Software Aids for Microprogram Development

Christopher Vickery

Queens College, CUNY

Summary: Debugging of microprograms can be approached in three ways: (1) with the aid of hardware test sets and monitors, (2) through interactive debugging programs, and (3) through simulation techniques. This paper discusses these three methods and describes an interactive debugging program and a simulator developed for debugging microprograms for the Interdata model 85 minicomputer.

INTRODUCTION

In the past several years almost all computer manufacturers have retreated from their original "users' hands off" attitude toward microprogramming and given at least begrudging support for user access to an alterable section of control storage. The division of control store into a read-only memory (ROM) implementing the vendor's target machine and a read/write memory (RAM) for the user to manipulate as he wishes has blunted the original fears that a user-microprogrammed processor would be impossible for the vendor to service. The maintenance engineer need only demonstrate that a failing system still operates correctly in "native" mode to prove that the problem lies in the design of the user's microcode.

Those uses of RAM control store which the manufacturers have anticipated have influenced the debugging techniques available to the user. Most vendors have anticipated the user wanting to "tune" his machine to a particular task through extension of the basic machine instruction set (vid. Reigel & Lawson, 1973). The organization of a machine built for this purpose is usually oriented toward a particular user-level instruction format, often providing fetch and decoding of user instructions in a single microinstruction. Hewlett-Packard, for example, goes so far as to provide a standard interface between Fortran programs and microcode (Park, 1973). Debugging support for this type of machine usually takes the form of interactive debugging programs. The user is presumed to be interested in how his new routine effects the user-level environment rather than in its interactions with the hardware per se. Memory searching and modification, breakpoints in microcode, and examination of the user-level environment are typically provided by the debug program.

Another class of RAM control store usage falls in the category of "emulation." Here the user wishes to implement a substantially different user-level instruction set (or subset) from that provided by the vendor. Machines designed for emulation tend to be flexible in their assumptions about user-level instruction and data formats, often employing "setup" registers to allow the microprogrammer to specify these parameters at execution time and still take advantage of hardware aids to instruction fetch and decoding. The Burroughs 1700, and to some extent the Varian 73, are examples of this class of machines. Since the emulator writer is likely to be using the hardware environment in unorthodox ways (relative to the machine's native mode of operation), software simulation of the processor can be very valuable; it allows the user to trace the entire processor state, including values that may not be directly available even to the microprogram, rather than just the conventional user-level view of the machine. The simulator may run either on the host machine itself, perhaps in an interpretive mode, or on another machine with perhaps a greater storage capacity.

Husson (1970) anticipated that with the advent of writable control storage, users might resort to the maintenance engineer's facilities for debugging the system. In addition to the usual logic testing equipment, the maintenance engineer is likely to have a test set or monitor designed expressly for servicing a particular machine. Operating as a sub-micro level operator's console, such test sets give the engineer the ability to examine and modify the hardware busses and registers, to set breakpoints in control store (both ROM and RAM), and to control the system clock (e.g., Interdata, 1973). However, such devices are designed to facilitate hardware fault detection and isolation rather than for firmware debugging. Even when they are made available to the end user, such aids are more likely to help the microprogrammer discover operational characteristics of the machine which were otherwise ambiguously defined rather than to find logical flaws in his microcode.

In the following two sections of the paper, I describe an interactive debug program and a software simulator developed for the Interdata 85 minicomputer. This machine is organized about three 16-bit busses connected to as many as eight hardware modules (a control unit, an ALU, and an I/O unit are standard). It addresses up to 64K bytes of semiconductor primary memory (270 ns) and up to 4K words (32 bits, 60 ns) of control store. The first 1024 words of control store is ROM implementation of the standard Interdata 70/80 instruction set; additional control store may be a mixture of RAM and ROM. The standard model 85 has 1024 words of RAM. The machine is highly optimized for emulating the standard instruction set, with many user-level instructions completely emulated in a single microinstruction. Thus, the machine falls in the first category of machines mentioned above, those intended for extension of the basic instruction set. The availability of this machine only with high speed semiconductor main memory reflects the vendor's belief that users would be interested primarily in the speed advantage a "tuned" machine would offer through microprogramming. Vendor-supplied microprogramming support for the system consists of a microassembler which runs under an operating system, a stand-alone interactive debug program, and a stand-alone diagnostic program. Also, user-level instructions were added to the 70/80 instruction set ROM which transfer information between user memory and RAM control store, and to branch into control store from user routines. A hardware test set is available as an option.

Micro-Delta

This is the interactive debug program we wrote to replace the one supplied by the vendor. Programs of this sort have much in common with those normally supplied by vendors for debugging user code (ours is named after DELTA, which is the debugging system for XDS Sigma computers). Memory search, examination, and modification, breakpoints, and disassemblies are the usual ingredients of such a program. Of course, the debugger for a microprogrammed machine must do double duty because it should provide the same functions for a user-level program as for a microprogram; this feature is especially important in the case of a machine which is using microcode to extend the user-level instruction set because the microprogram under test is usually invoked by a user-level program which sets up parameters, etc. In Micro-Delta, for example, the user types "M" or "U" to indicate which level of the machine he wishes to deal with ("micro" or "user"), then uses the same set of commands for either level.

One problem with a debug program is that not all of the processor environment is available even to the microprogram. Furthermore, that which is available may be destroyed by the process of interrupting an errant section of microcode (either through a breakpoint or by direct intervention through the console switches). For example, branches in Interdata microcode are of the branch-and-link type, so the instruction inserted at the breakpoint necessarily destroys one register just to escape from the tested program's execution sequence. Runaway microcode is particularly troublesome; an interrupt of some sort must be generated to stop the processor, which means that registers are destroyed by the ensuing automatic (ROM firmware) PSW swap. One solution to this problem is to use a second processor to monitor the one under test, perhaps with communication between the two through shared memory (Gasser, 1973). Aside from the obvious cost disadvantage, this solution still fails if the runaway microprogram never checks the sentinal in shared memory. Examples of information not available to the Interdata microprogram are the user instruction register (UIR), the primary memory address register

(MAR), the control store address and data registers, and the status flags from the ALU or I/O unit. All can be modified or tested by the microprogram, but none can be stored away for display by the breakpoint routine.

Micro-Delta differs from the debug program delivered with the system in two major respects: (1) It runs under an operating system. Although it modifies the OS and operates in the privileged mode, Micro-Delta restores the complete OS environment before exiting. The vendor's program was stand-alone, which required reloading of the OS after each use of the debugger, such as to reassemble a microprogram. (2) Micro-Delta is highly modular. The nucleus requires about 4K bytes of user memory, while subroutines perform the control store load, memory dump (one routine for control store and another for user memory), and disassembly functions. Calling any routine not linked at the time generates an appropriate nasty message for the user, who may then exit to the operating system to link the desired routine(s) from a library and return to where he left off in Micro-Delta. The vendor-supplied program is monolithic, and uses 8K bytes of memory.

The Simulator

If our interest had been just in extending the basic Interdata instruction set so we could do production runs of enhanced user-level programs, the debug program would probably have sufficed. As it is, we face a wide variety of interests in the machine by our faculty (emulation, operating system enhancement, etc.), as well as a potentially large number of users (there are 400 undergraduate majors in our department). The first factor suggested that a simulator would be a valuable research aid; the second suggested that it would be wise to take advantage of other computer facilities available in developing the simulator. To this end, the simulator was designed to run in the batch mode, with interactive facilities available optionally to the on-line user. Furthermore, we decided to write the program in Fortran so it could be run both at the campus computer center (twin Xerox Sigma 7s) and at the City University computer facility (IBM 168s). The price paid, of course, is that execution time is very long, especially when large amounts of output must be formatted by the Fortran edit routines. Simulated-to-real execution ratios as poor as 5000:1 may occur, if the user opts for a complete printout for a run.

The simulator has some valuable features not always found in such programs. For one, I/O functions are simulated as faithfully as possible, enabling meaningful simulation of time-dependent programs. For another, we wrote cross-assemblers (both micro and user level) to run on the Sigma 7s. We then added a loader which accepts the output from the assemblers and loads it into the simulated machine's memories (output from the assemblers ordinarily goes to magnetic tape which is transported manually to the Interdata for real-time execution). In addition, the standard model 85 ROM data and a core-image of one of the standard Interdata operating systems are in Sigma 7 files and can be included in the simulator's memories automatically. In a single batch job, a user can thus (1) assemble a user-level test program, (2) assemble a microprogram for RAM or ROM control store, (3) load the object code from both assemblies along with the standard ROM and OS image, and (4) trace the execution of the system to whatever level of detail is desired.

The structure of the simulator parallels the structure of the Interdata as much as possible (There is one subroutine for the ALU, another for the I/O unit, one for each of the control unit states, one for each I/O device controller, etc.). This structure means that the timing calculations for both the synchronous and asynchronous parts of the system were straightforward: Synchronous routines simply advance the system clock by an appropriate amount before returning to the calling routine; asynchronous routines post their anticipated completion time in a common area and return immediately to the caller. The result has been very close agreement between the simulated execution times and the vendor's published values.

CONCLUSION

I have suggested that small microprogrammable computers today are used in either of two ways: (1) as "tuned" versions of a vendor-designed processor, or (2) as emulators. Further, I have suggested that interactive debugging programs are appropriate for testing microprograms on the first type of machine, while simulation programs are more valuable when testing programs for the second type of machine. Examples of both a debug program and a

simulator for a machine of the first type (which is also used for emulation) were described.

REFERENCES

Gasser, M. An interactive debugger for software and firmware. Preprints of the Sixth Workshop on Microprogramming, College Park, 1973.

Husson, S.S. Microprogramming: Principles and Practices. Englewood Cliffs: Prentice-Hall, 1970.

Interdata, Inc. Model 80 test aid instruction manual. Interdata publication #29-344, 1973.

Park, H. Fortran enhancement. Preprints of the Sixth Workshop on Microprogramming, College Park, 1973.

Reigel, E.W. and Lawson, H.W. At the programming language - microprogramming interface. In R.L. Wexelblat (Ed), Proceedings of the ACM Sigplan-Sigmicro Interface Meeting, Harriman, 1973.