Hardware Acceleration of Logic Simulation using a Data Flow Micro Architecture

Gary Catlin - Bill Paseman

Daisy Systems Corp. - 700 Middlefield Road, Mountain View Ca.

Abstract

Current digital logic simulators running on engineering workstations lack capacity and speed. This paper discusses a hardware accelerator for a workstation simulator which addresses these problems. The accelerator runs 1ØØx faster than its software counterpart and can simulate up to 1 million gates. The accelerator has been built and is being sold commercially. The architecture of the accelerator is similar to that of a classical dataflow machine. We describe the architecture of the machine and illustrate how it would simulate a simple circuit. We then briefly discuss the relationship between event driven simulation and dataflow.

1.Ø Introduction

Logic Simulation capability is one of the key selection criteria for people evaluating workstations. Workstation based simulators provide a number of advantages over simulation alternatives on other machines. They run on the design engineer's stand alone station, providing results quickly without competing for mainframe resources. They usually exceed the modeling capabilities of most home grown simulators(12 state modeling, MOS, Bidirectionals, Functional modeling). They are integrated with the schematic capture facility, so that with a small number of steps, the user is able to go from schematic editing to simulation. However, there are two important areas of concern where workstation simulators fall short. This paper describes a product, the Daisy Megalogician, which addresses these areas.

1.1 Capacity

The first is capacity. Usually, only circuits with a few thousand simple elements can be successfully run on an engineering workstation. Simulators with this size limitation are useful for checking out pieces of a design, but they are only capable of completely simulating the smallest of IC and board designs. This problem can be partially overcome by adding extra memory to the system. This approach will allow users to perform simulations on most large ICs and small systems. Designs in this range include 16-bit VLSI microprocessors which are in the 1Ø-15K gate range, and a typical Multibus board with 15Ø TTL MSI IC's, which might require 2-3K primitives to describe.

However, there is a growing class of users who require the capability to simulate 1ØØk or more gates. Typically, these users are developing large computer systems. One can argue that behavioral models can be used to represent all of the blocks in the design which are not being immediately debugged, with a simulation vector "capture/compare" facility to verify consistency between the behavioral and gate level descriptions. However, even users who accept this method for debug-level simulations express a desire to run the complete system at the gate level at least once prior to commitment to physical implementation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Also, when one studies the growth of the size of typical IC and system level designs, one realizes that many engineers will be developing circuits in the 1ØØk - 5ØØk gate range in a few years. These engineers will also wish to simulate at the gate level at least some of the time. Therefore the class of users which require 1ØØk gate simulations will grow as well.

1.2 Performance

The second major concern is performance. When workstations are stocked with enough memory to completely solve all size problems, then simulation speed becomes a problem. Simulation speed is measured by how much real time elapses from the specification of the required simulation until results are presented in an analyzable format. Since logic design and debug is an iterative process, a fair measurement is found by seeing how long it takes to make a small change in a circuit, recompile and resimulate. For a 1ØK gate design the time breakdown might be 1 hour for compilation and 1 hour for simulation. However, for a 64K gate design, the time might be 6 hours for compilation and 1 day to simulate a test case. A whole set of diagnostics might take six monthes to simulate on such a design. Obviously, the engineer would have to be content with a piece by piece simulation, and perhaps a sampling of test patterns running on the complete system. He might build a breadboard version of the design to perform logic and timing checks that aren't possible on his simulator simply because it is too slow. Fault simulation is another area where simulation time is a limitation. Even for concurrent simulators, medium circuits require several hours to grade and medium large circuits must run overnight. Again, the job is hardly interactive.

Clearly, providing a significantly faster simulator allows:

(1) a speedup of 2-1Ø in the logic debug iteration cycle for large and very large designs,
(2) the capability of making fault simulation much faster, and
(3) the capability of moving system level debugging and diagnostic development from breadboards to the engineer's desktop.

This paper is organized into 9 main sections. After the introduction we will discuss alternative methods of meeting the time/speed constraint. We will then discuss the nature of the problem we are trying to solve in more detail. We will then discuss the design constraints on the architecture. We will follow this with a description of the system architecture, plus an example simulation. Particulars of the unit architecture will then be discussed. We will then discuss some product timing data. This will be followed by a section on how event driven simulation relates to data flow.

2.Ø Previous Solutions

Many approaches have been taken to increase logic simulation speed and capacity. Perhaps the simplest approach is the assembly language coding of the time critical parts of the algorithm. Unfortunately, even coupled with tricks such as loop unrolling, data structure reorganization, vectorization and branch reduction, this approach seldom gives more than a 3x speedup[Krohn81].

With the availablity of a higher performance microprocessor to execute the simulator engine, a further speed enhancement can be realized. For example, an 8 Mhz, 286 based "background" processor with high speed memory could perform 2-3 times as

© 1985 ACM 0-89791-172-5/85/0012/0117$00.75

fast as a simulator running on a workstation. This speedup is realized primarily by running the processor with faster memory. Cutting the number of wait states in a 286 architecture from 3 to Ø could result in a 2.5x speedup. Coupled with assembly language, this could give a 6-9x speedup.

Another approach is to develop special hardware that is exactly tailored to the problem. Pfister [Pfister82] reports on IBM's YSE machine which, although not event driven, provides the fastest raw evaluation speed of any existing simulator. It can contain up to 4 million gates and can evaluate 96Ø million gates/sec. Sasaki [Sasaki83] reports on NEC's HAL which is an IBM class machine that is event driven. It can contain up to 3 million gates and can evaluate 36Ø million gates/sec. There are other paper designs which capitalize on most of the parallelism inherent in the event driven simulation algorithm [Abramovici83], [Glazier84].

Although Special Hardware developed exclusively for simulation provides the greatest performance increase, it is usually difficult to modify in the field and cannot be used for anything else. On the other hand, microprocessor based systems do not have the necessary power.

In this paper we will present a special purpose attached processor, the Megalogician, which is especially well suited for running event driven simulation algorithms, although it is not limited to running only algorithms of this type. In addition, due to its microcoded nature, it provides a good balance of flexibility and raw power.

3.Ø The Nature of the Problem

In order to understand the megalogician's functionality constraints, we will briefly review aspects of the problem that it solves.

3.1 Overview of Modeling Levels

Simulators model a circuit's behaviour at various levels of abstraction(Table I).
The user expresses the details of his circuit at any of these levels, or with a mixture of abstractions, each at a different level. These levels range from the Analog level, where a node's state is modeled using real numbers (which represent currents and voltages), to the system level, where a node's state is modeled using boolean values (which represent boolean values). In modeling a particular circuit, modeling accuracy increases as one travels down the table and simulation speed increases as one travels up. Each modeling level distinguishes itself by its language of discourse. Each language has a type of data with which it deals, a medium in which it is expressed, a set of primitive expressions as a basis, a means of combining the primitive expressions into complex expressions, and a means of abstracting away from the complexity so that the abstraction seems itself to be a primitive expression. We will now briefly discuss Data Types and the different characteristics of the various modeling levels.

3.2 Data Types

At the Behavioral Level, a node's state could be represented using two valued logic. However, an extra state, "unknown" is useful for representing cases occuring at power-on, where an internal circuit node is either logic 1 or Ø, but it cannot be determined which.

At lower levels of abstraction, it is important to simulate tristate and open collector gates, which create implicit "wired-or" gates when wired together. In order to simulate various wired-or situations accurately, it is useful to introduce the concept of "strength". Strength is used to signify the driving capability of the node. If the node is actively driven to its current level, the strength is said to be forcing. If the node is pulled high or low to its current level through a resistor, the strength is said to be resistive. If the node is high or low due to the presence or absence of a capacitive charge, the strength is said to be high impedence. If a node's strength is indeterminate, it is said to be unknown. A table illustrating all possible level/strength combinations is shown below.

```
strength
 | +----+----+----+
 V |  Ø |  1 |  U |<-level
+--+----+----+----+
|F | FØ | F1 | FU |
+--+----+----+----+
|R | RØ | R1 | RU |
+--+----+----+----+
|Z | ZØ | Z1 | ZU |
+--+----+----+----+
|U | UØ | U1 | UU |
+--+----+----+----+
```

The example below shows a case where many of these strength level combinations are used. The circuit consists of two unidirectional transfer gates which drive a common inverter. Initially, the inverter's input is driven to an R1 by the upper gate. Next, the upper gate switches off, and the inverter's input becomes Z1. (If left in this state for a long enough period of time, the node will decay to a ZU.) Finally, the upper gate switches to an unknown state. If this state is "on", then the inverter's input should be R1. If it is "off" then the inverter's input should still be Z1. Although we cannot determine what strength the node will be, we

Table I. The Relationship Between Multiple Modeling Levels

Level	Data Type	Medium	Primitive Expressions	Means of Combination	Means of Abstraction	PMX Model
Behavioral	2,12 state	textual	"Behavioural" (Note1), "Gate"	Through syntax O1:=~I1|I2;	Procedure, Function	CPU,ALU (8Ø86)
Functional	12 state	textual	"Gate"	Through syntax O1=(Not I1)Or I2;	Macro (Note3)	Counter (74161)
Gate	12 state	graphic	"Gate" (see Note 2)	Through graphical connection	"Blocks" (Note4)	Nand (74ØØ)
Switch	"1ØØ" state	graphic	Bidirectional Transistor	Through graphical connection	"Blocks"	Mux (4Ø16)
Analog	continuous					

Note 1
Behavioural expressions include: Arithmetical, Shift, Bit Reduction, Logical and Relational operators as well as Conditional, looping, sequential, and parallel control constructs.

Note 2
Gate types include Input, Output, Delay, Logic(such as nand,nor), Tristate, Unidirectional, Ram, Rom, Pla, Latch, Flip-Flop, PMX, Setup_and_hold_check, Signal_relationship_check and Minimum_pulse_width_check.

Note 3
Macros cannot be combined at the functional level.

Note 4
Blocks are graphical "black boxes" with graphically defined bus and signal interfaces. Their contents can be examined by descending into them.

do know that the level will be 1. This means that we must model the state as being U1.

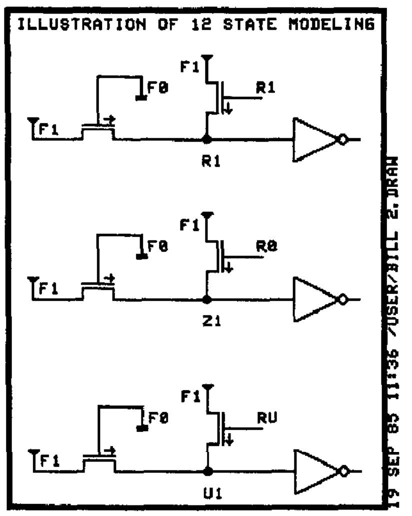


ILLUSTRATION OF 12 STATE MODELING
F1
F0
R1
F1
R1
F1
F0
R0
F1
Z1
F1
F0
RU
F1
U1
19 SEP 85 11:36 /USER/BILL 2.DRAW

At even lower levels of abstraction, 12 state modeling is not accurate enough. Switch level simulation uses 1ØØ's of values to model the state of wires attached to bidirectional transfer gates.

Though they differ in the number of states they have, all levels above the analog level can be described as modeling state using discrete values. This allows simulators to perform primitive evaluation at these levels using table lookup. The megalogician is optimized for this type of primitive evaluation. It currently does not do any type of modeling at the analog level.

3.3 Differences Between the Modeling Levels

At the gate and switch level, the circuit is expressed graphically, as shown in the figure above. Internally, the simulator represents the circuit at both these levels as a graph, where the nodes represent the primitives, and the arcs represent the wires between them. Again, the difference between these two levels is how state is modeled.

At the functional level, higher order primitives are modeled as a boolean combination of gates. This boolean combination is scheduled and evaluated as though it were a single gate. For example, an exclusive OR (Named XOR) with Delay N would be textually expressed as

```
XOR:EXPR<outputs:o1[N]; inputs:i1,i2>;
(o1=nand(nand(i1,not i2),nand(i2,not i1)));
```

where the delay for the entire operation is lumped into some single value N. This sequence of operations is described as a single expression. Whenever the simulator schedules this XOR function, the entire expression is evaluated at once, rather than as a number of individual events. This same function could have been defined at the gate level, with the difference that it would result in individual event scheduling for each of its primitive elements. Internally, the simulator represents functions as stack machine code, similar to pascal pcode. Again, the difference between the gate and functional level is the medium with which the concepts are expressed, and the fact that functional modeling allows coarser evaluation.

The behavioral level allows the user to create even more abstract descriptions of circuit elements using standard structured expressions such as IF THEN ELSE. It also allows the user to deal explicitly with time and event scheduling in his modeling. Constructs such as WHEN allow the user to sensitize an abstraction's inputs. So a designer may initiate a sequence of actions based on an expression such as WHEN CLOCK -> 1. Internally, the simulator represents behaviors as register machine code. Again the difference between the functional and behavioral level is that the user deals with a higher level of abstraction.

3.4 Physical Modeling

Sometimes it is not possible to create an accurate model of a circuit in a reasonable period of time, no matter what the level of abstraction. Microprocessor modeling is a good example of this. In these cases, the best model of the part is the part itself. The megalogician has a physical modeling extension (PMX) which allows the user to logically model a chip by plugging it into the megalogician[Stoll85]. This method of modeling cuts across all levels of abstraction, as shown in Table I.

4.Ø Megalogician Architectural Constraints

It was required that the megalogician support all the functionality described above. This meant supporting the interpretation of register machine code at the behavioral level, of stack machine code at the functional level and data dependency (data flow) graphs at the gate and switch levels. This required that the architecture be very flexible. Other requirements were that the architecture be able to support designs in the range of 1 million gates, and be able to run them 1ØØx faster than standard software simulators.

In order to meet the flexibility constraint, it was decided that the machine had to to be microcodeable. In order to meet the speed constraint, it was realized that a single processor architecture would not be fast enough. The question was then how to best partition the problem in order to parallelize it. There are two basic methods of parallelizing an algorithm. One is by data partitioning. The other is functional partitioning.

Data partitioning involves making several processors perform identical functions on different portions of the input data. This approach was rejected because it required complex interprocessor communication, and expensive processors.

Instead it was decided to exploit the structure of the simulation algorithm (i.e. use functional partitioning). It is possible to break down the algorithm into three pieces of approximately equal complexity. Each piece utilizes large data structures which it does not have to share with the other pieces. In addition the communication between the pieces is low bandwidth and simple compared to the data partitioning approach. We will now discuss the particulars of the implementation.

5.Ø The System Level Architecture

The system level architecture of the MegaLogician is diagrammed below.

```
..................................................................
.286 based workstation                                           .
.              +------+  +------+  +------+  +------+  +------+  .
.              |      |  |Sys.  |  |Disk  |  |Video |  |Key-  |  .
.              |8Ø286 |  |Mem.  |  |      |  |      |  |board |  .
.              |      |  |      |  |      |  |      |  |      |  .
.              +---#--+  +---#--+  +---#--+  +---#--+  +---#--+  .
.                  #         #         #         #         #     .
.                  #Multi-   # Bus     #         #         #     .
.              ##################################################.
...................#.........#.........#..........................
.Accelerator     +-#---+     #         #                         .
.                |E#a12|     #         #                         .
.    +------+  +---#--+|     #    +---#--+  +------+             .
.    |      |  |      ||     #    |      |  |      |             .
.    |Evall |==|Evall |---------->|Queue |==|Queue |             .
.    |Mem.  |  |Unit  |      #    |Unit  |  |Mem.  |             .
.    +------+  +------+      #    +------+  +------+             .
.                  |         #         |                         .
.                  |     +---#--+      |     +------+            .
.                  |     |      |      |     |      |            .
.                  +<----|State |<-----+     |State |            .
.                        |Unit  |============|Mem.  |            .
.                        +------+            +------+            .
..................................................................
```

A high performance 8Ø286 based engineering workstation serves as the nucleus of the MegaLogician, and is used for schematic preparation and compilation, and also supports the user interface during simulation. This system is interfaced to the special accelerator hardware via the workstation's Multibus.

The accelerator consists of four separate processing units, called the Queue, State, Eval-1 and Eval-2 units.

These four units are physically separate, i.e. each occupies a separate PC board.

The best way to understand the total architecture is to view the Queue, State and Evaluation units as being coroutines, communicating with each other through high speed fifo channels. Their memory spaces *are disjoint. They can only communicate through the* fifos. This fifo form of communication places restrictions on the efficiency of parallel processing among the units, since it makes it difficult for communication to occur among certain paths. For example, the SU can not conveniently query the QU for data. Rather, it is the responsibility of the QU to *provide the SU with all the data it needs to process* a particular command. The logic simulation algorithm we execute, however, lends itself to this approach. This approach offers performance advantages over others. A completely shared memory would eliminate the advantage of parallel processors, since all would become memory access limited. (This point has been brought up concerning hardware acceleration of certain AI paradigms such as Blackboard and Production systems.[Deering85] These systems, in their current forms, rely on memory for communication between tasks.) The fifo itself improves performance by smoothing out irregularities in the *"instantaneous" speeds of the units (their "average"* speeds must still be balanced). The lack of back-and-forth communication allows one unit to be performing tasks which are unrelated to what is occuring in the other units. By correctly partitioning the tasks and data, communication is minimized. In actual operation, all units process *simultaneously*, each recieving packets from behind, processing them and passing the results forward. Each unit handles a number of tasks associated with the contents of memory it contains.

5.1 The Queue Unit

The main data structure of the Queue unit is the event queue, which contains all output transitions which are scheduled to occur in the future as a result of current or past input transitions. The event queue is structured as a linked list of events. Associated with each event is:
1) a gate identifier
2) the simulation time at which the event is to occur, and
3) the new state(recall that state is defined as being a level/strength pair).
Routines exist which allow the queue unit to access the event queue by either time or gate identifier.

The queue unit's primary responsibility is to begin and halt the simulation. Once the circuit data has been loaded into the various units, actual simulation begins when the 8Ø286 instructs the Queue Unit to simulate the circuit for a certain amount of simulation time. The Queue Unit begins to "run" by incrementing time, and processing the events scheduled for that time.

5.2 The State Unit

These events are passed to the state unit, which enters them into a state array. The state array records the state at the current time step for all nodes in the circuit.

The state unit also contains the connectivity information for each element in the circuit. The connectivity is maintained in two lists, each list accessable via a gate identifier. The first list is called the fanin list, and lists all the gate identifiers that fanin to the given node. The second list is called the fanout list, and lists all the gate identifiers that fanout from the given node.

After the state unit has updated the state array for the particular time tick, the Queue unit again sends the events for that time tick to the state unit. For each of the events sent, the state unit discovers where the event fans out using the fanout list. For each of these gates, the state unit *bundles* together the gate identifier, gate type and the node's input states (using the fanin list and the state array) and sends them to the evaluation unit.

As can be seen, the state unit's primary responsibilities are to: enter current state transitions into the state array, *find all gates which need to be evaluated and their input states*, and to send these gates and their inputs to the evaluation units.

5.3 The Eval Units

The Eval-1 unit contains the functional models plus the behaviors for all simulation element types. The Eval Unit uses the behavior along with the list of input states to evaluate the correct output for the gate. It compares the evaluated output to the current output. If they differ, it passes the gate identifier and new output to the Queue Unit for scheduling. If the outputs are the same, it has the Queue Unit check for a spike on the node. The Queue Unit contains the rise and fall delays for each gate. It schedules an event by accessing the appropriate delay value, adding it to the current time, and entering a new event at the appropriate place in the event queue. It checks for a spike by seeing if any event is currently scheduled for that gate.

As can be seen, the primary responsibilities of the Eval1 unit are to: evaluate gates which have experienced an input transition and request an event to be scheduled if necessary. Eval-2 is used for the physical modeling of circuits.

By using this algorithm on a large circuit where it is likely that there are many events in a particular time "tick", it is possible to have all three processors performing useful work most of the time. The State Unit is collecting fan-in states, while the Evaluation Unit evaluates outputs and the Queue Unit schedules future events.

5.4 An Example Circuit

Figure I below illustrates a typical circuit. Figure II illustrates how the tables would be set up in the various units for this particular circuit.

The State Unit contains the state array and the Fanin and Fanout lists. Note that "A" and "B" have no fanin and that the fanout of "H" is not shown. "A" fans out to "C" and "D". In the example, we have consistently stripped off the strength part of the state representation. "A" has a state of logic 1 at the current time, which is 99 ns. "C" is the only *node to have a state of Ø.*

The delays for the various gates are kept in the queue unit. "C", "D", "E" and "G" have delays of 1Ø, 15, 12 and 18 ns respectively. The event pool contains a single event, which indicates that the state of "A" changes to logic Ø at time 1ØØ ns.

The eval unit simply contains the behavioral description of the various nodes.

Figure III illustrates the state update phase of the simulation algorithm at time 1ØØ ns. Here, the state unit updates the state of node "A" when it receives the appropriate Queue Unit packet.

Figure IV illustrates the second phase of the algorithm at time 1ØØ ns. 1) shows where the gate identifier portion of an event is propagated to the state unit. The state unit responds by propagating out instruction packets to be evaluated, 2) and 3). One packet is sent for each node to which "A" fans out (namely "C" and "D"). The eval unit evaluates the packets ands posts the results, 4) and 5), to the queue unit. Finally, the queue unit takes the results and discovers when to enqueue them by looking at its delay table. This happens in 6) and 7).

All these events occur simultaneously, that is while the Queue unit is processing events, the State unit is building instruction packets and the Eval unit is evaluating them.

The balancing of work among *various* units is based on some assumptions about the circuit being simulated. It is optimized for simple gate and boolean evaluation, assuming an average fan-out of 2.5. This means that there are 2.5 evaluations for each scheduling. For every 1ØØ clocks it takes to collect the fan-ins of the 2.5 gates affected by a state update, it takes 4Ø clocks to evaluate the output of each gate, and about 7Ø clocks to schedule an event and 15 to check for a spike.

Figure I: Circuit to be simulated

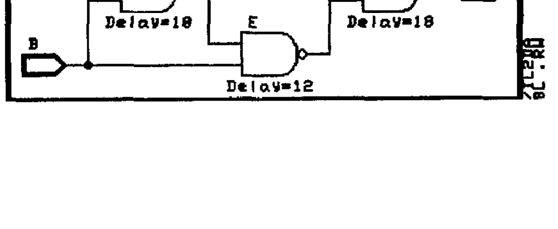


Figure II: Initial MegaLogician Configuration

```
          "and"  model  _
          "nand" model / \
          .           |   | Eval unit
          .           / \_/ \                +-+-+
                     /       \          1ØØ ->|A|Ø|
                    /         \              +-+-+
                   /           \             event
               _  /             \  _         pool
              / \/               \/ \
State unit   |   |<--------------|   | Queue unit
              \_/                 \_/

      Connectivity        State          Delay
         Table            Array          Table
--------+------+------++-----------+  --------+------+
Gate    | Fan  | Fan  || State     |  Gate    |Delay|
        | in   | out  ||(time 99)  |          |     |
--------+------+------++-----------+  --------+------+
  A     |  x   | C,D  ||    1      |    A     |  x  |
  B     |  x   | C,E  ||    1      |    B     |  x  |
  C     | A,B  | D,E  ||    Ø      |    C     | 1Ø  |
  D     | A,C  |  G   ||    1      |    D     | 15  |
  E     | B,C  |  G   ||    1      |    E     | 12  |
  G     | D,E  |  H   ||    1      |    G     | 18  |
  H     | G..  | ...  ||   ...     |    H     | ... |
```

(T denotes logic 1, F denotes logic Ø)

Figure III: State Update phase

```
      State              "and"  model  _
      Array              "nand" model / \
-------+----------+        .         |   | eval unit
Gate   |  State   |        .         / \_/ \
  (time 99->1ØØ)  |                 /       \
-------+----------+                /         \
  A    |   1 ->Ø  |            _  /           \  _
  B    |   1      |           / \/             \/ \
  C    |   Ø      |          |   |<-------------|   |
  D    |   1      |           \_/   +---+---+   \_/
  E    |   1      |          state <- | A | Ø | queue
  G    |   1      |          unit    +---+---+ unit
  H    |  ...     |                state update
                                     packet
```

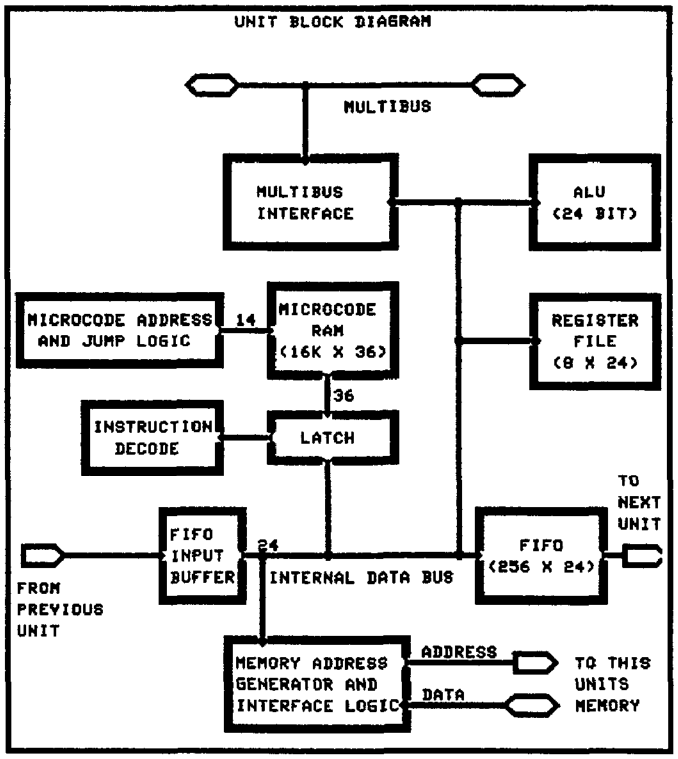
Figure IV: Event Propagation phase

```
                      eval  _
     +----+-+-+-+     unit / \      +-+-+
2)   |nand|C|Ø|1|         |   |  5) |D|1|
     +----+-+-+-+         / \_/ \   +-+-+
     +----+-+-+-+        /       \     +-+-+        +-+-+
3)   |nand|D|Ø|Ø|       /         \ 4) |C|1|  1ØØ->|x|x|
     +----+-+-+-+      /           \   +-+-+    |   +-+-+
                      /             \           |
instruction  _      /               \  _       V   +-+-+
packets     / \/                     \/ \  6) 11Ø->|C|1|
           |   |<-------------------|   |      |   +-+-+
            \_/        +---+         \_/       |
   State   state   <-  | A |        queue      V   +-+-+
   Array   unit        +---+        unit   7) 115->|D|1|
----+-----+          1) event                      +-+-+
Gate|State|             packet
----+-----+
 A  | Ø   |          (time = 1ØØ)
 B  | 1   |
 C  | Ø   |
 D  | 1   |
 E  | 1   |
 G  | 1   |
 H  |...  |
```

6.Ø The Unit Architecture

The Queue, State and Evall units are pipelined microcodable machines.

A block diagram for the processor card appears below. It consists of microcode storage, a microcode addressing mechanism, a register file, an ALU, a fifo buffer, a memory interface, a multibus interface, and a host of associated control logic.



Once the unit is running, the input port (in the multibus interface) is used to pass in commands and data. Like in most processors, a command is distinguished from data only in the manner in which it is interpreted. The processor is designed to appear to execute higher level commands than single microinstructions. A "command" in this sense is a sequence of microinstructions which perform a specific task. Within the simulation algorithm, these tasks can be such things as "schedule an event" or "evaluate a two-input NAND gate". A command is given to a unit through its fifo, or through the input port. This command is actually the starting microcode address of the sequence of microinstructions to be executed. Upon completion of the command, a new command is loaded. This is accomplished by a special microinstruction which performs the following function in hardware: 1) if the input port is full, transfer control to the address it contains, else 2) if the input fifo is not empty, take the next word from the fifo and transfer control to the address it contains, else 3) wait for condition 1 or condition 2 to occur. A command is likely to contain data. This data also comes from the fifo or input port. Each command must explicitly or implicitly specify the number of words of data to follow.

The ALU hardware is used mainly for decrementing loop counters and doing bit masking operations.

The machine used by each unit supports a fairly standard set of microcode operations. It differs from standard microcode machines in three ways. The first difference is that it has special primitives which allow structured access to its data on a microcode level. This allows for compact coherent coding of symbolic algorithms. The second difference is that it contains instructions tailored to the simulation algorithm requirements, such as special bit test and address calculation instructions. The third difference is that it contains a special set of instructions used for interunit communication. These fifo instructions support the high interunit communication bandwidth. They allow the units to communicate at register access speeds. This high bandwidth is essential to support the algorithm.

The communication is done through channels which connect the units together. The channels contain FIFOs to buffer temporary load imbalances between the units.

Eval2 is the special PMX processor. When Eval1 recieves a packet that it cannot process (such as one that involves evaluating a physical chip, such as the National 16ØØØ), it passes it to Eval2. Eval1 and Eval2 can process simultaneously. That is, given the appropriate mix of type E1 packets and type E2 packets, both processors can evaluate different nodes of a circuit graph at the same time.

7.Ø Measurement Data

Our benchmarks have shown the megalogician to be between 8Ø and 11Ø x as fast as our own software simulator. This is approximately 1ØØ,ØØØ evaluations/sec or 4Ø,ØØØ events/sec. Its event/second rate is greater on circuits with large gate counts than on circuits with small gate counts. This is due to the fact that larger circuits generally have more parallel activity going on per time tick, and that there is some overhead in starting and stopping a time tick. In one case, a 35,ØØØ gate design processed 5Ø% more events/sec than a 25Ø gate design.

The maximum activity measured was approximately 2 events/1ØØØ gate time-ticks. This occured on small circuits, larger circuits range from .13 to 1 event/1ØØØ gate time ticks. This means that event driven simulation does at least 5ØØx fewer evaluations than a non-event driven approach would.

In terms of a real benchmark, a single card computer design containing several PMX-modeled devices including an Intel 8Ø86, 8Ø89, 8288 and the Zilog Z8Ø SIO with a multibus card worth of software modeled MSI/SSI glue, ROM and RAM runs 1.3 msec of real time (1Ø,4ØØ cycles of the 8 MHz clock, or about 1ØØØ to 2ØØØ 8Ø86 instruction executions) in 17 minutes.

8.Ø Relation to Dataflow

Both logic simulators and dataflow machines are interested in evaluating networks of equations. In the data flow case, the network is called a data flow graph instead of a circuit. In order to understand the relationship between discrete simulation and dataflow, it is best to start with an example that compares the two.

We will start by looking at an example dataflow graph and see how it is processed.

```
"6"A  _                 A  _                 A  _
   \/ \                  \/ \ "3"             \/ \
   | - |\                | - |\               | - |\
   /\_/  \  _            /\_/  \  _           /\_/  \  _
"3"B      \/ \           B      \/ \          B      \/ \"3Ø"
    "7"E--| + |->          "7"E--| + |->          E--| + |->
"4"C  _   /\_/           C  _   /\_/          C  _   /\_/
   \/ \  /                \/ \  /              \/ \  /
   | * |/                 | * |/               | * |/
   /\_/                   /\_/  "2Ø"           /\_/
 "5"D                     D                    D

     (a)                    (b)                  (c)
 Example I
```

All nodes in dataflow graphs have a call by value semantics. This means that each node processes its inputs only after all of them appear. The example above calculates the expression 7+(A-B)+(C*D). In (a), the "-" and "*" nodes are free to fire, but the "+" node is not. This is because the "+" node has only one input, E, available. In (b), the "-" and "*" nodes have fired, so the "+" node is free to process its inputs. It then does so, producing the results shown in (c).

Let's now see how an event driven simulator would process the same graph. All nodes in event driven graphs have event driven semantics. This means that each node processes its inputs when any of them change. Let's assume in (a) that inputs A,B,C and D arrive at time Ø. Let's further assume that each node has a delay of 1. In (a), the "-" and "*" nodes are free to fire, but the "+" node is not. This is because the "+" node has no inputs that have changed. In (b), the "-" and "*" nodes have fired, and after a delay of 1, have arrived simultaneously at the input of the "+" node. The "+" node will now process its inputs since they have changed. This gives the result shown in (c).

On the face of it, this simulator example seems a little contrived. In particular, it gives the impression that the "-", "*" and "+" nodes fire only when both inputs are present. This is not true in general, since event driven semantics imply that these nodes fire when any input changes. It seems true due to the conditions stated above, which were that:

1) Inputs arrived to this functional block simultaneously and

2) Delays were added to the various components to align their processed results.

If we could somehow ensure that these conditions were always met for all functional blocks, then a dataflow graph evaluation would be indistinguishable from an event driven graph evaluation.

It can be shown that this can be done for a large number of cases[Paseman84].

The similarity between simulation and data flow extends to the data structures and algorithmic decomposition used in the MegaLogician and a token matching data flow computer. To better see the relationship, we have shown the MegaLogician architecture beside that of a typical token matching dataflow computer [Treleaven82].

MegaLogician Architecture

```
               +------+
               |Eval2|      --------
   +------+ +------+|    ( events )     +------+ +------+
   |      | |      ||     --------      |      | |      |
   |Eval1|=|Eval1|----------------->|Queue|=|Queue|
   |Mem   | |Unit  |                    |Unit  | |Mem   |
   +------+ +------+                    +------+ +------+
               |                            |
               |         +------+           |
               |         |      |           |
               +<--------|State|<-------+
   -------------         |Unit  |        ---------
  ( instruction )        +------+      ( events )
  (   packets   )           ||          ---------
   _____________         +------+
                         |      |
                         |State|
                         |Mem   |
                         +------+
```

Data Flow packet communication with token matching

```
                   -------------
          +------+ ( data tokens )   +------+
          |Proc.|  -------------     |Match|
          |Units|------------------->|Unit  |
          |P1-Pn|                    |      |
          +------+                   +------+
             |                          |
             |         +-------+        |
             |         |Fetch  |        |
             +<--------|Update|<--------+
  ------------         |Unit   |     ----------
 ( executable )        +-------+    ( sets of )
 (instructions)           ||        (  tokens )
  ____________         +------+      __________
                       |Mem.  |
                       |Unit  |
                       |M1-Mn|
                       +------+
```

The events passed from the queue unit to the state unit correspond closely to the sets of tokens passed from the matching unit to the fetch/update unit of a dataflow machine. The time wheel in the queue unit corresponds to the matching store in the matching unit, except that time is used as a tag. The instruction packets propagated from the state unit are identical to the packets passed from the fetch/update unit in a dataflow machine. And the events passed from the evaluation unit are identical to the data tokens passed from the processing units of a dataflow machine.

The points where they are not identical are that a dataflow machine has no analog to the state array, a data flow architecture usually has multiple state units, and dataflow matching is done by both node and iteration. The state array is an artifact of the simulator's implementation. It is the most efficient thing to do given the simulator's event driven sematics and the small number of processing units. It could be done away with entirely if all components were made call by value. In the case where we supported multiple state units, a distributed state array might cause the processor to be bottlenecked through heavily used state units. However, since the circular pipeline is currently well balanced, adding extra state units buys us nothing anyway.

It has been shown[Paseman84] that one can emulate call by value semantics easily in an event driven architecture by balancing the graph. In this case, the time wheel automatically does matching by both node and time. The state array is then made superfluous in the call by value case.

9.Ø Conclusions:

We have stressed that raw speedup should not be the only consideration in developing an accelerator. In particular, our goals were to balance speed with flexibility. We have shown how the relationship between event driven simulation and dataflow can be exploited in making a simulation accelerator.

Acknowledgements:

We would like to acknowledge the support of the other people on the Megalogician implementation team: Ben Lerner, Vered Ramon, Shan Shan Wang and Moshe Gray.

References:

[Deering85] Deering, Michael F. "Hardware and Software Architectures for Efficient AI." in Proceedings AAAI-84, 1984.

[Krohn81] Krohn, Howard E. "Vector Coding Techniques for High Speed Digital Simulation." Proceedings of the 18th Design Automation Conference, June 1981.

[Pfister82] Pfister Gregory F. "The Yorktown Simulation Engine: Introduction." Proceedings of the 19th Design Automation Conference, June 1982.

[Abramovici83] Abramovici et al. "A Logic Simulation Machine." IEEE Transactions on CAD of Integrated Circuits and Systems, Vol.CAD-2, No.2, April 1983.

[Sasaki83] Sasaki et al. "Hal; A Block Level Hardware Logic Simulator" Proceedings of the 2Øst Design Automation Conference, June 1983.

[Glazier84] Glazier et al. "Ultimate: A Hardware Logic Simulation Engine." Proceedings of the 21st Design Automation Conference, June 1984.

[Paseman84] Paseman W.G. "Processing Data Flow Graphs on an Event Driven Simulator." Daisy Systems Corporation, February 1984.

[Stoll85] Stoll, Peter A. "PMX: A Hardware Solution to the VLSI Model Availability Problem" Proceedings: ICCD '85 IEEE International Conference on Computer Design: VLSI in Computers, October 1985.

[Treleaven82] Treleaven P.C. et al. "Data Driven and Demand Driven Computer Architecture" Computing Surveys, Vol.14. No.1, March 1982. p114