LOGIC LEVEL AND FAULT SIMULATION ON THE RP3 PARALLEL PROCESSOR

Stephen H. Unger
Computer Science Department
Columbia University
LUCS 474-89

1. Introduction

Logic level simulation for circuits of the sizes currently being designed is indeed a formidable computational task. Chips are being built today containing over a million gates, with storage elements and RAMs. Ordinary logic simulation of systems of this size can take many hours of computation time on the fastest computers. Fault simulation for such large chips, is out of the question for anything but the largest supercomputer. Certainly this is a task justifying the use of parallel processing.

The 64 processor RP3 can serve as a useful test bed for experimenting with parallel processing techniques applied to the problem of logic level simulation and related DA algorithms, such as fault generation and simulation. I have been doing just this since last summer. I have developed a number of algorithms for general logic simulation and for fault simulation. These have been implemented with programs running on the RT under Mach. This permits testing the workability of the programs, since it simulates a parallel processor environment. But it gives no direct information as to how fast the programs would run on an actual parallel processor. I have run programs on the RP3, but my examples are not large enough to yield much meaningful data on speed.

In the following sections, I will describe the simulation algorithms that I have developed, and indicate what I propose to do in the future.

2. Combinational Logic Simulation - Overview

My goal is to develop a scheme that allows the work to be distributed among the processors in such a manner that:

(1) Each can do a substantial amount of work without having to wait for information from the other processors.

(2) The processors can share fixed information, principally the descriptions of the circuits being simulated.

The first requirement is necessary if the processors are to be usefully employed, i.e. if the useful work that each does is to be substantially greater than the overhead necessary to distribute and coordinate tasks.

The second requirement is important in keeping the memory requirements within reasonable bounds. If large circuits are to be simulated, it would be undesirable to require that each processor have a copy of the circuit description.

The approach that I have taken is to simulate on a level basis. That is, gates fed only by primary inputs (or latches) are on level 1, and a gate is on level 1 if the highest level gate feeding it is on level i-1. Given all of the primary input signal values, it is then possible to evaluate the outputs of all of the level 1 gates, independently of one another. Once all of the level i gates have been evaluated, it is then similarly possible to evaluate all of the gates on level i+1 independently of one another. Using a parallel processing system, we can attack the levels in sequence. At each level, the gates are partitioned equally among the processors. When every processor has completed the evaluation of the gates assigned to it, the gates on the next level are evaluated.

A single description of the circuit exists, and is accessible to all of the processors. The description consists of a list of gates (with their types) and, for each gate, pointers to the gates that feed it and to the gates it fans out to. Similarly, the current signal values at each gate output (as well as the primary input signal values) are also globally available. There is never an occasion for two processors to attempt to change the same signal value, nor is there any possibility of a processor changing a signal value that another processor must read.

The efficiency of this scheme may be improved (by how much is not clear) by evaluating only those gates for which at least one input has changed since the last time that gate was evaluated. (It is assumed
The basic fault simulation program is to determine which modules of a circuit or any size need to be evaluated.

4. Paul S. Sproull: Overview

Assignments for unknown logic values, the major improvement of the algorithm are the power of its elements as primitives, the assignment of unknown logic values, the major improvement of the algorithm are the power of its elements as primitives, the assignment of unknown logic values, the major improvement of the algorithm are the power of its elements as primitives.

The circuit evaluator is a part of my endeavor that is under development by a group of researchers who are actively working on this.

The circuit evaluator is a part of my endeavor that is under development by a group of researchers who are actively working on this.

The major improvement of the algorithm are the power of its elements as primitives, the assignment of unknown logic values, the major improvement of the algorithm are the power of its elements as primitives.

In order to fulfill the complete simulation of systems with multi-phase.

3. Sequential Circuit Simulation Overview

The most complex aspect of the program has to do with the simulation of sequential circuits, which are designed to simulate the behavior of circuits, especially those that have memory or storage elements such as registers, where implementation may be necessary. Simulation of sequential circuits is to be stand-alone. My simulation of sequential circuits is to be stand-alone. My simulation of sequential circuits is to be stand-alone.

In order to fulfill the complete simulation of systems with multi-phase.

Intermediate results are used in the feedback loops of the sequential circuit. It is also necessary to introduce a phase of the simulation in which some of the outputs of the sequential circuit are used to provide feedback to the circuit itself.
of sequential circuits, particularly those including RAMs. Various
lists of previous values of stored signals must be properly maintained.

5. Results To Date

The ideas discussed above have been implemented in programs written in
C, using the Mach C-threads system. They are running successfully on
both the RT and on the RP3. The examples used thus far are relatively
small circuits, the largest involving about 350 gates (most of them
are an order of magnitude smaller). These include sequential circuits
using small RAMs and also LS55 circuits. The results on the RP3 for
the two largest circuits run are as follows:

Circuit 38: 178 gate ALU (combinational logic) 21 inputs. 50 random
tests detected 418/1254 faults.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>699</td>
</tr>
<tr>
<td>2</td>
<td>376</td>
</tr>
<tr>
<td>4</td>
<td>221</td>
</tr>
<tr>
<td>8</td>
<td>164</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
</tr>
</tbody>
</table>

Circuit 30: A small circuit (designated DCDEX) designed for the RP3 by
Rory Jackson. 354 gates (including 20 latches), 24 inputs. 45 random
tests detected 396/1380 faults.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>332</td>
</tr>
<tr>
<td>16</td>
<td>219</td>
</tr>
<tr>
<td>32</td>
<td>185</td>
</tr>
</tbody>
</table>

The efficiency of the algorithm in utilizing multiple processors is
obviously a function of the size of the circuit being processed. For
example, if the number of gates per level is relatively small, then
there is not much work for each processor to do relative to the
overhead involved in getting them into play. Thus, I would not expect
this program to be really effective on circuits with fewer than many
thousands of gates.

At the input end, John Heaven has written some programs for converting
circuit descriptions generated via the SCALD graphics system to a form
that can be used by my programs. This would allow us to integrate my
simulator into the current design environment at Hawthorne and to
handle larger circuits. He has also done some preliminary work (on
the uni-processor version) to improve the input interface, and also to
reduce memory requirements. I have not yet incorporated these ideas
into my programs.

6. Proposed Further Work

I would like to continue along the following lines:

1. Thoroughly test the present version of my program on the RP3.
Include tests on real logic chips, such as those used in the RP3
itself.

2. Make measurements on the program to determine its speed and where
the bottlenecks are. Then determine how to eliminate them.

3. Modify the program to make it more efficient. For example, it
would certainly be useful to control memory allocation to ensure
that the variables used exclusively by a processor are assigned
to its local memory.

4. Incorporate into my program some of the new features mentioned
above (at the end of Section 5), particularly those pertaining to
the input interface.

5. Simulate some large, real, circuits to get some good speed
measurements. This depends on the enhancement of my program so
that it can fit in with the input interface referred to above.

6. Look into the problem of test vector generation.

7. Consider incorporating timing measurements.

8. Consider allowing simulation of multiple faults.

9. Consider applications to asynchronous sequential circuits.

10. Develop some ideas about program checking and debugging on an
RP3 type machine.
(11) Determine what characteristics of the RP3 are impeding faster operation of my program, and how might these be feasibly improved.

(11) Evaluate various RP3 features and generate some ideas for other new hardware and/or software features that would be useful and practical. (For example, how can the wait operation best be implemented?)

(11) Consider how to extend the ideas outlined here to systems using built-in-test.

(11) Since this is a research project, fruitful ideas for further work are likely to develop along lines not now evident.

7. Running Programs on the Current Version of the Simulator - Details

At present, circuit descriptions are in the form of a set of assignment statements, specifying the gates and the inputs to each gate. These descriptions are in files designated, for example as, cktls/38c.t. (All are in the directory cktls, the integer refers to the specific circuit layout in the letter s, such as 't', refer to variations in the description forms for various versions of the simulator. For example, versions 13 and 14 of the simulator work with circuit descriptions of type 't'.)

The circuit description includes variables ninpts (the total number of inputs - including feedback inputs), ngt (the total number of gates - including latches), nmems (the number of feedback inputs), and ncl (the number of clock signals). Circuit elements are, in general, multiple-input single-output devices, described by a structure labelled gate. (Input and clock signals are also specified as gate structures.) The gates are organized in a 1-dimensional array, g[], with indexes running from 0 to ngt - 1. Inputs are in an array i[] with indexes ranging from 0 to ninpts - 1, and clock signals are in an array c[] with indexes ranging from 1 to ncl. (This is true for the latest 2 versions of the simulator; in earlier versions all indexes start at 1.) For gate g[i], the type is given by g[i].fn. For example, g[i].type = 'b' specifies gate 3 as a NAND-gate. The statement g[i].fn = '4' specifies that g[i] has 4 inputs. The statement g[i].inp = new = gen_ptr_array(4) creates a pointer to an array of 4 pointers that will contain pointers to the 4 inputs. These are specified by further statements such as: *new = &g[2] (indicating that the first input is from g[2]). *new += 1 = &g[17] (indicating that the second input is g[17]), etc. A feedback input is an s receiving its signal from a latch g[6] would be specified by the statements: i[22].inp = new = gen_ptr_array(1), and *new = &g[6].

Circuit descriptions are compiled separately into object code and linked later to the compiled simulator programs. A shell script, sim, is used to compile and run. It calls on various make files to ensure that up-to-date versions are produced. Sim has 4 parameters. They indicate whether the program is to run on the RT or on the RP3, which version of the simulator is to be used, how the faults are to be supplied, and what circuit is to be tested. Thus, the command sim rt 14 m 38 would run version 14 on the RT with circuit 38. The parameter m specifies that the "m" faults are to be generated, meaning that both stuck-at-0 and stuck-at-1 (abbreviated here as 0 and 1) will be generated at the outputs and inputs of each gate, but there will be no duplication of the same fault at the output of a gate with fanout 1 and the input of the gate it feeds. Other options are "a", which does not eliminate the redundancy described above, "d", which does not include a fault if all tests for some other fault also test for it, and "i", which calls for a user supplied list of faults. (These are taken from a file tflts/38, where 38 is the circuit number.)

In all cases, the test vectors are taken (for circuit k) from file tsts/k. A prefix file, called tpre/k (prefix/k for versions of the simulator prior to 13) corresponds to each circuit. The first line of this file, is a y or n, indicating whether or not detailed printouts are desired. The next line specifies the number of processors to be used. Next comes a list, terminated by a line with a ""); of gates whose outputs are to be printed (if the first line is a "y", and then comes a similar list, also terminated by a ""); of the observable gates (i.e. observable for fault detection purposes). The sim shell concatenates tpre/k, tflts/k (for the "i") case), and tsts/k into a file called inpt. The executable program is placed in the file simprog. Where the RT was specified, sim causes simprog to be executed, with inpt redirected to it and the results redirected to res/k (for the "i") case). When the first argument given to sim is rp3, it is necessary for the user to ftp simprog and inpt to the RP3 and then telnet the appropriate command to call for execution. In my RP3 file, I have a shell called rrp3 which causes simprog to be
Acquiring control over all gates whose outputs it has changed.

Receives the simulation clock, wakes up all gates. If the clock signal is high, gate is activated. If the clock signal is low, gate is deactivated.

When a gate is activated, it updates its output based on the current inputs. The updated output is then passed to the next gate in the circuit. If the gate has an output that is not connected to another gate, it is considered as an output of the circuit.

For this purpose, all gates contain a flip-flop to store the current state of the gate. When the clock signal is low, the flip-flop is reset to its initial state. When the clock signal is high, the flip-flop is set to the current state of the gate.

In the output stage, the output of each gate is propagated to the next gate in the circuit. If the output of a gate is connected to the input of another gate, the output of the first gate is used as the input to the second gate.

8. Fault Simulation (of Combinational Logic Circuits - Closest Look)

In the directory simulation, all combinations of input values are tested to determine if the output of the circuit is consistent with the expected output. The purpose of this simulation is to identify any faults or errors in the circuit design.

The simulation process begins by selecting a specific input combination. The circuit is then simulated to determine the output of the circuit. If the output is not consistent with the expected output, a fault is identified.

The simulation process is repeated for all possible input combinations. The results are then analyzed to determine the type and location of any faults. The results of the simulation are used to improve the design of the circuit.
9. Fault Simulation of Sequential Logic Circuits—Closer Look

If a circuit has memory, in the form of storage elements such as latches, or FFs, then the fault simulation process must take this into account by keeping track of faulty states of such devices. Suppose that, as a result of some input acting on a circuit with a stuck-fault at some gate terminal, no observable output is changed (so that the fault is not detected), but the states of one or more latches are affected (i.e., are different from their valid circuit values as a result of the existence of the fault). Then it is possible that a subsequent input, in conjunction with these false signals may propagate to an observable output, hence revealing the fault. In order to simulate this behavior, the fault descriptors, introduced in the preceding section, include pointers (flitchstk) to stacks listing latches (references to latches also apply to FFs) whose values become false due to the original fault.

The checkfault routine pushes onto a stack, called newlitchstk, latches whose values have been changed from their valid values in the manner outlined above. For faults not yet detected, procfault attaches newlitchstk to the fault descriptor. Injflit copies the stack of latches with false signals (if such exists as part of the descriptor of the fault currently being processed) onto a stack called oldlitchstk. The latches involved are pushed onto the appropriate actgstsks. Injflit then treats the corresponding feedback inputs (if any) as though they were terminals with stuck faults. Checkfault takes into account the existence of such latches when determining fltlev (lowest level of gates to process). Procfault and restorez free memory allocated to these stacks when no longer needed.

Suppose that, during a faulty circuit evaluation of a latch, it is found that the clock input to that latch is 0. Then the output of that latch should be the same as the output it had during the previous input. But how is that value to be found? If the latch is on oldlitchstk, then the entry on that stack will contain the required value. But if the latch is not on oldlitchstk, the proper value is the valid output of the latch during the previous input. But the present value of the valid output of the latch (which is the result of the present input) may be different from the past value (the clock input for the valid circuit may be a 1, as opposed to the 0 for the faulty circuit). In order to take care of this situation, the data structure for a gate includes oldz, the value of the valid output after the last input, and a stack, chlitchstk, of latches whose values were changed by the current input is maintained during the simulation of the valid circuit. The routine upatoldz is part of this process. (I believe that oldz can be eliminated and the old latch value can better be kept on chlitchstk.)

10. Fault Simulation of Logic Circuits Containing RAMs

Since RAMs (usually of modest size) are sometimes included in logic circuits, it is useful for the simulator to be able to treat them as gates during simulations. This does, however, introduce some complexity into the process of fault simulation.

For the valid circuit simulation, a memory location is reserved for the contents of each memory location of each RAM. These are maintained in a straightforward way during the valid circuit simulations. (RAMs are treated as multi-input, single output gates, in a manner similar to the way latches or NOR-gates are handled.) For reasons similar to the need for the stack of changed latch values (see preceding section), it is necessary to maintain a stack (chmemstk) of RAM locations whose values have changed (along with the old values).

In order to handle fault simulation of such circuits, some further additions to the data structure are needed. Each fault descriptor must contain a pointer (fmsstk) to a stack of faulty memory locations (the index of the RAM, the local address within that RAM, and the faulty value are all stored on that stack for each false value in a RAM, whether due to a stuck fault in memory or to the consequences of other faults). During fault simulation, injflit generates a pointer (oldfmsstk) to this stack, whose contents may be altered during the simulation. In addition, a new stack (newfmsstk) of memory locations that acquire false values for the current input is produced by the checkfault. If the current input does not detect the fault, then oldfmsstk and newfmsstk are concatenated in procfault, and the result attached to the fault descriptor via fmsstk, so it becomes the oldfmsstk for the next input.
When faults are entered by list (the option) memory faults are entered in a special form (see the file crflist6.c). For example, \texttt{m12,3,0} specifies a memory fault in which location 3 of the RAM \texttt{g[12]} is \texttt{00}. This description is converted by the program to a form analogous to that for the other fault types. Its internal form beginning with a \texttt{0} for the type, would be \texttt{000,12,} and there would be a pointer \texttt{fmsk} to a stack of false memory values that would have, as its first (bottom) entry, a structure \texttt{(mvalst,k)} with components indicating false value \texttt{(mfval)}, gate index \texttt{(gtindex)}, address \texttt{(mloc)}, and a pointer to the next item on the stack \texttt{(nxt- initially NULL)}.

All RAMs with false memory contents are placed on the actgtstk by injft. When eval is evaluating the output of a RAM during a faulty circuit evaluation, a number of situations must be taken into account.

(1) If the operation is neither write nor clear, then the function chkmw is called to determine if the valid circuit simulation changed any memory bits in this RAM. If so, a false memory value listed on oldfmsk might have been corrected or changed, in which case an element of oldfmsk is deleted or changed (if it does not correspond to a stuck fault). If the memory location is not on oldfmsk, then a new false memory entry must be created and put on newfmsk.

(2) If the faulty circuit simulation is executing a read operation, then rdms searches oldfmsk for the location involved; if it is there, then the output is the associated false value. Else it is the valid contents of that memory location.

(3) If the operation is write, and the value \texttt{t} to be written differs from the valid circuit value stored at the specified memory address, then prmfch is called to see if that location is on oldfmsk. If it is, then prmfch updates the value on that stack if necessary. Otherwise prmfch adds a new item to newfmsk. If \texttt{t} is equal to the valid contents of the memory location, then another function, prnnfch, is called to search oldfmsk for an entry at this location and to delete it if it exists (and is not a stuck fault). In both cases, chkmwb is called to determine if the valid circuit simulation wrote at different location of the same RAM. If so, then it may be necessary to add a new fault to newfmsk.