AN EFFICIENT ALGOL-60 SYSTEM FOR THE PDP8

Roger H. Abbott
A.R.C. Unit of Muscle Mechanisms & Insect Physiology
Department of Zoology, University of Oxford
South Parks Road, Oxford OX1 3PS, U.K.

ABSTRACT

A one-pass compiler translates nearly full Algol-60 into an intermediate language, whose instructions and variable addresses are 6 bits long. The run-time system loads the intermediate language into core memory, and performs the operations specified by its 64 instructions. Execution speed is limited by floating point arithmetic, and is nearly as fast as programs written in machine code. It is about 6 times faster than OS/8 Fortran on a machine with EAE, although compiled programs occupy only one-third of the space. Minimum hardware is an 8K PDP8 with teletype. The system can run under Monitor or OS/8. A 12K machine can use Field 2 for array storage.

INTRODUCTION

The purpose of a compiler is to provide an interface between a program written in a symbolic language and the equivalent binary patterns on which the hardware of the machine can operate. This applied equally to machine code and the so-called high level languages. There are three fairly distinct ways to set about this task.
(a) A compiler can be written to translate the symbolic language into binary or a high level language into symbolic machine code or binary. The machine can then run the compiler output as it stands.
(b) The compiler may translate a high level language into a code which is not machine code, but whose instructions perform the functions which are needed in the high level language concerned. A run-time program then examines these codes, and executes the tasks they specify by means of subroutines. We could include in this class compilers whose output, although machine code, consists mainly of subroutine calls. This latter method is not very efficient, because it takes more instruction bits to specify a hardware subroutine jump and the address of the subroutine than it does to have a code number specifying which out of a list of subroutines should be executed.
(c) The high level language can be stored in the machine as it stands, with no compilation. A run time program interprets it, in a manner similar to (b). This method has the dual advantage that the program is easily modified, and the system can be made conversational. Focal works in this way, as do Basic interpreters. The disadvantage of the method is that programs run relatively slowly. It is not really a competitor to methods (a) and (b), which are used when speed and economy of memory are more important than user interaction with the running of the program.

Since the execution of a high level language requires operations more complex than are provided by the machine instructions of most computers (and certainly the PDP8!), a program translated by method (a) will be longer than one translated into an intermediate code (b), because operations which could be performed by subroutine are done by open code. In the PDP8, floating point arithmetic will limit the execution speed of a well-designed system, because it must be done by software, so method (b) should not be noticeably slower than method (a). A 6 bit instruction allows 64 different codes, which is quite sufficient for running Algol. It also suffices as an address length, since 64 variables in any procedure plus 64 in the main program are adequate. Two 6 bit instructions can be packed into a single PDP8 word. Therefore, method (b) using 6 bit instructions is the best one for the PDP8.

DEC do not offer such a system, the nearest being 4K Fortran which interprets 12 bit codes. It was decided to write an Algol compiler because this is a much more convenient and powerful language than Fortran, and because the PDP8 lacked an Algol-60 compiler.

Design Objectives

As well as being efficient, a high level language system should be convenient to operate. In practice, on a small machine, this means that the translation should involve the minimum number of passes, with the compiler output being as short as possible. The run-time system should also be short, and be designed in such a way that the Algol program can use any peripheral devices that the machine has. It should not be geared to any particular operating system, such as OS/8 or Monitor, but should be capable of running under any such system.

THE OBJECT CODE

It was therefore decided that the Algol should be translated in a single pass into a form which could be loaded directly into memory. Because of the desirability of being able to include machine code statements in the Algol program, the compiler output should be compatible with PAL, so that compiler output and copied machine code could be compiled

together into absolute binary. In the PDP8, it is essential that page boundaries be irrelevant, which means that all label addresses must be 12 bits. (Variable addresses are 6 bits, as already mentioned). This was achieved by having three types of loadable item:
(a) A signed decimal number, which represents two separate 6 bit instructions.
(b) A label address, consisting of the letter L followed by a decimal number.
(c) Floating point literals. These consist of the pseudo-op FLTG, followed by the literal, which is simply copied from the Algol text, followed again by the pseudo-op DECIMAL.
Labels are defined in one of the ways allowed by PAL, either by their occurrence followed by a comma, or by their definition with an equals sign. In the latter case, they are usually equated with a previously declared label. It is a simple matter to have the loader replace these symbolic labels by their binary equivalents. Floating point literals are read into the floating point accumulator by the same routine that reads floating point numbers when the program is running. The loader transfers them to the program area.

THE COMPILER

The most often quoted advantage of Algol over Fortran is that procedures can be called recursively with the evaluation of factorial being used as an example. This is doubly unfortunate, firstly because factorial is most naturally and efficiently evaluated without recursion, and secondly because the main advantage of Algol is that the language is defined recursively. For example, in the condition statement:

if Boolean then S1 else S2;

S2 may be any statement, concluding another conditional:

if Boolean then S1 else if B then S3 else S4;

As a further example, the statement brackets begin.. ...end may be nested, and variables and procedures can be declared after any begin. Evidently, a language which is defined recursively requires a recursively written compiler. Algol provides recursion, and as it is an Algol compiler which we wish to construct, the obvious thing to do is to write the compiler in Algol.

ALGOL -----	COMPILED --------
'IF' B 'THEN' S1; S2;	B; JUMP IF FALSE L1; S1; L1: S2;
'IF' B 'THEN' S1 'ELSE' S2; S3;	B; JUMP IF FALSE L1; S1; JUMP L2; L1: S2; L2: S3;

```
'ELSE''IF' BS=366 'THEN''COMMENT' 366 IS 'IF';
'BEGIN''INTEGER' L1,L2;
L1:=IFCLAUSE; 'IF' BS=366 'THEN' 'ANY(33); STATEMENT;
'IF' BS#212 'THEN' LDEC(L1)
'ELSE''BEGIN' ABS; L2:=JMPIE"; LDEC(L1);
        STATEMENT; LDEC(L2)
        'END'
'END' CONDITIONAL
```

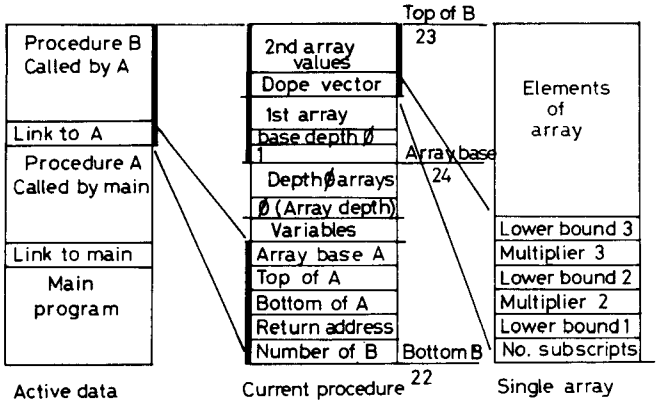
Fig. 1. Section of Algol Compiler

The top part of Fig. 1 shows, on the left, the two possible forms of Algol conditional statment. S2 may be another conditional statement, but S1 may not because the resulting statement is ambiguous. On the right are shown the translated equivalents, with the if, then and else removed. B stands for the code that evaluates the Boolean expression B, and S1, S2 and S3 for the codes that execute the Algol statements of the same names. "Jump if false" is a code whose job it is to examine the result of the Boolean expression, and jump to a label if it is false. Colons signify the definition of a label. Note that the compiled programs are the same up to the arrow. After the arrow, the code depends on whether S1 was terminated by ; or by else. The lower part of Fig. 1 shows the portion of the compiler which deals with conditional statements. It is part of procedure statement, which is called recursively in two places. Because of this recursion the label numbers of the two labels are held in locally declared integers, so that they remain intact through the recursive calls. integer procedure if clause compiles a Boolean expression, checks that the next symbol is then, outputs the conditional jump and returns as its value the label number of the conditional jump. The compiler then checks that the next symbol is not another if (S1 may not be conditional), and if not it compiles S1. Next it checks to see whether S1 was terminated by else (212). If it was not, all it has to do is declare the label L1, but if it was, it must compile a jump to a new label (L2), declare L1, compile S2 and finally declare L2.

The original intention was to write the compiler in full Algol-60, using a full compiler to compile itself. This proved to be impossible because of space problems. Firstly, it is necessary in a full system to check the types of procedure parameters at run time. This check is omitted in the compiler writing Algol system, which saves a great deal of space as the compiler consists mainly of procedure calls (the example in Fig. 1 consists entirely of procedure calls). Secondly, real quantities are not needed in the compiler, and so the compiler operating system does not have routines for dealing with them, leaving more space for identifier tables.

THE RUN-TIME SYSTEM

All run-time programs contain routines for doing arithmetic, evaluating Boolean expressions, entering subroutines and so on. The main feature which distinguishes Algol from Fortran is the way the data is organised, since variables are created as they are declared, and cease to exist when the block in which they are declared is left. In the PDP8 system variable allocation within a procedure is handled by the compiler. In addition recursion must be allowed for. Some text-books, modern ones included, state that this is one of the big difficulties of writing Algol systems, but in reality it is easy. The method is shown in Fig. 2. All that is necessary is to refer to variables by their position in the memory relative to a base pointer. This contains the address of Bottom in Fig. 2. Another pointer marks the next free space at the end of the variables. When a procedure is entered, the base pointer is set to the previous value of the next free space pointer, so that the new procedure has a section of memory all to itself. This arrangement is known as a stack.



ROGALGOL 8K DATA STORAGE

Fig. 2

Within the new procedure's memory are stored the previous values of the pointers, so that the machine can be restored to its previous state when exit from the procedure is called for. The return address is also held in this area. The first location in the procedure variable area contains a unique number which identifies the procedure. This is needed when a procedure called at a yet higher level refers to variables in the procedure under consideration. It is also used at labels declared in the Algol program, because such labels can be jumped to from procedures active at higher levels, in which case the pointers must be reset. The compiled program has at every label the identification number of the procedure in which the label is declared. At run time, this number is checked against the identifying number of the procedure level pointed at by the base pointer. If it is different, procedure levels are removed until the numbers correspond. Jumps into procedures which do not exist in the memory at the time of the jump are prohibited by the compiler.

Arrays present a special problem because they may appear and disappear within a single procedure and their size is not known until run-time. Arrays are held on a separate stack, which is embedded within the ordinary variable stack. Blocks in which arrays are declared are numbered by depth of declaration. At the beginning of each array level are two words, the first containing the current declaration depth, and the second the pointer to the base of the previous level. When the 12K overlay is in operation, a third word points at the next free space in field 2, where array elements are stored. The array base pointer is stored along with the top and base pointers in the link information. Each array starts with a dope vector, which contains all the information necessary to work out the address of an element, given the subscripts. This dope vector is set up at run time when the array is declared. In the 8K system, the array elements are immediately above the dope vector, but in 12K Algol the last word of the vector contains the address in field 2 where the array begins.

The operating system tape includes the loader, which occupies with its tables the memory which will be used for data storage when the program is running. Currently, the compiler output is loaded into field 0 starting at location 200, but the code is word-wise relocatable, and the system could easily be modified to load and run the code in any part of any memory field.

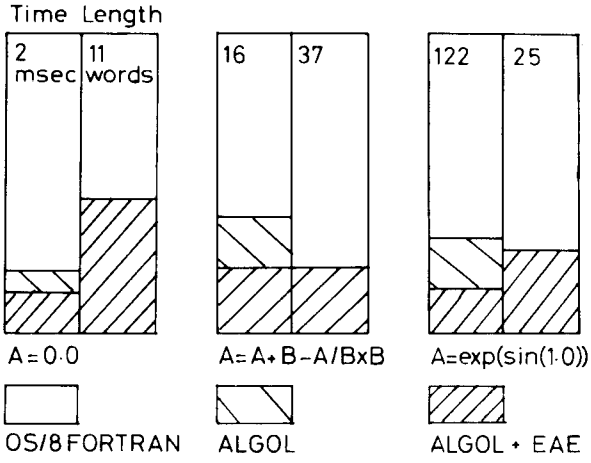
INPUT/OUTPUT

All the built-in input/output procedures have as their first parameter a device number, which must be in the range 0-7. The numbers are logical device numbers, and are used to address a table of input/output machine code routine addresses. Users can assign any device to any number by placing the address of the routine in the table, using an overlay to the run-time system. In the standard system device 0 gives a failure indication in input proccedures, but can be used to suppress output by the output procedures. Device 1 is the teletype and device 2 the high-speed reader/punch combination. Device 3 is the systems device, whose routines are written as an overlay to the run-time program, so that various operating systems can be catered for. Currently, Monitor and OS/8 overlays are available. Although the input/output procedures are normally used for just that, the organisation of the run time system allows them to be used for activating any piece of machine code.

SYSTEM PERFORMANCE

Speed

The speed attainable in a program which uses floating point arithmetic is limited by the speed of the floating point software. The statement A:=A+B-A/BxB has been timed in a program written in machine code and in Algol. In a machine which has no EAE Algol is only about 15% slower. If an EAE is available, Algol is about 80% slower, although it is nearly twice as fast as on a machine without EAE. It is believed that this extra time is spent mainly in needless arithmetic stack operations. It is planned to re-write the run time system to avoid these, and when this is done Algol should be nearly as fast as machine code on a machine with EAE.



EXECUTION TIME & CODE LENGTH

Fig. 3

Fig. 3 shows a more detailed comparison between OS/8 Fortran and Algol. In each case, the Algol values are represented as a fraction of the OS/8 Fortran values. Without EAE, Algol is about 3 times as fast, and with EAE about 6 times as fast. Fortran is hardly speeded up at all by use of the EAE, because its speed is not limited by the speed of the arithmetic routines.

Storage requirements

Fig. 3 also shows that the compiled Algol code is only one-third of the length of compiled Fortran code. However, the saving in space is greater, for two reasons. Firstly there is a greater amount of memory available for storing programs. Fig. 4 shows a memory map of an 8K machine, the cross-hatched areas are ones occupied by the system, and

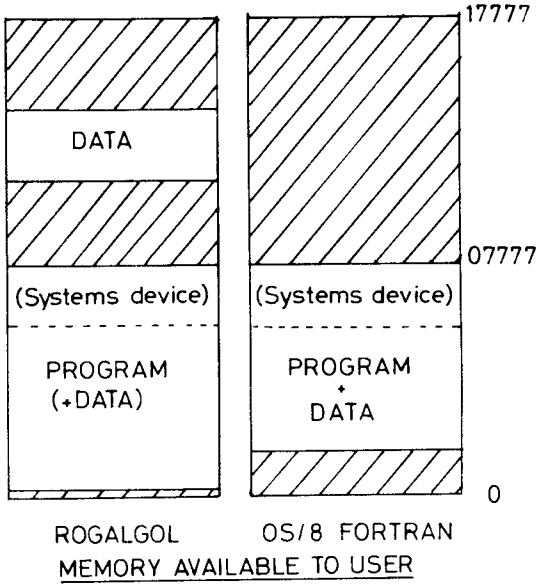


Fig. 4

The hatched areas are occupied by system routines. The items in brackets are optional.

not available to the programmer. The Fortran system is evidently much longer, although it has to be admitted that this is partly due to the greater facilities of the input/output handler.

The second reason is more subtle. When using machine code, we automatically think of writing a program as a series of subroutines, which are often short, because this saves space and makes the logic of the program easier to follow. Fortran is very bad at subroutines, because each one occupies at least one page, and has to be compiled separately. This is sometimes quoted as an advantage of Fortran, and although this may be true in general it is certainly not true of the PDP8 implementation. Algol is efficient in this respect. In the system described here, the minimum length of a compiled procedure is 3 words, compared with Fortran's 128 words.

The Algol compiler is about the same length as the Fortran compiler. The complete Algol run-time routines are about as long as the linking loader program needed by the Fortran system.

A good starting reference for those wishing to learn more about Algol Compilers is Vol. 3 of Annual Reviews in Automatic Programming.