The IBM Engineering Verification Engine

Daniel K. Beece*, George Deibert**, Georgina Papp*, Frank Villante***

* IBM Watson Research Center, PO Box 218, Yorktown Heights, NY 10598
** IBM Data Systems Division, P.O. Box 390, Poughkeepsie, NY 12602
*** IBM System Technology Division, 1701 North Street, Endicott, NY 13760

Abstract

In this review article, we describe IBM's Engineering Verification Engine, EVE, a special-purpose, highly-parallel programmable machine for the simulation of computer logic and currently being used at several locations within IBM. EVE, which is based on the architecture of the Yorktown Simulation Engine, can simulate two million gates at a speed of more than two billion gate evaluations per second, far beyond the capabilities of existing software simulators. In this paper, we give an overview of the EVE architecture, hardware and software, and describe some current applications.

(1) Introduction

The Engineering Verification Engine, EVE, represents the current state of the art in hardware attach processors. EVE hardware was designed and fabricated by IBM's System Technology Division, Endicott. Software support for EVE is provided by IBM's Engineering Design Systems in the General Technology Division, Poughkeepsie.

EVE is based on an architecture and prototype hardware and software developed in the IBM Research Division, known as the Yorktown Simulation Engine, YSE[*1,2,3*]. EVE was developed to bring YSE simulation technology to a production level for a wide range of users within IBM's development laboratories. While the basic architecture of the EVE and YSE are similar, there are differences between the machines (summarized in Section 9).

EVE is a combination of software and special purpose hardware designed for high speed and large capacity simulation. Simulation is performed by partitioning the design onto a number of processors which execute in parallel. There are two basic processor types on EVE, logic processors, LPs, and array processors, APs. EVE logic processors are architecturally similar to the YSE LPs and are described in Section 2. EVE array processors are constructed out of three sub-processors, as described in Section 3. Each LP and AP executes its own program in lock step with all the other processors; like the YSE, interprocessor communication is done via a statically scheduled cross-point switch (Section 4). All user interaction (load, test case drivers) with EVE is mediated by a special input-output processor, IOP, which also serves as the EVE controller (Section 5). Users interact with EVE via the IOP by running programs on a host System/370; the IOP is attached to the host via a System/370 channel.

The EVE switch is a full 256-to-256 cross-point. Each logic processor can simulate up to 8192 4-input gates and occupies 1 switch slot. Each array processor can simulate up to 12.4 Mbytes of storage and occupies 9 switch slots. The maximum configuration of EVE offered consists of 220 LPs and 4 APs, a total of 1.8 million gates and 50 Mbytes of array storage.

EVE is programmed by host software (Section 6) that converts a user's design data into the formats required to run on EVE, automatically partitioning it onto the LPs and APs and scheduling the switch. Several internal IBM design languages are supported by the EVE software, including structural descriptions and the register transfer level language BDL/CS[*2,4*].

Run time interaction with EVE is mediated by the IOP. In addition to basic simulation control, *e.g.*, changing data values (gate inputs and/or array contents), EVE software supports the automatic translation of internal IBM test case languages into EVE models, *i.e., virtual logic* (Section 7), which is merged with the real logic being simulated and is then executed directly on EVE.

In the following sections, we describe each of these EVE features in more detail. We also briefly summarize applications (Section 7) and performance considerations (Section 8).

(2) Logic Processors

The architecture of the EVE logic processors is similar to that of the YSE[*1*], so only a brief summary will be provided here.

EVE logic processors simulate arbitrary 4-input logic functions on two-bit data operands (four-valued logic, *e.g.,* 0, 1, undefined, high-impedance). Each LP operates in parallel with the other logic and array processors, executing its own unique program of instructions, where each gate is represented as a single logic processor instruction.

LPs have a data memory, which is subdivided into sections for local and switch input, and for rank order/unit delay simulation. Instructions, which describe the gates to be evaluated, are stored in an instruction memory on each LP. During execution, each LP obtains an instruction from its instruction memory, fetches four operands from its data memory, transforms each operand using what is called a *De Morgan translation* (for example, do nothing, complement, force to true), computes the function output via table look-up, and stores the result back into the data memory after a final translation. There are no conditionals or branch instructions.

Rank order simulation is done by sequencing the gates so that each gate is evaluated only after all of its predecessor gates have been evaluated. This ordering requirement prohibits feed backs in the logic, making it impossible to simulate memory. The advantage of rank ordered simulation over unit delay, which is described next, is that a single pass (simulation cycle) over the logic performs a complete evaluation, *i.e.,* a network of any depth is stabilized in one cycle.

Unit delay simulation is done by dividing the data memory into two halves, called "A" and "B". Each half holds the entire state of the model. During simulation, each successive cycle alternates fetches and stores between the two half-sections. That is, during even-numbered cycles, operand fetch is from the "A" section and result store is to the "B" section; during odd-numbered cycles, operand fetch is from "B" and store is to "A". The net effect is that each simulation cycle performs a single gate delay for every gate in the entire machine, so that a network N levels deep will require N cycles to stabilize. The order of the gates being evaluated has no effect, and memory can be simulated.

Most models are simulated on EVE by using a mixture of rank order and unit delay evaluations. This allows memory elements to be simulated (unit delay) while combinatorial logic between memory is simulated in one pass (rank order). Mixed mode simulation is accomplished by running in unit delay mode, but alternating the source addresses for rank order gates so that they always read from the "wrong" memory, *i.e.,* fetch and store to the "A" half section during even-numbered cycles and to the "B" half section during odd-numbered cycles.

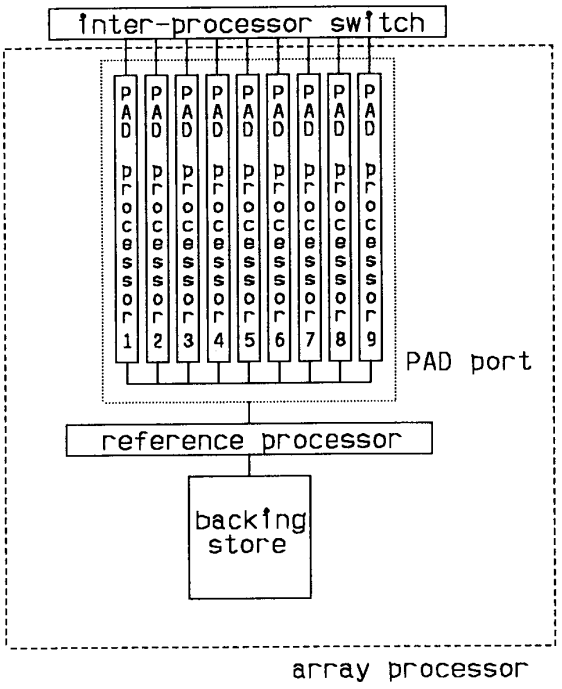
Functions on EVE are described via an index into a function memory, which selects one out of thirty-two different possible tables. In addition, each operand can be translated via the DeMorgan operations after fetch, but before function evaluation (for inputs) and after evaluation but before store (for output). Each instruction can also generate an interrupt when the result of an evaluation matches a predefined value.

During an instruction cycle, each LP sends and receives one value from the switch. The values received by the switch are stored in A/B mode in a unique location; the address of the operand to be send to the switch is coded as a field in each instruction. LP data memory is therefore divided into two parts twice, *i.e.,* quarters. The first division implements the A/B memory scheme necessary for unit delay simulation. The second division is into local memory, meaning the values generated by gates local to the LP, and switch input memory, for values received by the switch. Each operand address in the LP can access a value in any of the four quarters. Thus, since each EVE processor has 8192 instructions, each LP has 32768 data memory locations. Note that the entire data memory has five read ports (four function operands and one switch read operand).

(3) Array Processors

Array processing represents the most significant architectural hardware difference between the YSE and EVE. While partially conceptualized[*1*] no array processor for the YSE was ever designed; instead, RAM in the YSE's IOP was used to simulate memory. For EVE, a special array processor was architected and designed in IBM's System Technology Division, Endicott.

Each EVE array processor consists of 9 PAD processors, a reference processor and backing store, as shown in the following figure:



The PAD processors are similar to the logic processors, acting as an interface and buffer with the switch. These PAD processors work collectively as a unit called a *PAD port*. The reference processor sends and receives values with the PAD port, generating the physical array reference from the simulation values. The backing store provides the bulk storage that holds the simulated arrays.

The PAD and reference processor both have direct connections to the input-output processor. The backing store is only accessible through the reference processor.

(3.1) Parallel Device Adapter -- PAD: The PAD provides the interface between the array processor and the EVE switch. It buffers simulation values, such as address and data lines, between the switch and the reference processor, and provides additional information for simulated array references.

The PAD consists of nine processors, each occupying one switch slot. Each PAD processor is similar to a logic processor. During an instruction cycle, each processor fetches five operands from the switch input half of its data memory, which like the data memory in the LPs, is written in A/B mode from the switch. After fetch, each operand undergoes a De Morgan translation and is then sent to the reference processor. Simultaneously, two operands are received from the reference processor, translated and then stored in the local half of the PAD data memory which *does not* operate in A/B mode, unlike the results of the function evaluation stored in the local half of the data memory in an LP. On a PAD processor, this half of the data memory is called the *switch output* memory.

While each PAD processor is a separate entity, they operate collectively, *i.e.*, the different PAD processors are loaded independently but all control functions must be programmed in a coordinated way, so that during simulation the nine PAD processors act a single unit, called a *PAD port*. Thus, every instruction cycle, each PAD port sends a total $9 \times 5 = 45$ two-bit values to the reference processor and receives $9 \times 2 = 18$ two-bit values. When communicating with the switch, the PAD functions like nine logic processors, sending and receiving nine two-bit values.

Since only dynamically changing, data dependent information (address, data, *etc.*) are sent from the LPs to the PAD, one of the nine PAD processors contains additional fields, called *PAD control words*, in its instructions for data to be sent to the reference processor every instruction cycle. This static, data independent information tells the reference processor:

1. whether the access is a valid read, a valid write, or no-operation
2. which simulation array should be accessed
3. if multiple bundles (defined below) are needed, an identifier for the bundle being sent

(3.2) Reference Processor: As signals that model the interface circuits to an array, *e.g.*, address and data lines, are generated in the logic processors, they are sent out to the switch and collected by the appropriate PAD ports. When all the signals required to define the array access are collected, the data is sent to the reference processor, which then maps the simulation values into physical accesses in the backing store. If the access was a read, then the reference processor will send data back to the PAD, to be stored in the appropriate switch out memories, eventually to be sent back to the LPs. Note that given the finite band-width between the LPs and PAD, PAD and reference processor, and reference processor to backing store, it may be necessary to break a wide array into parts, called *bundles*.

The reference processor translates the simulated address values derived from the LPs and buffered by

the PAD into physical addresses in the backing store. The reference processor also provides packing, unpacking and mask functions for the backing store data access. Address translation, data value conversion and backing store access are based on the simulation values which originate on the LPs and the control word fields from the PAD.

The reference processor contains descriptions of each simulated array in the backing store in a memory called the *array descriptor table*, ADT. The ADT contains valid limits for the simulation address, the packing mode for the array (see below), the offset of the array in the backing store and, if multiple bundles are needed to access the array, how the information is divided across multiple bundles.

Because of the address translation and data conversion provided by the reference processor, it is the only means of access into an array processor's backing store. It is used not only during simulation to mediate transfers between the array processor and the LPs but also during load and unload transfers with the input-output processor.

(3.3) Backing Store: The backing store provides the actual data storage for the simulation arrays, organized as a fixed number of 36 bit words. Data can be stored in either *bit* or *peck* packing. For bit packing, only the low order bit of the two-bit simulation value is stored, so the amount of data the backing store can hold is essentially doubled, compared to peck packing. Bit packing is used any time it is known that a particular simulated array will always contain known values, such as when doing two-valued simulation (Section 7). For bit packing, the high order bit for each data word sent by the PAD to the reference processor is discarded during the write access and padded with "0" during the read access. For peck packing, both bits of the two-bit simulation value are stored in the backing store.

(4) Switch

The EVE switch consists of 256 identical switch ports. The logic processor connects to the switch via one switch port. The array processors communicate with the switch via nine switch ports, one for each PAD processor. Each switch port consists of a 256-to-1 multiplexer and memory to control the multiplier, called *switch memory*. Each port's switch memory feeds its multiplexer's address inputs and the switch outputs of all processors (PAD and LP) feed every multiplexer's data inputs. Each port's multiplexer output feeds the input memory of it's associated processor. Note that processors do not have to send the result of each instruction to the switch immediately after it is computed, as there is an address in each processor instruction which selects a data memory value to send to the switch. This allows a value to be sent any time after it is computed, or even multiple times, if necessary.

The switch is used as follows. All processors operate in lock-step, using a common, global clock, *i.e.*, all processors execute instructions $1 - 2 - 3 \ldots N$. As each processor performs an instruction, one data item is sent from the data memory to the processor's switch output port (multiplexer input), so that during each instruction cycle, each multiplexer has received a data item from all processors. At the same time, the switch memory tells each multiplexer which input line to select, *i.e.*, which of the possible 256 inputs should be received by this port. Note that on any given instruction cycle, any subset of values that are sent *to* the switch can be *received* by *any* subset of processors, provided that a given processor can receive only one value.

(5) Input - Output Processor

The input-output processor provides the means of communication between EVE and it's host. It consists of a host channel *adapter interface*, an EVE *bus controller*, a high-speed *direct memory access*, DMA, a microprocessor, Motorolla MC68000, and microprocessor RAM, interconnected on a *Versa-bus*.

The adapter interface connects the Versa-bus to the host's channel. It is the main communication link with the host, and is capable of transferring single words (16 bits) between the Versa-bus and the host in either direction. It also interprets commands from the host that activate other features of the IOP, *e.g.*, power-on-reset.

The DMA serves to transfer data between host and Versa-bus. Because of the overhead involved in setting up a DMA transfer, non-DMA transfers are also supported.

The bus controller is attached to the Versa-bus, and via an internal channel to all of the LPs, APs and switch. All data transferred between the input-output processor and any part of EVE must go through the bus controller.

The MC68000 handles interrupts from the EVE components (LPs, APs and switch) via the bus controller and from the host via the adapter interface. Interrupts from EVE are analyzed to determine which device generated the interrupt and in turn generate an appropriate interrupt to the host along with information about

the device and instruction that caused the interrupt and the type of interrupt.

The MC68000 starts and controls the EVE simulation cycles, *e.g.*, it starts the EVE by loading the number of cycles to simulate into a register in the bus controller, enables the appropriate interrupts, and then issues a *start* command to the EVE. While simulation is in progress, the MC68000 waits for either a host or EVE interrupt and is available for other purposes. For example, the IOP can store data values it collects from the EVE in IOP RAM, so that it can overlap the host I/O with EVE simulation, increasing the utilization of the engine.

It is possible for a user to write special routines for execution in the MC68000, for example, model specific interrupt handlers to help in the collection of data and/or test case interaction.

(6) Software Overview

The software support for EVE provides support for design and stimulus data conversion, model build, run time access and simulation control, and results processing. Many of the algorithms used in the EVE software are similar to the ones used for the YSE[2].

(6.1) Design Data Conversion: Automatic translation of internal IBM design languages, including structural representations and the register transfer language BDL/CS, are supported.

BDL/CS design translation is a two step process, a translation to a gate level description and then optimization of this description. The optimizer is multi-pass heuristic logic optimizer. It applies simple logic reductions that operate on the gate level logic to reduce the total number of gates. The reductions are applied repetitively; each application of all the reductions constitutes one "pass". It terminates when the reductions can produce no further results, or when the user-defined limit on the number of passes is reached.

The result of all the design translators is an intermediate representation called *NODES* that constitutes the basic language interface for the EVE compiler. NODES descriptions are a simple structural representation of the logic to be simulated. Connections are implicitly specified by nets, which must be either the output of a gate or an input to the model. If the logic has nets which have multiple sources, like a "wired OR", gate(s) must be used to represent this logic function.

(6.2) Compilation and Linking: The EVE *compiler* reads and processes NODES file(s). Its function is to partition the logic into a number of processors and generate the instruction loads for the LPs, APs and switch. The output of the compiler is called simply *a load module*.

Partitioning is based on a rank ordering of the gates. Rank ordering is performed by tracing the fast predecessors gate, beginning with a gate having no fast successors. This process also detects feedback loops within rank-ordered logic networks and signals an error condition.

There are two logic processor partitioning algorithms that can be used, both of which attempt to minimize the maximum depth of instructions across a number of processors. The first algorithm traces back from model outputs, attempting to place both a gate and its predecessors in the same processor. The second algorithm uses an annealing heuristic that tries to distribute the instructions equally among the available logic processors yet minimizing the number of processor interconnections. In this annealing process, the instructions are moved from one processor to another, scored according to an objective function, and then the move is accepted or not based on a time dependent probability distribution.

There are also two algorithms used for partitioning arrays among the APs. The first algorithm attempts to balance the number of signals required to simulate the arrays among the number of available APs. These signals are required to provide data, addresses, write masks, and enables for each of the arrays. By balancing these signals, which must be transmitted via the switch, bottlenecks are prevented and scheduling proceeds smoothly. In partitioning using this algorithm, the amount of storage used in the backing store is checked to prevent assignments requiring more space than is available. The second partitioning algorithm attempts to minimize the number of APs utilized. This algorithm only makes an assignment to a new array processor if there is not enough space available in the backing store of the array processor that have already been allocated for use.

After partitioning, the compiler must program the switch to receive and transmit inter-processor data at the appropriate times. Depending on the delay type (rank order or unit delay) specified, instructions may have to be scheduled to send or receive data from logical predecessor instructions in one (rank order) or several (unit delay) simulation cycles.

Since most models are developed in stages, a link facility is provided to combine outputs of several compilations. The EVE *linker* takes as input one or more load modules and connects gate inputs and outputs across modules, resolving inter-module scheduling and combines the PAD processor instructions. The output of the linker can be used as input to itself for further linking. The linker is also used to connect virtual logic representing simulation control functions (Section 7).

(6.3) Access Control and Run Time Interaction: Since EVE is a dedicated resource which must often be shared among several designers and/or projects, software is provided to give access, based on priorities which have been assigned.

Once the EVE hardware has been assigned to a specific job, the run time system provides the user interface which can be used either interactively or in batch mode.

Many functions are supported by the run time system, including

- loading the processor instructions
- loading test cases into arrays
- checkpointing and restarting a simulation job
- starting and stopping simulation
- setting and forcing net and array values
- displaying net and array values
- setting break point interrupts to stop simulation under specified conditions
- unloading net and array values to the host

(6.4) Results Processing: The user is able to specify the format of results gathered, *i.e.,* all events, (last) N cycles, selected nets on selected cycles only, *etc.* Results are presented as timing charts, with the ability to scroll (to view many net charts), and to window (to view many cycles).

(7) Applications

Because of model build and run time restrictions, most EVE jobs are large models running a large number of cycles.

In general, simulation is usually done in one of three timing models, zero delay, unit delay and nominal delay. In zero delay, combinatorial logic is modeled as zero delay gates, with special unit-delay primitives used to model latches. This *cyclic design style* timing model is often used with register transfer languages, such as BDL/CS. Unit delay is used for more *asynchronous* simulation. In this case, all logic is modeled in terms of unit delay gates, with latches often represented in terms of the gates used to build the latch, *e.g.,* NANDs. The most general case is nominal delay logic simulation, where more complex timing models are used, *e.g.,* variable rise and fall delays on gate outputs.

EVE is designed to support models which mix zero delay and unit delay logic. Zero delay combinatorial gates with special latch primitives are ideally suited for EVE, but mixed mode simulation is often used for unit delay as well, since gates which are modeled in unit delay may expand to a mixture of zero and unit delay EVE primitives, *e.g.,* a 6-input, 2-output block. In the simulation of gate-level equivalent descriptions of a register transfer language like BDL/CS, the fraction of gates which *need* to be evaluated every cycle is typically quite high, often more than 10%, so that the exhaustive simulation paradigm used on EVE is well suited to support the high amount of parallelism. Even for relatively simple asynchronous simulation using a unit delay model, where the fraction of gates which need to be simulated is less, an exhaustive paradigm on a highly-parallel machine like EVE is still significantly faster than many other approaches, particularly for large models.

(7.1) Test Cases: As is true for simulation in general, a simulation job on EVE is more than simply evaluating gates and arrays. For example, initial conditions must be established, test signals must be applied, and the model's response must be monitored and recorded. For a special purpose engine like EVE, such tasks can be done from the host using the run time system, but if the number of interactions with host are extensive, it can be a severely limiting factor to the performance of the simulator.

Since EVE is used for the simulation of large models, it is often the case that the model can be totally self-stimulating. Thus, one of the most widely used EVE applications is the testing and debug of large designs, such as processors and controllers, using actual software programs, either microcode, diagnostics or even architectural, *e.g.,* assembly language, programs on what are essentially complete systems. For this application, the host and IOP are used only as test case delivery and results gathering systems, and the amount of run time interaction after model initialization is minimal.

(7.2) Virtual Logic: Another way to reduce the run time interaction yet support more interactive test case drivers is to convert a test case description into "virtual" logic, *i.e.,* logic which is not part of the user's design, yet is connected with the model and exercised simultaneously with it. The use of virtual logic allows testing to become part of the simulation rather than a run time interaction with the model.

EVE model build software supports the conversion of an internal IBM test case language into virtual logic. Simulation is performed on the combined logic, as if it was one model. In reality, part of the logic is the logic under test, and the remaining logic is providing test inputs, response monitoring, and data collection. The use of virtual logic to implement testing ensures that overall simulation performance remains high.

(8) Performance and Capacity

On EVE, the instruction cycle time is 100 nanoseconds, so that the peak performance is 220 gate evaluations every 100 nanoseconds, 2.2 billion gate evaluations per second. For a maximally loaded EVE machine, 220×8192 gates, the *peak performance at maximum depth* is $8192 \times 100 = 820$ microseconds.

Realistically, 100% capacity is not achieved. But, depending upon the care taken during the model build phase, better than 95% can be achieved, with 80%-90% typical. It is usually easier to achieve high processor utilization for models with a large amount of unit delay logic, as this reduces the number of constraints that must be honored during model build.

Actual EVE performance depends upon three factors. First is the depth of the instructions in the processors. Suppose the maximum depth across all processors for a given model is N. The instruction cycle time is denoted by $T = 100$ nanoseconds. Therefore, the time for one *physical* simulation cycle or *pass* is $N \times T$. Multiple passes may be required to fully simulate a cycle on the model, due to imperfect scheduling (rank-order) or to insure stabilization of the logic between latches (unit-delay). If the number of passes is P, then the time for one *logical* simulation cycle is $N \times T \times P$. This assumes, of course, that EVE is simulating all the time, *i.e.,* the *utilization*, denoted by U, of the machine, which is the fraction of time spent simulating (as opposed to input-output for test case interaction) is 1.0. In practice, $0 < U < 1.0$, and depends upon the application. Thus the amount of time taken by EVE to simulate each logical cycle is $(N \times T \times P)/U$.

Most users try to run with the processors packed as deeply as possible, so that typically $N \simeq 8000$. The number of passes varies considerably, depending upon the application and the amount of time which can be devoted to model build; usually, $1 < P < 20$, with a typical value of $P = 5$. The utilization is also heavily application dependent. For jobs which rely on tight coupling with the host, the utilization depends upon the host load, varying from $.20 < U < .85$. For jobs which use virtual logic, or which otherwise consist of totally self-driving models (Section 7), utilization can be quite high, $U > .95$.

(9) EVE and YSE Differences

The YSE was a research project that developed prototype hardware and software to prove the viability of the concepts embodied in an architecture for a special purpose hardware engine. EVE is a production machine based on that architecture, but expanded and enhanced to meet the needs of real design projects.

Architecturally, as noted above, the most significant difference is the EVE array processor, which was conceptualized for YSE, but not fully specified. Other minor differences exist as well, such as maximum processor depth, 4196 for the YSE, 8192 for EVE, and instruction cycle time, 80 nanoseconds for the YSE, 100 nanoseconds for EVE.

Hardware for the machines are completely different. For example, the YSE was built out of non-IBM technology, using off the shelf components, mostly high speed TTL shift registers and RAM chips, while EVE employs several special gate array chips for improved density. The IOPs for the EVE and YSE are also completely different.

References

[1] Denneau, Monty M., "The Yorktown Simulation Engine," *ACM IEEE* 19th *Design Automation Conference Proceedings*, 1982.

[2] Kronstadt, E., and Pfister, G., "Software Support for the Yorktown Simulation Engine," *ACM IEEE* 19th *Design Automation Conference Proceedings*, 1982.

[3] Pfister, G., "The Yorktown Simulation Engine: Introduction," *ACM IEEE* 19th *Design Automation Conference Proceedings*, 1982.

[4] Dunn, L., "IBM's Engineering Design System Support for VLSI Design and Verification," *Design and Test*, vol. 1, no. 1, February, 1983.