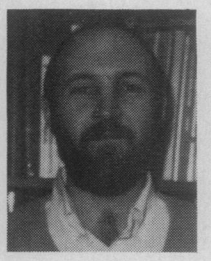
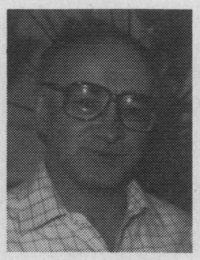
[6] *SDK-85 System Design Kit User's Manual*, Intel, No. 9080451B, 1978.
[7] *Operational Catalog*, Worcester Polytechnic Institute, Worcester, MA, 1981–82.
[8] K. L. Short, *Microprocessors and Programmed Logic.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[9] C. Guild, A. Rosenflack, and R. G. Willison, "Report on the commission on EMG instrumentation," *EEG Clin. Neurophysiol.*, vol. 28, pp. 339–413, 1970.



**Frederick J. Looft** (S'74–M'78) was born in Cleveland, OH, in 1951. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Michigan, Ann Arbor, in 1973, 1974, and 1979, respectively. He also received the M.S. in computer, information, and control engineering from the University of Michigan in 1976.

From 1979 to 1980 he was employed by Bell Laboratories, Inc., North Andover, MA. Since the fall of 1980 he has been Assistant Professor of Electrical Engineering at Worcester Polytechnic Institute, Worcester, MA. His teaching encompasses primarily digital and computer area courses, although he also advises directed research projects and M.S. theses in various areas of signal processing and biomedical engineering. His research interests are in sensory coding and biomedical signal processing.



**Wilhelm H. Eggimann** (S'59–M'62) received the Diploma in electrical engineering from the Swiss Federal Institute of Technology, Zurich, Switzerland, and the M.S. and Ph.D. degrees from Case Institute of Technology, Cleveland, OH.

He is presently on the faculty of the Department of Electrical Engineering, Worcester Polytechnic Institute, Worcester, MA. His interests are in electromagnetic theory, solid state electronics, and computer engineering.

# Teaching Assembly Language Programming with ZIP, a Z80 Assembly Language Interpreter Program

TREVOR N. MUDGE, MEMBER, IEEE, AND GREGORY D. BUZZARD, STUDENT MEMBER, IEEE

***Abstract*—This paper outlines the background of teaching needs that led to the development of an assembly language interpreter for teaching assembly language. The interpreter, called ZIP for Z80 interpreter program, is small enough to be used by a student on a modest sized Z80-based system. It greatly facilitates understanding the Z80 assembly language by allowing the student to learn it interactively. It can also be used as a debugging tool. An overview of the interpreter is given and its syntax and operation are discussed.**

## I. Introduction

WORLD-WIDE microprocessor (MPU chips and I/O chips) sales have risen from $52 million in 1975 to over $1 billion in 1981. Despite the recession-induced slowdown in 1981, strong growth should continue throughout the 1980's, with sales expected to reach $4.4 billion by 1986 (see Fig. 1). This represents a compound annual growth rate of 34 percent. Continued increases in sales are virtually assured as microprocessor usage expands in such diverse areas as household appliances, automobiles, military equipment, process control, robotics, communications, and medicine. The phenomenal growth of the industry has fostered a corresponding increase in the popularity of microprocessor-based digital system design courses.

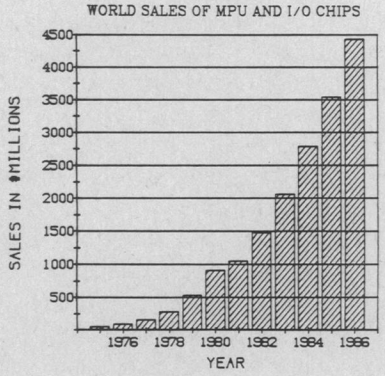



Fig. 1.

In keeping with this trend, enrollment in the introductory microprocessor course offered by the Department of Electrical and Computer Engineering (ECE) at the University of Michigan has increased dramatically. This increase has placed a considerable strain on the resources of the ECE Department resulting in, among other things, extremely large classes. In an attempt to offset the adverse effect of the class size and to increase the efficiency of the learning process, teaching aids are being developed. Principal among these is ZIP [1], [2]. ZIP is a teaching/debugging program designed to host the execution of

Manuscript received August 5, 1982; revised February 25, 1983. This work was supported in part by a grant from the Center for Research on Learning and Teaching, University of Michigan.

The authors are with the Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109.

0018-9359/83/0800-0091$01.00 © 1983 IEEE

"target" programs on Z80 microprocessor-based systems. This is achieved by interpreting the core image of machine level instructions and data in order to interactively emulate and display (in assembly language mnemonics) selected portions of the target program.

This paper describes the course sequence offered in microprocessor-based digital system design, outlines the teaching of the introductory course, and discusses those difficulties encountered by students which led to the development of ZIP. Design objectives and the implementation of ZIP are discussed, and a description of the program is given. Finally, the integration of ZIP into the classroom is described and observations on its use provided.

### A. Background

A sequence of three courses designed to educate students in the use of microprocessor-based digital system design is offered by the ECE Department [3]: ECE 365–Digital Computer Engineering, ECE 366–Digital Computer Engineering Laboratory, and ECE 466–Digital Design Laboratory. The first course in the sequence is intended to satisfy the educational needs of undergraduates whose main focus of interest is not digital system design. The complete sequence is intended to provide a firm background in microprocessor-based digital system design for students who plan to continue in our graduate program and whose graduate work would involve them in experimental computer science. The three courses are part of a larger commitment to revitalize experimental computer science in the ECE Department at the University of Michigan.

ECE 365 is, by far, the most popular course in the microprocessor sequence with yearly enrollment exceeding 500 students. Primarily intended for junior/senior level students in the ECE Department, this course serves a dual purpose. It must be self-contained for those students whose main interest is not microprocessor systems, while, at the same time, it must build a solid foundation for those who wish to continue in the sequence. The former purpose constrains us to limit the number of prerequisites required. As a result, ECE 365 is open to all students who have successfully completed only a basic digital computing course (i.e., one which covers an introduction to digital computer systems, flow diagrams, algorithms, and digital computer programming). This minimal prerequisite requirement results in students with very diverse backgrounds being enrolled in the same class. This diversity is most apparent in assembly language skills.

The teaching of ECE 365 employs three instructional formats: lecture, laboratory, and recitation. The lecture section meets for three 1 h sessions each week. Course material centers on digital computer organization, assembly language, and logic design; particular reference is made to microprocessor-based systems. ECE 365 differs from a more traditional course on digital system design in two important ways.

First, emphasis is placed on using microprocessors and other VLSI components such as PLA's, ROM's, RAM's, PIO's, SIO's, and timers as system components. Gate-level logic design is discussed in much less depth than is usual in more traditional courses on digital system design. Topics such as minimization of combinational logic and state reduction in sequential machines are only briefly mentioned, whereas design techniques that use PLA's and ROM's are emphasized. Finding a textbook to reflect the thrust of the course presents a problem. We are currently using *Computer System Architecture* by M. M. Mano [4], together with the Mostek *1982/83 Z80 Designers Guide* [5]. This material is supplemented with excerpts from the system description manuals of the RacPac[1] development system used in the class and from various other pertinent notes and articles.

Second, in addition to assembly language, the use of the RacPac operating system, the editor, loader, and other utility programs used in system development on the RacPacs is also taught. This represents a considerable increase in the amount of software taught compared to that presented in a traditional treatment of digital system design. Our experience had shown the time consumed in teaching this extra software presented an obstacle to covering the range of material that we felt was a minimum for such a course. Since assembly language appeared to give students the most difficulty, we decided to concentrate our efforts on developing methods to reduce this burden in the lecture section.

The laboratory sections of the course, which meet for 3 h each week, explore seven carefully chosen experiments. Four of these experiments involve the use of the RacPac microcomputer systems. The following two are digital logic design experiments which use attaché case "logic lab" kits containing a 5 V power supply, push button and toggle switches, breadboard IC sockets, wire, wire strippers, pliers, logic probes, and a selection of TTL logic. The final experiment requires students to interface the logic lab kits to the RacPacs through the parallel I/O ports. Circuit debugging for the logic lab experiments is facilitated by the logic probes and Hewlett-Packard logic analyzers–models HP-1602A and HP-1615A. Previously, program debugging was available through the monitor which permitted only the setting of breakpoints and the display and modification of memory and registers. The addition of ZIP has provided much more sophisticated program debugging capabilities.

The RacPac systems on which ZIP presently runs are comprised of a CRT, a keyboard, dual single density Shugart 8-in disk drives, an RS-232 interface, and connections for a printer and an EPROM burner. The logic is built on two boards designated 3744 and 3745. The 3744 (CPU) board contains the Z80 CPU, 32K × 1 bytes of dynamic RAM, 4K × 1 bytes of EPROM (containing the bootstrap loader), two parallel input/output chips, two serial input/output chips, a counter/timer circuit chip, a CRT controller chip, and a 2K × 1 byte static RAM for the CRT screen. The 3745 (expansion) board contains 64K × 1 bytes of dynamic RAM, a floppy disk controller chip, EPROM burner control logic, and parallel printer control logic.

The recitation section meets for 1 h each week. The main

[1] RacPac is the trademark for a Z80-based microprocessor development system manufactured by the Xycom Corporation.

focus is on discussing techniques for applying lecture material to the laboratory experiments. The idea is to provide ample information to aid in the successful completion of the laboratory experiments, without actually providing pseudocode outlines and/or actual assembly code source segments. Consequently, much of the time is spent covering the topics of code structuring and algorithm design. Emphasis is also placed on the efficient use of design and development tools. Accordingly, the majority of the information on ZIP is disseminated in the recitation section.

The large class sizes and the wide diversity in assembly language programming skills, coupled with the obstacle that assembly language presents to the coverage of sufficient course material, created the need for teaching aids to help students learn assembly language more rapidly. ZIP represents the principal effort in the development of such aids. Students receive a copy of this interpreter during the first laboratory period for use as a self-teaching aid and a debugging tool.

### B. Design Objectives

ZIP was designed to be both a self-paced educational tool and a powerful debugger. As an educational tool, ZIP had to be useful to both novices and moderately experienced users. To meet these goals ZIP has been designed to be easily invoked and to require knowledge of only a minimal number of commands for its basic use. As more experience is gained, the more powerful commands can be investigated. Another very important requirement for ZIP was that it had to illustrate to the student the relationship between the static representation of a program and its data (i.e., its image as a bit pattern in memory) and the dynamic aspect of that program during its execution. This relationship is fostered by providing a split screen format which allows the simultaneous viewing of the current (disassembled) section of program code together with the contents of the CPU registers, the top four stack entries, and a 32-byte portion of memory. This enables students to observe the current state of the machine as they step through the execution of the target program.

The apparent complexity of assembly language programs for the novice is often exacerbated by the limited amount of software support available on most microprocessor-based systems. This creates a frequent need for interactive debugging. To aid in this process, ZIP was provided with several debugging oriented commands. Among the more powerful of these are the "SET" and "trigger condition" commands. The SET command allows any byte in memory, or CPU register (except F—the condition code register) to be set to a specified value. Through the use of the SET command, errors can be corrected "on the fly," thus allowing many bugs to be located without frequent editing and reassembling of the source code. The "trigger condition" construct (a concept borrowed from logic analyzers) provides the ability to execute the target program until a specified (trigger) condition is met. The conditions are specific values of, or relations between, memory locations, registers, or the condition code flags. A natural extension to this construct is to allow the ANDing and ORing of trigger conditions; however, this is not currently implemented.

## II. Description of ZIP

### A. Program Overview

ZIP disassembles and interprets segments of memory containing Z80 machine code and/or data. The disassembled code is displayed on the left side of the CRT screen and the CPU registers, top four words of the stack, and 32 contiguous bytes of memory are displayed on the right side. This configuration provides a visual relationship between the assembly language source code and the state of the CPU registers, stack, and memory locations as the program is being interpreted. Commands can be entered to control the interpretation of the program, modify register or memory values, and control the display of information.

In order to display the disassembled program in an intelligible format, the executable code must be distinguished from data and unused memory. ZIP accomplishes this by searching, starting at a specified address, the target program core image for jump, call, and return statements. The location of these statements, along with their targets when necessary, is used to develop a table of origin and end point (ORG-END) pairs. This table of ORG-END pairs is then used to determine which segments of memory will be disassembled and displayed as source code and which will be displayed as data.

Upon completion of the ORG-END table the user is prompted for commands. After the execution of any command which affects the target PC (program counter), the appropriate segment of memory is disassembled and displayed. This can often lead to the disassembly of code that has already been disassembled; however, within the constraints mentioned later, this dynamic disassembly allows the effects of self-modifying code to be observed. The right side of the display is updated following the execution of commands which change the state of any of the CPU registers, displayed memory locations, or the stack.

### B. Screen Layout

The layout of the screen is shown in Fig. 2. The column on the left shows memory locations in hex (4 hex digits). Alongside these are 1 to 4 byte instruction codes also in hex (Z80 instructions can be from 1 to 4 bytes in length). Further to the right, the instruction codes are shown in their disassembled form. For example, consider the line enclosed by the shaded rectangle in the left center. At the left is a memory location ($90EE_{16}$). This location and the subsequent one contain the bytes $10_{16}$ and $F7_{16}$, respectively (the Z80 has byte oriented addressing). These disassemble to the Z80 instruction "DJNZ 90E7"—Decrement register B and Jump if B is NonZero to location $90E7_{16}$. Notice that addresses of operands or targets of jumps are not disassembled but are left as absolute addresses. Disassembling these would require access to the symbol table created when the program was originally assembled. In order to keep the first version of ZIP simple, the ability to recover symbolic addresses was omitted.

As noted above, ZIP automatically determines data areas in memory by examining jumps, subroutine calls, and returns. Memory locations that contain data rather than instructions

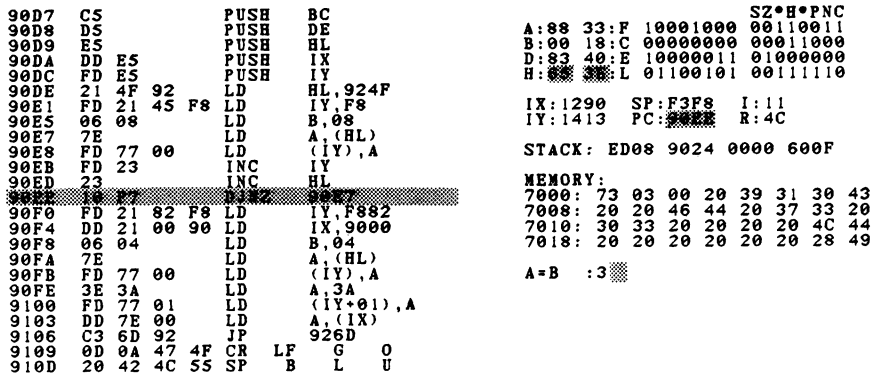



Fig. 2. Screen layout.

have their contents displayed as 1 to 4 pairs of hex digits in the same column as the instruction code mnemonics. Further to the right, in the same column as the disassembled instructions, the contents of the memory is repeated in its ASCII character form.

The right-hand side of the screen displays the contents of the Z80's CPU registers, the top four items on the stack, 32 bytes of memory, and the command line.

There are eight 1-byte CPU registers: A, F, B, C, D, E, H, L. These are displayed at the top right of the screen. For example, the second row at the top right shows the contents of A, the accumulator, in hex (88), the contents of F, the flag register, in hex (33), followed by the contents of A in binary (10001000), and the contents of F in binary (00110011). The binary display is useful for checking bit operations, shifts, and rotations. The F or flag register is not a general purpose register; instead, it holds six 1-bit flags which indicate the condition codes. Their position is shown in the binary display of F by the header "SZ*H*PNC" at the extreme top right. The conditions that the flags indicate are as follows: S = 1 if accumulator contains negative value; Z = 1 if accumulator contains zero value; H = 1 if a half-carry/borrow (between bits 3 and 4) occurred; P = 1 if number of ones in accumulator is even (parity) or, depending on instruction type, if overflow occurred; N = 1 if instruction is of the subtract type and if it affects the carry (C) flag; C = 1 if a carry/borrow occurred (see [5] for more details). Immediately below the 1-byte CPU registers are the 16-bit CPU registers IX, IY, SP, PC. IX and IY are index registers, SP is the stack pointer (points to the top of the stack), and PC is the program counter. The register pairs BC, DE, and HL can also be regarded as 16-bit registers, and the format of the display has been set up to allow this view. To the right of the 16-bit registers appear the two special 1-byte registers I and R. Below the 16-bit registers appear the top four stack items. These items are 1 word or 2 bytes each; thus, in Fig. 2, for example, the top of the stack is at location $F3F8_{16}$ (see contents of SP) and the top item is the 16-bit quantity $ED08_{16}$. The bytes within each word of the top four stack items are shown in the reverse order of which they appear in memory; left-to-right within each 16-bit word corresponds to high-to-low memory addresses. The stack grows toward low memory. The orientation of the bytes displayed in each word is consistent with the orientation of the bytes displayed in the 16-bit registers and the register pairs BC, DE, and HL. In all cases, 16-bit words are stored with their most significant byte at the higher memory location.

Below the stack display, a user-selected 32-byte area of memory is displayed in hex. Further below, the command currently being entered by the user is displayed.

The shaded rectangles drawn around characters in Fig. 2 indicate that those characters appear in reverse video on the screen. In the case of instructions, reverse video indicates the instruction to be executed next. In Fig. 2 this is the DJNZ mentioned above. In addition, the contents of the PC and the H and L registers are shown in reverse video, indicating that the most recently executed instruction—"INC HL"—caused their contents to be altered (PC was changed by the normal course of incrementing the program counter to fetch the next instruction). If any of the memory locations already displayed on the screen had been altered they would also be shown in reverse video.

The command line is shown with a reverse video square beside it to mark the cursor position. The particular command shown in Fig. 2 reads: beginning with the current instruction (the DJNZ) execute the program until the contents of registers A and B have been equal three times. The command is terminated with a carriage return ($c_r$); the return initiates ZIP's interpretation of the command line. The left side of the display scrolls so that the next instruction to be interpreted, i.e., the instruction displayed in reverse video, is always kept in the middle of the screen.

### C. Command Descriptions

Fig. 3 shows the syntax of ZIP's command structure in standard BNF notation. Nonterminal symbols are shown bracketed by "<" and ">." Rewriting rules are identified by "::="; the left-hand side can be rewritten as one of the alternatives on the right-hand side. The alternatives are separated by "|." The remaining symbols are the terminal symbols that appear in the commands. Spaces can be used freely to aid readability. The syntax is intentionally simple and, in fact, can be parsed by a finite state machine. Simplicity was a prerequisite for two reasons. First, size was an issue; we wanted to be able to use it in small systems. Second, Version 1 of the program was originally written by two students with limited programming experience.

When used in conjunction with the RacPac systems ZIP is loaded and run from a floppy disk operating system. Execu-

```
<command>::= GO<hex><hex><hex><hex>|<set>|<display>|
        <trigger_condition>|ON|OF|AU|UA|OL|NE|QU
<set>::= SE<reg>=<hex><hex>|SE<double_reg>=<hex><hex><hex><hex>|
        SE<memory>=<hex><hex>
<display>::= DI<memory>
<trigger_condition>::= <condition><taill>|<tail>
<condition>::= <reg><relation><rhs_reg>|
        <double_reg><relation><rhs_double>|
        <memory><relation><rhs_memory>|
        F=<bit><bit><bit><bit><bit><bit>
<reg>::= A|B|C|D|E|H|L
<relation>::= =|~=|~<|~>|<|>|
<rhs_reg>::= <reg>|<hex><hex>|<memory>
<double_reg>::= BC|DE|HL|SP|PC|IX|IY
<rhs_double>::= <double_reg>|<hex><hex><hex><hex>|<memory>
<rhs_memory>::= <hex><hex>|<memory>
<memory>::= @<hex><hex><hex><hex>
<hex>::= 0|1|2|3|4|...|F
<taill>::= cr|:<number>cr
<tail>::= <number>cr|cr
<number>::= <hex>|<hex><hex>|<hex><hex><hex>|<hex><hex><hex><hex>
<bit>::= 0|1|X
```

Fig. 3. ZIP's syntax.

tion begins by querying the user with

"Start Address?"

The user should respond with a 16-bit hex address indicating where in memory the interpretation should begin. The Z80 allows computed jumps: JP(HL), JP(IX), and JP(IY); see [5]. The current version of ZIP cannot handle these. If any of these are present in the region of memory to be interpreted, a further message is displayed as follows

"You are using Computed Jumps!"

If this is the case, the interpreter may try to disassemble areas in memory that are data areas. This is not critical, so the user is allowed to proceed if he/she wishes to (see later). At this point, ZIP is ready to receive any of the commands of Fig. 3.

The first production in Fig. 3 gives the major command classes. The first of these transfers interpretation to a specific location. For example,

GO 10B4 $c_r$

transfers the flow of interpretation to location $10B4_{16}$. The next alternative allows the modification of registers or memory locations. For example,

SE IX = 99AE $c_r$

sets the contents of the IX register to $99AE_{16}$. (See the second alternative of the second production: "<set>::=SE<double_reg>=<hex><hex><hex><hex>.") The third alternative of the first production (<display>) specifies which 32 bytes of memory appear on the screen display. For example,

DI 4F68 $c_r$

displays the 32 bytes beginning with the byte at location $4F68_{16}$. The fourth alternative to the first production identifies the most important class of commands. They define the trigger conditions. For example,

B = @A77D $c_r$

causes interpretation until the contents of register B equals the contents of location $A77D_{16}$ in memory. (Following the rewriting rules gives <trigger_condition>::=<condition><taill>::=<reg><relation><rhs_reg><taill>::= B = <memory><taill>::= B = @<hex><hex><hex><hex>$c_r$.) Or, for example,

F = OXX1X1 $c_r$

causes interpretation until the flags S = 0, P = 1, and C = 1 (the other three can be either 0 or 1). The simpler forms of this class of commands specify interpretation of the next fixed number of instructions. For example,

10 $c_r$

causes the next 10 instructions to be interpreted. The simplest form is just a carriage return–interpret the next instruction. (See the second alternative of the fourth production "<trigger_condition>::=<tail>" for more details.) The command ON turns on a large display of memory locations that temporarily fill the screen with a memory map beginning with the 32 bytes of memory usually displayed in the normal screen format. The command OF turns it off, reverting back to the normal format. The command AU causes the auxiliary register set to be displayed. The command UA causes the normal register set to be displayed. The command OL causes the old values of the registers–as they were when the last trigger condition was satisfied–to be displayed. The command NE causes the new (current) values of the registers to be displayed. Finally, the command QU quits the interpreter and returns control to the operating system.

### *D. Self-Modifying Code*

Disassembly of the displayable portions of the core image occurs during the execution of every command which affects the PC of the target program. This enables the students to observe the dynamic effects of self-modifying programs, as any modifications to the instruction code will be disassembled and displayed.[2] However, the target program is tracked only once, so the ORG-END table is created prior to program execution. Hence, if the target program dynamically modifies a memory area which was executable code into a data field, or vice versa, the display on the left side of the screen will sometimes become meaningless. An occurrence of this event, however, will not inhibit the correct execution of the program. Self-modifying code which does not change executable code into data, or vice versa, is interpreted without any adverse affects.

### *E. Interrupts*

Aside from the fact that interactive interpretation of computer programs distorts the real-time base in which many interrupts normally occur, interrupts pose another problem. All three interrupt modes available on the Z80 can affect the PC, and unless the target address of the altered PC is known in advance, retention of program control by the interpreter cannot be assured. This problem is most acute for interrupt modes 0 and 1.

Mode 0 interrupts allow the interrupting device to supply

[2] Although the undesirable effects of self-modifying code are made clear to the students, we feel that it is important to illustrate this feature of the von-Neumann computing model.

any one instruction to the CPU to be executed next. Provided that the instruction does not change the PC, interpretation of the instruction will appear transparent; but any changes in the CPU registers, top four stack words, or displayed memory locations will be shown. However, if the peripheral device supplies an instruction which changes the PC, it becomes the programmer's responsibility to assure that a proper return is made to the interrupted instruction in order to successfully resume interpretation.

Mode 1 interrupts cause an "RST 38H" (restart at location $38_{16}$) instruction to be executed. Again, control of the interpretation is lost, and it is the programmer's responsibility to assure its proper return. Any resulting changes in the items displayed on the screen will be shown as noted above.

Mode 2 is the vectored interrupt mode. The current version of ZIP will not interactively trace the execution of mode 2 interrupt service routines during the interpretation of the target program. The final CPU register status, top four stack values, and the displayed memory values will be shown, but the execution of the interrupt service routine will appear transparent to the interpreter.

For any of the interrupt modes, however, service routines can be interpreted (and their integrity verified) prior to execution of the main program. ZIP can be coerced to request the user to make entries into the ORG-END table. Next, ORG-END pairs which encompass only the interrupt service routines can be entered using the GO command, thus allowing the routine to be interpreted as a separate entity.

### *F. Implementation*

ZIP, which is written in Z80 assembly language, is implemented in five major modules: *exec*, *track*, *screen*, *disass*, and *simul*. The *exec* module contains the executive code which directs the overall program flow of ZIP. It also contains the command line interpreter, which comprises the majority of the code in this module, and error message routines. *Exec* performs an initialization sequence, queries the user for program start location information, calls the *track* routine, then enters the command line interpreter. The command line interpreter and its associated routines are then responsible for sequencing the rest of ZIP's activities. *Track* (which was described in the overview) creates the table of origins and endpoints which is used by the screen routine when deciding whether to disassemble memory contents into assembly language mnemonics or simply display the hexadecimal memory values and their ASCII equivalents. As previously mentioned, this module is executed only once. The *screen* module, whose routines are called almost exclusively by *exec*, is responsible for creating and maintaining ZIP's display formats. Hence, it also determines the memory locations that need to be disassembled for the display, and determines which of these disassembled lines is to be shown in reverse video. After determining which instructions to display, the *screen* routine invokes the disassembler. The *screen* module also contains all of ZIP's I/O routines. *Disass* determines the ASCII encoded mnemonic representation of the op-codes which are presented to it. These ASCII representations are then written into a buffer which is directly accessible by the *screen* module. Unimplemented op-codes are displayed as a string of asterisks.

*Simul* is responsible for the instruction-by-instruction simulation of target programs. This task is greatly simplified since the host machine is also the target machine. Simulation is then basically reduced to copying the target instruction into a work area within the *simul* module, swapping the values of all of the target machine's registers (with the exception of the refresh register—the effect of which is not faithfully simulated) into the actual register set, and executing the target instruction. Immediately after execution the values of the target machine's registers are resaved. Proper care has to be taken to ensure that control of the simulation is not lost by the execution of target instructions which may change the PC. This is done by identifying such instructions and temporarily modifying their destination operands. In the case of instructions which employ relative addressing, extra computation is required to determine the actual target destination addresses for any branches which are taken. Since the keyboard input on the RacPac systems is taken in a polled mode, and since no disk I/O is performed by ZIP, the interpreter is run with interrupts disabled. Of course, the execution of target program instructions may occur with interrupts enabled, depending on that state of the target machine's IFF (interrupt flip-flop) register (see [5] for more details).

The execution of ZIP is totally independent of any operating system. No disk I/O is performed by ZIP, and all of the necessary routines for reading keyboard input and writing to the screen reside within the screen module. The interface to the keyboard uses two parallel ports, one contains status information, the other latches the actual ASCII keyboard input. The video display is controlled by a memory mapped video RAM which can be bank selected into the upper 2K of memory address space. Routines which manipulate the video controller's control registers and the bank select register also reside within the screen module. Independence from any operating system makes ZIP easily transportable to other Z80-based systems which have similar hardware configurations. Details of key addresses for the keyboard inputs and the screen can be found in the listing in [2]. A discussion of the system currently running ZIP can be found in [3]. Of course, systems with interrupt driven keyboards or nonmemory mapped screens will complicate the porting of ZIP.

## III. Use of ZIP

### *A. Integration of ZIP into the Classroom*

ZIP was first used in teaching ECE 365 during the fall term of 1981. Although it was introduced in the lecture, it was not used as part of any formal presentations. Rather, it was offered to students to allow them to experiment at their own pace with the concepts presented in lecture. This simplified teaching to an audience of widely diverse backgrounds. Those students with a strong background in assembly language programming were not subjected to what amounts to remedial lectures, while students with a weaker background were provided with a means to explore new concepts in much more depth than is normally available to members of a class of 200

students. Being experimental, ZIP was required to be used in only one of the seven lab experiments, although its use was strongly encouraged in the others. Accordingly, much time and effort was devoted discussing ZIP's operating details and its usefulness as both a learning and debugging aid in the recitation section meetings.

The students' first exposure to ZIP is an overview of the program. Operating principles of ZIP are discussed in very general terms, as students are not assumed to have any previous experience with system programs such as assemblers, loaders, or interpreters. Emphasis is initially placed on covering the step-by-step instructions necessary to begin using the interpreter. After satisfying this immediate need, students are encouraged to explore the capabilities of ZIP. Then, as a better understanding of the concepts underlying program assembly, loading, and execution is gained, operating principles are covered in more depth. The command structure of ZIP is relatively simple and straightforward; hence, students can master its use in less than an hour. Of course, full proficiency is obtained through repeated use coupled with an understanding of the underlying concepts.

The mandatory use of ZIP occurs during the first laboratory experiment. Experiment 1 is a tutorial designed to acquaint the student with the RacPac systems and the available software. Source code for a bubble sort program, which sorts bytes, is provided, together with the instructions for editing, assembling, loading, interpreting, and executing it. The instuctions for interpreting the program demonstrate the more immediately useful commands and, via the display command, enable the students to see the numerically larger bytes bubble up through consecutive memory locations to the top of the list. No assumption of previous Z80 assembly language experience is made. However, students are informed that programming expectations increase rapidly.

The second experiment is intended to familiarize students with elementary assembly language mnemonics and constructs, and the relationship between source code, source data, and their representation in memory. Only a minimal programming effort is required. A very useful visualization of the code/data/memory relationship is provided by the left-hand side of ZIP's display (see the section on screen layout). Students can directly observe the hexadecimal contents of a given memory location and either its assembly language mnemonic or its ASCII equivalent representation in the case of instructions or data, respectively. While we recognize that not all program data are ASCII character strings, the majority of data items in our first few experiments are. Hence, we felt that the benefits of visualizing the relationship between ASCII text strings and their storage representations outweighed any possible confusion that the display might cause for nonASCII data.

The next two experiments stress program structure and efficient algorithm design. These problems are much more complex and require significantly more programming effort. To assist this effort, students are introduced to some of the more powerful instructions of the Z80, particularly the data manipulation instructions found in the block move, the bit rotation and shift, and the jump groups of instructions. By comparison to earlier classes, the introduction of ZIP made possible a much quicker understanding of the behavior of these instructions. The side-by-side display of the 8-bit register set in both hexadecimal and binary provides a very clear picture of how the rotate and shift instructions work. This display is also very useful for explaining the function of the condition codes register, particularly as it applies to conditional instructions. Block moves of a short distance can be seen in the display of the 32-byte memory segment. The ability to view the above displays simultaneously with the disassembled current instruction provides a unique look at the execution of complex instructions.

The following two experiments involve digital logic design using the attaché case "logic lab" kits. The final experiment requires students to interface the logic lab kits to the RacPacs. At this point, most students have a fairly good conceptual understanding of their programming task, and hence, ZIP is used primarily as an interactive debugger.

As an extension to the recitation, tutorial presentations are offered during the unused lab periods of the second experiment. These presentations supply detailed information on the operation of the interpreter and provide the opportunity to debug programs in a controlled atmosphere where questions and/or problems can be immediately addressed. In an effort to reach the entire class with such presentations, the use of in-class CRT monitors is being planned.

### *B. Student Acceptance and Use*

To date over 1000 students have used ZIP as part of ECE 365. During the final week of class students are asked to provide written comments about the class to the instructor. The students were asked to comment specifically on ZIP. In addition, the lab teaching assistants were also asked to comment on ZIP. The following summarizes the responses.

The introduction of ZIP into the classroom was well received. The interpreter was most often used as a debugging aid, a capacity in which ZIP performed very well. Teaching assistants reported a considerable reduction in the amount of debugging help required by those students using the interpreter. Although ZIP was used less frequently as a self-teaching tool, students using it for this purpose indicated that it significantly aided their understanding of assembly language. An encouraging indication of ZIP's capacity to generate interest in the class material was evidenced by the number of students who spent their free lab time exploring the RacPac systems in detail by interpreting the load modules of the operating system, monitor, and other system programs.

Now that ZIP's capabilities in a classroom environment are better understood, and with a staff of laboratory teaching assistants familiar with its use, more emphasis will be placed on the use of ZIP as a teaching aid. Using the interpreter in class will alleviate the dependence on the chalkboard and textbook for providing explanations of Z80 assembler instruction execution—both are "static" representations of what is essentially a dynamic process. In our opinion, the ability to immediately relate the static nature of an instruction mnemonic to the dynamic nature of its execution is a major factor in eliminating

one of the more common conceptual difficulties encountered by most students as they begin to learn assembly language.

## IV. Conclusion

The reasons for the creation and introduction of a teaching aid like ZIP have been presented. The use of ZIP has been discussed and its description given. Many of its capabilities are available only in logic analyzers such as the HP-1610A. ZIP provides a far less expensive alternative and offers the further advantage of being available for use on all of the systems simultaneously.

Subsequent versions of ZIP will enhance its capabilities by allowing more complex trigger conditions, interactively interpreting interrupt service routines within the context of the main program, and providing a "history" function to allow the easy recall of recently used commands. Expansion of the use of ZIP, both within ECE 365 and into courses such as Digital Computer Laboratory (ECE 366), is also planned.

The source listing for ZIP is available in [2]. Program maintenance and subsequent updates will be available through the authors.

## Acknowledgment

The authors would like to express their thanks to W. A. MacLeod for his help in the initial phases of the program development and to the referees for their helpful suggestions.

## References

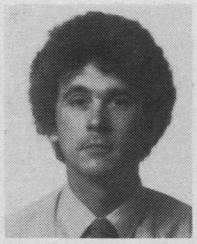
[1] T. N. Mudge, "Teaching assembly language using an assembly language interpreter," in *Proc. 1981 Nat. Conf. ASEE*, Univ. Southern Calif., June 1981, pp. 22–27.

[2] G. D. Buzzard and T. N. Mudge, "User manual for ZIP, a Z80 assembly language interpreter program," Univ. Michigan, Ann Arbor, SEL Rep. 154, Aug. 1981.

[3] T. N. Mudge, "A course sequence in microprocessor-based digital system design," *IEEE Trans. Educ.*, vol. E-24, pp. 14–21, Feb. 1981.

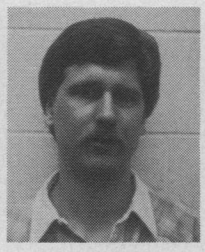
[4] M. M. Mano, *Computer System Architecture*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[5] *1982/83 Z80 Designers Guide*, Mostek Corp., Carrollton, TX, 1982.



**Trevor N. Mudge** (S'74–M'77) received the B.Sc. degree in cybernetics from the University of Reading, England, in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively.

He has been with the Department of Electrical and Computer Engineering at the University of Michigan since 1977, and currently holds the position of Assistant Professor. He is the faculty adviser for Eta Kappa Nu. He was responsible for developing a state-of-the-art microcomputer lab that handles 500 students a year. His research interests include computer architecture, VLSI circuit design, large scale computation, and robotics. His research in large scale computation is concerned with exploiting the potential of VLSI to produce high speed multiprocessors. His research in robotics is centered on developing special purpose VLSI systems to handle the high computation rates associated with vision and real-time robot arm control.



**Gregory D. Buzzard** (S'79) received the B.S. and M.S. degrees in electrical engineering from the University of Michigan, Ann Arbor, in 1981 and 1982, respectively. He is currently pursuing studies for the Ph.D. degree in electrical engineering at the University of Michigan, in the area of object-based multiprocessor architecture.

He is a Kodak Fellow, and is currently affiliated with the University's Robotics Research Laboratory.

Mr. Buzzard is a member of Eta Kappa Nu and Tau Beta Pi.