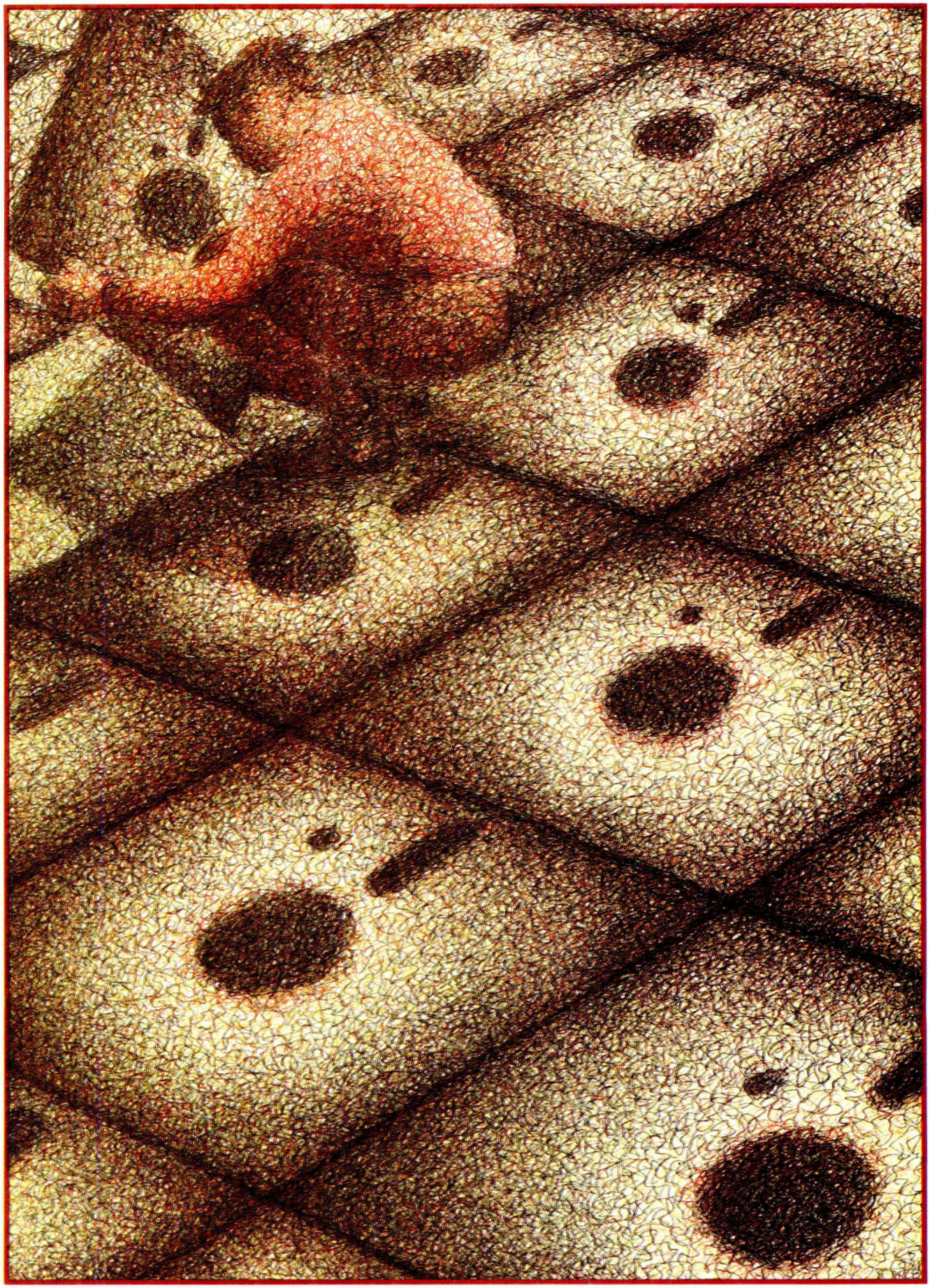
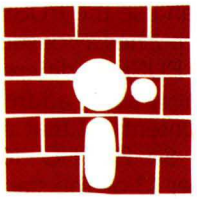


ILLUSTRATED BY DAVE RIDLEY

BUILDING A COMPUTER IN SOFTWARE

BY JONATHAN AMSTERDAM

The realization of one software hacker's dream of designing a computer



I must admit that as a software hacker, I sometimes feel envious of my hardware-inclined colleagues. I dream occasionally of building a computer from the ground up, out of transistors, gates, adders, barrel shifters, and other such esoterica. It's not so much the hands-on feel of construction that I desire (though I do at times long for a whiff of melting solder); rather, I crave the power of making every decision in the design process. I would decide how much memory the machine has, how many registers there are, and what the machine's built-in instructions would be. My hardware dreams will never be fulfilled. But my design dreams can still be realized—in a program.

In other words, I can build my computer in software. And I have. That's what this programming project is all about.

VIRTUAL MACHINES

A software simulation of a computer—a so-called *virtual machine*—can be a helpful tool. Its utility is obvious in designing new processors: You can test out your design in a program before committing it to silicon. But building a virtual machine has also been proved worthwhile in implementing programming languages. The idea is to design a computer ideally suited to the execution of the high-level language you want to implement. Since chances are that no real computer will fill the bill, the dream machine is simulated on a real machine. You are then free to compile the high-level language into instructions for the virtual machine, which will then be interpreted by the simulator. Because the virtual machine was designed with the high-level language in mind, writing a compiler should be relatively easy. And there is an added bonus of portability: The compiler will work on any computer on which the virtual-machine simulator is running. Writing a virtual-machine simulator for a new machine is usually much easier than retargeting a compiler to produce native code for that machine. Furthermore, because the virtual machine's instruction set can be optimized for the high-level language, the program's object code is often much smaller than a corresponding program translated directly to machine language.

The price paid for all this is speed. Because the virtual machine is interpreted in software, programs compiled into virtual machine language run more slowly than native-code programs.

(continued)

Jonathan Amsterdam is a graduate student at the Massachusetts Institute of Technology Artificial Intelligence Laboratory. He can be reached at 1643 *Cambridge St. #34, Cambridge,* MA 02138.

An early use of the virtual-machine technique was in one of the first Pascal compilers. Later, Kenneth Bowles adopted the technique for his UCSD Pascal system, which is still going strong. Virtual machines for Pascal are often called P-machines, and their native language, p-code. Xerox's exciting object-oriented language, Smalltalk, also uses the virtual-machine scheme. The Smalltalk virtual machine takes the burden of manipulating objects off the compiler writer's shoulders. And Niklaus Wirth's Modula-2 has also been implemented using a virtual-machine code called m-code. In fact, the system that I used for this project, MacModula-2 from Modula Corporation (950 North University Ave., Provo, UT 84604 (801) 375-7400), is implemented this way. Wirth's computer, the Lilith, solves the virtual-machine speed problem: Its processor executes m-code instructions directly. The virtual machine and real machine are one and the same.

Soon there will be another programming language joining the virtual-machine club. I designed it, and in the near future I will be discussing a compiler for it in these pages. But before I discuss the compiler, I need to specify the virtual machine.

VM2 Specification

My virtual machine is called VM2 (VM for virtual machine, M2 for Modula-2). VM2's memory is divided into an unspecified number of units called words. The size of a word is also unspecified, but it must be large enough to hold the address of any word in VM2's memory. A word is addressed by a number between 0 and one less than the size of memory. A typical "real"-size VM2 might have 64K 16-bit words, addressed from 0 to 65535. A word is the smallest, indeed the only, unit of storage. There are no bytes. I'm trying to keep things simple.

VM2 has three word-length registers. A register is a special storage location that is not part of memory. The programCtr register holds the address of the next instruction to be executed. The stackPtr register holds the address of the top of the stack. (See the text box above for an explanation of stacks.) VM2's stack grows down from the top of memory, so when the machine is reset, the stackPtr register points to the highest word of memory. The stackLimit register is a special feature designed to protect programs, which reside in the low end of memory, from being overwritten by the stack. Every time something is added to the stack, the value of the stackPtr register is compared with stackLimit. If they are ever equal, a stack-overflow error results. If you set stackLimit to point just past the top of your program, you can be sure that the stack won't overwrite the program.

The instruction set for VM2 is found in figure 1, along with an explanation of each instruction. All VM2 instructions occupy one word of memory. Some instructions (PUSH, for example) take a one-word argument, which is always located in the word immediately following the instruction in memory. I've tried to keep the number of instructions down, although I've left some redundancy in the instruction set to make it easier to write programs. For instance, you can get the effect of a PUSH by using a PUSHC followed by a CONTENTS, but I figured that PUSH would be very common.

VM2 is a so-called stack machine. That means that many instructions either expect values on the stack, place their results on the stack, or both. The ADD instruction is a prime example. First it pops two words off the stack. Then it adds them together, treating them as signed integers, and pushes the result back on the stack. The SUB, MUL, DIV, and NEG instructions work similarly. To get something onto the stack in the first place, you can use the PUSHC instruction, which pushes a constant value on the stack. PUSH is different; it treats its argument as an address and pushes the contents of that address on the stack. So PUSHC 13 will put the number 13 on the stack, while PUSH 13 will put the contents of memory location 13 on the stack. For example, the three lines

```
PUSHC 3
PUSHC 12
ADD
```

will put the numbers 3 and 12 on the stack, add them, and leave the result, 15, on the stack.

Getting things off the stack and into another memory location is done with the POPC instruction, which puts the top of the stack into the memory location specified in the instruction. If we put a POPC 25 instruction after the ADD in the program above, then the value 15 would be stored in location 25. POP is similar, but instead of the address being an argument to the instruction, it is one of the values on the stack. For instance, if the number 13 is on the top of the stack with 24 just below it, then the POP instruction will put 24 into location 13.

Several instructions change the value of the programCtr register.

Stacks

My virtual machine, the VM2, makes use of a data structure called a stack. Abstractly, a stack is a sequence of elements with two operations, PUSH and POP. Pushing an element onto a stack adds it to one end of the sequence, called the top of the stack. The POP operation removes the top element from the stack. A stack data structure acts just like a stack of dishes: Things are always added and removed from the top.

There are many ways to implement stacks. VM2 uses one of the simplest. A contiguous region of memory—an array—holds the elements of the stack. A pointer into the array indicates where the top of the stack is. In VM2, the stack pointer is set initially to point to the highest memory location. When a PUSH occurs, the element is placed in the location pointed to by the stack pointer, and the stack pointer is then decremented. For a POP, the stack pointer is first incremented, then the contents of the location it points to is returned. If a POP is attempted when the stack is empty, an error occurs.

BRANCH takes an argument and sets the programCtr register to it. This allows one to transfer control to any other instruction in the program. JUMP is like BRANCH but gets the new value for the programCtr from the stack. It's useful for returning from subroutines and implementing jump tables. BREQL, BRLSS, and BRGTR pop the top value off the stack and set the programCtr to their argument if the value is equal to, less than, or greater than 0, respectively.

Seven instructions work with Boolean values. A 0 is taken to mean false, and anything else is interpreted as true. The machine itself indicates true by the number 1. The EQUAL instruction, for example, compares the top two words on the stack, popping them off in the process. If the words are equal, it pushes a 1 on the stack; if they're not equal, a 0. The five instructions NOTEQL, GREATER, LESS, GTREQL, and LSSEQL work similarly. NOT complements the value on the stack; 0 is changed to 1, and anything else is changed to 0. These instructions will come in handy when I discuss the compiler.

There are four instructions for I/O (input/output). WRCHAR and WRINT write the value on the top of the stack to the screen. They differ in whether they treat the value as a character or integer. RDCHAR and RDINT read either a character or an integer from the keyboard and push it on the stack. A couple of miscellaneous instructions round out the set. CONTENTS replaces the address at the top of the stack by its contents, and HALT stops the machine.

Mnemonic	Instruction	Arg?	Function
Arithmetic			
ADD	Add	no	Add the top two values on the stack.
SUB	Subtract	no	Subtract the top of stack from the value below it.
MUL	Multiply	no	Multiply the top two values on the stack.
DIV	Divide	no	Divide the second value on the stack by the top one. Truncate and discard the remainder (like Pascal's DIV operation).
NEG	Negate	no	Negate the top of the stack.
Boolean			
EQUAL	Equal	no	If the top two items on the stack are equal, push a 1; else push a 0.
NOTEQL	Not Equal	no	If the top two items on the stack are not equal, push a 1; else push a 0.
GREATER	Greater	no	If the top of the stack is greater than the value below it, push a 1; else push a 0.
LESS	Less	no	If the top of the stack is less than the value below it, push a 1; else push a 0.
GTREQL	Greater or Equal	no	If the top of the stack is greater than or equal to the value below it, push a 1; else push a 0.
LSSEQL	Less or Equal	no	If the top of the stack is less than or equal to the value below it, push a 1; else push a 0.
NOT	Not	no	If the top of the stack is 0, replace it with 1; else replace it with 0.
Stack Manipulation			
PUSHC	Push Constant	yes	Put the argument on the stack.
PUSH	Push	yes	Put the contents of the location specified by the argument on the stack.
POPC	Pop Constant	yes	Put the top of the stack into the location specified by the argument.
POP	Pop	no	Put the second value on the stack into the location specified by the top of the stack.
Control			
BRANCH	Branch	yes	Set the program counter to the location specified by the argument.
JUMP	Jump	no	Set the program counter to the top of stack.
BREQL	Branch if Equal	yes	If the top of the stack is 0, branch to the argument.
BRLSS	Branch if Less	yes	If the top of the stack is less than 0, branch to the argument.
BRGTR	Branch if Greater	yes	If the top of the stack is greater than 0, branch to the argument.
Input/Output			
RDCHAR	Read Character	no	Read a character from the keyboard and put its ASCII value on the stack.
RDINT	Read Integer	no	Read an integer from the keyboard and put it on the stack.
WRCHAR	Write Character	no	Write the top of the stack to the screen, treating it as an ASCII value.
WRINT	Write Integer	no	Write the top of the stack to the screen, treating it as a signed integer.
Miscellaneous			
CONTENTS	Contents	no	Replace the top of stack with the contents of the location specified by the top of stack.
HALT	Halt	no	Stop the machine.

Figure 1: *The instruction set of the* VM2 *virtual machine.*

THE VM2 INSTRUCTION CYCLE

The job of VM2 is to laboriously but accurately execute machine instructions. The basic algorithm is simple: A loop fetches the next instruction from memory and executes it. If a HALT instruction is ever seen, the loop stops. I'll now explain how to implement the loop, which is called the *instruction cycle* of the machine:

1. Get the next instruction from mem-

(continued)

My approach to implementing VM2's instructions was straightforward.

ory. You know which instruction the "next" one is because, by convention, the programCtr register points to it.
2. If the instruction is a HALT instruction, stop the machine.
3. Increment the programCtr register. It will now point to the next instruction or to the argument of the current instruction if it has one.
4. Execute the instruction. This means looking up the instruction in a table and jumping to the appropriate subroutine.
5. Go to step 1.

Obviously, most of the work is done in the subroutines that implement each instruction. An instruction with an argument can find the argument by seeing where programCtr is pointing, but it has to remember to increment programCtr to point to the next instruction.

Implementing VM2

Ideally, VM2 should be implemented in assembly language for speed. But I don't enjoy programming in assembler, and there is the additional drawback that there is no one standard assembly language that everyone can read and understand (after all, portability is one good reason for using a virtual machine). Instead, I wrote VM2 in Modula-2, which is my programming language of choice these days.

Writing a machine simulator in a high-level language is a bit tricky because the machine "hardware" likes to be able to interpret memory values in different ways: sometimes as an address, sometimes as a signed integer, sometimes as a character, or perhaps even as an instruction. But many high-level languages, the so-called strongly typed languages, are rather insistent about assigning a unique type—integer, character, real, etc.—to every location. You have to implement VM2's memory words as some type, but you need to be able to treat those words as being of several different types.

Pascal programmers have long known how to defeat the Pascal type system. You set up a variant record that has one variant for each of the different types you want to use. Accessing the record as r.int, say, lets you view its contents as an integer; using r.ch lets you use it as a character. If you program in Pascal, this solution is necessary, but it is rather inelegant. It's also somewhat dangerous because its correctness depends on the computer you're using. On some machines, characters and integers may be compatible; on others, perhaps not. The Pascal compiler won't tell you.

Modula-2 has a better solution. I find it much more elegant, and while it's still machine-dependent, it's considerably safer than the Pascal approach. It also makes it easier to identify the machine-dependent parts of your code. Modula-2 has a built-in type called WORD, whose actual size is implementation-dependent, but is considered to be the smallest useful unit of storage on the machine. You can change the type of a WORD variable easily. For instance, if w is of type WORD and you want to convert it to an INTEGER, just write INTEGER(w). No actual computation is performed in converting between types; INTEGER(w) just indicates to the compiler that a value of type WORD is to be treated as an INTEGER. If INTEGERs and WORDs aren't compatible (i.e., if INTEGERs occupy more than one WORD of storage), the compiler will let you know. Happily, CHARs and INTEGERs, as well as the representations for VM2 addresses and instructions, are all compatible with WORDs in the implementation of Modula-2 I use and, I strongly suspect, in nearly all others.

With this little problem out of the way, building VM2 is a straightforward matter. Memory is an array of WORDs; registers are global variables; a CASE statement takes care of dispatching to the appropriate subroutine for executing an instruction. Instructions (called op codes in my implementation) are represented by a scalar type,

```
TYPE opCode = (Add, Sub, Mul,
  Div, . . . );
```

The Modula-2 compiler takes care of mapping the elements of this type—Add, Sub, and so on—to numbers. For instance, Add turns out to be 0 when viewed as an integer. It's important not to confuse these op codes with the mnemonics ADD, SUB, etc., which are character strings. The monitor program I've provided has facilities for converting between mnemonics and op codes.

The two workhorses of the simulator are the pushWord and popWord procedures; nearly every instruction uses one or both of them. They handle the stack, including checking for underflow (trying to pop something off an empty stack) and overflow (trying to push something past stackLimit).

In addition to the VM2 simulator, I've also made a simple monitor program available. It is of little theoretical interest but of enormous practical value in using VM2. It provides facilities for examining and storing into VM2's memory, printing out the contents of the stack and registers, running programs, and single-stepping (executing programs one instruction at a time). By the way, if you know Pascal but not Modula-2, you shouldn't be afraid to look at the program. The two languages are so similar that you ought to have little difficulty reading the code, and the only trouble involved in translating it into Pascal is the problem of type conversion that I discussed above. [*Editor's note: The code for* VM2 *can be found on* BYTE*net Listings. The phone number is* (617) 861-9774].

Notes on Efficiency

My VM2 implementation is designed for clarity and portability, not efficien-

(*continued*)

Instructions and data each occupy one word.

cy. It is possible both to increase the speed of the interpreter and to reduce the size of the instructions.

My approach to implementing VM2's instructions was straightforward. I wrote some low-level routines to do pushes and pops, then used them in implementing the instructions. So the code for ADD, for example, actually pops two values off the stack, adds them, and pushes the result back on the stack. But the pops and pushes are expensive: Each one checks for an error, increments or decrements the stack pointer, and accesses a memory location. Furthermore, using one of the routines means doing a subroutine call. We can get the same effect as my ADD more cheaply by first checking if there are at least two things on the stack (a quick test of the stackPtr register) and then, if there are, adding the top item to the one below it and incrementing the stack pointer. Although I find this stack juggling a bit inelegant, it is certainly faster.

For reasons of simplicity and portability, I stipulated that instructions and data each occupy one word. But if a word is 16 bits—a typical value, and in fact the size of the WORD data type in MacModula-2—then the instructions are much too large. An instruction size of a byte (8 bits) is more reasonable; it provides room for 256 instructions, which is plenty. You would have enough instructions to fill all your needs and still have enough op codes left over to provide useful optimizations of common instructions, which will provide a further reduction in space. For example, pushing small constants like 0 and 1 is a common operation. PUSHC 0 takes two words (or 4 bytes) in the current implementation; by making instructions 1 byte long, you could get it down to 3 bytes; but by providing a special PUSH0 instruction, you could reduce it to only 1 byte.

You could also save space by allowing byte-size as well as word-size arguments. A "Push Small Constant" instruction, which took a 1-byte argument, would make it possible to push values from 0 to 255 with only 2 bytes. Or it might treat its argument as a signed integer, allowing values from −127 to 128. The control instructions are also ripe for byte shaving. Since most branches are to nearby locations, a variety of branch instructions that took a 1-byte offset from the current address rather than a 2-byte absolute address would reduce the size of most branch sequences. It would make the most sense in this case to treat the byte as an integer between −127 and 128.

All these space-saving hacks and more can be found in the instruction sets of "real" computers. You should definitely consider them if you want to write a virtual machine for serious work on a microcomputer. But as you'll see when I discuss the compiler, you can get an even greater space reduction by encoding complicated high-level language operations, which would take many instructions on a conventional architecture, into a single virtual-machine instruction.

Where To Go from Here?

When you get your virtual machine up and running, you will discover a somewhat depressing fact: It's not much fun to program. Poking instructions into memory one by one using the monitor is only a few steps above flipping toggle switches on a front panel. Now that may send a nostalgic shiver down your spine, but being a software person, I would rather have something do the job for me. I'd like to be able to write my program as a list of mnemonics, and I'd like to be able to use symbolic names for memory locations rather than numerical addresses. I'd like a program that translated the mnemonics and symbolic names into the op codes and addresses that VM2 prefers. In short, what I'd like is an assembler, and that is what my next programming project will be about. ■