MACHINE LANGUAGE PROGRAMMING IN AN UNDERGRADUATE COMPUTER SCIENCE CURRICULUM

Ian Sommerville,
Dept. of Computer Science,
Heriot-Watt University,
Edinburgh, Scotland.

ABSTRACT

This paper examines the advantages and disadvantages of teaching machine lnaguage programming to computer science undergraduate students. A teaching language based on reverse Polish notation, but with high-level control constructs, is presented as an alternative to conventional assembly language. Experiences with using this language are described.

INTRODUCTION

Since the introduction of undergraduate computer science courses in universities an assembly language programming module has been widely regarded as an essential part of the curriculum. Such a module is usually included either at the end of the first or at the beginning of the second year after students have learnt some high-level language. Their programming experience, at this stage is usually fiarly limited.

The inclusion of such a machine language programming course offers some benefits:-

(i) Assembly code is used a great deal in the commercial programming environment. Exposure to a low-level language provides a practical training for the student.

(ii) Programming at machine level provides knowledge and understanding of the host machine and of concepts of machine architecture.

(iii) Assembly language programming gives students an indication of how high-level language statements may be translated to machine code. This provides background material for a compiling course.

Benefits (ii) and (iii) above are very real but the first 'benefit' is of very dubious nature. Commercial utilisation of assembly languages is often based on ill-considered ideas of efficiency. Thankfully more and more users are realising that machine efficiency and system economy are not synonymous terms and are turning away from assembly language programming.

As a result, the market for machine language programmers is contracting and probably only a fiarly small percentage of graduates will be employed in this task. Time spent in 'practical training' can therefore be time wasted for most students.

However, there are also a number of disadvantages involved when a course in assembly language programming is included in the curriculum:-

(i) A course of this nature using a language such as IBM S/360 assembler is time consuming for both staff and students. Typically a course like this occupies 20-25 lecture hours. About half this time is spent covering background material needed before the student can run a program.

(ii) Assembly language programming is error prone and most assemblers are not noted for the quality and readability of their error diagnostics. This is discouraging and the students motivation is rapidly eroded.

(iii) 'Bit twiddling' constructs are available most assemblers and inexperienced programmers see no reason why these should not be used. Hence, the more able students tend to develop "dirty" programming habits.

Our experience has been that, in teaching a conventional assembler, the time spent by both staff and students is disproportionate to the returns obtained in terms of the student's understanding of computing concepts. However, we recognise the advantages which

accrue from learning a low-level language and are relectant to lose them by merely teaching high-level languages.

It was, therefore decided to design and implement a low-level language for teaching purposes. The language had to illustrate principles of machine architecture and the translated form of high-level language statements. Essential requirements for this system were identified as:-

(i) The language should be easy to use and understand.

(ii) Cleanliness and consistency should be inherent - "dirty" programming should be impossible.

(iii) The implementation should be such that discouraging and inexplicable errors should not occur.

(iv) The language should be teachable in not more than eight one hour lectures.

It was finally decided that the most suitable type of language was not a conventional assembler-like language, but a form of reverse Polish notation.

As the students had previously attended an ALGOL programming course, a reverse Polish language is eminently suitable for illustrating the translation of ALGOL statements to some lower level form. Our language, POLLY, is described below.

THE PROGRAMMING LANGUAGE POLLY

A POLLY program consists of variable and procedure declarations followed by a reverse Polish string of instructions. Basically these have a one-to-one correspondance with machine instructions but higher-level control statements have been included in the language. The language description below is very informal and illustrates, mostly by example, the language POLLY. The system is described fully in the appropriate reference manual [1].

Variables

All variables used in a POLLY program must be declared and initialised before they are used in instructions. Variables are not typed and both simple variables and arrays are declared using a %DEF declaration. For example:-

```
%DEF  A=1, B=2, C(3)=(1,2 3);
```

This declares A and B as simple variables, initialised to 1 and 2 respectively, and a 3 element array C with the elements initialised to 1, 2, and 3.

```
%DEF  STR(16) = "THIS IS A STRING",

      BIGARRAY(100) = (0*50, 1*50);
```

Declares two arrays, STR initialised to the character string "THIS IS A STRING" and BIGARRAY. The first 50 elements of BIGARRAY are initialised to 0, to last 50 to 1.

Procedures

The POLLY programmer may name sections of code by declaring them as a procedure. Procedures do not have parameters but local variable declarations are allowed. An example of a procedure declaration is:-

```
%PROC  ANYPROC
  [
   %DEF  P=0, Q=0;
         :
         :
      POLLY instructions
         :
         :
  ];
```

Procedures are activated using the %CALL statement thus:-

```
%CALL ANYPROC
```

Parameters may be passed to a procedure by leaving their addresses or values on the machine stack before calling the procedure and storing them in local variables on entry to the procedure.

Statements

The machine instructions in a POLLY program are written as a reverse Polish string with the elements of that string separated by commas. For example:-

```
@C, A, B, +, 6, -, %STORE
```

corresponds to the assignment statement

```
C := A + B - 6
```

Notice that variable and constant values are stacked merely by writing the variable name or constant itself. A variable address is stacked by preceding the name by an @ symbol.

The usual arithmetic and conditional operators +, -, *, /, =, >, etc. are provided along with stack operations such an %UNSTACK, %DUP (push a copy of the top element) and %SWAP (swap the top two elements).

There is no mechanism in POLLY for array indexing and this must be handled by the programmer. For example, element P+Q of array A would be loaded onto the machine stack as follows:-

```
@A          ! Load base address of array !
P,Q,+,+,    ! Compute index and add to
                base !
%LOAD       ! Load value onto stack !
```

The language has a number of operations for working with arrays:-

%LOAD	Replace address on top of stack with its contents.
%LDSAVE	Push contents of address on top of stack onto the stack.
%STSAVE	Store top of stack at address in second top element. Leave address on the stack.
%INC	Add 1 to top of stack.

Control Statements

Unlike most languages at this level POLLY does not use conditional and unconditional goto statements to control the flow of the program. Rather, two higher level statements, the IF statement and the REPEAT statement are provided.

The IF statement is the familiar two-armed conditional but with the condition preceding the if part. For example:-

```
A, B, =, %IFTRUE X %ELSE Y,
```

If A=B then X otherwise Y is pushed onto the stack.

```
A,B,=,C,D,>,%AND.%IFTRUE(A,B,+)
```

If A=B and C D then A+B is pushed onto the stack. Notice that brackets group POLLY instructions into a compound statement.

A loop may be programmed by preceding a compound statement with the command %REPEAT. This will cause that statement to be repeatedly executed until either a %BRKTRU (conditional break) or %BREAK (unconditional break) instruction is executed. Control is then transferred to the following POLLY instruction. For example:-

```
%REPEAT (1,+,%DUP,10,=,%BRKTRU)
```

This will add 1 to the top stack element until it is equal to 10. The compiler checks that there is always a break statement as part of the compound statement thus avoiding one source of infinite loops. These control statements were chosen for POLLY on the basis of their simplicity, their relatively structured nature and because they fit into a reverse Polish notation.

Input-Output Instructions

Input/output instructions are often difficult to understand and use at this low-level. Hence a simple but adequate set of I/O instructions have been included in POLLY.

These are:-

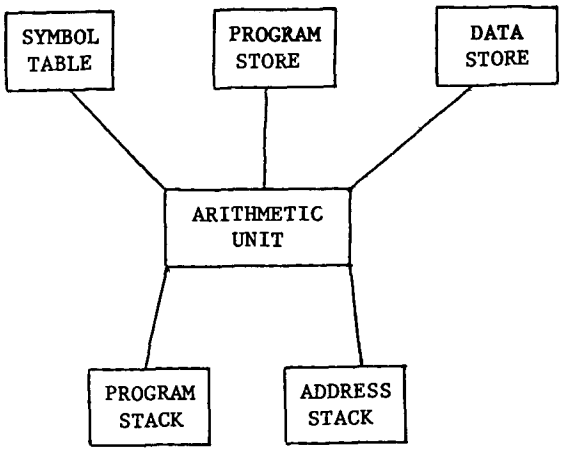
%READ	Reads a number onto the stack.
%READCH	Reads a character onto the stack.
%READSTR<array name>	Reads a string of characters into the given array.
%WRITE	Prints the top of the stack as a number.
%WRITCH	Prints the top of the stack as a Character.
%WRISTR<array name>	Prints the string held in the given array.
%NEWCARD	Go onto next card.
%NEWLINE	Print on new line.
%NEWPAGE	Print on new page.

Experience has shown that these commands are easy to use and not particularly error prone.

IMPLEMENTING POLLY

POLLY is implemented using a compiler/interpreter system with the underlying abstract machine designed to provide diagnostic facilities and to protect the programmer from himself.

The organisation of this machine (STAC) is diagrammed below in figure 1.



Organisation of the STAC Machine

Figure 1

Notice that the machine has five distinct data areas:-

(i)	A program stack	Used for computation
(ii)	An address stack	Stores return addresses
(iii)	A data store	Stores program variables
(iv)	A program store	Stores compiled program
(v)	A symbol table	Stores information about declared names.

The organisation ensures:-

(i) That the student cannot overwrite his program code
(ii) That return addresses cannot be accidentally used as data.
(iii) Variable names are available for diagnostic purposes.

All POLLY variables and procedures are addressed indirectly via the symbol table. For example, the statement

%CALL ANYPROC

compiles into

<call op code> <symbol table address of ANYPROC>.

This addressing technique makes the implementation of diagnostic facilities very straightforward. For example a statement

%TRACE A, B, C

which prints the values of A, B and C whenever they are changed may be simply implemented by flagging A, B and C in the symbol table.

In addition to this trace statement the following diagnostic facilities are provided:-

(i)	CTRACE	Diagnostic information for each instruction executed is printed.
(ii)	PTRACE	A trace of procedure calls is printed
(iii)	SDUMP	Prints the program stack.
(iv)	VDUMP	Prints values of all declared variables.

POLLY IN USE

In this section, we discuss the reaction of the students to POLLY, and compare that reaction with the feelings of students who were taught a conventional (S/360) assembler language.

To our surprise, students who had no previous knowledge of low-level programming reacted to POLLY extremely well. The class, in general, progressed very quickly and the notion of reverse Polish notation was easily understood. Students enjoyed the course, found the language interesting (many tackled signigicant personal programming projects) and some suggested extensions to the language.

Programming projects with a time limit set on the basis of conventional assembler projects were, in many cases, completed in less than half the allotted time and were generally well programmed.

After learning and using POLLY, the student's understanding of ALGOL concepts was improved. This is especially true of procedure parameter passing techniques, which POLLY illustrated very well.

These reactions may be compared with those of another group of slightly more advanced students who were concurrently attending a course in S/360 assembler. In general this language was disliked for its inconsistent and unstructured nature. Projects were late, badly programmed and, in some cases, were carried out mechanically by programming in ALGOL and hand translating this to assembler. They questioned why they had to learn that language rather than POLLY.

Subsequent questioning (6 months later) of each group of students revealed that those students who studied POLLY appeared to have assimilated the concepts of stack machines and reverse Polish programming. On the other hand, the S/360 group seem to have forgotten almost everything they learned about the 360 structure.

A total of seven one-hour lectures were spent teaching POLLY. These were broken shown as follows:-

Lecture 1	Stacks and reverse Polish notation
Lecture 2	POLLY Declarations
Lecture 3	POLLY Statements
Lecture 4	The STAC Machine
Lecture 5	Procedures and parameter passing
Lecture 6	Arrays
Lecture 7	Use of diagnostics

As a result of our experiences, conventional assembly language teaching has now been dropped from our course. The time saved (15 hours) is devoted to a course in comparitive machine architecture. We feel this benefits the students more than wrestling with the idiosyncracies of assembly code.

REFERENCES

1. POLLY - A Reverse Polish Programming Language.
Ian Sommerville,
Department of Computer Science,
Heriot-Watt University.
Revised edition June 1976.