SLAC-PUB-2246
January 1979
(M)

PASCAL/P-CODE CROSS COMPILER FOR THE LSI-11 *

Bruce L. Hitson
Stanford Linear Accelerator Center
Stanford, California

ABSTRACT

This paper describes the implementation of a cross compiler for Pascal that produces code that can be executed on an LSI-11 minicomputer. The approach taken is to first compile the source Pascal program (using an existing compiler) into an intermediate form known as P-Code. The P-Code is then cross compiled to LSI-11 assembly language. Once this has been achieved, the assembly language programs can be assembled using existing assemblers (such as MACRO-11) to produce relocatable load modules. These are linked together into an absolute load module and reformatted for transmission via serial line to the LSI-11. The details of the implementation are described. A comparison is also made between the approach taken in this implementation (cross compiling to the host machine's assembly code) and the approach where P-Code is interpreted directly.

INTRODUCTION

This paper describes the implementation of a cross compiler for the LSI-11 that converts Pascal pseudo-code (P-Code)[1] into assembly code that is suitable for processing by existing LSI-11 software. Unlike many other implementations of Pascal for minicomputers, this approach does not interpret P-Code, but instead produces LSI-11 assembly code [2] by cross compiling the P-Code statements that are output by the Pascal compiler. One of the goals of this approach is to generate code that will execute significantly faster than existing interpretive implementations without a severe increase in the total program size (program code plus runtime support routines). We would also like to allow programs written in Pascal and other languages such as Fortran, PL-11[3] and assembly language to be compiled separately and linked together into software packages that make use of the best features of each language. We are able to protect our large investment in existing software and yet be able to write new programs in a high-level language that should be easier to debug and maintain.

HARDWARE

The system is designed to be used on LSI-11 systems that have a minimal hardware configuration. A minimal system might consist of the LSI-11 processor, an EIA RS232 serial line interface for connection to the host computer, and a ROM kernel that can communicate with the host computer and download LSI-11 core images via the serial line from the host computer. In our case, the host computer is referred to as the TRIPLEX. It consists of two IBM 370/168s running OS/VS2 and a single IBM 360/91 running OS/MVT. All three processors operate under the ASP job management system. An alternative configuration could be based solely on an LSI-11 or other PDP-11 family computer with floppy drives and a large enough memory to execute the Pascal compiler. We have chosen to implement the first configuration for our current applications.

The systems in use at Stanford Linear Accelerator Center (SLAC) typically consist of an LSI-11 with serial line interface, 4K of ROM kernel routines, and .24K words of RAM. This allows execution of reasonably large software packages without having to be overly concerned about the efficient use of memory. Typical programs use only a fraction of the available memory for code storage, leaving the remainder for runtime stack and heap.

SOFTWARE

The software used to produce code that can be executed on the LSI-11 is, for the most part, written in Pascal. The only exceptions to this are the Pascal runtime routines which are written in assembly language for the sake of efficiency. The main programs used in the process of making an LSI-11 absolute load module are described briefly below.

1) Stanford Pascal compiler [4] - This is a highly modified version of the Zurich P2 compiler[5]. Modifications to produce P-Code that is also cross compiled into efficient IBM 370 code were done by Sassan Hazeghi of SLAC. This is the same P-Code that is cross compiled to LSI-11 code that eventually runs on the LSI-11.

2) P-Code Cross Compiler (PCC) - This is a 1500 line Pascal program that takes as its input P-Code produced by the Stanford Pascal compiler, and produces assembly code suitable for processing by standard LSI-11 assemblers such as MACRO-11. The detailed implementation of this program is the topic of the following sections.

* Work supported by the Department of Energy under contract number EY-76-C-03-0515.

(Contributed to DECUS Fall Mini/Midi Symposium, San Francisco, CA, November 29-30, 1978.)

3) Pascal runtime support - This collection of routines provides the standard procedures of the Pascal language (e.g., PUT, GET, EOF, etc.). It is currently written in assembly code for the sake of efficiency, and is in the process of being coded in Pascal.

4) SLAC LSI-11 Software [6] - This consists of implementations of programs such as MACRO-11 that run on the TRIPLEX and are used for assembling, linking, and loading LSI-11 code. This also includes routines for downloading programs via the serial line interface to remote LSI-11 systems.

All program development, compiling and linking is currently done on the TRIPLEX. The LSI-11 is simply downloaded from the TRIPLEX via the serial line and started executing at the beginning of the program that was loaded. Note that complex program systems may be loaded which may themselves consist of compilers, interpreters, etc.

IMPLEMENTATION DETAILS

Memory Organization

Memory is conceptually divided into three areas: Pascal monitor, program code, and runtime stack/heap. These are shown in Figure 1. The Pascal monitor performs the necessary initialization before entering the main Pascal program. It also does clean-up operations when the LSI-11 has finished executing and before control is returned to the TRIPLEX. The program code is the actual code for the routines of the Pascal program that is to be executed. The rest of the memory space is allocated to runtime stack and heap. The heap starts at the end of the program code and grows towards higher memory locations. The stack starts at the highest memory location and grows towards lower memory locations.

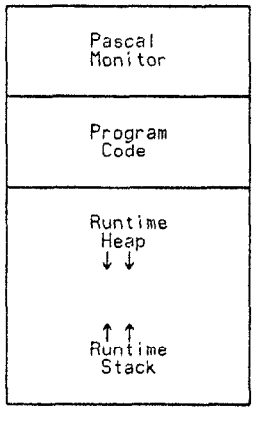

Figure 1 - Memory Organization

Data Types

Before discussing P-Code, it is useful to know the structure of the data that it will be referencing. There are six basic types of data: addresses (A), boolean (B), character (C), integer (I), real (R), and set (S). Boolean and character variables occupy one byte of storage each. Addresses and integer variables occupy one word (2 bytes) of storage each. Reals are represented in standard DEC floating point format, and occupy two words (4 bytes). Sets occupy four words (8 bytes), and can have up to 64 members. Alignment for each data type is provided for by the compiler according to the number of bytes it occupies. Thus, reals are aligned on 8 byte boundaries, while characters and booleans are aligned on single byte boundaries.

P-Code

P-Code is a pseudo-assembly code designed for a mythical stack computer (the P-machine[5]). There are two basic types of instructions: instructions that manipulate the top few items of the stack, and instructions that move data to and from "memory". The "memory" is actually part of the stack, and is accessed by specifying a pointer into the stack. A general P-Code instruction consists of four fields: OP, T, P, and Q.

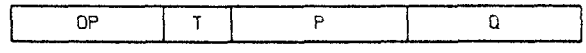

Figure 2 - P-Code Format

OP is a string of characters that specifies the operation to be performed. T is a single character that specifies the type of the operand to the instruction (e.g., I=integer, R=real). P and Q are used for a variety of purposes. They are most commonly used to specify a level and offset for instructions that load or store variables. The P-Code instruction set is described in the paper by Gilbert and Wall[1].

Referencing Variables

The P-Code produced by the Stanford Pascal compiler references variables by specifying two numbers: a level number and an offset. The level number specifies the lexical level of the variable being referenced. The scoping rules of the Pascal language require this to be interpreted as the lexical level of the most recently invoked procedure at the level specified. The offset specified is the number of bytes from the base of the specified lexical level where the variable being referenced is stored.

In order to make references to variables as quickly as possible, we would like to use the indexing capabilities of the LSI-11's general purpose registers. A number of registers, referred to as DISPLAY[1]..DISPLAY[n] are used to hold pointers to the base of the most recent activation of the lexical level (i.e., procedure or function) associated with n. To access a particular variable at lexical level n, we can use the indexed addressing mode of the LSI-11. Thus, to implement the P-Code instruction

LOD I *<level>*, *<offset>*

which loads an integer onto the stack, we can say

MOV -*<offset>*(DISPLAY[*<level>*]), -(SP).

A problem with this scheme is that we may want to access variables in more lexical levels than there are registers to hold

their base pointers. A solution to this problem involves keeping only the most commonly used DISPLAY registers in actual registers of the LSI-11. The remaining display registers are stored in memory, and loaded into registers only as they are needed. The concept of display registers has been discussed by Gries[7] and others.

As it turns out, the structure of many Pascal programs is such that most variables accessed are either local to the currently invoked procedure or are global variables (i.e., declared in the body of the program and *not* in a procedure or function). Taking advantage of this fact, only two registers are dedicated to holding DISPLAY register pointers. DISPLAY[1] is referred to symbolically as "GMP" (Global Memory Pointer), and DISPLAY[*n*] (where *n* is the level of the currently executing procedure) is referred to symbolically as "CMP" (Current Memory Pointer). References to variables in lexical levels other than those specified by GMP and CMP require that the value of DISPLAY[*n*] first be loaded into a temporary register which is then used for indexing.

In order to allow recursive procedures and functions, the value of CMP must be saved at each invocation. This process is described in the section on the Runtime Stack. The value of GMP need not be saved and is, in fact, fixed for the duration of a programs execution since a Pascal *program* (as opposed to a Pascal procedure or function) is not allowed to call another program at lexical level 1.

Runtime Stack

The format of the runtime stack is shown in Figure 3. Starting at the high end of memory, we have the stack frame for the main program. This consists of the return address to the Pascal monitor. Following this are five words of system variables, and three words of I/O buffer addresses. The I/O buffer locations contain pointers to buffers for up to six different devices. The default I/O device is the tty. The global variables are stored after the I/O buffer addresses.

When a procedure or function is invoked, a new stack frame is allocated. The first word of this stack frame is the return address to the procedure that invoked it. (In the case of the first procedure call, this will be the main procedure). The value of the CMP register must also be updated. This consists of 1) save the old value of DISPLAY[<*level*>] in the next stack location 2) load DISPLAY[<*level*>] with the current value of CMP 3) load CMP with a pointer to the return address that was pushed onto the stack in step 1...this is the base of the new stack frame.

The next four words on the stack are used to store the result of calls to routines that are functions. These four words are unused if the routine is a procedure. Local variables (variables declared in the level that we are now entering) appear next on the stack. The code for the routine whose stack frame was just created is now executed. At some random point in this routine, another procedure or function call may occur. If the call is to a function that is embedded in a calculation, some intermediate results of the calculation being done may be stored on the top of the stack. These are referred to in the diagram as temporary variables since they represent intermediate results At this point, a new stack frame for the function being called is created, and execution proceeds as described above.

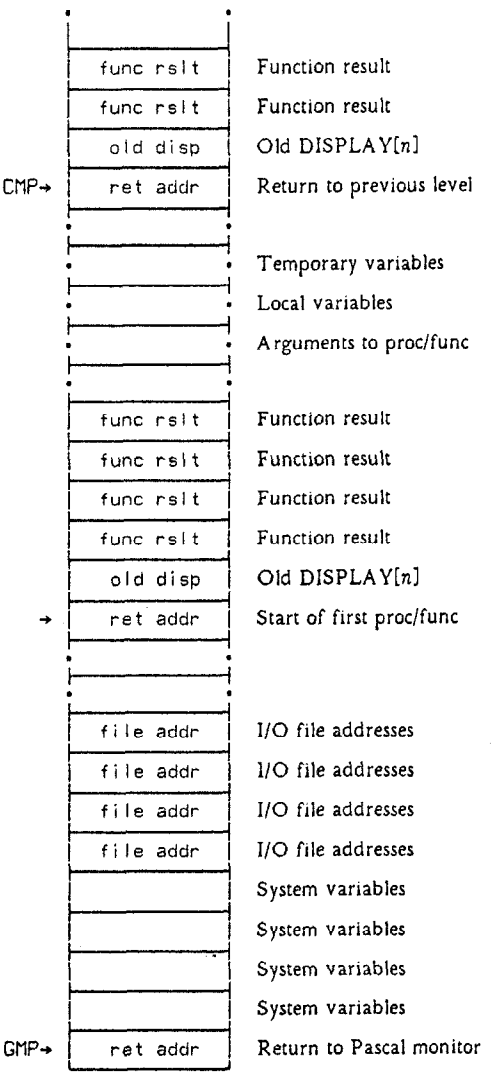

Figure 3 - Runtime Stack Format

OPTIMIZATION

At the present time, only a slight degree of optimization has been implemented. This manifests itself as not executing the standard system calls to do the initial reset/rewrite of the tty (since the tty is already initialized by the Pascal monitor). Also, routines to "start I/O (SIO)" and to "end I/O (EIO)" to the tty have been optimized out for the same reasons.

There are many places where the LSI-11 translation of a sequence of P-Code statements are relatively inefficient. Consider the Pascal statement "i:=i+1". If the variable "i" is an integer located at an offset of 14 in level 1 (the main program), the following P-Code might be produced:

```
LOD I 1,14          ;Get variable "i" onto stack
LDC I 1             ;Load the constant "1" onto stack
ADD I               ;(Top-1)←Top+(Top-1); Top:=Top-1 ;
STO I 1,14          ;Store the value back in "i"
```

The LSI-11 assembly code produced would be:

```
MOV -14(GMP),-(SP)  ;SRC=> LOD I 1,14
MOV #1,-(SP)        ;SRC=> LDC I 1
ADD (SP)+,(SP)      ;SRC=> ADD I
MOV (SP)+, -14(GMP) ;SRC=> STO I 1,14
```

Obviously, this is not the optimum solution since the single LSI-11 assembly statement

INC -14(GMP)

would have had the desired result. The increment instruction takes two words of memory, whereas the sequence produced above takes seven words!! This inefficiency is due largely to the differing architectures of the register oriented LSI-11 and the stack oriented P-machine. Efforts are currently under way by various people to produce optimized P-Code[8] that would eliminate some of the more obvious inefficiencies. We are working on a much less extensive optimization of replacing the above *<load><load><operation><store>* operations with a more nearly optimal solution. This should have a fairly significant effect in reducing program size.

INTERFACE TO OTHER LANGUAGES

One of the main objectives in cross compiling Pascal to LSI-11 assembly code was to allow Pascal routines to be linked together with routines from other languages. In this way, our existing software library of Fortran, PL-11, and assembler routines can be used along with Pascal routines to produce significant software systems. Also, this allows the use of a language that is most appropriate for the problem at hand. As an example, it is fairly inefficient to deal with low-level concepts such as bit masking or trap handling through Pascal routines (although not impossible). These routines can be implemented in PL-11 or in assembly language and linked in with the Pascal routines to produce a usable software package.

PERFORMANCE EVALUATION

As was mentioned in the introduction, one of the main motivating factors for the cross compiling implementation of Pascal as opposed to the interpretive approach is the speed with which the code executes. Although extensive testing has not yet been completed, a preliminary comparison of the Pascal system produced using PCC (PASLSI) and two other systems has been made. The performance was also compared to DEC Fortran running under RT-11 on the LSI-11. A simple integer bubble sort was used as a benchmark. It is a fairly good example since it tests a combination of performance characteristics such as loop efficiency and array indexing efficiency. Execution time was tested for the sorting of 500 integers (arranged in reverse order so that every integer must be moved). Three Pascal systems were compared: PASLSI, UCSD Pascal[9], and Stanford Pascal. The results shown below are typical of the three systems from measurements made so far.

Execution time of bubble sort of 500 integers
(measured in seconds)

Stanford Pascal (IBM 370) - 0.6
DEC Fortran (LSI-11) - 22.0
PASLSI (LSI-11) - 84.0
UCSD Pascal (LSI-11) - 463.0

Based on the preliminary results above, we can make a couple of comments. PASLSI seems to have a significant speed advantage over UCSD Pascal (which uses an interpretive approach). Compared to Fortran, PASLSI runs about four times slower at the present. It should be kept in mind, however, that the DEC Fortran has been optimized to a significant extent, whereas the PASLSI system still has considerable room for improvement. With some of the optimizations mentioned under "Optimizations", it seems reasonable to assume that PASLSI in its final form will probably execute within a factor of two of DEC Fortran.

CONCLUSIONS

The cross compiling approach to making Pascal available on a minicomputer such as the LSI-11 is a useful addition to our existing software package of Fortran, PL-11, and assembly language routines. It allows us to use programs written in Pascal together with existing software written in other languages with only minor changes to the existing software. High level programs can be written quickly and cleanly in the block structured environment of Pascal. Low level routines can be written that perform critically time dependent tasks or that can more easily access the lower level constructs of the LSI-11 than Pascal. Thus, a particular task can be written in the language that is most nearly suited to the task (be it execution-time critical, memory usage critical, or software development and debugging time critical). The system is in a continual state of improvement and extension, and will no doubt have many new features added by the time this paper is printed.

ACKNOWLEDGEMENTS

I would like to thank Sassan Hazeghi of SLAC for his help with Stanford Pascal and for the many Pascal programs he has graciously shared with me. I especially want to thank Les Cottrell, whose initial encouragement got me started on this project, and whose continued support, help, and advice keeps me going.

REFERENCES

[1] Erik J. Gilbert and David W. Wall "P-Code Intermediate Assembler Language (PAIL-3)", Stanford Artificial Intelligence Laboratory, on-line documentation, March 1978.

[2] Digital Equipment Corporation, "Microcomputer Handbook 1976-1977", Maynard Massachusetts.

[3] Russell, Robert D. "PL-11: A Programming Language for the DEC PDP-11 Computer" European Organization for Nuclear Research (CERN), Geneva, 1974.

[4] Sassan Hazeghi, "Stanford Pascal Compiler" online documentation at SLAC.

[5] K.V. Nori, U. Ammann, K. Jensen, H. H. Nagel, "The PASCAL <P> Compiler: Implementation Notes", Berichte des Institutis fur Informatik, Zurich.

[6] R.L.A. Cottrell and C.A. Logg, "An IBM 370/360 Software Package for Developing Stand Alone LSI-11 Systems", Proceedings of the Digital Equipment Users Society, vol. 4, no. 4, pp 985/991, April, 1978.

[7] Gries, David, "Compiler Construction for Digital Computers", John Wiley & Sons, N.Y. 1971.

[8] Sites, Richard L. "Progress Report, June 1978: Machine-Independent Code Optimization", Department of Applied Physics and Information Science, University of California, San Diego.

[9] UCSD (Mini-Micro Computer) PASCAL Documentation, Release Version 1.4, January 1978.