An approach to firmware engineering*

by DAVID A. PATTERSON
University of California
Berkeley, California

INTRODUCTION

Although microprogramming has only been a research topic for the past 10 years, the concept is nearly as old as computers. Wilkes originally defined microprogramming as a systematic and orderly approach to the design of the control section of a computer. Rather than using an ad hoc approach with counters and decoders to generate control signals, Wilkes proposed that the control be organized as a matrix. This memory-like approach to control remained largely of academic interest until the late 1950s. The first large-scale microprogrammed computer was the IBM 7950 in 1961. However, it was the introduction of the IBM System/360 line (with all but the largest system microprogrammed) that revived interest in microprogramming. The advance in speed and the reduction in cost of read-only memory (ROM) technology transformed microprogramming from an academic concept of mild theoretical interest to a practical approach to the design of commercial computers. IBM was able to offer several internally different models with the same instruction set in order to provide a spectrum of cost and speed through the use of microprogramming. With the success of the 360, several microprogrammed computers were and are being built. In the late 60's some manufacturers made microprogramming available to the users. Nearly all computer manufacturers have at least one computer implemented via microprogramming, and the expanding list of manufacturers that provide user microprogrammable computers include Burroughs (E1700), Control Data Corporation (Cyber-17), Data General (Eclipse), Digital Equipment Corporation (PDP-11/60) and Hewlett Packard (HP-21MX). This has expanded the potential domain of microprogrammers from computer architects to systems and applications programmers.

While microprogramming and “ordinary” programming have existed for almost the same amount of time, little of the considerable research in programming languages and methodologies has impacted the microprogramming field. Unfortunately, the following statement is a fairly accurate description of the state of the microprogramming:

"At present, microprogramming is an elite activity, performed effectively only by a small number of expert practitioners. The work is detailed, precise, time consuming, and considerably more expensive than present-day software programming. But, computer manufacturers have found they can get dramatic improvements in system performance by converting software into microprogrammed form."

An approach to expanding the group of practitioners of microprogramming is to reduce the complexity and expense of firmware by providing tools similar to those that have reduced the cost of software. The Strum system was designed as a prototype to explore the benefits and detriments of such an approach.

SOFTWARE ENGINEERING TECHNIQUES

The most obvious technique to enhance the production of microprograms is to provide a high level microprogramming language (HLML). One reason HLMLs have not been used is the importance of efficient firmware. With restricted size and execution times being primary characteristics of microprogramming, some doubt whether a HLML could ever be useful. With the newer computers control memory has expanded (e.g., 16K control memory of the HP-21mx) thereby relieving some of the concern about size. A more important aspect is the influx of applications microprogrammers who have a very different set of constraints from the microprogrammer of the native instruction set. In the latter case one may be able to justify doubling the effort to reduce execution time by 10 percent. In the former case it is likely that ease of use, development cost and maintainability are much more important than small gains in execution speed. The necessity of a high level microprogramming language is further illustrated by the Figure 1. It is widely accepted that the programmer spends half his time testing the program. This figure indicates that a microprogrammer may spend half his time coding the microprogram. As it is widely accepted that a high level language will reduce coding time it makes eminent sense to provide a high level microprogramming language if it can produce reasonably efficient code.

Another technique that has reduced the time to develop software is structured programming. This phrase is so widely used that the criteria that determines whether a program is

* This research was supported in part by the Department of Energy under Contract EY-76-S-03-0034, PA214.
structured is vague. By structured programming we mean a disciplined approach to programming that starts with the highest level of abstraction and proceeds with a toptdown hierarchical development of levels of modules that are successively refined and expanded until the bottom refinement is the program in some programming language. A key aspect of this approach is for every module at each level to be "intellectually manageable," i.e., restrict the size and flow of control to create modules that are easy to read and to understand. Although assembly level programming does not preclude structured programming, it is easier to follow these concepts if the programming language supports this methodology. We believe that a cleverly designed microprogram language can hide the "unstructured" aspects of microprogramming plus provide tools conducive to structured microprogramming.

A final aspect of software engineering is validation. Here an attempt is made to determine whether a program is "correct." Unfortunately testing does not guarantee that a program is correct; it only means that someone (probably the creator of the program) is satisfied that the program performs properly within a subset of the total input domain. As Dijkstra states:

"Program testing can be used to show the presence of bugs, but never their absence!"

If we conclude that testing is inadequate, our only alternative is formal program verification. This work began with the inductive assertion method of Floyd that was elaborated by Hoare and Manna. This method first requires the addition of redundant text called assertions. Assertions are nonexecutable comments that describe the desired state of the variables of the program. These assertions can then be mechanically combined with the program text to produce logical formulas. If proven true, these formulas show a kind of consistency, usually called partial correctness, between the text and the assertions.

It is not clear that program verification is a practical tool for validating microprograms. In fact, there is a great deal of skepticism of the practicality of proofs of correctness in any environment. The criticism is that only "toy" programs are proved, i.e., programs that use only integers or vectors of integers, deal with numeric algorithms (e.g., division), and are relatively short. Using these measures, one would have to conclude that microprograms are "toy" programs. Microprograms deal with integers (usually two's complement binary numbers) or integer vectors (memories), deal with numeric algorithms (instruction sets), and are relatively short (frequently less than 1024 words). Thus, the application of correctness proof techniques to microprograms may prove beneficial.

STRUM SYSTEM

The prototype system developed to test this approach to firmware engineering is called Strum (from STRUCTured Microprogramming). This system includes a Pascal-like high level language, an optimizing compiler that produces microcode for the Burrough's D-machine, and a verification system. The Burroughs D-machine is a general purpose microprogrammable, multiprocessor computer. The computer was designed by the Burroughs Advanced Development Organization in 1968-1969 to be used in future Burroughs systems. The D-machine has several innovations. It was designed to be used in multiprocessor configurations. The arithmetic section of the machine is expandable in 8-bit increments from 8 bits through 64 bits. It was the first machine to use two levels of control memory (micromemory and namememory) to try to obtain the advantages of horizontal and vertical microprogramming. And at the same time it was built, it was one of the few computers to offer a control memory that could be altered by the user during program execution. The D-machine allows up to 4096 56-bit microinstructions.

THE EXPERIMENT

As a meaningful test of the system we decided it was essential to emulate a real computer. If a paper computer were to be emulated one might wonder whether the design of the new instruction set had not been compromised in order to avoid difficulties with the high level language or to simplify the verification of the emulation. Although this test would be emulation of a real computer on a real microprogrammable machine, some would wonder whether program verification, high level languages, and structured programming would not lead to correct yet inefficient code. As speed and size are very important in the microprogramming environment, it is possible that the overhead incurred using such techniques makes this approach unusable in the "real world."

To overcome these possible objections, an emulation of the same real computer was done twice. The first version was implemented via techniques traditionally used with microcode on the D-machine, i.e., assembly level coding, debugging the microcode on the machine, and testing the microcode using programs written for the machine being emulated. The other version was written using techniques proposed in this paper. Unfortunately it is difficult to create tests that are free from the appearance of bias. If one person did both emulations, one could correctly argue that the experience gained on the first emulation project would improve the results of the second project. To avoid this criticism, a different person did each project. The argument, then, is that with two people the differences in education, experi-
ence, motivation, etc. affect the accuracy of the results and possibly affect the fairness of the comparison. One could also claim that the choice of a minicomputer for the experiment rather than a large-scale computer does not allow generalization of the results.* We can only respond that both microprogrammers are bright, capable people, that there are fewer and fewer distinctions between minicomputers and large-scale computers, and that we can only hope that the reader will appreciate our attempt to make a fair experiment. The test case was the emulation of the HP-2115 computer.

The Hewlett-Packard HP-2115 computer

The Hewlett-Packard HP-2115 is a predecessor of the HP-2100 and HP-21MX computers currently marketed by Hewlett-Packard. The HP-2115 has 70 basic instructions that operate on two accumulators, two flags, and (in this emulation) 8192 words of 16-bit main memory. This instruction set includes several special input-output instructions for up to 64 devices and provides for multi-level indirect and paged addressing. The HP-2115 will handle up to 64 different interrupts. Although the Hewlett-Packard manual lists 70 basic instructions, most of the opcodes in the shift-rotate group and alter-skip group can be combined to form more than 100 additional useful instructions.

HP-2115 emulation using traditional techniques

The emulation using traditional techniques was implemented by a graduate student in Spring 1975. This microprogram was written in TRANSLANG, the assembly language for the D-machine. He did not attempt to use the techniques of structured programming. He debugged the microprogram directly on the D-machine by running HP-2115 programs and examining the registers at each microstep. As we indicated above, this primitive approach is unfortunately typical of the way most microprograms are developed. This student had ideal conditions for using this approach in creating the emulation:

(1) He was thoroughly familiar with the HP-2115. He had spent two years as an M.I.T. undergraduate using this computer.
(2) He was thoroughly familiar with the D-machine. This student was in charge of maintaining the D-machine and the implementing enhancements to it.
(3) He was the sole user of the D-machine during that period. Usually a microprogrammer must compete with other users for access to the machine.
(4) The hardware was fixed and operational. With computer manufacturers the computer is usually being built and debugged while the microprogram is being debugged. † It is often difficult to distinguish between an error in the microprogram and an error in the hardware. With formal microprogram verification, the hardware is not needed until after the microprogram is verified, i.e., much later in the development cycle.

(5) There was software available to test his program. When a new instruction set is created, there is usually no program available to test the microprogram.

After he was satisfied that his microprogram worked, we tried running diagnostic programs for the HP-2115 on the emulation. Diagnostic programs are used to test if the hardware is working properly. This is a widely used technique to test accuracy of emulation. The argument is that if the diagnostics believed that the HP-2115 “hardware” was working correctly, the emulation of the HP-2115 must be working correctly. Running the diagnostics uncovered two errors in this emulation.

Subsequently we discovered that one of the unassigned opcodes performed a useful function. This opcode was obviously not tested by the diagnostics, but its use was apparently so widespread by users of the HP-2115 that they defined this opcode in a later programming manual. We were also surprised that the diagnostics did not test some apparently unusual combinations of options in the I/O instructions. The student left this out of his emulation because he doubted that it would ever be used. (In March 1976 he completed the emulation of all unusual I/O instruction options.) In writing the specification for the STRUM HP-2115 emulation, another error was discovered. Memory reference instructions can use the current page address to supply a portion of the operand address. When an interrupt occurs the program address register is not modified but the computer selects and executes an instruction from the first 64

<p>| TABLE 1.—A Comparison of Size and Speed of Microassembly, Unoptimized Strum, and Optimized Strum Emulations of the HP-2115 |</p>
<table>
<thead>
<tr>
<th>Computer</th>
<th>Microassembly</th>
<th>Unoptimized Strum</th>
<th>Optimized Strum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Micro-instructions*</td>
<td>946</td>
<td>1124</td>
<td>940</td>
</tr>
<tr>
<td>Number of Nano-instructions*</td>
<td>162</td>
<td>123</td>
<td>161</td>
</tr>
<tr>
<td>Size in bits*</td>
<td>25504</td>
<td>25856</td>
<td>25344</td>
</tr>
<tr>
<td>Average number of cycles executed per instruction**</td>
<td>72.9</td>
<td>66.4</td>
<td>58.6</td>
</tr>
</tbody>
</table>

* The Burroughs D-machine has two levels of control memory so the size should be compared either in bits or number of microinstructions.
** The Burroughs D-machine comes in several microcycle times from 50 to 500 nanoseconds.

† A referee mentioned that usually a microprogram simulator is available while the hardware is under construction. Our experience is that the simulator and its corresponding hardware quickly diverge. While the simulator is useful this divergence combined with time constraints often force the microprogrammers to have access to the computer before the hardware is debugged.
locations in memory. It was not clear from Hewlett-Packard documentation whether a memory reference instruction that requested address modification from the current page would get the information from the program address register or from the current page of the instruction (page 0). This emulation used the program address register but a test of another Hewlett-Packard computer revealed that the address bits always came from page 0. These three examples show the danger in relying on diagnostic programs as the only method of verifying an emulation. This emulation used 946 microinstructions which required 162 unique nanoinstructions. The execution speed is shown in Table I. The values in these tables are given in terms of D-machine microcycles. The D-machine at UCLA runs at 800 nanoseconds per microcycle although Burroughs has developed versions that run as fast as 30 nanoseconds per microcycle.

Strum HP-2115 emulation

The second emulation of the HP-2115 using a high level language, structured programming and formal program verification was implemented in Spring 1976 by the author. We had serious reservations about this experiment, mainly because this microprogram would be the first microprogram that used the STRUM system. A simpler test case to catch any bugs in the STRUM compiler and program verification system was planned, but lack of time ruled this out. The result of the output from the optimized code is shown in Table I.

Testing the HP-2115 emulation

The testing of the STRUM HP-2115 emulation was performed on November 11, 1976 by the creator of the other emulation and the author. The test consisted of executing the HP-2115 diagnostic routines. These routines exhaustively test all opcodes and most register values for the memory reference, alter skip, shift rotate and overflow instructions. We did not know what to expect, because this was the first test of the STRUM compiler, and the first large-scale test of the STRUM verification system. The verification process uncovered 30 errors in the emulator divided equally between the executable microprogram, the specifications, and the assertions. The program and verification formulae were corrected by hand. Although one large (about 2000 lines of NUCLEUS code) program has been verified, this program was never executed. We were in the uncomfortable position of using untested software tools to verify a program that would immediately be subjected to exhaustive testing.

Surprisingly no errors in the STRUM emulation were uncovered by the HP-2115 diagnostic programs. Four errors were found in the code generated by the compiler, but had we run a simple compiler test program beforehand, all these
errors would have been detected. This was indeed a gratifying result. To the best of our knowledge the STRUM emulator is the largest formally verified and tested program.

Summary and results

Although all the results were gratifying, we did not expect the STRUM version of the emulator to be faster than the TRANSLANG version. Many believe that high level language microprogramming is too inefficient to be useful. Possibly an explanation for the unexpected result is the difference between “microprogramming-in-the-small” and “microprogramming-in-the-large.” For small examples (10 to 20 microinstructions) it is feasible for a microprogrammer to spend hours optimizing microcode. However, with a 1000 instruction microprogram, it is extremely difficult to consistently convolute one’s thinking to write efficient microcode. Also, high level languages and hierarchical structure make it easier to perform the global optimizations that may save more time or code than any instruction-by-instruction optimization. Perhaps the most exciting result was the success of the verification of microprogram. Although working with only a prototype system, we were able to verify a complete emulation of a real computer on a real microprogrammable computer. Moreover, although the exhaustive testing performed by the diagnostic programs did find four bugs in compiler, it did not reveal any errors in the STRUM emulation. Not only did the verification uncover program errors, it uncovered errors in the specification and assertions. No matter how practical testing may seem, it will not uncover errors in documentation (specification) nor in the comments (assertions). The microprogramming language STRUM proved to be very easy to use. The use of macros and good control structures resulted in a well-structured microprogram. Figure 2 shows the structure of the HP-2115 emulation. Each of the 14 boxes corresponds to a functional section of the emulator. Only two sections occupied more than one page of the listing and 12 of the 14 boxes are macros rather than procedures. STRUM object code and the original microassembly emulation contained many more jumps, labels and called statements than the STRUM source program (see Figure 3). The STRUM source program also contained fewer loops because the “handshaking” firmware used to access memory includes loops that are generated by the STRUM compiler.

CONCLUSION

With respect to the microprogramming field we believe this experiment has demonstrated the validity of the use of each facet of the STRUM approach: use of a high-level microprogramming language, structured microprogramming, and formal microprogram verification. While we believe it is most effective to use all facets of the STRUM approach, the reader should be forewarned that the development of such a system is a tremendous undertaking. The development of production versions of an efficient optimizing compiler and verification system is probably a five to ten man year effort. Hopefully tools will be created which will simplify this task.

Readers interested in a more detailed analysis should see References 4, 8, 11 and 12.

REFERENCES
