pillar reference manual (Rev. 0.0) was distributed for review in November, 1985. The proposed language was closely tied to VAXELN Pascal. As a result of the review the language was redesigned, and it is no longer coupled to Pascal. In February 1987, the redesigned Pillar was made available via the SIL compilers that support most, but not all, of the language.

The documentation distributed with SIL V1.0 was incomplete in regards to SIL, and the SIL language is not exactly a subset of Pillar, which has continued to evolve. The next true Pillar manual will be Rev 1.0. It is scheduled for December, 1987. As soon as the manual is available, the next language review will start.

The style, structure, and principal features of Pillar are set. However there is still plenty of opportunity for improvement via the review. To get as good a language as possible within the

project constraints, the review will be done online over a three month period. All comments and proposed language changes will be published in a notes file as they are received. This way all interested parties will get up to date information on language issues, and people's comments can be timely. A review team will consider all significant issues, and all language changes will be listed in the notes file.

The language resulting from the review will be described in Rev 2 of the Pillar manual, and implemented in the Pillar V1 compiler, which will be the first production Pillar compiler. This compiler will

- provide an absolutely sound base for system software development and further compiler development.
- produce clean, moderately optimized code.
- implement a smooth language that is functionally a superset of SIL.
- run on VAX/VMS, produce PRISM object modules, and use the PRISM libarian and linker (which will also temporarily operate on VAX/VMS).

It is neither possible nor desirable to make Pillar exactly compatible with SIL. To ease the transition from SIL to Pillar. the compiler will have two features:

1. Whenever possible, the compiler will recognize an obsolete SIL construction, issue a very specific error message, and emit an LSE diagnostic record with correction information. This should make conversion from SIL to Pillar rather easy.
2. There will be a /SIL command option for compatibility. Under this option, the compiler will, whenever possible, recognize an obsolete SIL construction and do the right thing without any error message.

The cases excluded by the phrase "whenever possible" are expected to arise only from code depending on accidental features of SIL and the limitations of SIL's type checking and range checking. The compatibility option will not be removed until after both PRISM and VAX versions of the Pillar compiler are available.

The language supported by the V1 Pillar compiler will be almost the complete language required for FRS of the PRISM system, and the missing pieces will be provided rapidly. (Rev 1 of the manual will indicate which pieces are expected to be deferred until after V1.) Most of the compiler development resources after V1 will be devoted to code optimization, the VAX back end for Pillar, and the PRISM C compiler. However, after people have some experience with Pillar V1, there will be another language review to consider possible language extensions that might be made before or after PRISM FRS.

VAX Pillar requires some additional system-specific features, e.g.; for VAX descriptors. Shortly after V1, there will be a review of VAX features. This review should also cover any issues about the integration of Pillar with the VMS environment, e.g.;

in regards to message files. The schedule for the Pillar VAX compiler depends on project priorities. If it has high enough priority, it can be ready for use in December 1988.

## 4.2 Pillar Documentation

There are two manuals for PRISM Pillar.

- The Pillar Reference Manual is a concise language reference manual. For the most part it is system independent. In the few instances where the manual does deal with system dependent rules, it will cover both VAX and PRISM. Until the Pillar language is frozen, this manual serves as the language standard. It's being written by Don MacLaren and edited by Bill Muse. Once Rev 2 is published, Bill will convert it to a normal reference manual.

- The Pillar User Manual. This manual is for experienced programmers, but it does not require knowledge of Pillar. It explains how to use Pillar on PRISM, emphasizing the solution of problems that occur in systems programming. There will also be a VAX version of this manual. Liz Hunt is the technical writer for this manual.

Rev 1 and Rev 2 of the Pillar manuals will be distributed (hard copy and labeled company confidential) to everyone on the Pillar and SIL interest lists. The open review procedures will be announced along with Rev 1.

Requests for documentation and general questions about Pillar should be sent to DECWET::PILLAR.

## 4.3 Pillar Definition Modules

This section discusses Pillar definition modules, which can be used by other compilers and utilities to get information about routines programmed in Pillar. This is especially important for system routines.

Compilation of a Pillar source module, ALPHA, generally produces a definition module in addition to an object module. The definition module contains the declarations of all of ALPHA's exported symbols in a compiled form. An exported symbol is one that may be used in another module. If another module uses symbols from ALPHA, it names ALPHA in an import statement, and the compiler reads ALPHA's definition module. This is more efficient than compiling ALPHA's declarations each time they are used in another module, and it prevents errors in ALPHA from showing up while compiling another module.

In large systems or application programs there is always a danger of inconsistency resulting from failure to recompile a module when declarations on which it depends have changed. To prevent this, Pillar definition modules contain a signature for each exported declaration, and both definition and object modules

record the signatures on which they depend. Consistency of signatures can be checked by the compiler or by the linker (a feature of the MICA linker). Because there is a signature for each symbol, it is not necessary to recompile everything just because a simple change is made to a module.

Pillar definition modules are the means for making system interfaces available to all languages. When the declaration of a system interface is needed in a language other than Pillar, it can be obtained two ways. The other language's compiler can directly import the definition module, or the definition module can be translated into an include or require file in the other language. The direct import method has these advantages:

- It's less bother; there are no require files to manage.
- Module consistency checking can be used.
- The other language's compiler can understand some things that can't be expressed in the language or that require nonstandard expressions such as %DESCRIPTOR.

Because the direct import method has not been used before, and because customers may prefer the concrete form of a require file, we are supporting both methods. There is a definition module utility that translates Pillar definition modules into other languages: C, FORTRAN, Pascal, and Bliss. Additional languages will be supported as they are implemented on PRISM. This utility is structured as a shell plus a set of back ends, one for each target language. The shell reads the definition modules and builds a symbol table in memory. Each language-specific back end accesses the symbol table through a set of routines provided as part of the shell. The shell and C backend are implemented by DECwest, the other backends by SDT.

The parts of the shell that build and access the symbol table will be packaged so that they can be incorporated into other utilities. In particular, this will be the way in which the SDT compilers directly import Pillar definition modules.

## 5 PRISM C

For systems programming, PRISM customers are most likely to use the C language, and this usage will be intermixed with applications programming in C. (There is no clear boundary between systems and applications programming.) The compiler will be heavily used in the scientific, technical, and educational markets. It will feature the industry's most advanced code optimization methods: interprocedural analysis, vectorization and decomposition, instruction scheduling, global register allocation, and performance profile feedback to improve all aspects of code generation and optimization.

Within the PRISM project, PRISM C will be used for ULTRIX-related software (including ULTRIX itself if it is ported to PRISM). C will also be used for parts of the Applications Interface Architecture (AIA) on MICA.

PRISM C will be an implementation of the ANSI C standard. This standard specifies certain syntax for implementation-specific extensions to the standard language. This syntax will be used in VAX C, V3.0, for both old and new extensions. PRISM C will contain the extensions from VAX C that do not depend on the target architecture. A point to note here is that the VAX C, V3.0, extensions related to optimization are being defined in a way that is not dependent on the architecture, so they will be supported by PRISM C.

PRISM C will support 64-bit integers as well as 16- and 32-bit integers. This may require an extension. Built-in functions for privileged operations and atomic memory access may be required, especially for the ULTRIX executive. No other language extensions are planned for PRISM C. However, the compiler's ability to import PRISM definition modules will be a valuable feature for users of PRISM C on MICA.

Validation for the C compiler will use a test system derived from the VAX C test system. The compiler will also be tested via its use in compiling ULTRIX software.

PRISM C is documented in the Guide to PRISM C. This has the same organization as the Guide to VAX C, and the two manuals differ only where the systems differ. Helen Custer is the technical writer for this manual.

Direct questions about the PRISM C language to Lu Anne Van de Pas.

For information on the C runtime library on MICA, see the MICA project plan.

# 6 MISCELLANEOUS LANGUAGE SOFTWARE

This section covers the other PRISM software being developed by the DECwest compiler group.

## 6.1 Message File Compiler

Pillar has features for defining conditions and messages either casually (in a Pillar program) or in a message file. The Pillar message file compiler accepts a Pillar message source file. It produces a message file, an object module, and a Pillar definition module. The definition module can be imported by any module that needs to reference a message in the file. Note that there is a single source for all information about messages and conditions defined in the file.

To produce messages in a different natural language, the message file is edited and recompiled. The message file compiler will be able to import the original definition module and check the modified source file for consistency with it.

Pillar message files will be the system message files for PRISM. For VAX/VMS Pillar, the compiler will generate VMS message files.

## 6.2 SPASM. the Simplified PRISM Assembler

SPASM is the new assembly language for PRISM. It is not compatible with the language accepted by the interim assembler. SPASM will use the Pillar lexical analyzer, so it will be possible to use the full Pillar compile time facility with SPASM. However, to meet the diagnostic group's short term requirements for a macro capability, SPASM will have a simple intrinsic macro language.

SPASM will be used only in special contexts where normal programming language concepts do not apply. Customers can only obtain the assembler by special arrangement.

Gary Kimura is responsible for SPASM, including definition of the language.

## 6.3 Pillar Runtime Support

Pillar object code may use out of line complex code sequences and a few runtime routines that are not known to the user. These will be implemented by the DECwest compiler group. Because the executive of the PRISM operating system (MICA) is written in Pillar, this runtime will be packaged with the executive.

## 6.4 DST Analysis

The MICA ANALYZE command will have an option to analyze the debug symbol table in an object module or image. The DST specification is being done by the debug group in SDT. The DECwest compiler group is doing the analysis program.

# 7 THE DECWEST PRISM COMPILER

This is the PRISM compiler for Pillar and C, and it also includes the SPASM assembler. It will feature the industry's most advanced code optimization methods: interprocedural analysis, inline routine expansion, vectorization and decomposition, instruction scheduling, global register allocation, and performance profile feedback to improve all aspects of code generation and optimization.

The general goal for the compiler group is to produce a compiler that will meet PRISM performance goals and that will be perceived as being of higher quality than any existing VAX compiler. Here quality encompasses ease of use, compile speed, reliability (correct behavior), and object code performance. Object code performance is receiving special emphasis in the PRISM project, but all the goals are important. In particular, reliability must

not be compromised.

The compiler has roots in the family of compilers using the VAX Code Generator (VCG), but the design is completely new. It has been influenced by eight years of experience with the VCG compilers and by new work done at DECwest in the context of the SIL compilers and their Pascal predecessors. This section describes some of the newest features of the design: the modular organization and the code optimization methods.

## 7.1 Compiler Organization And Command Interface

The compiler is structured so that it can be easily integrated with new environments and hosted on a variety of systems. It contains:

1. a small super shell that contains all functions related to the host operating system and command interface,
2. a small language driver routine for each language,
3. a compiler shell providing general routines used by all parts of the compiler,
4. a separate front end for each language,
5. a back end that does optimization and code generation including all target-dependent code generation.

A complete compilation is controlled by the language driver. Supported by the super shell, it interprets the command line and establishes an environment for the compilation. The driver then calls the compiler shell to initiate the real work. To integrate the compiler with a new program-development environment, one only needs to modify the language drivers. For example, a fancy program development system can provide its own drivers thus bypassing the normal command interface.

The expansion of displayed text (e.g.; error messages) can be controlled by the language driver (using the super shell). There is complete flexibility in regards to the translation of messages into national languages.

The compiler can be packaged in various ways: one big image, a set of related shareable images, three separate images, etc. Whatever arrangement is finally chosen, the DECwest compiler group regards development of this compiler as one project being done by one team.

The shell and super shell are used in the Pillar definition module utility and the message file compiler.

## 7.2 Code Optimization

Almost all optimization is done in the compiler's common back end so that it will apply to all languages except SPASM. As in the

VCG compilers, the operators of the intermediate language are n-tuples, but all traces of PL/I have been removed, and the basic classification of data types deals only with size and alignment. From the point of view of the front ends, certain key operators, such as those for data references and procedure calls, are simplified. The intermediate language is always processed by the compiler's global optimizer, which converts the difficult operators into forms most appropriate for optimization.

The first phase of the global optimizer scans the intermediate language and extracts information about the calling relations between procedures and the usage of variables within procedures. This information is then analyzed to get sharp information about procedure calls and data aliasing. Here, as in many places in the back end, there is an option for quick analysis or deeper, more time consuming analysis. The user will not see these options directly, rather there will be a few practical command options to control the compiler. The default mode is for the maximum optimization consistent with quick compiling.

The second phase of the global optimizer performs conventional global optimization on the intermediate language. Procedures are processed separately using the results of the preceding inter-procedural analysis. The flow analysis method is a variation of recursive descent analysis that works on flow graphs and accommodates moderate usage of goto's. This method's running time is linear in the number of graph nodes, and the storage required for bit vectors depends (more or less) on the nesting depth of control structures rather than the number of nodes in the graph. Recognition of equivalent expressions uses a combination of hashing and self-adjusting binary trees, so the time spent is at worst n*log(n).

Inline procedure expansion and inter-procedural analysis both yield many opportunities for value propagation, which can result in the recognition of constant conditionals. The flow analysis and recognition of equivalent expressions is designed to exploit this, and the flow graph is simplified whenever possible. The flow analysis can be repeated on the simplified graph, and there will be a provision for repeating the interprocedural analysis using the sharper information found by flow analysis.

The global optimizer does standard optimizations such as loop unrolling and result incorporation. Over time we add many specialized optimizations of this sort, the most interesting being vectorization and decomposition (into parallel threads of execution). The optimizer also collects interference and life-time information for later optimization phases. When it's finished with a procedure, it produces the optimized intermediate language in the form expected by the local code generator.

The Local Code Generator (LCG) reads the intermediate-language operators produced by the global optimizer. It generates unbound code blocks which implement the operators. The unbound code blocks look something like instructions, but use register temporaries and symbol nodes rather than hardware registers to keep track of the storage of operands. The actual instructions being emitted by the LCG may also contain pseudo-opcodes rather than real instructions.

The code blocks are unbound in the sense that their interrelationship is not constant at the end of the code generation phase of the compiler. This allows the code scheduler to freely reorder the instruction stream based on the machine which the code is being generated for.

The LCG may also output more than one sequence of instructions for a given operator. For example, when a string of characters is to be moved from one location to another, the code generator might provide code blocks to move the string three different ways:

1. a straight inline code sequence, or

2. a loop to move the string, or

3. a call to a complex code sequence to move the string.

It is then up to another phase of the compiler which runs after the code generator to select which sequence should be used based on profile information, register usage, instruction stream cache information, etc.

The Code Block Optimizer (CBO) works on the unbound code blocks produced by the LCG. Actions of the CBO include:

1. Keeping track of the constants which are too large to be placed in the instruction stream itself and allocating the storage in linkage section to hold the constants.

2. Keeping track of constants which have been loaded into register temporaries. This allows the CBO to ensure that no constants are loaded into register temporaries which have already been loaded, therefore cutting down on the number of memory load instructions which are performed.

3. Coalescing register temporaries so that two register temporaries may exist in the same register temporary. This cuts down on the number of register temporaries that the register allocator must deal with. It also guarantees that a value in one register temporary which is simply being moved to another register temporary will not use two separate hardware registers.

4. Performing some peephole optimizations that apply before scheduling and register allocation. The CBO takes out some unnecessary instructions or changes their sequences so that there are fewer instruction code blocks.

Because the CBO is dealing with all of the code blocks output by the LCG at once, it has a much better view of the operations actually being performed. It also has the graphs that the optimizer built which it can use to make some decisions about how to optimize out code blocks. All of this allows the CBO to actually peephole code blocks across basic blocks.

The instruction scheduler runs after the Code Block Optimizer. It rearranges the order in which PRISM instructions are issued to minimize execution time due to stalled instruction issue cycles. It is an optional phase of the PRISM compiler. For each specific

PRISM processor, the scheduler uses a different model to describe its characteristics with respect to when it will stall on an issue cycle.

The scheduling algorithm can be tuned for various levels of optimizations. It can schedule basic blocks or entire procedures. Scheduling basic blocks is the simplest and fastest scheme where the instruction scheduler only rearranges instructions within a single basic block, one basic block at a time. When scheduling an entire procedure, the instruction scheduler is allowed to rearrange and move instructions anywhere within the procedure. The scheduler will use flow graph information (provided by the optimizer) and profile information to help schedule entire procedures. Intermediate degrees of scheduling will also be defined as warranted.

The register allocator runs after the instruction scheduler. Its task is to assign register temporaries to actual hardware registers, insert spill code as needed, finalize the decision on which procedures need or do not need a call frame, insert prologue and epilogue code, and complete the storage allocation. It uses flow graph information, call graph information (both provided by the optimizer), and profile information to help it assign register temporaries to hardware registers. An underlying goal in register allocation is to minimize hardware register usage, spill, and the need for call frames.

Like the scheduling algorithm, the register allocation algorithm is tuned for various levels of optimizations. A fast allocator will only process one procedure at a time and use a simple fixed point method for allocation. The more thorough allocator will use all of the flow information available and deal with global register usage. It will also assign parameters to nonstandard registers for procedure calls where it is possible and beneficial.

Several different register allocators, each using different algorithms, were experimented with during the development of the SIL compiler. The final allocator selected for use in that compiler uses a non-backtracking form of coloring algorithm and allocates registers across procedures within a compilation unit. This allocator was selected for that compiler based on its output relative to the actual processor time required to complete the compilation.

The optimizations discussed so far are based on the analysis of the procedures in a single compilation unit. The compiler is designed to work with very large programs, but there is still a need to carry out interprocedural analysis and register allocation across separate compilation units. Within DEC, this has been named universal optimization. A long-term goal for both PRISM compiler projects is to provide universal optimization in a way that is not tied to a single language or compiler.

To efficiently support universal optimization and processor sensitive optimization, especially instruction scheduling, the DECwest compiler will provide deferred code generation. The intermediate language representation of a module (or multiple modules) can be saved in a deferred object module and compiled later. The deferred compilation starts with the global optimizer

phase. The target processor can be specified at this time, and profile information or the results of universal optimization may be used.

## 7.3 Performance Analysis

Performance analysis of program code is receiving special attention in the PRISM Pillar and C compiler. In addition to its value for programming compute-intensive applications and system software, this sort of performance analysis contributes to the development of the compiler's code optimization methods and to the evaluation of hardware architectures and designs. The traditional separation between hardware performance analysis and compiler development has handicapped both VAX and PRISM development.

On PRISM, the most useful data for performance analysis appears to be execution profiles generated by code inserted by the compiler. This can be related to the structure of the program as seen by the user and also to the fine structure used in code optimization. The compiler will have the capability to generate this form of profile code, and the results can be fed back to improve optimization in a subsequent compilation of the same program. We have successfully experimented with this in the PRISM SIL compiler, including an experiment where the compiler varied the number of hardware registers and reported the resulting numbers of loads and stores.

The compiler will have a special option to accept information produced by the PRISM timing simulator. It will be able to display this information, the regular profile information, and its own code scheduling assumptions as part of the machine code listing. In addition, it can display interesting statistics. The point of this is to get all the relevant information together in a useful form for design feedback. This should eliminate inconsistencies in the hardware- and compiler- design assumptions.

# 8 VAX SOFTWARE

Pillar will be a product on VAX/VMS. The software involved is the Pillar compiler, the related runtime support, and the Pillar definition module utility.

The VMS Pillar compiler differs from the PRISM compiler only in the super shell and in those parts of the back end that depend on the target architecture. All of the compiler's general optimization apparatus is used on both VAX and PRISM.

The Phase 1 review for VAX Pillar will be held sometime after the PRISM Pillar V1 compiler is available. At that time, VMS-specific specifications for the compiler will be available for review.

### 8.1 VAX Software Dependencies

The plan for the VAX Pillar compiler depends on VAX Debug accepting the new format debug symbol table and providing language-specific support for Pillar ("SET LANGUAGE PILLAR").

## 9 BOOTSTRAP SOFTWARE

PRISM has a new hardware implementation architecture, a new operating system, and a new systems programming language. Developing the new system requires an elaborate bootstrap process. Most of the compiler group's work prior to June 1987 has been on software to be used in the bootstrap and then discarded. The compiler group is responsible for:

- The SIL cross compiler. This produces PRISM object code packaged in VAX object modules. The code can be used on the PRISM emulators or under PRISM simulators running on VAX. This compiler features instruction scheduling and global register allocation across procedures. It uses the PRISM calling standard and has options to assist in hardware evaluation.

- The VAX/VMS SIL compiler. This is used for the development of modules and programs that do not require PRISM-specific functions (e.g.; privileged instructions). The compiler does support the vector operations via a runtime package. Programs such as the PRISM and C compiler and the MICA linker will be developed using this compiler.

- An interim macro assembler. This is a modification of the VAX macro assembler that produces PRISM object code packaged in VAX object modules. Note that this assembly language will not be supported on PRISM.

The SIL compilers have an option to translate Pillar data declarations and procedure declarations into interim Macro.

The SIL compilers (PRISM and VAX) will be supported until the Pillar compilers (PRISM and VAX, respectively) are available.

Preliminary 64- and 32-bit Pascal compilers for PRISM were developed, used for a while, and retired.