An Interactive Assembly Level Debugging System

S. PANCHAPAESAN, S. SUBRAMANIAN AND H. VENKATESWARAN
National Aeronautical Laboratory, Bangalore-560017, India

SUMMARY
An interactive assembly level debugging system has been developed to facilitate program development on an INTEL 8080A/8085 based microcomputer. It has features such as decoding machine level instructions into the assembly language, relocating programs in memory, changing instructions interactively at assembly level etc. This paper deals with the design of the assembly level debugging system and the various facilities and features it provides. The debugging system requires only 4-k bytes of RAM besides the memory requirements of the application program that has to be debugged.

KEY WORDS Interactive assembler Assembly debugging system Microcomputers

INTRODUCTION
An interactive assembly level debugging system has been developed for an INTEL 8080A/8085 based microcomputer. The design of the debugging system and its features are presented. A program developed in an assembly language is first assembled and loaded into the microcomputer memory for subsequent execution. When the assembled program does not execute properly or as desired by the programmer, it becomes necessary to have a tool to facilitate debugging. Usually, the tools for debugging need knowledge of machine code, as all debugging monitors provided by manufacturers or developed individually work at machine code level with short mnemonic commands. These monitors decode the instructions/data into hex or octal code for display. So it is mandatory to know the hex or octal codes in order to carry out debugging on a machine code program. Though these debug monitors provide very many features to facilitate debugging of programs, they on the other hand expect a good deal of machine code knowledge on the part of the user to interpret correctly the codes and data.

The interactive assembly level debugging system provides facilities for debugging programs at the assembly level itself. One such facility is to reconvert portions of the assembled program to the assembly language to know what exactly are the instructions in memory and to pin-point the bugs. The other facilities are modifying instructions/portions of a program, moving program units in the memory of the computer to test such units and to do patching of these units after the successful completion of tests, and to do relocation of programs without invoking re-assembly.

THE DEBUGGING SYSTEM
The debugging system has been designed to convert machine code programs of INTEL 8080/8085 based microcomputers into assembly language instructions. The

© 1985 by John Wiley & Sons, Ltd. Revised 13 September 1982

Received 14 March 1980
assembly language mnemonics used for reconversion of machine code programs are as described in Reference 1. The characteristic byte table (CBT) is extensively used to achieve conversion back to assembly language. The debugging system requires 4.5 K bytes of RAM besides the memory requirements of the user programs.

After assembly and loading of the application program into the memory of the microcomputer, the debugging system can be invoked. A prompt character (?) appears on the TTY or a display terminal, prompting the user to enter the debug commands. The mnemonic commands provide many useful features such as copying, converting back to assembly mnemonics and relocating programs all at assembly level. Thus the mnemonic commands help development of several program units and integration of these into a single large program. The program units can be developed and/or tested using the debug facilities. Large and complex programs can be split into small program units and can be tried for various types of inputs and conditions. Also the grafting (patching) of working program units into the main-line program is easily done with the powerful commands of the debugging system.

DESIGN OF THE INTERACTIVE ASSEMBLY LEVEL DEBUGGER

When an instruction in memory has to be converted back into the assembly language, the debug system looks at the opcode and identifies the type of instruction from its characteristic byte. The characteristic byte consists of information bits such as the length of the instruction, the number of opcodes involved and the position bits of the operands in the instruction word. The use and description of the characteristics byte can be found in Reference 1.

Once the type of instruction to be assembled into mnemonic form is identified, the debug system follows strictly the assembly language syntax rules which are embedded in it. Then the characters required for mnemonic opcodes, the operands, the directives, separators and other delimiters completing the syntax structure of the language are generated.

SEARCH FOR THE OP-CODE

The opcode table, consisting of the mnemonics, machine codes and their corresponding characteristic bytes, is arranged in a specific order to facilitate scanning. A machine code to be assembled into mnemonic form is taken and compared with the entries in the opcode table, and if a match is found, the debug system uses the CBT information to assemble back the instruction in its mnemonic form. This matching can be done in a single sweep for certain instructions such as RAL, BAR, PPP, FSS, BSA, etc. and the generation of the mnemonics can be carried out immediately. The instructions in this group contain only the operation code bits, and no operand bits are embedded in the instruction word. A list of such simple instructions that can be identified in a single sweep over the opcode table is provided in Appendix I.

There are other instructions which contain operand information, such as the register number or condition flags. These instructions will not obtain a match as the opcode table entries do not contain this information bits.

Hence it becomes necessary to assume certain bit patterns and remove the appropriate information bits in the instruction word before scanning the entries in the opcode table for a match.

RELOCATION OF PROGRAMS

When programs are copied in memory they have to be relocated. Apart from relocating the jump and call addresses, it may be necessary to relocate some addresses used in instructions such as LRP H, A.....LSP SP, A......also. These instructions may have been used to define data areas for the program.

Hence two types of commands are provided for relocation. The CPYRLC command followed by a subsequent command PART copies and relocates program units, effecting relocation only for branch instructions such as BRA, BCT, BCF and for call instructions such as CAL, CFI, and CCF. In the case when data-addresses have to be replaced, the command CPYRLC followed by FULL must be used.

However, the FULL command should be exercised with caution, as it would inadvertently change the codes in the Load Register Pair instructions which are intended to represent immediate data and not pointers to data area.

DEBUG SYSTEM COMMANDS

The commands use six character names to provide the following functions:

1. copying program units in memory and relocating them, partly or fully (CPYRLC PART/FULL)
2. decoding and converting the machine instructions at the specified address in memory to assembly language (MN6ASM MNEXIT)
3. assembling an assembly language program from a specified address in memory (ASSEMBLE)
4. producing listings of programs in the assembly language for documentation (GODOCM)
5. displaying the contents of all registers (DSPALL)
6. executing instructions in the memory (EXECUT).

The exact commands and examples of these functions are listed in Appendix III.
CONCLUSION

The debug system is useful for developing debugging programs at the assembly language level. The debug system has been working quite satisfactorily for the past two years on an INTEL 8080A based system located in our Laboratory.

ACKNOWLEDGEMENTS

The authors wish to thank the Director, NAL for his kind permission to publish this paper. The authors wish to thank the referee for his valuable comments and criticism in the light of which this paper has been revised.

APPENDIX I

Opcodes mnemonics that can match directly on the first sweep with the entries in the opcode table are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Mask</th>
<th>Opcodes that get a match</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>370</td>
<td>ADD, ADC, SUB, SBB, AND, EXR, INR, CMP</td>
</tr>
<tr>
<td>2</td>
<td>317</td>
<td>BCT, COT, RCT</td>
</tr>
<tr>
<td>3</td>
<td>307</td>
<td>RST, PPR, PPR, BCF, CCF, RCF</td>
</tr>
<tr>
<td>4</td>
<td>300</td>
<td>TRN</td>
</tr>
<tr>
<td>5</td>
<td>017</td>
<td>ORP, ARP, LAI</td>
</tr>
<tr>
<td>6</td>
<td>007</td>
<td>LDA, INC, DEC, LRP, IRP, SAI</td>
</tr>
</tbody>
</table>

APPENDIX II

Second level commands in monitor

1. CPYRLC XXXX YYYY
   FULL
   Copy the program from location XXXX (hex) to the location YYYY (hex) and relocate fully all the addresses of jump and call instructions as well as LRP instructions.
the result is
0200  LRP  H0400
?  MNENXT
0203  CAL  H0252
?  MNENXT
0206  HLT
?
5. ASMBLE XXXX
Assemble a program to be input from memory location XXXX (hex) till a directive
FIN is encountered.

Example
? ASMBLE 0200
0200  LRP  H.A1024
0203  CAL  A0594
0206  HLT
0207  FIN

Here the addresses on the left are printed by the assembler part of the monitor.
Instructions are given by the user through keyboard, which are assembled till the FIN
directive.

6. GODOCM XXXX YYYY
Print out the program residing in the memory from location XXXX to location YYYY
in assembly language.

Example
? GODOCM 0200 0206
0200  010004  LRP  B.H0400
0203  CD 15 02  CAL  H0215
0206  76  HLT

The output consists of the memory address, the machine code and the assembly
language statement. This will be useful for documenting. Note that labels and
labelled names cannot appear in this listing but only absolute addresses.

7. DSPALL
Display the contents of all the register.

Example
? DSPALL
A = 15  B = 20  C = 25  D = 30
E = 35  F = 42  H = 40  L = 60
?

8. EXECUT
Execute the program from its starting location defined in the start of the
debugging/developing session.

REFERENCE
1. S. Panchapakesan, H. Venkateswaran and S. Subramanian, 'Assemblers for micro-computers',

An IKBS Implementation

PHILIP LEITH
Faculty of Mathematics, Open University, Milton Keynes MK7 6AA, U.K.

SUMMARY
This paper details the practical approaches and solutions taken in the implementation of an
INTERLISP programmed knowledge based program. Although some of the solutions used
are particular to the specific problem domain of this program, the general pragmatic
methods are of wider interest. The program excerpts are given in Pseudo-LISP code.

KEY WORDS: IKBS  INTERLISP

INTRODUCTION
Much of the currently available literature on IKBS techniques describes systems in
relatively high level terms. This situation is unfortunate in many ways, but most
particularly because there is a very necessary requirement for basic introductory
writings on the tools and techniques rather than the goals and general methodologies
towards which IKBS systems designers are working. The techniques and tools are, in
fact, just those which are used in other areas of computing—such a fact is often
obscured by high level system descriptions. In this paper I describe the actual
implementation of a system which has been discussed previously at a higher level.
Although this system is not representative of all currently available systems, it does
accord with the generic 'expert system' type.

There are basically three problems in the design of an IKBS system. The first, and
most difficult, is to choose the data structures which will represent the area of
expertice handled by the system. As with all computer programs it is necessary to
choose one of several possible structures. Each of the possible data structures might be
advantageous in solving one particular problem, but not another. So it is with IKBS
programs—one might use production rules, another frames and a third, associative
networks. Just as likely as a system which uses only one data structure to handle its
expertise is a system which uses more than one (EMYCIN uses a tree structure—
called a context tree—as its prime data structure; it is built and transformed by
production rules held in a different structure).

The second area is that of interpretation—how can the information which is held in
these data structures be extracted from it.

The third area is that of filling up the data structures with the information which,
when interpreted, is to become the system's expertise.

The ELI system was designed to handle one type of information—legislation
covering British Welfare Rights. The legislation is transformed by an expert within
the field into rules (sometimes called production rules or productions). The general
approach is outlined in Reference 2. In this text, I deal with the handling of rules

0038-0644/85/010065-22$02.20  Received 7 December 1983
© 1985 by John Wiley & Sons, Ltd.
Revised 17 May 1984