UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT

The Thoth Assembler Writing Kit

Michael A. Malcolm
Gary J. Stafford

September 1977
CS-77-14

The Thoth Assembler Writing Kit

Michael A. Malcolm
Gary J. Stafford

Department of Computer Science
University of Waterloo

September 1977

This research was supported by the National Research Council of Canada.

The Thoth Assembler Writing Kit

Michael A. Malcolm
Gary J. Stafford

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Abstract: The Thoth assembler kit is designed to support the Thoth portable programming system being developed at the University of Waterloo. Thoth Assemblers are capable of cross or on-site operation.

Approximately 85% of the code remains unchanged across implementations. About 5% of the code is generated automatically from descriptions of the target machine. The remaining 5% to 10% must be provided in the form of small well-specified functions which do error detection and combining of opcodes with operands.

To date, assemblers for six different machines have been generated. The average time to completion for each of these assemblers has been less than eight man hours.

1. Introduction

In this paper we describe one component of a programming system being developed at the University of Waterloo. This system, called Thoth, consists of a set of software tools that can be used for programming on and for a variety of different machines. Other components of the Thoth system software include a portable compiler for a high-level language (Braga, 1976), and a portable real-time operating system (Cheriton, et. al., 1977). The compiled and assembled modules are relocated and linked by a portable linking loader (Sager, 1977).

It is not possible to avoid assembly coding altogether, even though the Thoth language provides the ability to emit machine instructions inline. We find that on most machines, assembly language is needed for initializing interrupt vectors, processess activation and deactivation, interrupt and fault handling, and other similar functions. We also find assembly language useful for certain functions critical to system performance. Since the Thoth compiler output is incompatible with that of assemblers supplied by machine vendors, we implement our own assemblers. To avoid having to write a new assembler for each new machine, we have implemented a "kit" which substantially reduces the effort required to build each new assembler.

The Thoth operating system requires approximately 500 lines of assembly code for each implementation. Other system and user programs require no assembly code. Therefore, our assemblers need not be efficient or powerful; we have not attempted to make them so.

Before proceeding, we will define a few terms. An *environment* of a program is the combination of hardware and system software required by the program. The *host environment* is the environment in which the program executes; the *target environment* is the environment for which the output of the program is intended. All programs have host environments, but only certain system programs such as assemblers, have target environments. We say that a program is *portable* over a set of environments if it is significantly easier to modify it for each environment than to implement and maintain separately. If moving a program to new host environments requires no modification, it is said to be *machine independent*. A program is said to be *machine invariant* if it requires no effort to convert it to a new target environment. A program that is not machine invariant is said to be *machine specific*.

Thus, portability is a matter of degree, and portability problems may take many forms. Our assemblers are machine independent over the set of machines which use the Thoth operating system. In this paper, we will focus on the design features which make it easy to adapt a Thoth assembler to a new target environment.

The Thoth assembler kit evolved from an earlier "portable assembler writing kit" described by Malcolm, Sager and Stafford (1976). Recent changes include the elimination of certain directives and the addition of six new directives, more operators, and the addition of manifest constants and a file indirection capability. The present assembler is somewhat more portable than the earlier version; this is mainly due to improved programming style rather than the development of new techniques.

2. Structure of the Assembler

On close inspection, one may observe that large portions of a typical assembler consist of algorithms which can be expressed independently of the target or host environments. These algorithms include symbol table managment, processing of assembler directives, lexical scan, expression evaluation, and creation of the output module. Our experience which is similar to that of others (e.g., Mueller (1976) and Wick (1975)). indicates that these algorithms constitute approximately 85% of the code in an assembler. If this invariant portion is written in a machine independent fashion, it is only necessary to produce the remaining 15% when porting to a new target environment.

The syntax for Thoth assembly programs is given in Appendix 1. Each Thoth assembly language includes directives, pseudo opcodes, machine-specific instruction opcodes, identifiers, constants and a set of operand symbols. The lexical scanner classifies tokens into these categories; a token which satisfies the syntax rules for identifiers is compared with symbols in the symbol table to categorize it as a directive, pseudo opcode, instruction opcode or identifier. If the symbol is an instruction opcode, it is further categorized into one of the machine-specific instruction classes. Tokens beginning with a numeric character, **$** or **'** are categorized as constants. For numeric constants, the first character specifies the radix: **0** indicates base 8, **$** indicates base 16, otherwise the base is 10. Character constants consist of a single ASCII character enclosed with single quotes. The value of a character constant is the value of its ASCII representation. Some nonprinting characters can be represented in an "escaped" notation; the rules for this are the same as for the Thoth base language described by Braga (1976).

The 15% of each assembler which is machine specific consists of functions for error checking and specifying the semantics of machine opcodes and certain pseudo opcodes. The structure of these functions is predetermined both by the manner in which data is provided to them by the higher level machine-invariant functions and by the information they must pass on to lower level machine-invariant functions. Because of this structure, it it possible to construct them in skeletal form from parameters describing the target machine. This is done by an interactive program called Helper (see Appendix 3). Helper asks the assembler implementer for parameters describing the sizes of storage cells on the target machine, the sizes of instructions, etc. Based on this information, Helper outputs skeletal forms of the functions which must be augmented by hand. Typically, the amount of code added by the implementor is 5% to 10% of the total assembler, or from 125 to 250 out of a total of 2500 lines of code. In addition to these functions, the implementor must prepare a table of machine opcodes and their values, and categorize each operation into a "class" which indicates the number and position of operands within the instruction.

Of the time required to implement the assembler on a new machine, approximately 50% is spent building the table of machine opcodes, 5% in interaction with Helper and 45% augmenting the code skeletons.

3. Expressions and Relocatability

Expressions are evaluated in left-to-right order, but this can be over-ridden with parentheses. The standard set of operations available include addition +, subtraction −, bit-wise *and* &, bit-wise *or* |, *exclusive or* ↑, multiplication *, division /, modulus %, left shift <<, and logical right shift >>. The special symbols ++, −−, @, !, #, [and] are not processed by the evaluator, but their presence at either the beginning or end of an expression is reported to machine specific functions, which may interpret them as appropriate to the target machine. For example, the implementor may choose to have the symbol @ indicate indirection.

The machine-invariant code of the assembler evaluates "relocatable" expressions which are subject to modification at load time. Here we have the advantage of knowing fully the operational characteristics and capabilities of our relocating loader, (Sager, 1977). Our loader allows programs to be loaded in up to 8 separate relocatable "sections" and have up to 8 different fields it can relocate. The programmer indicates the relocatable sections of his assembly program with the directive:

.rel <expr>

where <expr> must evaluate to an absolute number between 0 and 7. The value must be chosen to be compatible with the relocations used by the compiler, so the user is expected to have some knowledge of the language implementation before doing any serious assembly language programming.

Values in the symbol table have an associated attribute which may be either absolute (i.e. known at assembly time) or relocatable (resolved at load time). In the case of relocatable values, a further distinction is made to indicate the relocation which will apply at load time. External symbols have attributes which cannot be

determined until load time, so Thoth assemblers give them a relocatable attribute which will not match that of any other symbol at assembly time.

As an expression is evaluated, each operation and the attributes of its operands are used to determine the attribute of the result. For + and − operators, the rules for deriving attributes of results is best presented in the form of tables:

		right operand		
	+	A	R	R′
left operand	A	A	R	R′
	R	R	E	E
	R′	R′	E	E

In this table, A stands for absolute, R is a relocatable and R′ is a relocatable whose attribute is different from R (i.e., it refers to a different relocatable "section" of code). The symbol E is used to indicate an error due to an illegal combination of operands. The table then tells us, for example, that it is not possible at any point during an expression evaluation to add a relocatable to another relocatable. The assembler will flag these types of errors. The table for the subtraction operator is:

		right operand		
	−	A	R	R′
left operand	A	A	E	E
	R	R	A	E
	R′	R′	E	A

For the operators *, /, %, << and >>, the only legal combination is absolute for both the left and right operands. Any other combination is in error.

4. File Inclusion and Manifests

Two facilities of our compiler have been incorporated into the assembler. One is the ability to "include" or "redirect" the source input stream via directives in the source. The other is a form of textual macro (without parameters), called *manifests*. These two features are related in the sense that we often use file inclusion to redirect the input stream to a file of manifest definitions. Manifests are usually used to parameterize the code, describe record formats, or to delay and localize the binding of certain decisions which are likely to change. The combination of file inclusion and manifests allows the programmer to use the same file of manifest definitions for high-level language modules and assembly modules.

File inclusion directives can appear anywhere in the source; they must have a % in column 1. The remainder of the line is a pathname of a file. In effect, the contents of the file replace the file inclusion directive line.

Manifest definitions can appear anywhere in the source input stream; they must have a # in column 1. The format for a definition is:

 # manifest name =manifest text

The *manifest name* can be any legal identifier, which must be unique throughout the assembly. The *manifest text* begins immediately after the = and is terminated by the first newline or \ character. (Note the \ begins a comment.) All further occurrences of the manifest name, except in lines beginning with % or #, are replaced by the manifest text as the assembler scans the source input.

5. Implementing a New Assembler

After studying the new target machine, the assembler implementor should have an interactive session with the Helper program, as illustrated in Appendix 3. This results in the creation of files which contain manifest definitions for machine-specific parameters of the invariant code, and skeletal forms of functions which must be completed by the implementor.

The implementor must provide a function for each class of operation code which he specifies in the opcode table. These functions perform a number of tasks, as outlined below:

```
Opcode_n()
  {
    global variable declarations
    local variable declarations

    if( pass 1 )  skip to end of input line

    if( pass 2 )
      {
        call expression evaluator for each operand

        verify range and attribute of each operand

        output assembled data with relocation information
      }
    increment location counter
  }
```

Since the operations which must be performed in an Opcode function are fairly standardized, lower level functions are provided to simplify the implementation. These lower level functions do expression evaluation, code emission, etc.

In order for the output modules to load correctly with those of the compiler, the compiler and assembler implementors must agree on the fields which are to be relocated, how many relocatable sections are to be used, and on how this is represented in the load code. For some machines, the assembler uses more types of relocation than the compiler.

6. Conclusion

We have used the Thoth assembler kit to generate assemblers for the Honeywell HIS 6000 series, Texas Instruments TI 990, Data General NOVA, Microdata 1600/30, Interdata 70 and Motorola M6800. The average time to completion for each of these assemblers is less than eight man-hours for a person experienced with Thoth assembler generation but not with the target machine. Time to completion for the Microdata by a person unfamiliar with both the Thoth assembler kit and the target machine was approximately 13 hours.

The HIS 6000, TI 990 and NOVA assemblers have been used to write support software for Thoth, our portable real-time operating system (Cheriton, 1977).

A more formal approach to automating the generation of assemblers has been investigated by Wick (1975). It is difficult to compare the effectiveness of Wick's approach with that discussed here.

7. **Bibliography**

Braga, R. S. C. (1976), Eh Reference Manual, University of Waterloo, Computer Science Department, Research Report CS-76-45, November.

Braga, R. S. C., Malcolm, M. A. and G. R. Sager (1976), A Portable Linking Loader. *Symposium on Trends and Applications 1976: MICRO and MINI Systems* (an IEEE/NBS conference). May, 124-128.

Cheriton, D. R., M. A. Malcolm, L. S. Melen and G. R. Sager (1977), Thoth, a Portable Real-Time Operating System, Univerisy of Waterloo, Computer Science Department, Research Report CS-77-11, October. (to appear: 1977 ACM SIGOPS Conference)

Malcolm, M. A., Sager, G. R. and G. J. Stafford (1976), A Portable Assembler Writing Kit. *Mini- and Microcomputers '76 Symposium,* Toronto, Ontario, Canada, November.

Mueller, R. A. (1976), Automatic Generation of Microcomputer Software. Master's Thesis, Dept. of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, April.

Sager, G. R. (1977), The Thoth Linking Loader, University of Waterloo, Computer Science Department, Research Report CS-77-15, October.

Wick, J. D. (1975), Automatic Generation of Assemblers. Ph.D. Thesis. Department of Computer Science, Yale Univeristy, December.

Appendix 1. **Syntax of Assembly Programs**

The following characters are used in source programs:

1. letters: **A B C D E F G H I J K L M N O P Q R T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z . _**
2. digits: **0 1 2 3 4 5 6 7 8 9**
3. delimiters: + − ( ) [ ] ! ' **$** | & << >> * / % : ; , \ # @ ***n** and blank.

The ***n** is a "newline" used to delimit input lines; the character \ will cause the scanner to ignore the remainder of the input line (this is for entering comments).

In addition to the delimiter characters, the tokens recognized by the scanner are:

1. special tokens: ++, −−
2. directives: **.align, .ext, .ent, .equ, .loc, .rel, .ptr, .ds, .ptr, .str .dc1, .dc2, .dc3, .dc4, .formal, .function, .nargs, .stack**
3. implementor-defined opcode mnemonics
4. user-defined constants and identifiers

Since the forms of opcode argument lists vary from one target machine to another, it is not possible to give syntax rules that apply fully to all Thoth assemblers. The following grammar is intended to give the general flavor.

As with our high-level language, identifiers are defined as a letter followed by zero or more letters and digits, not to exceed 32 characters in length. Similarly, the syntactic rules for forming a string-constant are the same as for the Eh language, and the same escaped characters are allowed (see Braga, 1976).

module−list	::=	*module module−list*
module	::=	*line−list* .end
line−list	::=	*line line−list*
line	::=	*label−list statement* '*n'
label−list	::=	*label label−list* null
label	::=	identifier :
statement	::=	.align *expr* .ds *expr* .ent *ident−list* .ext *ident−list* .formal *expr* , *expr* .function .loc *expr* .nargs .rel *expr* .ptr *expr* .stack *expr* .str string−constant .dc1 *expr* .dc2 *expr* , *expr* .dc3 *expr* , *expr* , *expr* .dc4 *expr* , *expr* , *expr* , *expr* opcode *argument−list* null
ident−list	::=	identifier *ident−list* , identifier
argument−list	::=	*operand* *argument−list* , *operand*
operand	::=	*expr* *expr* [*expr*]

special	::=	++
		−−
		@
		!
		#
expr	::=	(*expr*)
		expr & *expr*
		expr + *expr*
		expr − *expr*
		expr | *expr*
		expr << *expr*
		expr >> *expr*
		expr * *expr*
		expr / *expr*
		expr % *expr*
		expr ↑ *expr*
		− *expr*
		special expr
		expr special
		identifier
		constant

Appendix 2. **Directives and Pseudo Operations**

The following directives are used in all implementations

.align : starts the next assembled cell at an address which is a multiple of the expression

.dc1, **.dc2**, **.dc3**, **.dc4** : define a constant word having 1, 2, 3 or 4 subfields, respectively

.ds : reserves words of storage specified by expression

.ent : specifies external names defined in this module

.ext : specifies external names defined in other modules which may be referenced in this module

.formal : specifies the number of arguments expected by the function

.function : specifies that a value is returned by the function

.loc : the location counter corresponding to the attribute of the expression is set to have the value of the expression, and is used for the following code

.nargs : specifies that a function needs to know the number of actual arguments passed to it

.rel : specifies which location counter to use for the following code

.ptr : defines a word containing a word pointer

.stack : specifies the number of words of stack used by a function

.str : defines a string constant

Appendix 3. **An Example Session with Helper**

I am here to help get your new assembler started.

What machine are you writing an assembler for? *INTERDATA 70*
What is the file name for this assembler? */interdata*

Where is the file containing your list of opcodes? *tla/interdata/mnemonics*

Now tell me about the INTERDATA 70.
How many bits in a byte? *8*
How many bits in the smallest addressable cell? *8*
How many bits in the largest machine address? *16*
How many bits in the smallest instruction? *16*
How many bytes in a no−op instruction? *2*
What is the numeric representation of a no−op instruction? *$0800*

Now tell me about how Thoth is implemented on the INTERDATA 70.
How many bits in an word? *16*

Will you need a .dc1? *yes*
What is the maximum value allowed? *32767*
What is the minimum value allowed? *−32768*
Is the operand relocatable? *yes*
Which relocation descriptor? *0*

Will you need a .dc2? *no*

Will you need a .dc3? *no*

Will you need a .dc4? *no*

The INTERDATA 70 assembler has had four files generated for it:
/tla/ph1/interdata/manifests
/tla/ph1/interdata/functions
/tla/ph0/interdata/externals
/tla/ph1/interdata/externals

Best of luck with the rest of the assembler!